Highly Concurrent Linked List and Skiplist

Di Shang
Dr. Vincent Gramoli
School of Information Technologies
FACULTY OF ENGINEERING & INFORMATION TECHNOLOGIES

Introduction

Need for Highly Concurrent Data Structure

- As single core speed become bottlenecked, high performance computing infrastructure relies on expanding the number of cores.
- To harness the power of multicore platform efficiently we need data structures and algorithms able to coordinate concurrent access to shared memory within multi-threaded application environment.

Sorted Linked List

- A linked list is a good example to illustrate high concurrency algorithm design, as each update operation only affects a small number of nodes, it potentially allows highly concurrent data access patterns.

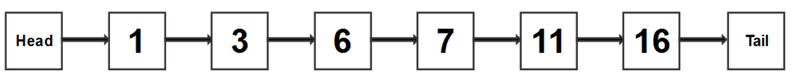


Figure 1. A sorted linked list

- This is a sorted linked list implementing a set of unique integers, there are 3 operations defined on the set: *contains(i)*, *insert(i)*, and *remove(i)*.

Skiplist

- A skiplist is basically a multi-layered sorted linked list, with the invariant that each layer contains a subset of the elements on the layer below.

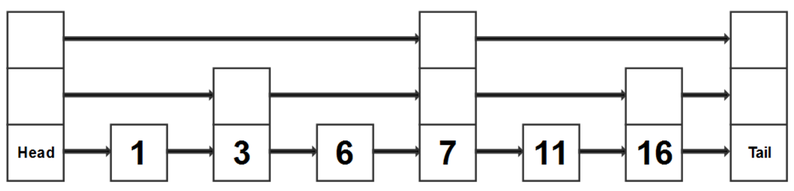


Figure 2. A skiplist

- The skiplist based set also supports the same 3 basic operations as the linked list, it has logarithmic lookup time but involves more complex update operations as insert and remove need to modify the linked list at each layer.

Contribution

- We define the concurrency metric and optimality in terms of the ability to accept correct schedules.
- We introduce the *versioned try-lock*, a novel synchronization technique that can be applied to optimistic lock based algorithms to achieve high concurrency and contention tolerance.
- We show that applying the versioned try-lock to lock based set implementations using linked list and skiplist could achieve high theoretical concurrency, which is supported by our performance benchmarks.

The Versioned Try-Lock

- We propose a new synchronization technique, the versioned try-lock, inspired from the field of transactional memory.
- The versioned try-lock basically encapsulates a locking state with a version number that can be modified together atomically, it supports the following atomic operations:

```
interface VersionedTryLock {
    int getVersion();
    boolean tryLockAtVersion(int version);
    void spinLock();
    void unlock();
    void unlockAndIncrementVersion();
}
```

- Hence we can transform a generic optimistic lock based algorithm like this:

```
1: read data on node
2: lock node
3: re-read & validate data integrity:
                    if fail unlock and restart
4: modify data
5: unlock node
```

```
1: read node version ver
2: read & validate data integrity:
                    if fail restart
3: tryLockAtVersion(ver):
                    if fail restart
4: modify data
5: unlockAndIncrementVersion
```

Concurrency Metric and Optimality

Correctness and Consistency

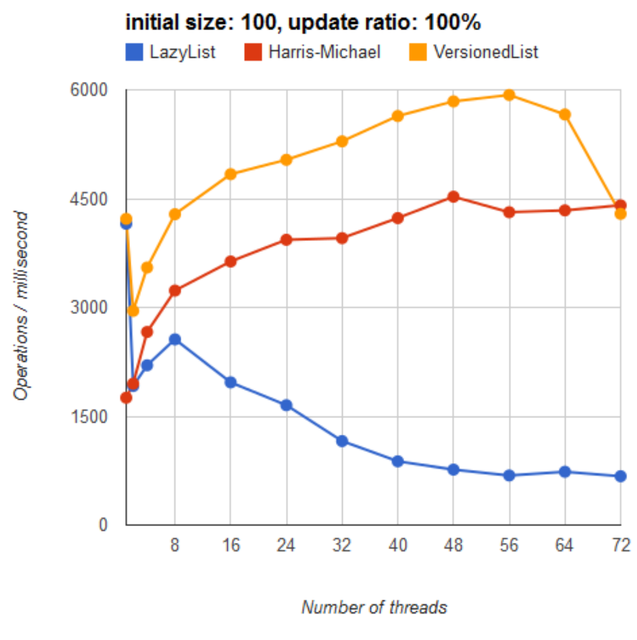
- The most commonly used consistency condition for concurrent data structures is *linearizability* [1], which we used as our main correctness criteria.
- Linearizability says that each operation on the data structure should appear to take effect at some instant point during its execution, so that concurrent operations will take effect in some sequential order.

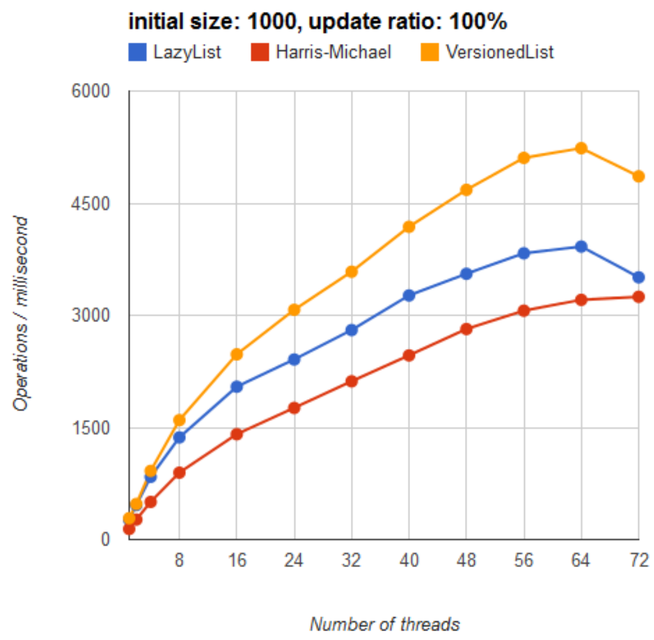
What does “highly concurrent” mean? How can we measure it?

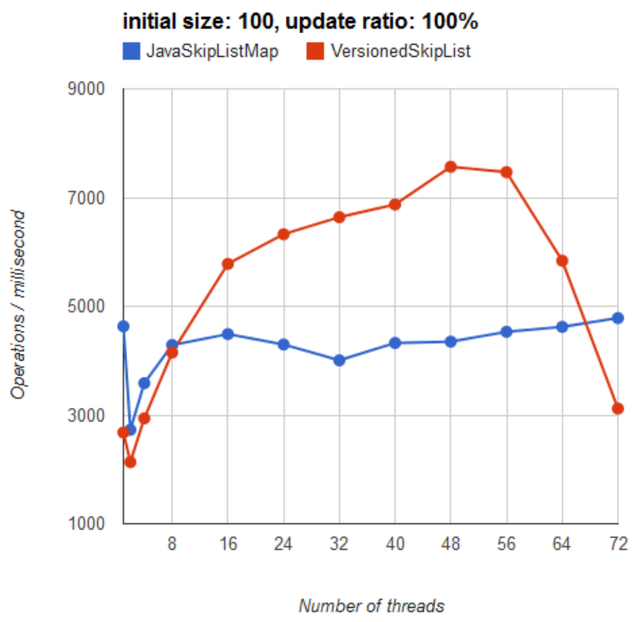
- A *schedule* is a sequence of instructions to access shared memory potentially consisting of instructions from multiple threads interleaved in an arbitrary order.
- The level of concurrency allowed by a data structure can be seen as its ability to accept correct schedules, the more schedules it accepts, the more concurrent it is.
- Correct algorithm accepts *only* correct schedules (schedules that do not violate the correctness criteria).
- (Concurrency) Optimal algorithm accepts *only* and *all* correct schedules.
- To prove that an algorithm is concurrency optimal, it is sufficient to show that the algorithm is correct, and that all rejected schedules are necessarily incorrect.

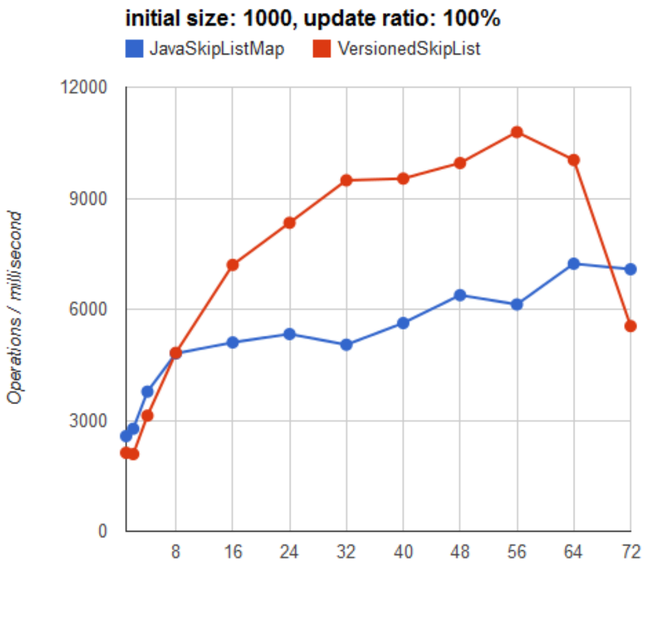
Experiments

- We tested our linked list and skiplist implementations in Java and benchmarked their throughput against known state-of-the-art algorithms [2][3][4]. Below is part of the results where we observed the most performance gain under heavily contended workload.









References

1. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
2. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.
3. S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2006.
4. Lea, D. ConcurrentSkipListMap. In java.util.concurrent.