Brief Announcement: Fast Shared Counting using $O(n)$ Compare-and-Swap Registers

Pankaj Khanchandani
ETH Zurich
kpankaj@ethz.ch

Roger Wattenhofer
ETH Zurich
wattenhofer@ethz.ch

ABSTRACT

We consider the problem of building a wait-free and linearizable counter using shared registers. The counter supports a read operation, which returns the value of the counter, and an increment operation, which increments the value of the counter and returns nothing. The shared registers support read, write and compare-and-swap instructions. We show that given n processes and $O(n)$ shared registers, the increment operation is in $O(\log n)$ and read operation is in $O(1)$.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; **Concurrent algorithms**;

KEYWORDS

shared counter; wait-free; linearizable; compare-and-swap

1 INTRODUCTION

The compare-and-swap (CAS) instruction is widely available in commercial hardware and correspondingly used in software. It is therefore an incentive to design parallel algorithms with optimal time and space complexity using CAS instructions.

One of the basic problems in shared memory systems is to implement a linearizable and wait-free shared counter for n processes. Jayanti et al. [5] showed that if the shared registers support only read and write instructions, then the read operation on a counter takes $\Omega(n)$ steps. An interesting question is how fast we can get with the shared counter operations using registers that support CAS instructions as well and what is the smallest number of registers required for the fastest algorithm. In this brief announcement, we show an implementation using $O(n)$ shared registers supporting CAS instructions where the increment operation takes $O(\log n)$ time and the read operation takes $O(1)$ time.

2 RELATED WORK

We consider the shared counter where the read operation returns the value of the counter where as the increment operation increments the counter and does not return anything. This is in contrast to the fetch-and-increment counter that increments the counter and returns the value prior to the increment. Ellen et al. [2] give an $O(\log n)$ implementation of fetch-and-increment counter using $O(n)$ load-link/store-conditional registers. It is possible to simulate m load-link/store-conditional registers for n processes using $O(n^2 + m)$ CAS registers so that the load-link and store-conditional operations are wait-free and take $O(1)$ steps [4]. Thus, we can implement an $O(\log n)$ fetch-and-increment counter and an $O(\log n)$ shared counter using $O(n^2)$ CAS registers. Regarding a lower bound on the time required, one can use the construction from Alistarh et al. [1] to show that the total number of steps required for a sequence of $\Theta(n)$ read and increment operations is $\Omega(n \log n)$. Thus, at least one of read or increment operation takes $\Omega(\log n)$ steps.

3 MODEL

The shared memory consists of registers. Each register R supports the following operations: (i) `read(`R`)`, which returns the current value of register R, (ii) `write(`R`,` x`)`, which writes x to R and returns $\perp$, and (iii) `CAS(`R`,` v`,` v'`)`, which writes v' to R if and only if $R = v$ and returns whether v' was written to R. The register R allows concurrent operations from different processes.

We define the sequential counter object using the state set $\mathbb{N}_0$ and the following operations: (i) `get()`, which returns the current state $s \in \mathbb{N}_0$ without changing the next state, and (ii) `inc()`, which changes the next state to $s + 1$ and returns $\perp$. We give a wait-free and linearizable implementation [3] of the sequential counter for n processes. We assume that $n = 2^x$ for an integer $x \geq 0$ and that the processes have ids $1, 2, \ldots, n$.

4 ALGORITHM

We denote the counter object with id i by C_i^k, where k is the number of concurrent `inc()` operations that the counter supports (no restriction on the number of concurrent `get()` operations). Algorithm 1 gives a recursive construction for C_i^k using a register R_i and two other counter objects that support $k/2$ concurrent `inc()` operations each. These are $C_{2i}^{k/2}$, called the *left child* and $C_{2i+1}^{k/2}$, called the *right child*.

The counter that supports all the n processes is C_1^n. From Algorithm 1, the counter C_1^n uses the register R_1 and the counters $C_2^{n/2}$ and $C_3^{n/2}$. Recursing further, we end up with a heap of $2n - 1$ registers $R_1, R_2, \ldots, R_{2n-1}$. The processes that access the counter C_1^n have ids $1, 2, \ldots, n$. The sets L and H computed in Line 9 can be $\{1, 2, \ldots, n/2\}$ and $\{n/2+1, n/2+2, \ldots, n\}$, respectively. Thereafter, the range of process ids that access any counter C_i^k is contiguous. Thus, the sets L and H can be computed by splitting the range in the middle. Figure 1 shows the construction for $n = 4$ for processes.

PODC '17, July 25-27, 2017, Washington, DC, USA

ACM ISBN 978-1-4503-4992-5/17/07…$15.00
https://doi.org/http://dx.doi.org/10.1145/3087801.3087841

```
1  C_i^k.get()
2      return read(R_i);
3  C_i^k.inc()
4      if k == 1 then
5          T ← read(R_i);
6          write(R_i, T + 1);
7          return ⊥;
8      else
9          Partition the id set of k processes that access this
             function into two equal sized sets, L and H;
10         Let p be the current process id;
11         if p ∈ L then
12             C_{2i}^{k/2}.inc();          // increment left child
13         else
14             C_{2i+1}^{k/2}.inc();        // increment right child
15     V ← read(R_i);
16     sum ← C_{2i}^{k/2}.get() + C_{2i+1}^{k/2}.get();
17     success ← CAS(R_i, V, sum);
18     if success ≠ true then
19         V ← read(R_i);
20         sum ← C_{2i}^{k/2}.get() + C_{2i+1}^{k/2}.get();
21         CAS(R_i, V, sum);
22     return ⊥;
```

Algorithm 1: The algorithm for `inc()` and `read()` operations on the counter C_i^k.

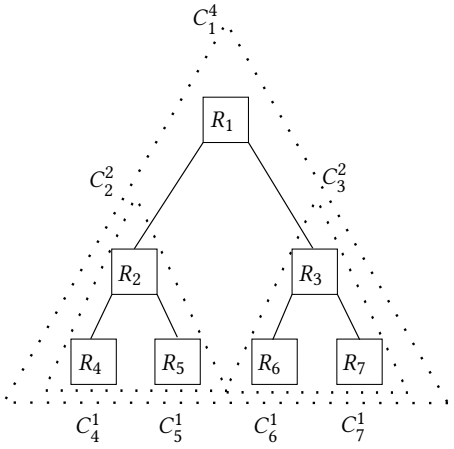

Figure 1: The counter construction for $n = 4$ processes (C_1^4). It includes two counters that support two concurrent `inc()` operations each (C_2^2, C_3^2) and four single process counters (C_4^1, C_5^1, C_6^1, C_7^1).

5 ANALYSIS

Lemma 5.1. *Algorithm 1 is a linearizable implementation of shared counter C_i^k.*

Proof. The claim is true for $k = 1$ because there is a single process that increments the register R_i by one upon each `inc()` operation.

For $k > 1$, assume that the claim is true for $k/2$, i.e., both $C = C_{2i}^{k/2}$ and $C' = C_{2i+1}^{k/2}$ are linearizable counters. Then, we can consider the operations `get()` and `inc()` on both C and C' as atomic [3]. The value returned by a successive C.`get()` operation can only be larger as there can only be C.`inc()` operations in between. This is also true for C'.`get()`. Thus, the sum of C.`get()` and C'.`get()` and the value written to R_i never decreases. Also, we can associate each distinct value I written to R_i with a unique pair of values i, i' so that $I = i + i'$ and i, i' are the values returned by C.`get()`, C'.`get()` respectively. We say that all the increment operations C.`inc()` and C'.`inc()` leading up to the value i and i' were *applied* to the register R_i.

Assume w.l.o.g that the process with id $p \in L$ and invokes C.`inc()`. To show linearizability, we have to show that C.`inc()` is applied when C_i^k.`inc()` returns (the linearization point being the point of application of C.`inc()`). Clearly, the operation is applied when either the CAS in Line 17 or Line 21 succeeds. So, we only need to check the case when both the CAS operations fail.

Assume that both the CAS operations fail. As the value of R_i can only change by successful CAS operations and the CAS in Line 17 fails, a successful CAS operation by another process q must have occurred between Lines 15 and 17. Process q may or may not apply C.`inc()` from process p depending on whether process q calls C.`get()` before or after p calls C.`inc()`. If it was after, then process q applied C.`inc()` to R_i. If it was before, then C.`inc()` is not applied until the next successful CAS. But, as we know that the CAS in Line 21 also fails, a successful CAS from a process q' must have occurred between Line 19 and Line 21. This CAS occurs on R_i that is at least the value updated by CAS operation performed previously by process q. Thus, the *sum* calculated by q' occurs after C.`inc()` from process p and this is applied to R_i by process q'. □

6 CONCLUSION

In this brief announcement, we gave an $O(\log n)$ time shared counter implementation using $O(n)$ space. This is both optimal in time and space. It is an interesting question to see if this technique can be applied to other problems such as fetch-and-increment counter using $O(n)$ space.

REFERENCES

[1] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. 2014. Tight Bounds for Asynchronous Renaming. *Journal of the ACM (JACM)* (2014).

[2] Faith Ellen and Philipp Woelfel. 2013. An Optimal Implementation of Fetch-and-Increment. In *27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel.*

[3] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1990).

[4] Prasad Jayanti and Srdjan Petrovic. 2005. Efficiently Implementing a Large Number of LL/SC Objects. In *9th International Conference on Principles of Distributed Systems (OPODIS), Pisa, Italy.*

[5] Prasad Jayanti, King Tan, and Sam Toueg. 2006. Time and Space Lower Bounds for Nonblocking Implementations. *SIAM Journal on Computing* (2006).