Fast counting with the optimum combining tree

Author[s]:
Wattenhofer, Roger; Widmayer, Peter

Publication Date:
1998

Permanent Link:
https://doi.org/10.3929/ethz-a-006652266

Rights / License:
In Copyright - Non-Commercial Use Permitted
Fast Counting with the Optimum Combining Tree

Roger Wattenhofer       Peter Widmayer

Institut für Theoretische Informatik
ETH Zürich, Switzerland
{wattenhofer,widmayer}@inf.ethz.ch

Abstract

A distributed counter is a concurrent object which provides a test-and-increment-operation on a shared value. On the basis of a distributed counter, one can implement various fundamental data structures, such as queues or stacks. We present a fast, linearizable counting scheme for processors that increment at arbitrary rates. Our counter is efficient in both, a message passing and a shared memory environment; we describe in detail the former. We analyze the expected behaviour of our scheme using queueing theory. In our simulations, we compare our scheme with Counting Networks and Diffracting Trees.

1 The Problem

We observe an ever increasing importance of distributed data in our network-centric universe. Even though data is distributed over several processors, any processor should be able to access data at any time. Such an access may be triggered either by a human user or a running application program. The design and analysis of distributed data structures draws theoretical interest from the fact that, contrary to distributed algorithms, processors in distributed data structures generally compete against each other rather than cooperate: An operation triggered by processor $p$ may interfere with an operation of processor $q$, and it may thus make the work invested by $q$ futile.

In particular, distributed solutions have been proposed for counting, a basic step in virtually any computation. A distributed counter is a variable that is common to all processors in the network, and that supports an atomic test-and-increment operation: It delivers the counter value to the requesting processor and increments it. The counter is required to satisfy an elementary soundness condition: Whenever no operation is active in the system (it is in a quiescent state), the mechanism has delivered consecutive counter values, with none missing and none delivered twice. Sometimes applications require that in addition, a counting scheme be linearizable [HW90] in the sense that whenever the first of two operations finishes before the second starts, the first gets a lower counter value than the second.

The quest for finding a most efficient scheme for distributed counting is strongly related with the quest for finding an appropriate measure of efficiency for distributed data structures. The obvious measures of efficiency for distributed systems, such as time or message complexity, will not work for distributed data structures. For instance, even though a data structure could be message and time optimal by just storing the whole data structure with a single processor and having all other processors access the data structure with only one message exchange, such an
implem entation is clearly unreasonable: This solution does not scale — whenever a large number of processors operate on the data structure, the single processor handling the data structure will be a bottleneck. In other words, the work of the algorithm should not be concentrated at any single processor or within a small group of processors, even if this optimizes some measure of efficiency.

If the primary goal is a good behaviour of the counter in a practical situation, Counting Networks [AHS91] are an excellent solution. They make sure that message contention at each individual node of the network is low — an important precondition for efficient counting. In addition, however, any single operation in an efficient counter should not be forced to travel through a large number of nodes. These two goals appear to be in conflict with each other, and any fast counting scheme must respect both of them. Diffraclng Trees [SZ94] have an ingenious design that also avoids contention at nodes (by means of diffraclors) and thereby achieve very high speed (see also Section 5). Both, Counting Networks and Diffraclng Trees, are not linearizable; Counting Networks can be made linearizable [HSW91], with a significant extra effort that makes them by far less efficient. One might ask whether in principle, the hot-spot problem of a central counter can be overcome without losing other desirable properties, such as the ability to compute arbitrary functions (instead of counting only). One approach here tries to minimize “bottleneck complexity” — the number of messages which a “busiest” processor of the system must exchange [WW97b]. Even though this tells about the lower bound of how much a basic distributed data structure such as counting can be decentralized, it is not advisable to construct a real counting scheme directly alng this theoretical suggestion, just because the bottleneck complexity abstrasts too much from reality.

In this paper, we propose a scheme on the basis of the Optimum Combining Tree (OCtree) of [WW97a]. The OCtree itself follows the basic idea of Combining Trees [YTL86, GVW89]. We show how both conflicting goals, low message contention at individual nodes and short paths along which messages travel, can be assessed uniformly by means of traditional queueing theory. Our analysis measures the operation time — the time spent from the start of a test-and-increment-operation to its completion. We restrict ourselves to the M/M/1-model for every single processor in the system. This model, where the rate for arriving messages as well as the service rate model Markov processes and we have only one server, is the most fundamental and simplest to calculate with. In addition, it appears to be a good approximation for the real world.

The model of computation that we use to explain our counting scheme is presented in the next section; The scheme applies similarly to a parallel environment, such as an EREW PRAM. In Section 3, we study the extreme cases where access of the processors to the counter is very low or very high. Section 4 presents a counting scheme that can be adjusted to any access rate whatsoever. We argue on the performance of the scheme using queueing theory. In Section 5, we show simulation results for several counting schemes on a distributed virtual machine. Finally, Section 6 discusses properties beyond pure speed. Charts and the Java source and class code of the simulation will be found at the web site http://www.inf.ethz.ch/personal/watten/sim/

2 The Model

Consider an asynchronous, distributed system of \( n \) processors in a message passing network, where each processor is uniquely identified with one of the integers from 1 to \( n \). Each processor has unbounded local memory; there is no shared memory. Any processor can exchange messages
directly with any other processor. A message arrives at its destination an unbounded, but finite amount of time after it has been sent. No failures whatsoever occur in the system.

An abstract data type distributed counter is to be implemented for such a distributed system. A distributed counter encapsulates an integer value val and supports the operation inc (short for test-and-increment): When a processor initiates the inc operation, then the system's counter value is returned to the initiating processor and the system's counter value is incremented (by one).

For concreteness in our subsequent calculations, let us assume that a message takes $t_m$ time on average to be transferred from some processor to some other processor. In addition, assume that some amount of local computation at a processor takes $t_c$ time on average.

Since more than one message may arrive at a processor $p$ at (approximately) the same time, $p$ queues all incoming messages and consumes them sequentially in first-come-first-serve order.

### 3 Counting when Load is Extreme

Let us study the cases of extreme load first. Assume that accesses of the processors to the counter are very sparse: The time between any two increment initiations for some processor $p$ is long enough that even a Central Scheme, where the current counter value is stored at a distinguished central processor $c$, can handle all requests one after the other, without being a bottleneck processor.

Let us assume that the time that elapses between the start of two consecutive increment operations by the same processor is distributed exponentially with expected value $t_i$, for every processor. If $t_i \gg n \cdot t_c$, there will be no queue usually when a request arrives at the central processor $c$. Therefore $c$ can respond immediately, resulting in an operation time of roughly $2t_m + t_c$. But, although it is very unlikely, it is still possible that many requests arrive at $c$ at more or less the same time. Then these requests are handled sequentially one after the other. Therefore, the last request in the queue takes about $2t_m + nt_c$ operation time, a delay that is certainly not competitive with decentralized solutions (even when access is sparse).

Queueing theory has proven to be a very powerful tool to argue about expected queue sizes and service times. When choosing an exponential distribution for every involved random variable, we can use the simple standard $M/M/1$ queueing theory to analyze the performance. The arrival rate at $c$ is $\lambda = n/t_c$, the service rate $\mu = 1/t_c$. The ratio $\rho = \lambda/\mu$ is the utilization of $c$. In order to have bounded expected queue size, we must assure that $\rho < 1$, that is $t_i > nt_c$. Note that this restriction is quite natural. Whenever $t_i \leq nt_c$, the Central Scheme is not meaningful, since there is an unbounded queue at $c$; we therefore must choose some decentral counting scheme to have good estimated operation time. But also when $\rho$ is very close to 1, the average operation time is not very good. To not only guarantee limited, but also a quite small expected queue size, we restrict $t_i$ such that $t_i \geq 2nt_c$. From queueing theory, we know that the expected time that a request spends at $c$ (known as the expected response time $t_s$ = time in the queue + time to be processed by $c$) is $t_s = \frac{1}{\mu(1-\rho)}$. Then, the expected operation time is $2t_m + t_s$, with $t_s = \frac{t_c}{1-t_c/1}$. Since $t_i \geq 2nt_c$, we get $t_s \leq \frac{t_i}{1-t_i/2} = 2t_c$. Thus, the expected operation time is bounded from above by $2(t_m + t_c)$, which is optimal up to a constant factor.

On the other extreme, let us study the case in which we have very high access rates. In [WW97a], we proposed a counting scheme, the OCtx, and proved that the OCtx has a
maximal operation time of $O(T_m \log T_m / T_c \; n)$, where $T_m$ resp. $T_c$ is the maximum message resp. calculation time.

Since our scheme for low load (the Central Scheme) can be seen as a special case of the OCtree (an OCtree with height 1), the OCtree can be adjusted to any access rate whatsoever.

4 Counting when Load is Arbitrary

In this section, we will present a counting scheme that adapts gracefully to any access rate whatsoever. We have a rooted tree with height $h$: the root is on level zero, all leaves of the tree are on level $h$. Every inner node $u$ on level $0, \ldots, h-2$ has $k$ children with $k > 1$. Every inner node on level $h-1$ has $k' := \lceil n/k^{h-1} \rceil$ children. Note that the tree has $k' \cdot k^{h-1} \geq n$ leaves. We will make sure that $k^{h-1} < n$, thus $k' > 1$.

Since $k > 1$ and $k^{h-1} < n$, it is possible to give each inner node a distinct number between 1 and $n$. Furthermore, we choose $n$ of the leaves and give them distinct numbers from 1 to $n$. All other leaves are discarded. We identify each inner node resp. leaf with its number. Because for each $p$ with $p = 1, \ldots, n$, there is exactly one leaf $p$ and at most one inner node $p$ in the tree, processor $p$ acts for its leaf and for its inner node (if it exists). To achieve this, processor $p$ will communicate with every neighbour of these two nodes in the tree. We make sure that every processor knows its neighbours.

A simple strategy to handle the inc-operations is the following. The current counter value $val$ is stored at the root of the tree. Whenever a processor $p$ wants to increment, the leaf $p$ sends a request message to its parent, which in turn will send a message to its parent and so on until the request arrives at the root. The root then assigns in its response a value $val$ to this request, and $val$ is sent down the tree along the same path until it arrives at the leaf $p$.

However, this simple approach is bound to be inefficient since the root node is a hot-spot, and the performance of the system is not better than a Central Scheme. To overcome this problem, we let the inner nodes combine several requests from their children. That means, instead of just forwarding each individual request up the tree, an inner node tries to combine requests for as many counter values as requested by its children at “roughly the same time” (within a certain time frame). On the other hand, we have to guarantee that requests resp. counter values are forwarded up resp. down the tree quickly. Consequently, we only need two kinds of messages: the upward and the downward messages. An upward message is sent up the tree and consists of only one integer, the number of increment operations requested by the subtree. A downward message is sent down the tree and consists of an interval of counter values (assigned by the root), specified by the first and the last counter value of the interval. Let us describe this counting scheme more precisely by defining the behaviour of the nodes:

Leaf

Let leaf $p$ initiate the increment. To do so, leaf $p$ immediately sends an upward message (asking for one counter value) to the parent node in the tree.

Later, the leaf gets a downward message from the parent with an assigned counter value.

Root

When receiving an upward message from a child asking for $z$ counter values, the root instantaneously returns a downward message with the interval $\{val, \ldots, val + z - 1\}$ to the child and increments $val$ by $z$. 

4
In order to arrive at a fast scheme, we set the parameters as follows:

To simplify bookkeeping, all we do in our analysis is counting twice as many incoming upward messages plus outgoing upward messages at a node and forget about the incoming and outgoing downward messages.

The way down is symmetric. Whenever $p$ receives a downward message from its parent, it distributes the given interval to the children, according to the entries at the beginning of its queue.

Let us now argue on the performance of this counting scheme, assuming that every processor knows $t_m$ and $t_z$, where $t_m$ is sufficiently larger than $t_z$, more specifically, $t_m \geq 4t_z$. For the moment, let us assume that $t_i \geq t_w$; we will later show how to get around this restriction. Moreover, let $t_i$ be the same for every processor.

In order to arrive at a fast scheme, we set the parameters as follows:

$$k := \left\lceil \frac{t_m}{t_c} \right\rceil, h := \left\lfloor \log_k \left( \frac{4nt_c}{t_i} \right) \right\rfloor + 1, t_w := 4t_m.$$

**Up-Down Proposition:** At each inner node, the amount of local computation from handling all upward messages is the same as the amount of local computation from handling all downward messages.

**Proof:** An inner node $p$ might receive far more upward messages than downward messages. On the other hand, upward messages can always be handled in a constant amount of local computation steps, whereas an interval from a downward message has to be distributed to possibly many children. To simplify the argumentation, let us model a downward message as a whole set of downward entities, every one meeting the demands of exactly one pending entry in the queue. This can be done in a constant amount of local calculation steps, too. Moreover, since every upward message generates a queue tuple and every downward entity removes a queue tuple, the proposition follows. \(\square\)

To simplify bookkeeping, all we do in our analysis is counting twice as many incoming upward messages plus outgoing upward messages at a node and forget about the incoming and outgoing downward messages.

**Timing Lemma:** At every inner node, any upward message is forwarded to the parent after $8t_c$ expected time.

**Proof:** Since inner nodes on level $h - 1$ are an exception, we restrict ourselves to inner nodes on level 0, ..., $h - 2$ for the time being. The arrival rate of upward messages at inner node $p$ is $k/t_w$. Inner node $p$ sends an upward message to its parent after time $t_w$. By taking care of the downward messages and some time for leaf $p$ for initiating (and consuming) an increment after time $t_i$, we have an arrival rate $\lambda \leq 2(k/t_w + 1/t_w + 1/t_i)$. With $t_i \geq t_w = 4t_m$, we get $\lambda \leq \frac{2(k+2)}{t_w} = \frac{k+2}{2t_m}$. Since each of these events takes $t_c$ expected time, we have $\mu = 1/t_c$.

Therefore, the utilization ratio is $\rho = \lambda/\mu \leq \frac{(k+2)t_c}{2t_m}$. Since $k + 2 = \left\lceil \frac{t_m}{t_c} \right\rceil + 2 \leq t_m/t_c + 3$, we
know that $\rho \leq \frac{(\ell_w/t_w+3t_c)}{4t_m} = \frac{t_c+3t_c}{4t_m}$. With $t_m \geq 4t_c$, we get $\rho \leq 7/8$. Using the bounds for $\rho$ and $\mu$, the expected response time is $t_s = \frac{1}{\mu(1-\rho)} \leq 8t_c$.

One can argue analogously on inner nodes on level $h-1$. Again, the number of children $k'$ and the height of the tree $h$ are chosen carefully, such that the requests do not “overheat” the node. In detail, the arrival rate of inner node $p$ is $\lambda \leq 2(k'/t_i+1/t_w+1/t_i)$. Since $h = \left\lfloor \log_\lambda \frac{\lambda^k}{t_i} \right\rfloor + 1$, we have $k^{h-1} \geq \frac{4\mu}{t_i}$. Therefore $k' = \left\lfloor \frac{n}{k^{h-1}} \right\rfloor \leq \frac{n}{k^{h-1}} + 1 \leq \frac{4\mu}{t_i} + 1$. Thus, using the facts that $t_i \geq t_w = 4t_m \geq 16t_c$, we get $\lambda \leq 2(k'/t_i+1/t_w+1/t_i) \leq 2(\frac{4\mu}{t_i}+1/t_i+1/t_w+1/t_i) \leq 2(\frac{4\mu}{t_i}+3/t_w)$.

Again, each of these events takes $t_c$ expected time, that is $\mu = 1/t_c$. Therefore, the utilization ratio is $\rho = \lambda/\mu \leq \frac{2t_i}{4t_c} + \frac{6t_c}{t_w} \leq 1/2 + 6/16 = 7/8$. Using the bounds for $\rho$ and $\mu$, the expected response time is $t_s = \frac{1}{\mu(1-\rho)} \leq 8t_c$. \hfill\square

This immediately leads to the following.

**Counting Theorem:** The expected operation time for an increment operation is

$$O \left( t_m \log_{t_m/t_c} \frac{nt_c}{t_i} \right).$$

**Proof:** The Timing Lemma shows that forwarding a message at an inner node takes only $O(t_c)$ time. From the Up-Down Proposition, we know that this holds for sending down too. Handling one message goes along with transferring one message ($t_m$) and waiting until one might send it up ($t_w$). All up the tree and down again. Using $t_w = 4t_m \geq 16t_c$, we get for the total expected time $2h(t_c + t_m) + 3t_w < 7h \cdot t_m$. With $h = \left\lfloor \log_\lambda \frac{\lambda^k}{t_i} \right\rfloor + 1$ and $k = \left\lfloor \frac{l}{t_i} \right\rfloor$, the theorem follows. \hfill\square

In this section, we have introduced two restrictions to simplify the arguments. Let’s get rid of them. One constraint was $t_i \geq t_w (= 4t_m)$. When processors are very active and initiate the increment operation very often ($t_i < 4t_m$), we don’t allow them to send an upward message to the parent immediately, but we force them to wait for at least time $4t_m$ instead, and to already combine several requests into one message. Since $t_i = O(t_m)$, this does not introduce any extra waiting time asymptotically.

The other constraint is $t_m \geq 4t_c$. Since sending/receiving a message includes always at least some local computation, this constraint will usually be satisfied naturally. However, if the opposite is the case, all we have to do is setting up a binary tree with $k = 2$ and adjust $h$ resp. $t_w$ to $\left\lfloor \log_\lambda \frac{nt_c}{t_i} \right\rfloor$ resp. $4t_c$. Again, one can show a similar Timing Lemma and Counting Theorem with an upper bound of $O(t_c \log_{t_c} \frac{nt_c}{t_i})$.

With arguments similar to those in [WW97a], we can prove a lower bound on the expected operation time of

$$\Omega \left( t_m \log_{t_m/t_c} \frac{nt_c}{t_i} \right).$$

Therefore, the presented counting scheme is asymptotically optimal in the expected case.

## 5 Simulation

A real implementation of a counter would operate differently in detail. For instance, an inner node would not wait for an exponentially distributed time to send a combined upward message to its parent, because there are simpler methods (like sending an upward message immediately if none was sent within a certain amount of time – possibly $t_w$) having the same effect. Although queueing theory is a suitable tool when arguing about systems where everything is Markov
distributed, the analysis gets rather complicated for non-standard distributions. Therefore, we complement our analysis with a simulation. Simulation has often been an excellent method to assess the performance of a distributed scheme, starting with the work of [HLS95]. For purity and generality reasons, we decided to simulate the counting schemes not for a specific machine (such as the preferred Alewife), but for a distributed virtual machine (DVM), in analogy to PVM in the parallel world.

The DVM consists of \( n \) processors, communicating by exchanging messages. A constant number of local computation steps cost \( t_c \) time for a processor. The time it takes a message to be sent to its destination is denoted by \( t_m \). Since several processors might send a message to the same processor, every received message is stored in a queue of incoming messages and consumed one after the other. Note that both, sending and consuming a message, need some local computation (for sending: specify the receiver and start the sending process; for consuming: mark the message as read and remove it from the queue of incoming messages). With our DVM, it is not possible to broadcast a message to several other processors in one step. In other words, if a processor is to send a message to every other processor in turn, the last processor will not consume the message before \( O(t_c + t_m) \) time. Both, \( t_c \) and \( t_m \), do not have to be constants, but can be random variables with arbitrary probability distributions. This way, one can easily simulate the performance of a system where some messages are delayed or some processors might have some temporary performance problems (due to caching, for example). As a virtual machine, the DVM abstracts from reality in a number of aspects. Although every counting scheme we have tested uses no more than a constant number of local computation steps to work on a pending event, the constants do certainly differ – a fact simply ignored by the DVM. But the constants do not differ that much; when we compare the work of a simple element (e.g. a balancer in a Counting Network: receiving a message, toggling a bit, sending the message according to the bit) with a complex element (e.g. an inner node in an OCl tree when receiving an upward message: receiving a message, storing the message in a queue, adding the value \( z \) to \( \text{sum} \)), the difference is not that big. We accepted this simplification to have the opportunity to experimentally evaluate the counting schemes on very large scale DVM's with up to 16k processors, a dimension which is surely intractable for a simulation model that is close to a real machine, such as the favoured Proteus simulator [BDCW92].

On our DVM, we have implemented four counting schemes. We are not going to discuss them in detail here, but merely give a hint of how we have chosen the parameters. Of course, choosing the parameters is not a simple task, in particular because some of the schemes were proposed for shared memory and not for message passing. However, instead of relying on analysis only, we have tested the schemes with various parameters to find the best.

Central Scheme

The Central Scheme has no parameters; the implementation is straightforward. Whenever \( \rho \geq 1 \) for the central server, the system collapses. One simple solution to get around this severe problem is only allowing the processors to have at most one pending request and combine all initiations in one packet until the last request has been answered.

Bitonic Counting Network introduced by [AHS91]

One natural transformation from the original shared memory model to our message passing model is by assigning every balancer and every output counter to a distinct processor. That is, no processor is to act for more than one balancer or output counter. The only parameter of interest is the width \( w \) of the network. On one hand, \( w \) has to be small enough, so that the number of processors in the system suffices to instantiate every balancer and the output
counters, resulting in $n \geq \frac{w}{4/(\log_2 w + 1) \log_2 w + w}$. On the other hand, $w$ should be large enough to handle the requests, that is $\rho < 1$ for every processor, resulting in $n \cdot t_c/t_i < w/2$. Between these two bounds, we were looking for the width with minimal operation time.

**Diffraction Tree** introduced by [SZ94]

Again, every prism element, every toggle element and every output counter are implemented by a single processor, such that no processor has more than one job. To set the parameters, the height of the binary tree, the prism size, and the spinning time, we follow the rules of [SUZ96]. As for the Bitonic Counting Network, there are constraints on parameters.

**OCtree**

As for Bitonic Counting Networks and Diffraction Trees, we tested several combinations of parameters, to find the best performance.

![Simulation Results](image)

**Figure 1: Simulation Results**

Figure 1 shows the throughput of the four counting schemes in a system from 1 up to $16k$ processors. For this chart, we have been using the following system characteristics: Each processor executes a loop that increments the counter whenever it received the value for its previous operation. [HLS95] call this the Counting Benchmark. Calculation time and message time are deterministic with 1 resp. 5 time units each.

As long as the number of processors in the system is small, all schemes show equal performance, since every scheme degrades to a Central Scheme. When the number of processors is rising, the Central Scheme is getting useless. When having $n$ processors in the system, the average operation time for the Central Scheme is about $n \cdot t_c$. Since the Bitonic Counting Network has a log^2 depth, it also loses time when the number of processors is very high. The two most efficient schemes are Diffraction Trees and the OCtree.

Although efficiency is very important, there are other qualities a counting scheme can offer. We will discuss some of the more important ones in the next section.
6 Discussion

Linearizability

The proposed OCtree implementation is similar in speed to the Diffracting Trees, but distinguishes itself clearly from the latter by realizing a linearizable counting scheme. Therefore, the OCtree is the only scheme that is linearizable in its very nature and efficient at the same time.

Powerful Operations

Simple counting schemes allow only the test-and-inc-operation. To implement various important data structures (e.g. stacks), one needs more powerful operations. [ST95] have shown that Diffracting Trees can be extended to Elimination Trees offering both, a test-and-inc and a test-and-dec-operation, and implementing stacks and pools that way. The OCtree is able to add or subtract any value, and many operations beyond that. The only restriction on the operations are that they are combinable without using too many bits. Since the Central Scheme does not have any restrictions at all, it is the only known structure to implement more complicated data structures (e.g. priority queues). There is also another advantage of having the possibility for powerful operations: Imagine having a system where the load is very high – [YTL86] call this an “unlimited” access system. Using an OCtree, the processors initiate the inc-operations in bulk. They always ask for several values at once, and they therefore do not overload the system. This is not as easily possible with Diffracting Trees or Counting Networks.

Adaptivity

Any counting scheme of general relevance should be adaptive – the scheme should perform well under high load as well as under low load. Since the Central Scheme has no system parameters, it is not adaptive at all. Counting Networks do not change their structure incrementally; it is a difficult task to give them an adaptive extension. For Diffracting Trees, [DS97] have proposed an appealing adaptive version, the Reactive Diffracting Trees. The very same idea, the folding and unfolding of nodes, can be used for the OCtree also. Future experiments will prove whether an adaptive OCtree is efficient for more general dynamic load balancing problems.

Acknowledgments

We’d like to thank the Swiss National Science Foundation for partially supporting this research.

References


