The counting pyramid

The Counting Pyramid

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, the Counting Pyramid. We analyze the expected behaviour of the Counting Pyramid using queueing theory.

1 The Problem

We observe an ever increasing importance of distributed data in our interconnected world. Even though data is distributed over several processors, any processor should be able to access data quickly. 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 co-operate: 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 message complexity, are not satisfactory for distributed data structures. For instance, even though a data structure could be message 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 implementation 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 certain practical situations, Counting Networks [AHS91, AHS94] are a nice 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, not every operation in an efficient counter should be forced to travel through a large number of nodes (but this is the case for counting networks). These two goals appear to be in conflict with each other, and any fast counting scheme must respect both of them. Diffracting Trees [SZ94, SZ96] have an ingenious design that also avoids contention at nodes (by means of diffractors) and thereby achieves very high speed. Both, Counting Networks and Diffracting 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 loosing other desirable properties, such as the ability to compute arbitrary functions (instead of counting only). A natural approach here tries to minimize the "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 a counter can be decentralized, it is not advisable to construct a real counting scheme directly along this theoretical suggestion, just because the bottleneck complexity abstracts too much from reality.

In this paper, we propose a new counting scheme – the Counting Pyramid. We start by introducing the model of operation in Section 2. In Section 3, we advocate the use of queueing theory to uniformly assess the efficiency of a distributed counting scheme. We give an example by analyzing the behaviour of a centralized counter.

Section 4 proposes the Counting Pyramid. Its design loosely follows the basic idea of (Optimized) Combining Trees [YTL86, GVW89, WW97a] to combine requests along upward paths in a tree, and to decombine the answers on the way down. The Counting Pyramid extends this concept in two ways. First, a node forwards a request up the tree to any one of the nodes on the next higher level, not necessarily to its parent. This target node is selected at random for each individual request, and the response travels down the tree along the same path as the request went up. This helps to spread out the load evenly across entire levels. Since this concept departs from a tree substantially, we name it differently for clearer distinction: The scheme is a pyramid, a level is a floor, and a node is a brick. Second, a processor decides freely on the floor in the pyramid at which it initially sends its request. We propose that this decision be based on the frequency with which the processor increments: If it increments very often, it should send the request to a leaf; if increments are very rare, it should go directly to the root of the tree. With this strategy, the Counting Pyramid is the first counting scheme that adapts to any access pattern without delay.

Due to its adaptivity and because counting is not the only operation that the Pyramid supports, the Counting Pyramid is quite a flexible structure with high potential for a variety of applications. For instance, to realize various important data structures (e.g. stacks), one needs more powerful operations (e.g. decrement, in addition to increment). [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 Pyramid is able to add or subtract any value, and many operations beyond that. The only restriction on the operations is 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 a Pyramid, the processors initiate the increment 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.

In Section 5, we present a performance analysis of the Counting Pyramid by means of queueing theory. We show how both conflicting goals, low message contention at individual bricks and short paths along which messages travel, can be assessed uniformly in this way. Our analysis measures the operation time – the time spent from the start of a test-and-increment-operation to its completion. It turns out that the expected operation time of the Counting Pyramid is asymptotically optimum, and at the same time the Pyramid adapts gracefully to changing access patterns, guarantees linearizability, and offers more general operations than just counting.

2 The Model

Consider a 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 *val* 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. Apart from spontaneous initiations of increment operations, actions of processors are driven by the events of arriving messages. 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, with ties broken arbitrarily. Whenever an incoming message is in the queue, the processor reads the message, performs a constant number of local computation steps, and potentially sends a message. Assume that the local computation takes t_c time on average. We aim at keeping the time short that elapses from the start of an *inc* operation to its completion; we call this the *operation time*.

3 The Central Scheme

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 $t_i \gg nt_c$, there will be no queue usually when a request arrives at the central processor c. Therefore processor 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 processor 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 response times. By using the average-case as a yardstick, we get a simple and consistent way to assess the performance of the Central Scheme. When choosing an exponential distribution for the time to handle a message, and processors have independent Poisson access patterns, we can use the simple standard M/M/1 queueing theory to analyze the performance. For an introduction into queueing theory, see [GH81, Nel95].

The key attributes for processor c are the *arrival rate* λ (the time between two message arrivals has expected value $1/\lambda = t_i/n$), and the *handling rate* μ (the time to handle a message is $1/\mu = t_c$ on average). The *utilization* of processor c, denoted by ρ, is the fraction of time where processor c is busy. It can be calculated as $\rho = \lambda/\mu$. Since both, λ and μ, are positive, we have $\rho > 0$. An important constraint is that processor c must not be overloaded – that is, $1/\mu < 1/\lambda$, which results in $\rho < 1$. Thus ρ can vary between 0 and 1, a natural restriction. Whenever $\rho \geq 1$, thus $t_i \leq nt_c$, the Central Scheme fails, since an unbounded queue builds up at processor c; we therefore must choose some decentral counting scheme to have good expected operation time. Whenever $\rho < 1$, the expected *response time* t_r (the time when a message is in the queue of processor c + the time for processor c to handle the message) is $\frac{1}{\mu(1-\rho)}$.

Theorem 1 (Central Scheme Performance) *Let $n\frac{t_c}{t_i} = \rho \leq 1 - \frac{1}{k}$ for some positive constant k. Then the expected operation time is $O(t_m + t_c)$.*

Proof: The expected response time of processor c is $t_r = \frac{1}{\mu(1-\rho)} \leq kt_c = O(t_c)$. As the request message has to be sent from the initiating processor to processor c and back, there is an additional term of $O(t_m)$ for the operation time. □

As every counting scheme has to transfer some messages and has to do some local computation, the expected operation time for the Central Scheme is asymptotically optimal, with low load.

On the other extreme, there is the case in which we have very high access rates. In [WW97a], we proved a lower bound for counting of $\Omega(t_m \log_{t_m/t_c} n)$. To show this bound to be tight, we proposed a counting scheme, the OCtree, and proved that the OCtree has an expected operation time of $O(t_m \log_{t_m/t_c} n)$. The OCtree, however, is not sufficiently flexible to qualify as a universal, practical distributed counter. Ideally, a distributed data structure should behave efficiently when access is low as well as when access is high, and it should adjust to changes of the situation instantly. In the next section, we present the Counting Pyramid, an adaptive scheme which can be seen as a randomized and generalized Combining Tree.

4 The Counting Pyramid

In this section, we will present a counting scheme that adapts gracefully to any average access rate whatsoever: the Counting Pyramid. Each processor p has a completely local view of the counting speed – the performance of the Counting Pyramid for processor p does not depend on the access rates of the other processors, but solely on processor p. Obviously, this individual treatment of the processors is bound to fail if their access patterns are correlated. Our approach will work nicely whenever processors initiate the *inc* operation with a Poisson access pattern and completely independently of each other.

A *Counting Pyramid* consists of h *floors*. Floor f ($f = 0, \ldots, h-1$) is made of m^f *bricks* where $m > 1$ is an integer. The single brick in floor 0 is the *top* of the Pyramid, with floor numbers increasing from top to bottom. The *height* h of the Counting Pyramid is defined as $h := \log_m n$; for simplicity, assume that h is an integer. The number of bricks in the Pyramid is smaller than n, because $\sum_{f=0}^{h-1} m^f = \frac{n-1}{m-1} < n$. Thus, it is possible to give each brick a distinct number between 1 and n. We identify each brick with its number – processor p will act for brick p. Regrettably, we cannot claim that the Pyramid concept is truly novel. It has been recognized as quite a universal paradigm in ancient times already, and even the distinguished role of the top has been acknowledged previously (Figure 1).

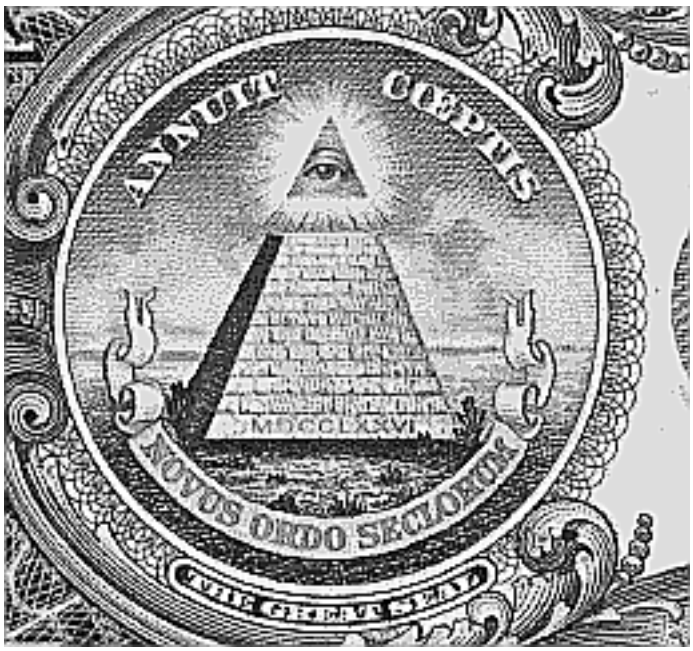

Figure 1: The Counting Pyramid on the US One Dollar Bill

The current counter value *val* is stored at the top. Whenever a processor p wants to increment, p sends a request message to a random brick b in floor f ($f = 0, \ldots, h-1$; a good choice for f is to be specified later). A brick in floor $f > 0$ that receives a request picks a brick in floor $f-1$ at random and forwards the request to that brick. The top (brick in floor 0), upon receipt of a request, assigns in its response a value *val* to the request, and *val* is sent down the Pyramid along the same path that it took on its way up. As soon as the response value arrives at b, it is returned to the initiating processor p.

This simple approach, however, is bound to be inefficient since the top is a hot-spot, and the performance of the system is not better than that of a Central Scheme. To overcome this problem, we let a brick combine several requests into one. That means, instead of just forwarding each individual request up the Pyramid, a brick tries to combine requests arriving at "roughly the same time" (within a certain time frame). We still have to guarantee that requests resp. counter values are forwarded up resp. down the tree quickly, i.e., without waiting too long. Let us distinguish two kinds of messages: upward and downward messages. An *upward* message is sent up the Pyramid and consists of two integers: z, the number of increment operations requested, and s, the sender identification of the message. A *downward* message is sent down the Pyramid and consists of an interval of counter values, specified e.g. by the first and the last counter value of the interval. Let us describe the counting scheme more precisely by defining the behaviour of the participating entities:

Top

When receiving an upward message from q, asking for z counter values, the top returns a downward message with the interval $\{val, \ldots, val+z-1\}$ to q and increments val by z. Initially, val is 0.

Initiating Processor

Let processor p initiate an increment. Processor p immediately sends an upward message (asking for one counter value) to a random brick b in floor f. The value of f will be specified in the next section – it is a function of t, the time that has passed since p has initiated the last inc operation (if there has been one) resp. the time since setting up the system (if p has not initiated an inc operation yet), and it will be denoted as $f(t)$. Later, processor p will get a downward message from b with an assigned counter value. Then, the inc operation is completed.

Brick (not the top)

As already sketched, bricks are to combine upcoming messages and decombine them on the way down the Pyramid. Also, a brick keeps track of all *open requests* sent up to the top whose response did not come down yet. An open request is a set of received upward messages that were combined by the brick and sent to a brick on the next higher floor (i.e., the floor with the next lower number). The brick administrates the open requests in local memory.

Let us first describe the process of sending a request up in the Pyramid: The brick b in floor f keeps track of the number of increment requests that have not yet been forwarded up the Pyramid to some brick in floor $f-1$, denoted as sum (initially 0). Whenever brick b receives an upward message from q asking for z counter values, b stores the tuple (q, z) in the current open request and it adds z to sum. From time to time, after waiting for time t_w on average, b sends an upward message to a random brick in floor $f-1$, asking for sum counter values, and b locally resets sum to 0. We will discuss how to choose t_w in the next section.

The way down is symmetric. Whenever b receives a downward message, it distributes the given interval according to the tuples of the corresponding open request. Afterwards, the open request is removed from local memory.

5 A Performance Analysis

In this section, we will argue on the performance of the Counting Pyramid by means of queueing theory. This analysis is more complicated than the one in Section 3 for the Central Scheme, since there is not a single processor with one queue, but a whole network of processors. This scenario, known as *queueing networks*, attracted the interest of many researchers. Queueing networks are usually presented directly upon the single processor queueing system in many textbooks [BG87, Nel95]. The basic idea of how to make queueing networks analytically tractable comes from Jackson [Jac57]. Simplified, it says that under certain restrictions, every single processor in a queueing network can be modeled as an isolated M/M/1-system.

It turns out that the following argumentation gets simpler when we introduce two restrictions for the time being. First, let the expected time for transferring a message be significantly larger than the time used for some local computation on average, that is, $t_m \geq 6t_c$. Moreover, we restrict the access rate, for every processor. Let us assume that the time that elapses between the initiations of two consecutive *inc* operations by the same processor is Poisson with expected value t_i. We restrict t_i such that $t_i \geq t_w$. Mind that an individual time t since the last initiation of processor p may be smaller than t_w, but not the average. Later, we will show how to get around these two restrictions. In order to arrive at a fast scheme, we set the parameters of the Counting Pyramid as follows:

$$m := \left\lceil \frac{t_m}{t_c} \right\rceil, t_w := 6t_m, f(t) := \min\left(\left\lceil \log_m \frac{nt_w}{t} \right\rceil, h \right) - 1.$$

Directly from these definitions and the restrictions on t_m and t_i follows:

Fact 2 (Relative Durations) $t_i \geq t_w = 6t_m \geq 36t_c$.

Lemma 3 (Up-Down) *At each brick, 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: A brick b might receive far more upward messages than downward messages. On the other hand, upward messages can always be handled in t_c expected time, whereas an interval from a downward message has to be distributed to possibly many tuples in the corresponding open request. To simplify the argumentation, let us model a downward message as a whole set of downward entities (often called a *bulk* arrival), every one meeting the demands of exactly one pending tuple in the open request. Each entity task can be handled in t_c expected time, too. Moreover, since every upward message generates a tuple and every downward entity removes a tuple, the Lemma follows. □

To simplify bookkeeping, we count twice as many incoming upward messages plus outgoing upward messages at a brick and forget about the incoming and outgoing downward messages, in the following analysis.

Lemma 4 (Brick Arrival Rate) *The arrival rate at a brick is*

$$\lambda \leq 2\left(\frac{m}{t_w} + \frac{m}{t_w} + \frac{1}{t_w}\right).$$

Proof: Let us count the rate of the upward messages arriving at a brick b in floor f.

(1) Brick b receives upward messages from bricks in floor $f+1$. Bricks (in floor $f+1$) send no two upward messages within time t_w on average. As there are m^{f+1} bricks in floor $f+1$, and m^f bricks in floor f, the arrival rate for upward messages at brick b from floor $f+1$ is no more than $\frac{m}{t_w}$.

(2) Brick b is receiving upward messages directly from initiating processors, choosing floor f as start floor. In the worst case, all n processors in the system choose f as the start floor for their increments.

(2a) If $f < h-1$, then

$$f = f(t) = \left\lceil \log_m \frac{nt_w}{t} \right\rceil - 1 \Leftrightarrow f+1 \geq \log_m \frac{nt_w}{t} \Leftrightarrow t \geq \frac{nt_w}{m^{f+1}}.$$

Having at most n independent processors starting at f (with a choice of m^f bricks), the arrival rate is bounded by

$$\frac{n}{t}\frac{1}{m^f} \leq \frac{nm^{f+1}}{nt_w}\frac{1}{m^f} = \frac{m}{t_w}.$$

(2b) If $f = h-1$, the arrival rate at a brick in floor $h-1$ (with a choice of m^{h-1} bricks) is bounded by

$$\frac{n}{t_i}\frac{1}{m^{h-1}} = \frac{m}{t_i} \leq \frac{m}{t_w}.$$

(3) The waiting poses no additional problems when we assume the waiting time to be Poisson, with expected waiting time t_w. To stay within the queueing theory model, we introduce a virtual waiting message; when this messages is consumed by the brick, waiting is over and a combined upward message is sent. Thus, virtual waiting messages arrive at brick b with arrival rate $\frac{1}{t_w}$.

From Lemma 3, we know that handling downward messages is as expensive as handling upward messages. When adding up the cost for handling the types (1), (2a) resp. (2b), and (3) of upward messages, and doubling the result (for downward messages), the Lemma follows. □

Corollary 5 (Brick Arrival Rate) *The arrival rate at a brick is* $\lambda \leq \frac{5}{6t_c}$.

Proof: We simplify Lemma 4 using Fact 2 and the definition of m ($m = \lceil \frac{t_m}{t_c} \rceil \leq \frac{t_m}{t_c} + 1$):

$$\lambda \leq 2\left(\frac{m}{t_w} + \frac{m}{t_w} + \frac{1}{t_w}\right) = 2\frac{2m+1}{t_w} \leq 2\frac{2(\frac{t_m}{t_c}+1)+1}{t_w} = \frac{4t_m}{t_c t_w} + \frac{6}{t_w} \leq \frac{4}{6t_c} + \frac{1}{6t_c} = \frac{5}{6t_c}.$$

□

Corollary 6 (Processor Arrival Rate) *The arrival rate at a processor is* $\lambda \leq \frac{8}{9t_c}$.

Proof: A processor p is not only acting as a brick in the Pyramid, but p is also initiating *inc* operations from time to time with a delay at least t_i, on average. Also with an expected delay at least t_i, p receives a counter value for a preceding operation. The arrival rate at processor p is therefore bounded by the arrival rate at brick p (with Corollary 5) and $\frac{2}{t_i}$ (initiating and receiving). By Fact 2, we get

$$\lambda \leq \frac{5}{6t_c} + \frac{2}{t_i} \leq \frac{8}{9t_c}.$$

□

Corollary 7 (No Overload) *No processor is overloaded since* $\rho \leq \frac{8}{9} < 1$.

Proof: Every message takes t_c time to be handled, on average. With Corollary 6, we get $\rho = \frac{\lambda}{\mu} \leq \frac{8}{9t_c} t_c = \frac{8}{9}$. □

Corollary 8 (Brick Response) *At a brick, any upward message is consumed within* $9t_c$ *expected time.*

Proof: From queueing theory, we know that the expected response time for a message at a brick is $t_r = \frac{1}{\mu(1-\rho)}$. With $\mu = 1/t_c$ and Corollary 7, the Corollary follows immediately. □

Theorem 9 (Pyramid Performance) *The expected operation time of the Counting Pyramid is*

$$O\left(t_m \log_{t_m/t_c} \hat{n}\right),$$

where $\hat{n} = n \min(1, t_c/t)$, *and* t *is the time that has passed since the initiation of the last inc operation.*

Proof: Corollary 8 shows that the response time at a brick takes only $O(t_c)$ time. Please note that the time spent in the queue at a brick is included. From Lemma 3, we know that this holds for downward messages too. Handling one message goes along with transferring one message (t_m) and waiting until the brick might send it upwards (t_w). All up the Pyramid and down again. Using Fact 2, the expected operation time when entering in floor f is $2f(9t_c + t_m) + ft_w \leq 11ft_m$. With the definition of $f(t)$ and $t_w = 6t_m \leq 6mt_c$ (Fact 2 and definition of m), the Theorem follows because

$$\begin{aligned} f(t) &= \min\left(\left\lceil \log_m \frac{nt_w}{t} \right\rceil, h\right) - 1 \\ &= O\left(\min\left(\log_m \frac{n \cdot 6mt_c}{t}, \log_m n\right)\right) \\ &= O\left(\log_{t_m/t_c} \min(nt_c/t, n)\right). \end{aligned}$$

□

Corollary 10 (Memory) *The expected amount of local memory needed at a processor is*

$$O(mh) = O\left(\frac{t_m}{t_c} \log_{t_m/t_c} n\right).$$

Proof: From Corollary 5, we know that messages arrive at the brick with arrival rate less than $\frac{1}{t_c}$. More or less every second message is a potential upward message, with the consequence that a tuple has to be stored in memory. With Theorem 9, the expected operation time is $O(t_m \log_m \hat{n})$, thus every tuple is removed after $O(t_m \log_m n)$ time ($\hat{n} \leq n$). □

In this section, we have introduced two restrictions to simplify the arguments. Let's get rid of them. One constraint was $t_i \geq t_w (= 6t_m)$. When processors are very active and initiate the increment operation very often ($t_i < 6t_m$), we don't allow them to send a request immediately, but force them to wait for at least time $6t_m$ instead, and to already combine several requests into one message. Thus $t_i = O(t_m)$; this does not introduce any extra waiting time asymptotically, and the constraint is gone.

The other constraint is $t_m \geq 6t_c$. 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 $m = 2$ and adjust h resp. t_w to $\log_2 n$ resp. $6t_c$. Again, one can show a in similar way an upper bound on the performance of $O(t_c \log_2 \hat{n})$.

Note that the Pyramid favours processors that do not initiate the *inc* operation frequently. Whenever t is quite large, for example $t = \Omega(nt_c^2/t_m)$, the logarithmic factor in Theorem 9 will be constant and thus, the operation time is $O(t_m)$. On the other hand, when t is very small, we have $\hat{n} = n$, and the operation time is $O(t_m \log_{t_m/t_c} n)$. By setting $t_c = 1$, $t_m = m$, and $t = t_c$, the expected operation time of the Counting Pyramid coincides with that of the OCtree and with the lower bound [WW97a].

Acknowledgments

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

References

[AHS91] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks and multiprocessor coordination. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 348–358, New Orleans, Louisiana, 6–8 May 1991.

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.

[BG87] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1987.

[DS97] Giovanni Della-Libera and Nir Shavit. Reactive diffracting trees. In *Proc. 9th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'97*, pages 24–32, 1997. Extended abstract.

[GH81] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory*. John Wiley and Sons., 1981.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, Massachusetts, 3–6 April 1989. ACM Press.

[HSW91] Maurice Herlihy, Nir Shavit, and Orli Waarts. Low contention linearizable counting. In IEEE, editor, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 526–537, San Juan, Porto Rico, October 1991. IEEE Computer Society Press.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Jac57] J. R. Jackson. Networks of waiting lines. *Oper. Res.*, 5:518–521, 1957.

[Nel95] Randolph Nelson. *Probability, Stochastic Processes, and Queueing Theory.* Springer-Verlag, 1995.

[ST95] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95*, pages 54–63, Santa Barbara, California, July 1995.

[SZ94] Nir Shavit and Asaph Zemach. Diffracting trees. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, pages 167–176, New York, NY, USA, June 1994. ACM Press.

[SZ96] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, November 1996.

[WW97a] Roger Wattenhofer and Peter Widmayer. Distributed counting at maximum speed. Technical Report 277, ETH Zurich, Departement Informatik, November, 1997.

[WW97b] Roger Wattenhofer and Peter Widmayer. An inherent bottleneck in distributed counting. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 159–167, 1997.

[YTL86] Pen-Chung Yew, Niau-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large scale multiprocessor. In *International Conference on Parallel Processing*, pages 51–58, Los Alamitos, Ca., USA, August 1986. IEEE Computer Society Press.