Report

Distributed counting at maximum speed

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

Publication Date:
1997

Permanent Link:
https://doi.org/10.3929/ethz-a-006652199

Rights / License:
In Copyright - Non-Commercial Use Permitted
Distributed Counting at Maximum Speed

Roger Wattenhofer       Peter Widmayer
Institut für Theoretische Informatik
 ETH Zürich, Switzerland

1 The Problem

The importance of distributed data structures for sharing a high data access load among the processors of a network with no shared memory has been widely recognized in the past few years. Each processor holds some share of all the data, and access operations are triggered spontaneously at any processor (by a human user or by a running application program). The theoretical interest in the design and analysis of distributed data structures has been fueled by the fact that unlike for distributed algorithms, processors in distributed data structures do not primarily cooperate to solve a problem, but rather compete against each other: The operation triggered at one processor may (adversely) affect the result of the operation triggered at some other processor. For instance, whenever two processors simultaneously try to delete the minimum in a priority queue (used in a distributed branch-and-bound computation), only one of them can succeed; the other has to go for the second smallest element.

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. Its implementation must aim at avoiding the hot spot problem of a central solution, and at keeping the number of exchanged messages small. Early proposals towards this end suggest to combine requests in a tree [YTL86, GVW89]. To avoid having a processor play the governing role of the root in combining trees, counting networks (topologically identical to sorting networks, but with balancing elements instead of compare-and-exchange modules) have been proposed [AHS91, AHS94] and developed further [AA95, AA92, BM96, KP92].

Applications often require that 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. Counting networks are not linearizable, but merely 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. They can be made linearizable [HSW91, HSW96], with a significant extra effort that makes them by far less efficient (for restricted machine models, [LST96, MPT97] show that counting networks are linearizable). The literature on counting networks reflects the substantial effort that has been invested in the search for a fast linearizable counter as a fundamental distributed object. In spite of this effort, it seems that no satisfactory solution has been found until now. Diffracting trees, the third major family of counters [SZ94, SZ96, SUZ96] and their variations [DLS97, ST95], are no exception, because they are also not linearizable; that is, they are limited to applications where the order of counter values is of no importance.
In this paper, we propose an efficient linearizable counter. The definition of efficiency for distributed data structures in general and for a distributed counter in particular is not as immediate as in the uniprocessor case. The number of messages exchanged among processors, or the time needed from the start of an operation to its completion (provided that a message is delivered in unit time) are reasonable measures, but do not suffice: According to these measures alone, it is best to store the entire data set in one processor. This, however, is clearly undesirable, because such a central processor is bound to be a hot spot that slows the system down. Decentralization is therefore an important explicit criterion [WW97b]. Whenever a theoretical analysis is difficult and one must resort to experiments, speed (measured in throughput, latency, and the like) is the measure of choice that subsumes all others. In fact, the counting schemes described above have not been analyzed as to their theoretical performance; experiments [HLS95] indicate that counting networks might outperform combining trees, and that diffracting trees might be the fastest [WW97a].

The linearizable distributed counter that we propose in this paper needs only the minimum amount of time for its operations, asymptotically in the worst case. We prove this by presenting a theoretical analysis of its asymptotic behavior, and by proving a matching lower bound. The proposed counter avoids hot spots by combining requests along a tree, and by spreading the delivered counter values accordingly. It is in fact an optimized combining tree (OCtree). This tree turns out to be also efficient in practice: Our experiments show that it is faster than standard combining trees or counting networks, and at least as fast as diffracting trees [WW97a]. That is, the safety of a theoretical worst-case analysis and the linearizability can be achieved without any loss of efficiency. In addition, it not only offers counting as an operation, but it allows to read-and-write the value of a shared variable in an atomic step in a system without shared memory. Such a variable can be used for incrementing and decrementing by arbitrary amounts, or for mutual exclusion. The scheme we propose can also be used directly for implementing stacks, queues and pools of data.

We start our description with the synchronous case. The model of computation for this case is presented in the next section; this will also be the basis for the asynchronous model that follows later. In Section 3, we describe our counting scheme for the synchronous case, and we analyze its performance. Section 4 proves a lower bound on the synchronous time needed for counting; it implies that the counting scheme of Section 3 is optimum. In Section 5, we show how the scheme carries over to the asynchronous case, and how it leads to an optimum solution also in this setting.

2 The Model

Consider a synchronous, 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. The processors operate in synchrony. Within each clock cycle (of a global clock), each processor may receive one message, perform a constant number of local calculations, and send one message, in this order. Every message takes exactly \( m \) (with \( m > 1 \))
cycles to travel from the sending processor to the receiving processor. 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 \( \text{val} \) and supports the operation \( \text{inc} \): When a processor initiates the \( \text{inc} \) operation, then the current counter value \( \text{val} \) is returned to the initiating processor and the system’s counter value is incremented (by one).

### 3 A Synchronous Solution

In this section, we present a decentral counter called OCtree. It is inspired by our upper bound solution in [WW97b] and the Combining Tree in [YTL86, GVW89].

The OCtree structure is as follows. Each inner node (including the root) in the OCtree has \( m \) children. The height of the tree is \( h \); the root is on level zero, all leaves of the tree are on level \( h \). Hence the number of leaves is \( m^h \). For simplicity, let us assume that \( n = m^h \).

Since \( m \geq 1 \), it is possible to give each inner node a distinct number between 1 and \( n \). Furthermore, we number the \( n \) leaves with distinct numbers from 1 to \( n \). We call the inner node resp. leaf with number \( p \) the **inner node** \( p \) resp. **leaf** \( p \). Therefore, for each \( p \) with \( p = 1, \ldots, n \), there is exactly one leaf \( p \) and at most one inner node \( p \) in the OCtree. When setting up this tree numbering scheme, we take care that for every \( p \), when the leaf \( p \) is the \( i \)th (\( i = 1, \ldots, m \)) child of its parent, then inner node \( p \) is not the \( i \)th child of its parent. A processor \( p \) will act for its leaf and for its inner node; to achieve this, it will communicate with every neighbour of these two nodes in the tree. Every processor knows its neighbours.

Let us first explain informally a simple strategy to handle the \( \text{inc} \)-operations. The current counter value \( \text{val} \) 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 a value \( \text{val} \) to this request, and \( \text{val} \) is sent down the tree along the same path until it arrives at the leaf \( p \).

However, this simple approach is bound to fail since the root node is a hot-spot and the performance of the system is not better than a central solution. To overcome the 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.

In order to simplify the following reasoning, we will make strong use of the synchronicity. Every processor is to know the current cycle \( c \) of the system. Each node in the OCtree that is the \( i \)th (\( i = 1, \ldots, m \)) child of its parent node is allowed to send a message to its parent only in cycle \( c = (2k - 1)m + i \), for integer \( k \). Similarly, every inner node in the OCtree is allowed to send a message to its \( j \)th (\( j = 1, \ldots, m \)) child only in cycle \( c = 2km + j \), for any integer \( k \). From the sending restrictions we get immediately:

**One Message Proposition**: No node sends (or receives) more than one message per cycle.

Proof: Sending: \( (2k - 1)m + i \neq 2k'm + j \), for integers \( k, k' \) and \( 1 \leq i, j \leq m \). Receiving: Since messages take exactly \( m \) cycles to get to the receiving node, a node (that is the \( i \)th child of its parent) receives a message from the parent only if \( c = 2km + i + m = (2k + 1)m + i \), and receives
a message from the jth child only if \( c = (2k' - 1)m + j + m = 2k'm + j \), for \( j = 1, \ldots, m \). Again, 
\( (2k + 1)m + i \neq 2k'm + j \), for integers \( k, k' \) and \( 1 \leq i, j \leq m \). □

**Communication Lemma:** A processor sends (receives) at most one message per cycle.

**Proof:** The numbering of the inner nodes in the OCTree takes care that for every \( p (p = 1, \ldots, n) \), when the leaf \( p \) is the ith child of its parent, then inner node \( p \) is not the ith child of its parent. Using the fact that the leaf has no children and the argument from the One Message Proposition, the Lemma follows immediately. □

Therefore, a processor can handle every received message instantaneously. Let us describe the counting scheme by defining the behaviour of the nodes:

**Leaf Node:**

The leaf \( p \) initiates the increment. Since the ith child is allowed to send a message up the tree only in cycle \( c = (2k - 1)m + i \), the leaf will send a message containing the number of all increments that were generated by the client application in the last \( 2m \) cycles.

Some cycles later, when the message went all the way up the tree and down to leaf \( p \), \( p \) will receive an interval of counter values.

**Root:**

The root receives a message from its jth child asking for \( z \) counter values in the cycle \( 2km + j \). The root returns the current counter value \( val \) immediately to child \( j \) and increments it by \( z \).

**Inner Nodes:**

As already sketched, inner nodes are to combine incoming messages and decombine them on the way down the tree. Also, an inner node keeps track of all operations it sent up to the root that did not come down yet. These operations are stored in a local first-in-first-out queue.

Let us first describe the process of sending up in the tree: The inner node \( p \) has to keep track of \( sum \), the number of increment requests by the leaves in the subtree that have not yet been forwarded up the tree (initially 0). When inner node \( p \) receives a message from child \( j \) asking for \( z \) counter values, \( p \) enqueues a record \((j, z)\) to queue and adds \( z \) to \( sum \).

The children’s upward messages arrive at cycles \( 2km + 1, \ldots, (2k+1)m \). If there are any messages from children that have not yet been forwarded up the tree, the inner node sends a combined message in cycle \( c = (2k+1)m + i \) to its parent. This message contains the number of requested increments \( sum \). Also in cycle \( c \), \( sum \) is reset to 0.

The way down is symmetric. In cycle \( (2k + 1)m + i \), the inner node \( p \) gets a message with the starting counter value \( start \) from the parent. Then, if at cycle \( c = (2k + 2)m + j, j = 1, \ldots, m \), the first record of queue is \( j \), inner node \( p \) removes this record and sends a message with \( start \) to child \( j \). Then, inner node \( p \) adds the removed record’s quantity \( z \) to \( start \).

From the description of the OCTree’s nodes we get:

**Forw arding Proposition:** There is only a constant number of operations per processor per cycle, and the information is forwarded up (down) the tree in \( 2m \) cycles at every node.

**Synchronous Counting Theorem:** Every \( inc \) operation is executed in \( O(m \log m n) \) cycles.

**Proof:** The time to travel along the links alone is \( 2h \cdot m \) cycles. The Forw arding Proposition states that we have additional delays of at most \( 2m \) cycles at every node, thus in total at most \( 2h \cdot 2m \). Therefore, the total number of cycles is at most \( 2hm + 4hm = 6hm \). Since \( m^h = n \), we get \( h = \log_m n \). □
If \( n \) is very small, this time degenerates to \( O(m) \) which is trivially optimal, since some processor has to send at least one message to some other processor in any correct counting scheme.

To show that the distributed counter that we propose is linearizable, we will argue on how counter values are handed out. Let \( t_{\text{dis}} \) be the time when the distributed counter decides which processor gets the counter value \( \text{val} \).

**Linearizability Lemma:** A distributed counter is linearizable, if \( t_{\text{inc}}^a \leq t_{\text{dis}} \) if and only if \( a \leq b \).

Proof: Obviously, the \( \text{inc} \) that gets value \( a \) starts before and terminates after \( t_{\text{inc}}^b \). Hence, whenever \( a < b \), the \( \text{inc} \) that gets \( b \) cannot terminate before the \( \text{inc} \) that gets \( a \) started. \( \square \)

### 4 A Lower Bound

We will derive a lower bound on the time it takes to increment the counter value in the worst case. To do so, let us first discuss the problem of broadcasting information.

Let some processor \( p \) broadcast a piece of information in the shortest possible time. According to our model, the number of processors that know the information in cycle \( c \) is defined by

\[
f_m(c) = f_m(c-1) + f_m(c-m), \text{ if } c \geq m, \text{ and } f_m(c) = 1 \text{ if } c < m
\]

**Dissemination Proposition:** Let \( m \geq 4 \) and let \( f_m(c) = f_m(c-1) + f_m(c-m) \) if \( c \geq m \) and \( f_m(c) = 1 \) for \( c = 0, \ldots, m-1 \). Then \( f_m(c) \) is bounded from above by \( f_m(c) \leq m^{c/m} \).

Proof: For \( c = 0, \ldots, m-1 \), the claim is true since \( 1 \leq m^{c/m} \). If \( c \geq m \), we have by induction

\[
f_m(c) \leq m^{(c-1)/m} + m^{(c-m)/m} = m^{c/m} \frac{m + m^{1/m}}{m - m^{1/m}}
\]

Since \( \frac{m + m^{1/m}}{m - m^{1/m}} < 1 \) for \( m > 3.2932 \), the lemma follows. \( \square \)

**Broadcasting Theorem:** For every \( m \geq 1 \), broadcasting takes \( \Omega(m \log_m n) \) time.

Proof: The Dissemination Proposition says that in \( c \) cycles, we can inform no more than \( f_m(c) \leq m^{c/m} = n \) processors, when \( m \geq 4 \). Therefore, informing \( n \) processors takes at least \( f_m^{-1}(n) \) cycles, where \( f_m^{-1}(n) \geq c = m \log_m n \). For the special cases \( m = 2, 3 \), one can easily show that \( f_m(c) \leq 2^c \) and therefore \( f_m^{-1}(n) \geq \log_2 n \). Because for \( m = 2, 3 \), we get \( \log_2 n = \Theta(m \log_m n) \), and the theorem follows. \( \square \)

By symmetry, cumulating information from \( n \) different processors at one processor takes the same time as broadcasting to \( n \) processors.

**Cumulation Proposition:** For every \( m \geq 1 \), cumulating information from \( n \) processors takes \( \Omega(m \log_m n) \) time.

Finally, we use the Cumulation Proposition to prove a lower bound for every distributed counting protocol.

**Lower Bound Theorem:** An increment costs \( \Omega(m \log_m n) \) time, for every \( m \geq 1 \).

Proof: At cycle \( c-1 \), assume that the system is quiescent and the counter value is \( \text{val} \). Assume that \( s \) processors initiate an \( \text{inc} \)-operation at cycle \( c \) and no processor initiates an \( \text{inc} \)-operation at cycle \( c+1, \ldots, c+t \), for sufficiently large \( t \). The elementary soundness condition requires that the \( s \) processors get the counter values \( \text{val}, \ldots, \text{val} + s - 1 \). Assume processor \( p_w \) is one of these \( s \) processors and gets the counter value \( \text{val} + w \), \( w = 0, \ldots, s-1 \). For this to be possible, \( p_w \) has to cumulate information from \( w-1 \) of the \( s \) involved processors. As we know from the Cumulation
Proposition, this takes $\Omega(m \log_m w)$ cycles. Since for the majority of the $s$ processors, $w = \Omega(s)$, the result cannot be expected before cycle $c + \Omega(m \log_m s)$. Whenever $s = \Omega(n)$ (a substantial part of the processors), this bound is $c + \Omega(m \log_m n)$. □

Note that this lower bound does not only hold for linearizable counting protocols, but also for weaker correctness conditions such as the elementary soundness requirement that diffracting trees and counting networks satisfy.

The lower and upper bounds together imply:

**Optimum Counting Corollary:** The OCtree counts in optimum time.

## 5 The Asynchronous Case

Synchronicity helps in describing a fast counter. Let us now adapt our proposition for a synchronous counter to the asynchronous case. Assume for the moment that each message in an asynchronous system arrives at its destination processor a finite, but unbounded time after it has been sent. Since any distributed counting scheme must exchange messages to operate on the data structure, this implies that we cannot guarantee an inc-operation to be completed in bounded time. This is not an interesting situation. Therefore, let us assume from now on that there is an upper bound of $t_m$ on the transfer time of a message. In addition, assume that the local computation step at a processor takes no more than $t_c$ time.

The counting scheme we propose works as follows. Similar to the synchronous case, we use an asynchronous communication tree (ACtree) whose root holds the counter value. The leaves of the tree are the processors that request inc operations; the inner nodes of the tree manage requests. Each inner node in the ACtree has $m$ (with $m > 1$) children. The height of the tree is $h$: The root is on level zero, all leaves of the tree are on level $h$. For simplicity, let us assume that $n = m^h$.

Since more than one message may arrive at a processor at (approximately) the same time, a receiving processor queues incoming messages, and it accepts (consumes) them in FIFO order. Queues are a critical item, because long queues slow the system down arbitrarily. We resolve this issue by proposing a scheme that bounds the length of queues in the degree of the nodes of the tree.

In the ACtree, each node repeatedly accepts messages from all its neighbours, performs a local computation, and then sends messages to all its neighbours. For the sake of analysis, we will argue on a point in time after a node received but before it sent its messages; let us call this a **checkpoint** for the node. For each node, we number the checkpoints according to their order in time, with numbers that start with $l - h$ for a node on level $l$ and are incremented in steps of 2. The nodes operate as follows.

### Initialization:

Before the first increment, the system is initialized. The root sends a wakeup message to each child. Whenever a node receives a wakeup message, this defines its first checkpoint. The node sends an upward message to the parent and a wakeup message to each of its children.

After this, a node does not send before it received an upward message from each child and a downward message from its parent. This defines its next checkpoint. Then the node sends an upward message to the parent and a downward message to each child.

### Leaf Nodes:
A leaf node only sends inc operations up to its parent after a checkpoint (since a leaf has no children, a downward message from the parent alone defines a checkpoint). In the message to its parent, the leaf tells the number of all inc-operations that were initiated since the leaf’s most recent message (maybe zero).

**Root:**
The root consumes an upward message from each of its children $j$ with $j = 1, \ldots, m$; this defines a checkpoint. For a child $j$ that asks for $z(j)$ counter values, the root returns an interval of values $val, \ldots, val + z(j) - 1$ in a downward message, and it increments the counter value by $z(j)$.

**Inner Node:**
An inner node waits until it received an upward message from every child and a downward message from the parent; this defines a checkpoint. It keeps track of the requested quantity $z$ from every child, by appending it to a bookkeeping queue of pending requests. Then it forwards the total of the requested quantities $z$ up to its parent. Finally, it spreads the interval it received from its parent among its children, according to their pending requests. The efficiency of this scheme is based on the fact that queues remain short at all times.

**Queue Lemma:** For any processor at any point in time, there are at most $m + 2$ messages in the queue of incoming messages.

Proof: Every processor $p$ has at most two roles, as a leaf $p$ and as an inner node $p$. The leaf $p$ gets downward messages from its parent only. The inner node $p$ gets downward messages from its parent and upward messages from each of its $m$ children only. All these neighbours send a message to processor $p$ only after a checkpoint. A new checkpoint for any of these neighbours, however, is defined only after the reception of a message from processor $p$. □

**Time Lemma:** From the checkpoint of a node sending a message, it takes $O(mt_c + tm)$ time until the message is consumed by the receiving node.

Proof: An inner node prepares $(m + 1)$ messages for sending; this can be done in time $t_c$ per message. The transfer of a message into the queue of the destination node takes at most $tm$ time. By the Queue Lemma, the number of messages in this queue is at most $m + 2$, and each message in the queue can be consumed in no more than $t_c$ time. □

When the last of the messages is consumed by the receiving node, this defines a checkpoint. We will now argue on the timing constraints of this checkpoint. Let $c_p(k)$ be the global time of the checkpoint $k$ of node $p$. Let $l_p$ be the level of node $p$. Recall that $l_p - h$ is the first checkpoint defined for node $p$.

From the initialization, we know that node $p$ consumes a downward message from its parent node $q$ before $p$’s checkpoint $k$, if that message was sent after $q$’s checkpoint $k - 1$. Similarly, $p$ consumes an upward message from its child $j$ before $p$’s checkpoint $k$, if that message was sent after $j$’s checkpoint $k - 1$.

Without loss of generality, assume that $h$ is even. Note that for nodes on even (odd) levels, only even (odd) checkpoints are defined. Let us now argue on the first and the last point in time in which checkpoint $k$, for $k = 0, 1, 2, \ldots$, is defined for some node in the tree:

$$first(k) := \min_p c_p(k), \quad last(k) := \max_p c_p(k).$$

Although some node might define a higher checkpoint before some other defines a lower checkpoint, there are timing constraints.

**Speed Lemma:** $last(k) \leq first(k + 2h)$, for every $k \geq 0$. 

7
Proof: $\exists$ node $p$ with $last(k) = c_p(k)$ and $\exists$ node $q$ with $first(k + 2h) = c_q(k + 2h)$. Any two nodes in the tree are connected by a path in the ACtree with length $d \leq 2h$. Let us follow the messages on the links of the path, starting with checkpoint $k$ of $p$, until we get to $q$. Since all transfer times of messages are nonnegative, we get $c_p(k) \leq c_q(k + d) \leq c_q(k + 2h)$. □

**Checkpoint Lemma:** $last(k + 1) - last(k) = O(mt_c + t_m)$.

Proof: $\exists p$ with $last(k + 1) = c_p(k + 1)$. From the Time Lemma, we know that it takes $O(mt_c + t_m)$ time from $c_q(k)$ for a message to be consumed by $p$, where $q$ is a neighbour of $p$. After the last of these neighbour messages is consumed, $p$ defines a checkpoint. With $c_q(k) \leq last(k)$, we get $last(k + 1) = c_p(k + 1) \leq c_q(k) + O(mt_c + t_m) \leq last(k) + O(mt_c + t_m)$. □

**Asynchronous Counting Theorem:** Every inc operation is executed in $O(t_m \log_{t_m/n})$ cycles.

Proof: Assume that processor $p$ initiates an inc-operation at time $t$, and let $k$ be the checkpoint at which leaf $p$ sends the request to its parent. The request travels through $2h$ nodes, with an incremented checkpoint at each node. Therefore the response arrives at leaf $p$ at checkpoint $k + 2h$. The total time is bounded from above by $c_p(k + 2h) - c_p(k - 2)$, since the initiation might have just missed checkpoint $k - 2$. With $c_p(k + 2h) \leq last(k + 2h)$ and $c_p(k - 2) \geq first(k - 2)$, we get a bound on the total time of $c_p(k + 2h) - c_p(k - 2) \leq last(k + 2h) - first(k - 2)$. By the Speed Lemma, the total time is no more than $last(k + 2h) - last(k - 2h - 2)$, and with the Checkpoint Lemma, this bound becomes $(4h + 2) \cdot O(mt_c + t_m)$. By choosing $m = \Theta(t_m/t_c)$, the time is bounded by $h \cdot t_m$ with $h = \log_{t_m/n}$, and the theorem follows. □

By setting $t_c = 1$ and $t_m = m$ in the lower bound theorem, this proves that the ACtree is asymptotically optimal.

**Acknowledgments**

We’d like to thank Nicola Galli for helpful discussions and pointers to the literature.

**References**


