Brief Announcement: A Concurrency-Optimal List-Based Set

Vincent Gramoli[1,2], Petr Kuznetsov[3]*, Srivatsan Ravi[4], and Di Shang[2]

[1] NICTA
[2] University of Sydney
vincent.gramoli@sydney.edu.au,dsha5693@uni.sydney.edu.au
[3] Télécom ParisTech
petr.kuznetsov@telecom-paristech.fr
[4] TU Berlin
srivatsan.ravi@tu-berlin.de

Measuring concurrency. Multicore applications require highly concurrent data structures. Yet, the very notion of concurrency is vaguely defined, to say the least. What is meant by a "highly concurrent" data structure implementing a given high-level object type? Generally speaking, one could compare the concurrency of algorithms by running a game where an adversary decides on the schedules of shared memory accesses from different processes. At the end of the game, the more schedules the algorithm would accept without hampering high-level correctness, the more concurrent it would be. The algorithm that accepts all correct schedules would then be considered *concurrency-optimal*.

The lack of concurrency. To illustrate the difficulty of optimizing concurrency, let us consider a highly concurrency-friendly data structures: the sorted linked list. Since updates on a list-based set affect only a small number of contiguous list nodes, most of them could, in principle, run concurrently without conflicts.

The Lazy Linked List [1] achieves high concurrency by holding locks on only two consecutive nodes when updating but suffers from an overly conservative post-locking validation scheme. More precisely, both insert(v) and remove(v) traverse the structure until they find a node whose value is larger or equal to v, at which point they acquire locks on two consecutive nodes. Only then the existence of the value v is checked: if v is found (resp. not found), then the insertion (resp., removal) releases the locks and returns without modifying the structure. To illustrate that this concurrency limitation may lead to poor scalability, consider Figure 1 that depicts the performance of a 100-element Lazy Linked List under a workload of

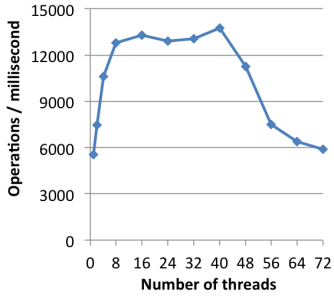

Fig. 1: The concurrency limitation of the Lazy Linked List based set leads to poor scalability as operations potentially contend on meta-data even when they do not modify the structure

* The author is supported by the Agence Nationale de la Recherche, ANR-14-CE35-0010-01, project DISCMAT

10% updates (insertions/removals) and 90% of contains on a 64-core machine. The list is comparatively small, hence all updates (even the failed insertions and removals) are likely to contend. We can see that when we increase the number of threads beyond 40, the performance drops significantly. This observation raises an interesting question: Does there exist a concurrency-optimal list-based set algorithm?

Our contribution. We answer this question in the affirmative. We propose the *Versioned List*, the first concurrency-optimal list-based set algorithm to date [2]. Its key feature is a *versioned try-lock*, a novel synchronization step inspired by transactional memory (TM). It allows us to implement a pre-locking validation: an update operation uses a CAS to set a versioned try-lock immediately after the validation of the node succeeds. In short, a lock is taken and schedules are rejected only if the data structure has to be modified under the effect of either a successful insertion or a successful removal. The versioned try-lock can be implemented using a StampedLock since Java 8 and a uint in C/C++. The Versioned List algorithm combines this new version try-lock with existing efficient mechanisms: the logical deletion technique of the Harris-Michael algorithm [3, 4] and the wait-free traversal of the Lazy Linked List [1]. If acquiring the try-lock fails because of a version change, then the operation re-reads some nodes.

We show that the Versioned List algorithm implements a linearizable set and rejects a concurrent schedule only if otherwise the linearizability of the set type is violated. Our algorithm is thus provably concurrency-optimal: no other correct list-based set algorithm can accept more schedules. This observation unveils an interesting desirable data structure property by which concurrent operations conflict on metadata only when they "conflict" on data, for which we need to exploit the semantics of the high-level data type. Note that this property extends the formal definitions of DAP [5–7] that are all trivially ensured by classic linked list implementations simply because all their operations "access" the *head* node and, thus, are allowed to conflict on the metadata.

References

1. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. (2006) 3–16
2. Gramoli, V., Kuznetsov, P., Ravi, S., Shang, D.: A concurrency-optimal list-based set. Technical Report 1502.01633, arXiv (2015)
3. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. (2001) 300–314
4. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. (2002) 73–82
5. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. In: SPAA. (2009) 69–78
6. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2010)
7. Ellen, F., Fatourou, P., Kosmas, E., Milani, A., Travers, C.: Universal constructions that ensure disjoint-access parallelism and wait-freedom. In: PODC. (2012) 115–124