Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 64 (2004) 449–460
Journal of Parallel and Distributed Computing
http://www.elsevier.com/locate/jpdc

The counting pyramid: an adaptive distributed counting scheme ☆

Roger Wattenhofer* and Peter Widmayer

Department of Computer Science, ETH Zurich, Switzerland

Received 21 May 1999; revised 5 July 2002

Abstract

A distributed counter is a concurrent object which provides a `fetch-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 the *counting pyramid*, an efficient implementation of a distributed counter in a message passing system, which is based on software combining. The *counting pyramid* adapts gracefully to changing access patterns, guarantees linearizability, and offers more general `fetch-and-`Φ operations. We analyze the expected performance of the *counting pyramid*, using queueing theory and simulation. We show that the latency of the *counting pyramid* is asymptotically optimal.

Keywords: Queueing theory; Contention; Latency; Linearizability; Decentralization

1. The problem

A distributed counter is a variable that is common to all processors in a distributed message passing system, and supports an atomic `fetch-and-increment` operation: It delivers the counter value to the requesting processor and increments it. The counter is required to satisfy elementary soundness conditions: Whenever no operation is active in the system, the mechanism has delivered consecutive counter values, with none missing and none delivered twice, a property known as *quiescent consistency* [2]. In addition, applications may require that a counting scheme be *linearizable* [14] in the sense that whenever the first of two operations is completed before the second is initiated, the first gets a lower counter value than the second.

The quest for finding an efficient scheme for distributed counting is strongly related to the quest for finding an appropriate measure of efficiency for distributed data structures. The traditional measures of efficiency for algorithms in distributed message passing systems, such as message complexity, are not satisfactory. For instance, even though a structure could be message optimal by just storing the whole data structure with a single *central* processor and having all other processors access the data structure with one message exchange only, such an implementation is clearly unreasonable because it does not scale—with a growing number of accesses, the central processor will become a *bottleneck*. In other words, the work of an algorithm should not be concentrated at any single processor or within a small group of processors, even if this optimizes some measure of efficiency. We advocate the use of queueing theory to uniformly assess the efficiency of a distributed counting scheme: We analyze the expected *latency*—the time spent from the initiation of a `fetch-and-increment` operation to its completion.

Counting networks [1,2] are an original and seminal solution to the distributed counting problem. They make sure that message contention at each individual node of the network is low—an important precondition for efficiency. In addition, however, not every operation in an efficient counter should be forced to travel through a somewhat 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; Counting networks achieve the first goal, but not the second.

☆A preliminary version of this paper was presented at the 5th International Colloquium on Structural Information and Communication Complexity, 1998 [27]. A short time after the publication of our suggestion, a similar approach called "Combining Funnels" has been proposed by Nir Shavit and Asaph Zemach [23]. We would like to thank the Swiss National Science Foundation for partially supporting this research.

*Corresponding author. Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland. Fax: +41-1-632-1172.

E-mail addresses: wattenhofer@inf.ethz.ch (R. Wattenhofer), widmayer@inf.ethz.ch (P. Widmayer).

0743-7315/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0743-7315(03)00114-X

Diffracting trees [21,22] have an ingenious design that also avoids contention at nodes (by means of diffractors) and thereby achieves low latency. Both, counting networks and diffracting trees, are not linearizable; counting networks can be made linearizable [13], 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). A natural approach here tries to minimize the "bottleneck complexity"—the number of messages which a "busiest" processor of the system must exchange [26]. 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 concept, just because the stand-alone bottleneck complexity abstracts too much from reality.

In this paper, we propose a new counting scheme—the *counting pyramid*. Its design follows the basic idea of combining several requests along upward paths in a tree, and to decombine the answers on the way down. The idea of combining several requests to one message which is sent to a destination processor instead of sending every request alone, is well known in computer science. The first hardware proposals towards this end date back to the early eighties. Gottlieb, Lubachevski, and Rudolph introduced the combining metaphor in 1983 [10]. More articles on how to combine messages in networks followed shortly thereafter [5,8,16,19]. Later, research started separating the combining and the underlying hardware of the network which resulted in the *software combining tree* [29]. Another analysis and several applications are due to [7].

The counting pyramid extends the concept of combining 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 tree develops into a pyramid, a level becomes a floor, and a node becomes 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 itself increments: If processor p increments often, p should send the request to a low floor; if processor p increments very rarely, p may go directly to the top of the pyramid. We will show that this technique makes the counting pyramid adaptive to any access pattern.

There have been other constructions of adaptive distributed counters: The Reactive Synchronization Algorithm [17] keeps a global state that tells which counting structure is currently used. The reactive diffracting tree [6] grows and shrinks the size of the tree in order to keep up with the current load situation. In contrast to these two algorithms, the counting pyramid does modify neither form nor state when access patterns change, and hence there is no learning curve for a new access pattern. In other words, the adaptivity of the counting pyramid is not *reactive* but *instantaneous*.

Due to its adaptiveness and because counting is not the only operation that is supported, the counting pyramid is quite a flexible structure with high potential for a variety of applications. To realize various established data structures, one often needs more powerful operations. For instance, diffracting trees can be extended to *elimination trees* [20] offering both, a `fetch-and-increment` and a `fetch-and-decrement` operation, which is sufficient to implement stacks and pools. The counting pyramid is able to add or subtract any value. Even more generally, it naturally supports a `fetch-and-`Φ [9] operation, where Φ is an arbitrary associative binary operation defined on a set, resulting in `fetch-and-add`, `swap`, `fetch-and-or`, or `fetch-and-max` [4,28]. In a system where accesses are frequent (the load is high)—[29] call this an "unlimited" access system—the processors can use the `fetch-and-add` operation in the pyramid to increment in bulk; this is not possible with diffracting trees or counting networks.

A short time after the first conference publication of the counting pyramid [27], a similar approach called "combining funnels" has been proposed by Shavit and Zemach [23]. The combining funnel is designed for the "shared memory" computational model, while the counting pyramid is presented in the "message passing" model; both computational models are standard in distributed computing, and as the combining/counting funnel/pyramid shows, often equivalent. Moreover, [23] focus on empirical testing while [27] give a detailed analysis.

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 behavior of a centralized counter. Section 4 proposes and Section 6 analyzes the counting pyramid, while Section 5 proves elementary correctness conditions. We show that the performance of the counting pyramid is asymptotically optimal with a lower bound argument in Section 7. In Section 8, we simulate the counting pyramid against the most important competitors. Section 9 concludes this paper.

2. The model

Consider a distributed message passing system of n processors, 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 a 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 `fetch-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).

Each processor in the system has two roles: It is part of the counting pyramid infrastructure that realizes the distributed counter; in this role it receives messages from other processors that are to be handled according to the specifications. In the other role each processor runs an application program that triggers the `inc` operation at random.

The application program running at a processor is interrupted by a message arrival. We will see that (with one exception) such an interrupt can be handled with a constant number of local computation steps. Since more than one message may arrive at a processor p at (approximately) the same time, p queues all incoming messages and dequeues them sequentially in first-come-first-serve order, with ties broken arbitrarily.

For concreteness in our subsequent calculations, we make the following assumptions:

- It takes time t_m on average to transfer a message from one processor to some other processor.
- It takes time at most t_c to do the constant number of local computation steps in the case of a message arrival interrupt. In Section 4 we will see that (with one exception that we will take care of in Section 5) the interrupt handling routines consist of two to three simple pseudocode statements. Each pseudocode statement is either an assignment, a coin flip, an $O(1)$ operation in a local data structure, or a message send, and therefore takes constant time.
- The expected time that elapses between two consecutive increment operations initiated by processor p's application program is t_p, for a processor p.

Apart from these system parameters we need two support variables. For completeness we present them here and use them later:

- There is one interrupt which is not triggered by a message arrival but by a timeout. There is one timeout per time interval t_w.
- The *response time* t_r is a queueing theory term that determines the time a message spends in a queue of a processor plus the time for the processor to handle the message. We will make extensive use of t_r in the next two sections.

We aim at minimizing the time that elapses from the initiation of an `inc` operation to its completion; we call this the *latency*. Let us now provide the basis of our reasoning by first focussing on the most simple implementation of a distributed counter, the central scheme.

3. The central scheme

In the central scheme, a distinguished "central" processor c administrates the counter, by storing the counter value *val*, which initially is 0. When a processor p initiates an `inc` operation, processor p sends a request message to processor c. When processor c dequeues a request message, it returns a copy of *val* to the requestor and increments *val* by one.

Assume that accesses of the processors to the counter are very sparse (low load): The time between any two increment initiations for a processor p is long enough that even a central scheme can handle all requests one after the other, without being a bottleneck processor.

If $t_p \gg nt_c$ for all p, there will usually be no queue when a request arrives at the central processor c. Therefore, processor c can respond immediately, resulting in a latency of roughly $2t_m + t_c$. However, although it is unlikely, it is possible for many requests to arrive at processor c at more or less the same time. These requests will be dequeued and handled sequentially. Therefore, the last request in the queue will have a latency of up to $2t_m + nt_c$, a delay that grows linearly with the number of processors; the system does therefore not scale.

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. By choosing an exponential distribution for the time to handle a message, and by letting processors have independent Poisson access patterns, we can use standard M/M/1 queueing theory to analyze the performance. (If the handling time does not have an exponential distribution, one can apply M/G/1 queueing theory.) For an introduction into queueing theory, see [11,18]. For a closer look at queueing or probability theory in the context of analyzing the performance of distributed counting, see [24] or [21].

The key attributes for processor c are the *arrival rate* λ (the time between two message arrivals has expected value $1/\lambda = t_p/n$), 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. Whenever $\rho \geqslant 1$, we have $t_p \leqslant n t_c$ and 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 latency. Whenever $\rho < 1$, the expected *response time* t_r is $\frac{1}{\mu(1-\rho)}$.

Theorem 1 (Central scheme performance). *Let* $n\frac{t_c}{t_p} = \rho \leqslant 1 - \frac{1}{k}$ *for some positive constant* k. *Then the expected latency is* $O(t_m + t_c)$.

Proof. The expected response time of processor c is $t_r = \frac{1}{\mu(1-\rho)} \leqslant k t_c = O(t_c)$. Since 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 latency. □

Since every counting scheme has to transfer some messages and has to do some local computation, the expected latency of $O(t_m + t_c)$ for the central scheme is asymptotically optimal if the load is low.

In the other extreme, there is the case in which we have very high access rates. In [24,25], we proved a lower bound on the latency for counting of $\Omega(t_m \log_{t_m/t_c} n)$, which is similar to the proof of Section 6. To show this bound to be tight, we proposed a counting scheme, the "optimal combining tree" (OCtree), and proved that the tree has an expected latency 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 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. That is, the latency experienced by processor p will only depend on t_p, the expected time between two `inc` operations initiated by 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 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.

The current counter value *val* is stored at the top. Whenever a processor p wants to increment, it sends a request message to a random brick b in floor f ($0 \leqslant f < h$; 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 brick b, it is returned to the initiating processor p.

This simple approach 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 use the combining technique [10] and 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 and counter values are forwarded up and down the pyramid quickly, i.e., without waiting too long. We distinguish two kinds of messages: upward and downward messages. An *upward* message is sent up the pyramid and contains z, the number of increment operations requested, and an *id*, from which the requestor later can match the reply to its original request. A *downward* message is sent down the pyramid and contains an interval of counter values, specified by the first value of the interval and the same *id* as in the request. Let us describe the counting scheme more precisely by defining the behavior of the participating entities:

Top

The top works as in the Central Scheme; in pseudocode:

```
Top manages local variable:
    val: (of type) integer, initially 0.
Top, upon receiving upward(z, id) from processor q {
    send to processor q: downward(val, id);
    val:= val+ z;
}
```

4.1. Initiating processor

Let processor p initiate an `inc` operation. Processor p 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_p, the expected time between two inc operation initiations of processor p. Later, processor p will get a downward message from brick b with an assigned counter value. Then, the inc operation is completed. In pseudocode:

```
Application on processor p, initiating an inc operation {
  on floor f, choose a brick q uniformly at random;
  send to brick q: upward(z, id);
}
```

Brick (not the top)

As already sketched, bricks are to combine upcoming messages and decombine them on the way down the pyramid. In local memory, 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. Whenever a timeout occurs (every t_w), the brick sends the combined open request to a random brick on the next higher floor. We use an *id* to find the matching open request when a downward message is received. We give the details in pseudocode:

```
Brick manages local variables:
  req: array of linked lists, initially all lists are nil.
  free: a set of integers, initially 1,2,...,[size of req]-1
  id: integer, initially 0.
  sum: integer, initially 0.

Brick p, upon receiving upward(z, ref) from processor q {
  sum := sum + z;
  add the record(z, q, ref) to the linked list req[id];
}

Brick p, upon receiving a downward(val, ref) {
  while the linked list req[ref] is not empty do
    remove the record(z, q, i) from the linked list req[ref];
    send to processor (or application) q: downward(val, i);
    val := val + z;
  endwhile;
  free := free + {ref};
}

Brick p, when a timeout happens {
  if sum > 0 then
    send to random brick on next higher floor: upward(sum, id);
    id := pop an arbitrary integer from the free set;
    sum := 0;
  endif;
}
```

5. Correctness

In this section, we give safety and liveness proofs of the counting pyramid presented in Section 4. In particular, we will proof that (a) every initiated `inc` operation will terminate, (b) the counting pyramid is correct, and that (c) the counting pyramid is linearizable.

A *counting scheme* is *correct* if it fulfills these correctness criteria at any time t:

(1) *No duplication*: No value is returned twice.
(2) *No omission*: Every value returned before time t is smaller than the number of `inc` operations initiated before time t.

The system is in a *quiescent state* when every initiated *inc* operation is completed. If a system is in a quiescent state and k *inc* operations were initiated, exactly the values $0, 1, \ldots, k-1$ were returned by a correct counting scheme.

A counting scheme is called *linearizable* [14], when, in addition to accomplishing the correctness criteria, the values assigned reflect the order in which they were requested. More formally, a correct counting scheme is *linearizable* [14] if the following is granted: Whenever the first of two `inc` operations is completed before the second is initiated, the first gets a lower counter value than the second.

Lemma 2 (Termination). *An* `inc` *operation does terminate eventually*.

Proof. An `inc` operation is initiated by an application and sent to a random brick on floor f. After a timeout, the `inc` is then included and forwarded in an upward message to a brick in floor $f-1$, then for the same reason to $f-2$ and so on, until it reaches the single brick in floor 0 (the top). After reaching the top, it is forwarded in downward messages along the same bricks in floors $1, 2, \ldots, f-1$ until it reaches the brick f, and is from there returned to the application. □

At every brick, upward messages that arrive within a small time-frame (between two timeouts) are combined into a single upward message. For our correctness proof it is enough to put all the `inc` operations that jointly arrive at the top (that is, in the same upward message) into the same "group."

Lemma 3 (Correctness). *The counting pyramid is correct.*

Proof. A *group* of `inc` operations is defined as the set of `inc` operations that arrive in the same upward message at the top. Each `inc` operation is in exactly one group.

All counter values are assigned centrally at the top. The top gives intervals of values to the groups. The first group receives the interval $\{0, 1, \ldots, z-1\}$, where z is the number of `inc` operations in the group. The next groups receive a subsequent intervals, so that no value is omitted and none is given twice. That is, at any time the values $\{0, 1, \ldots, val-1\}$ are given to the group, where $val-1$ is the largest value given.

The bricks distribute the values of the interval of a group to the `inc` operations of the group. Both correctness conditions follow immediately. □

Lemma 4 (Linearizability). *The counting pyramid is linearizable.*

Proof. With Lemmas 2 and 3, we can identify each `inc` operation by the counter value it will eventually receive. Let the top decide at time $t^{\mathrm{d}}(v)$ that some `inc` operation will receive counter value v. This `inc` operation is initiated (by an application) at time $t^{\mathrm{i}}(v)$ and is completed (the value is returned to the application) at time $t^{\mathrm{c}}(v)$. Because the time for decision must be between initiation and completion, we know that $t^{\mathrm{i}}(v) \leqslant t^{\mathrm{d}}(v) \leqslant t^{\mathrm{c}}(v)$. Moreover, we have $t^{\mathrm{d}}(a) < t^{\mathrm{d}}(b) \Rightarrow a < b$ because the local variable *val* is never decremented at the top. (Note that if a and b both belong to the same group, $t^{\mathrm{d}}(a) = t^{\mathrm{d}}(b)$.) The linearizability condition "whenever the first of two operations is completed before the second is initiated $(t^{\mathrm{c}}(a) < t^{\mathrm{i}}(b))$, the first gets a lower counter value than the second $(a < b)$" is fulfilled since $t^{\mathrm{d}}(a) \leqslant t^{\mathrm{c}}(a) < t^{\mathrm{i}}(b) \leqslant t^{\mathrm{d}}(b) \Rightarrow a < b$. □

Lemmas 2–4 directly lead to Theorem 5.

Theorem 5 (Safety and Liveness). *The counting pyramid is correct and linearizable.*

6. A performance analysis

In this section, we will argue on the performance of the counting pyramid by means of queueing theory. We essentially need to show that all the queues at all the bricks (processors) are small on average. We do this by a cascade of queueing theory arguments.

This analysis is more complicated than the one in Section 3 for the central scheme, since there is not just a single processor with one queue, but a whole network of processors. The counting pyramid forms a *queueing network*, a well-studied topic in queueing theory. Queueing networks are usually presented directly upon the single processor queueing system in many textbooks [3,18]. The basic idea of how to make queueing networks analytically tractable comes from Jackson [15]:

Theorem 6 (Jackson). *Given a queueing network in which every handling rate is distributed exponentially, the system arrivals are Poisson distributed, and the*

messages are sent to other stations with fixed probabilities, then, in the equilibrium, every processor in the system can be modeled as an isolated M/M/1 processor, totally independent of the rest of the system.

Jackson's Theorem makes us having Markov distribution for all random variables. Even though we would not recommend exponentially distributed random variables (e.g. t_w) in an implementation, we will assume so for this analysis.

We simplify the analysis of the counting pyramid by introducing two restrictions for the time being. First, let the expected time for transferring a message be significantly larger than the time for handling a message arrival, that is, $t_m \geqslant 6t_c$. In Section 2 we have defined the expected time between two consecutive `inc` operations as t_p; we restrict t_p such that $t_p \geqslant t_w$, for every processor p. 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:

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

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

Fact 7 (Relative durations). $t_p \geqslant t_w = 6t_m \geqslant 36t_c$.

Lemma 8 (Up equals down). *At each brick, the local computation time for handling all upward messages is the same as the local computation time for handling all downward messages.*

Proof. Since every upward message generates a record and every downward entity removes a record in the open requests, the lemma follows. (Queueing theory purists may want an M/M/2 system at each brick; downward messages are immediately dequeued and handled by a separate thread.) □

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

Lemma 9 (Brick arrival rate). *The arrival rate at a brick is*

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

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 more than one 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 the start floor. In the worst case, all n processors in the system will choose f as the start floor for their increments.

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

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

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

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

(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

$$\frac{n}{t_p}\frac{1}{m^{h-1}} = \frac{m}{t_p} \leqslant \frac{m}{t_w}.$$

(3) For the analysis, we have assumed that the waiting time is Poisson, with expected waiting time t_w. To stay within the queueing theory model, we introduce a virtual waiting message; when this message 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 8, we know that handling downward messages costs as much local computation time as handling upward messages. When adding up the cost for handling types (1), (2a) or (2b), and (3) of upward messages, and doubling the result (for downward messages), the lemma follows. □

Corollary 10 (Brick arrival rate). *The arrival rate at a brick is* $\lambda \leqslant \frac{5}{6t_c}$.

Proof. We simplify Lemma 9 using Fact 7 and the definition of m ($m = \lceil \frac{t_m}{t_c} \rceil \leqslant \frac{t_m}{t_c} + 1$):

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

Corollary 11 (Processor arrival rate). *The arrival rate at a processor is* $\lambda \leqslant \frac{8}{9t_c}$.

Proof. A processor p is not only acting as a brick in the pyramid, but processor p is also initiating `inc` operations from time to time with a delay of t_p, on average.

Also with an expected delay of t_p, processor p receives a counter value for a preceding `inc` operation. The arrival rate at processor p is therefore bounded by the arrival rate at brick p (with Corollary 10) and $\frac{2}{t_p}$ (initiating and receiving). By Fact 7, we get

$$\lambda \leqslant \frac{5}{6t_c} + \frac{2}{t_p} \leqslant \frac{8}{9t_c}. \qquad \square$$

Corollary 12 (No overload). *No processor is overloaded since* $\rho \leqslant \frac{8}{9} < 1$.

Proof. Every message takes t_c time to be handled, on average. With Corollary 11, we get $\rho = \frac{\lambda}{\mu} \leqslant \frac{8}{9t_c} t_c = \frac{8}{9}$. $\square$

Corollary 13 (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 12, the Corollary follows immediately. $\square$

Theorem 14 (Pyramid performance). *The expected latency of the Counting Pyramid for processor p is*

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

Proof. Corollary 13 shows that the response time at a brick takes only $O(t_c)$ time. Handling an upward message goes along with transferring one message (t_m) and waiting until the brick sends it upwards (t_w). From Lemma 8, we know that downward messages have at most $O(t_w/t_c)$ records, on average; thus the last record is handled $O(t_w)$ after arrival of the downward message. Using Fact 7, the expected latency when entering in floor f is $2f(9t_c + t_m) + ft_w \leqslant 11ft_m$. With the definition of $f(t_p)$ and $t_w = 6t_m \leqslant 6mt_c$ (Fact 7 and definition of m), we have

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

Since $t_p \geqslant t_c$ the theorem follows. $\square$

Corollary 15 (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. With Corollary 10 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 record has to be stored in memory. With Theorem 14, the expected latency is $O(t_m \log_m n)$. Thus every record is removed after at most $O(t_m \log_m n)$ expected time. $\square$

We have used two restrictions to simplify the arguments. We now remove them. One constraint was that $t_p \geqslant t_w$ $(= 6t_m)$. When processors are very active and initiate the `inc` operation very often $(t_p < 6t_m)$, we do not 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 $t_p = \Omega(t_m)$; this does not introduce any extra waiting time asymptotically.

The other constraint was that $t_m \geqslant 6t_c$. Since sending/receiving a message includes always at least some local computation, this constraint will usually be satisfied naturally. However, if not, we set up the counting pyramid with $m = 4$ and choose $h = \log_4 n$ and $t_w = 12t_c$. This gives an upper bound of $O(t_c \log_2 n)$ for the latency.

Note that the pyramid favors processors that do not initiate the `inc` operation frequently. Whenever t_p is large, that is $t_p = \Omega(nt_c^2/t_m)$, the logarithmic factor in Theorem 14 will be constant and thus, the latency will be $O(t_m)$. On the other hand, when t_p is very small, the latency will be $O(t_m \log_{t_m/t_c} n)$. By setting $t_c = 1$ and $t_m = m$, the expected latency of the counting pyramid matches with the lower bound in the next section.

If processor p's application does not know its access frequency $1/t_p$, it can, for each initiation, use the time that has elapsed since the last initiation of the `inc` operation. This is a good practical estimate of the real access frequency.

7. A lower bound

For a lower bound, we need a computational model that is a synchronous version of the model in Section 2, where 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 computations, and send one message, in this order. Transferring a message takes exactly m (with $m > 1$) clock cycles. Everything else is as in Section 2.

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

Let a processor p broadcast a piece of information in the shortest possible time. If a processor q knows the information in cycle c, then (a) processor q already knew the information in cycle $c-1$ or (b) processor q received a message with the information in cycle c, that is, another processor sent a message to q in cycle $c-m$.

Therefore, the number of processors that know the information in cycle c is defined by

$f_m(c) = f_m(c-1) + f_m(c-m)$,
if $c \geqslant m$, and $f_m(c) = 1$ if $c < m$.

Lemma 16 (Dissemination). *Let $m \geqslant 4$ and let $f_m(c) = f_m(c-1) + f_m(c-m)$ if $c \geqslant m$ and $f_m(c) = 1$ for $c = 0, \ldots, m-1$. Then $f_m(c)$ is bounded from above by $f_m(c) \leqslant m^{c/m}$.*

Proof. For $c = 0, \ldots, m-1$, the claim is true since $1 \leqslant m^{c/m}$. If $c \geqslant m$, we have by induction

$$f_m(c) \leqslant m^{(c-1)/m} + m^{(c-m)/m} = m^{c/m} \frac{m + m^{1/m}}{m m^{1/m}}.$$

Since $\frac{m+m^{1/m}}{m m^{1/m}} < 1$ for $m > 3.2932$, the lemma follows. □

Theorem 17 (Broadcasting). *For every $m > 1$, broadcasting information to n processors takes $\Omega(m \log_m n)$ cycles.*

Proof. Lemma 16 says that in c cycles, we can inform no more than $f_m(c) \leqslant m^{c/m} = n$ processors, when $m \geqslant 4$. Therefore, informing n processors takes at least $f_m^{-1}(n)$ cycles, where $f_m^{-1}(n) \geqslant c = m\lceil \log_m n \rceil$. For the cases $m = 2, 3$, one can easily show that $f_m(c) \leqslant 2^c$ and therefore $f_m^{-1}(n) \geqslant \log_2 n$. For $m = 2, 3$, we have $\log_2 n = \Theta(m \log_m n)$, and the theorem follows. □

By symmetry, accumulating information from n different processors at one processor takes the same time as broadcasting to n processors.

Corollary 18 (Accumulation). *For every $m > 1$, accumulating information from n processors takes $\Omega(m \log_m n)$ cycles.*

Finally, we use Corollary 18 to prove a lower bound for every synchronous distributed counting scheme.

Theorem 19 (Lower bound). *In the worst case, the latency of an* `inc` *operation is $\Omega(m \log_m n)$ cycles, for every $m > 1$.*

Proof. At cycle $c-1$, assume that the system is quiescent and the counter value is *val*. Assume that s processors initiate an `inc` operation at cycle c and no processor initiates an `inc` operation at cycle $c+1, \ldots, c+t$, for sufficiently large t. The quiescent consistency (Section 1) requires that the s processors get the counter values $val, \ldots, val + s - 1$. Assume processor p_w is one of these s processors and gets the counter value $val + w$, $w = 0, \ldots, s-1$. For this to be possible, p_w has to accumulate information from $w-1$ of the s involved processors. Using Corollary 18 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 schemes, but for the weaker quiescent consistency (Section 1) that is used for counting networks and diffracting trees.

One can see that the lower bound of Theorem 19 and the upper bound of Theorem 14 match when we set $t_c = 1$, and $t_m = m$. Since every lower bound for a synchronous setting is also a lower bound for an asynchronous setting, the counting pyramid is optimal. However, we used different computational models for deriving the bounds, and hence, such a conclusion cannot be drawn directly. In [24] we solve the dilemma by giving a synchronous analogy for the counting pyramid.

8. Simulation

Although queueing theory is a suitable tool when arguing about systems where the parameters (e.g. the handling time) are Markov, the analysis gets intractable for non-standard distributions. Therefore, we complement our analysis with a simulation. Simulation has often been an excellent method to assess the performance of distributed counting schemes, starting with the work of [17]. For purity and generality reasons, we decided to simulate the counting schemes not for a specific machine, but for a distributed virtual machine.

First, we will present and discuss the simulation model and the distributed virtual machine. Then we give the results of the so-called "counting benchmark". For other benchmarks the reader is advised to refer to [24].

The *distributed virtual machine* (DVM) is essentially the n processor message passing system described in Section 2. For this benchmark we have chosen the following parameters: Transferring a message (t_m) takes exactly 5 time units, dequeueing and handling a message takes always exactly 1 time unit, with the exception of a downward message, where handling takes as many time units as we have records in the open request. This simplification allows us to experimentally evaluate the counting schemes on very large scale DVMs with up to 2^{14} processors.

On the DVM, we have implemented the central scheme (Section 3), the bitonic counting network [2], the diffracting tree [22], and the counting pyramid (Section 4).

The efficiency of all counting schemes but the central scheme varies according to a few structural parameters (e.g. network width, waiting time). For each scheme, we

have chosen the best parameter settings based on analysis and exhaustive simulation in [24]. We have implemented the system, the counting schemes, and methods for statistics in the programming language Java. The source code and instructions can be found at `http://www.inf.ethz.ch/personal/wroger/sim/`

In the benchmark, we measure the average latency. The other popular criterion to estimate the performance of a counting system is the *throughput* where one counts the number of `inc` operations that are managed by the counting scheme within a certain amount of time. For this benchmark, latency and throughput are inversely proportional to each other.

In the benchmark, each processor executes a loop that initiates an `inc` operation whenever it receives the value for its previous `inc` operation. We measure the average latency of an `inc` operation. This benchmark produces a high level of concurrency; Herlihy et al. [12] call this the "counting benchmark". Fig. 1 shows the average latency as a function of the number of processors of the four implemented counting schemes.

As long as the number of processors in the system is small (up to 16 processors), all schemes show equal performance, since every scheme degenerates to the central scheme (the bitonic counting network has width 1, the diffracting tree has height 0 and counting pyramid has height 1); the latency is about twice the message time because the initiating processor sends one message to the "central" processor which returns the current counter value immediately. When the number of processors is high, the central scheme is very slow: With n processors in the system, the average latency for the central scheme is about nt_c.

Since the bitonic counting network has depth $O(\log^2 n)$, it is left behind when the number of processors is very high. In the chart, one can see that one has to double the width of the bitonic counting network from time to time in order to keep congestion low; doubling the width is a major influence on performance and therefore, the latency grows significantly whenever one has to do it. For our setting, the width was doubled at 256 and 2048 processors, respectively.

The two most efficient schemes are the diffracting tree and the counting pyramid, with a small advantage for the counting pyramid. The latency of both schemes is logarithmic in the number of processors. The counting pyramid aims to combine m (where m is the ratio of message transfer time over local computation time) requests in a brick, which renders the logarithm to have base m. The diffracting tree always diffracts 2 "tokens", such that the base of the logarithm is 2. The advantage of the counting pyramid therefore grows with the ratio of message transfer time over local computation; i.e. in loosely coupled distributed systems such as the Internet.

Considering previous simulation results such as [12] or [22] (where the combining tree did not perform as well as the bitonic counting network and the diffracting tree), one might be surprised by our results. There are at least two justifications for the discrepancy. First, the counting pyramid differs significantly from the version that [12,22] implemented: They used a binary tree, we have a children/parent ratio of 5. And even more severely, their combining tree implementation did not use our waiting technique that promotes combining just the right amount of messages. The second reason might be that we ignored the different constants involved when handling a message; Herlihy [12] and Shavit and Zemach [22] use a more realistic machine model that takes these constants into account.

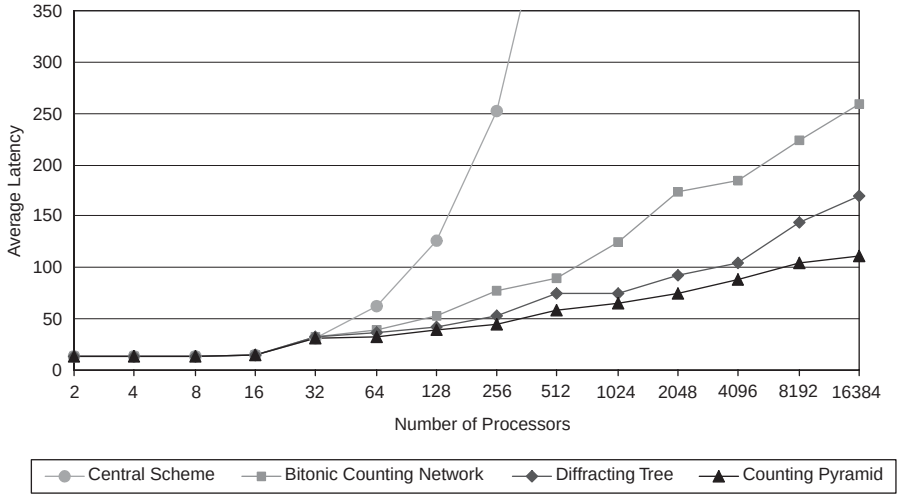

Fig. 1. Counting benchmark.

9. Conclusions

We have presented the counting pyramid for a distributed message passing system. As a basis, the counting pyramid uses the combining paradigm introduced by Gottlieb et al. [10]. By adding randomization and by tuning the parameters we get high performance.

We have analyzed the expected latency of the counting pyramid, and given evidence that the counting pyramid has asymptotically optimal latency with a lower bound argument.

The counting pyramid is adaptive in the sense that processors can change their access frequency and immediately get the desired performance. Moreover, processors have a local view of the system; if a processor does initiate the operation rarely, it will get the same performance as if the processor would access a "private" counter, even when all other processors initiate the operation frequently.

For a distributed virtual machine, we have simulated the counting pyramid against its major competitors: the central scheme, bitonic sorting networks, and diffracting trees. In a benchmark that measures the average latency, the counting pyramid outperforms the other counting schemes. We have seen that the relative advantage of the counting pyramid over its competitor counting schemes grows when a local computation step is much faster than transferring a message.

The counting pyramid is linearizable, and offers not only counting but more general `fetch-and-`Φ operations, both not possible with counting networks and diffracting trees.

We believe that the counting pyramid is a scalable scheme with a high practical potential, and that it should be implemented when distributed data structures experience a bottleneck.

References

[1] J. Aspnes, M. Herlihy, N. Shavit, Counting networks and multiprocessor coordination, in: Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing, New Orleans, Louisiana, 6–8 May 1991, pp. 348–358.

[2] J. Aspnes, M. Herlihy, N. Shavit, Counting networks, J. ACM 41 (5) (September 1994) 1020–1048.

[3] D. Bertsekas, R. Gallager, Data Networks, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[4] W.C. Brantley, K.P. McAuliffe, J. Weiss, RP3 processor-memory element, Proceedings of the 1985 International Conference on Parallel Processing, 1985, pp. 782–789.

[5] L.A. Cohn, A conceptual approach to general purpose parallel computer architecture, Ph.D. Thesis, Columbia University, New York, 1983.

[6] G. Della-Libera, N. Shavit, Reactive diffracting trees, J. Parallel Distrib. Comput. 60 (7) (July 2000) 853–890.

[7] J.R. Goodman, M.K. Vernon, P.J. Woest, Efficient synchronization primitives for large-scale cache-coherent multiprocessors, in: Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, 3–6 April 1989, pp. 64–75.

[8] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, The NYU ultracomputer: designing a MIMD, shared memory parallel computer, IEEE Trans. Comput. C-32 (2) (1983) 175–189.

[9] A. Gottlieb, C.P. Kruskal, Coordinating parallel processors: a partial unification, Comput. Architect. News 9 (6) (October 1981) 16–24.

[10] A. Gottlieb, B.D. Lubachevsky, L. Rudolph, Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors, ACM Trans. Programming Languages Systems 5 (2) (April 1983) 164–189.

[11] D. Gross, C.M. Harris, Fundamentals of Queueing Theory, Wiley, New York, 1981.

[12] M. Herlihy, B.-H. Lim, N. Shavit, Scalable concurrent counting, ACM Trans. Comput. Systems 13 (4) (November 1995) 343–364.

[13] M. Herlihy, N. Shavit, O. Waarts, Low contention linearizable counting, in: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, San Juan, Porto Rico, October 1991, pp. 526–537.

[14] M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Programming Languages Systems 12 (3) (July 1990) 463–492.

[15] J.R. Jackson, Networks of waiting lines, Oper. Res. 5 (1957) 518–521.

[16] G.Y. Lee, C.P. Kruskal, D.J. Kuck, The effectiveness of combining in shared memory parallel computer in the presence of 'hot spots', in: International Conference on Parallel Processing, Los Alamitos, CA, USA, August 1986, pp. 35–41.

[17] B.-H. Lim, A. Agarwal, Reactive synchronization algorithms for multiprocessors, in: Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), October 1994, pp. 25–35.

[18] R. Nelson, Probability, Stochastic Processes, and Queueing Theory, Springer, Berlin, 1995.

[19] G.F. Pfister, A. Norton, Hot spot contention and combining in multistage interconnection networks, IEEE Trans. Comput. C-34 (10) (1985) 943–948.

[20] N. Shavit, D. Touitou, Elimination trees and the construction of pools and stacks, in: Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95, Santa Barbara, California, July 1995, pp. 54–63.

[21] N. Shavit, E. Upfal, A. Zemach, A steady state analysis of diffracting trees, Theory Comput. Systems 31 (4) (July/August 1998) 403–423.

[22] N. Shavit, A. Zemach, Diffracting trees, ACM Trans. Comput. Systems 14 (4) (November 1996) 385–428.

[23] N. Shavit, A. Zemach, Combining funnels: a new twist on an old tale..., in: PODC: 17th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 1998.

[24] R.P. Wattenhofer, Distributed counting—how to bypass bottlenecks, Ph.D. Thesis, ETH Zurich, Institut fuer Theoretische Informatik, September 1998.

[25] R. Wattenhofer, P. Widmayer, Distributed counting at maximum speed, Technical Report 277, ETH Zurich, Departement Informatik, November, 1997.

[26] R. Wattenhofer, P. Widmayer, An inherent bottleneck in distributed counting, in: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, 1997, pp. 159–167.

[27] R. Wattenhofer, P. Widmayer, The counting pyramid: an adaptive distributed counting scheme, in: Proceedings of the Fifth International Colloquium on Structural Information and Communication Complexity, June 1998.

[28] J. Wilson, Operating system data structures for shared-memory machines with fetch-and-add, Ph.D. Thesis, New York University, Department of Computer Science, 1988.
[29] P.-C. Yew, M.-F. Tzeng, D.H. Lawrie, Distributing hot-spot addressing in large scale multiprocessor, in: International Conference on Parallel Processing, Los Alamitos, CA, USA, August 1986, pp. 51–58.

Roger Wattenhofer received a M.S. in computer science and operations research and a Ph.D. in computer science from the Swiss Federal Institute of Technology (ETH) Zurich. He is currently an assistant professor at ETH Zurich. His research interests include distributed data structures and algorithms.

Peter Widmayer received a M.S. in industrial engineering and a Ph.D. in computer science from the University of Karlsruhe, Germany. He is currently a professor at the Swiss Federal Institute of Technology (ETH) Zurich. His research interests are algorithms and data structures.