Concurrent Extensible Hash Table

https://xinzhu-cai.github.io/418project.github.io/

Team Members: Danié Alvarado, Xinzhu Cai

Background & Goal

- Goal: Implement a concurrent extensible hash table library using C++
- Target Machine : shared memory multiprocessor
- Library Interface:

```
ConcurrentExtensibleHashTable (int load_factor)
bool find(hash_key_t key, hash_value_t *value)
bool put(hash_key_t key, hash_value_t value)
Bool remove(hash_key_t key)
```

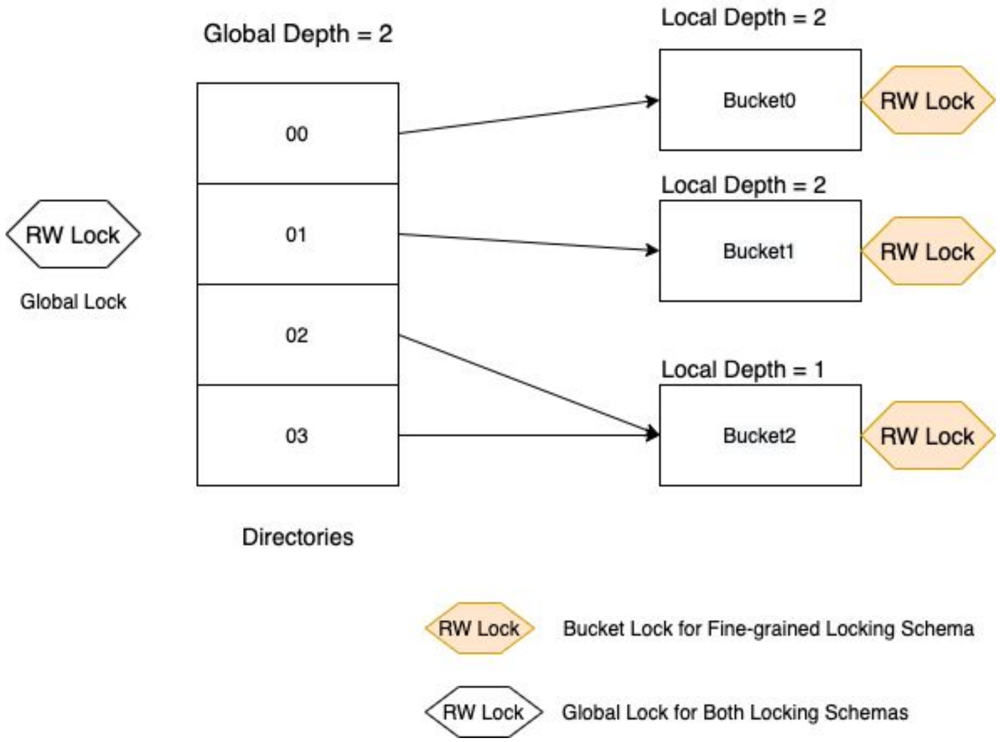
➔ Implementations potentially allow running all three operations in parallel
  - ➔ Coarse-grained Lock-based extensible hash table
  - ➔ Fine-grained Lock-based extensible hash table
  - ➔ Lock-free extensible hash table

Lock-based Extensible Hash Table

- Coarse-grained Locking Schema
  A single RW lock to protect the whole data structure
- Fine-grained Locking Schema
  A single RW lock to protect directories
  A RW lock per bucket

Pros: Simple to implement

Cons: Resizing and bucket splitting become a bottleneck as they require taking the global write lock

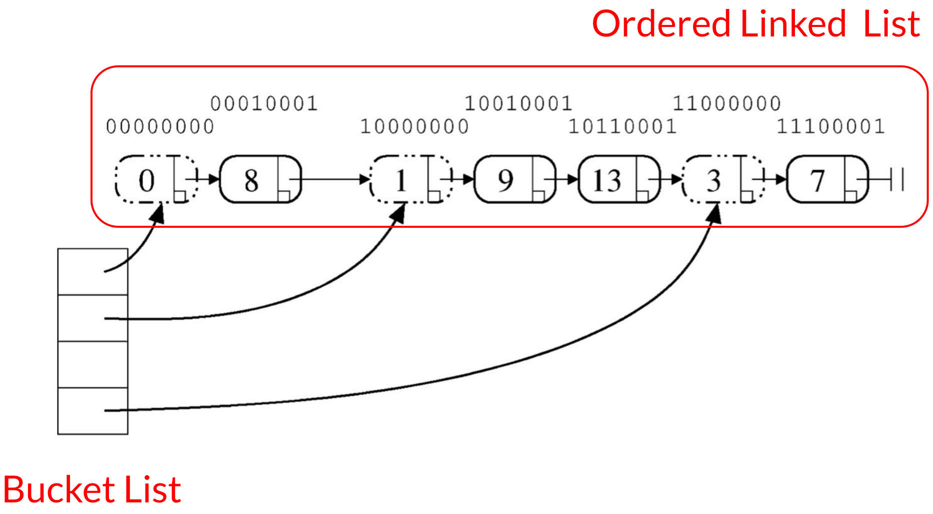


Lock-free Extensible Hash Table [1]

- A Recursive split-ordered linked list
  - Sorted on binary reversal of keys
  - eg : the split-order of key 13 is reverse(00001101) = 10110000,
- A bucket list
  - a dummy node at the start of each sub-list (bucket)

Pros: Perform the resizing and bucket splitting by only directing additional pointers into the list, and does not move any item

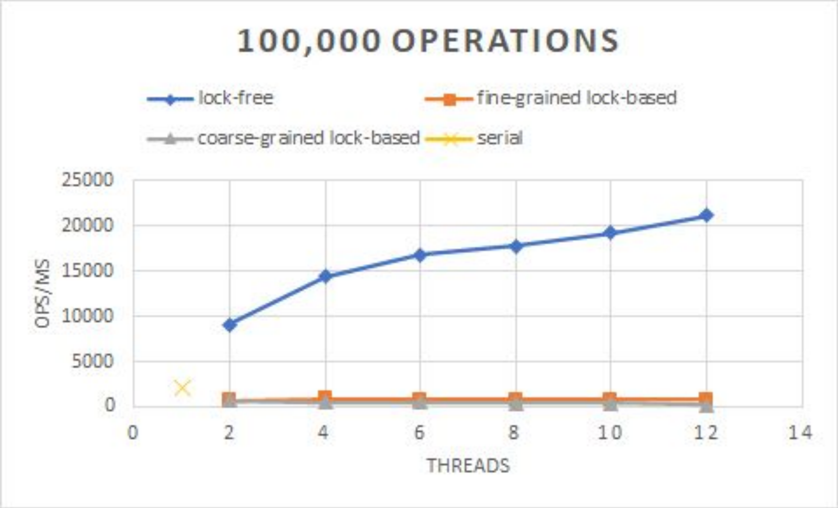
Cons: Implementation complexity and specific assumption

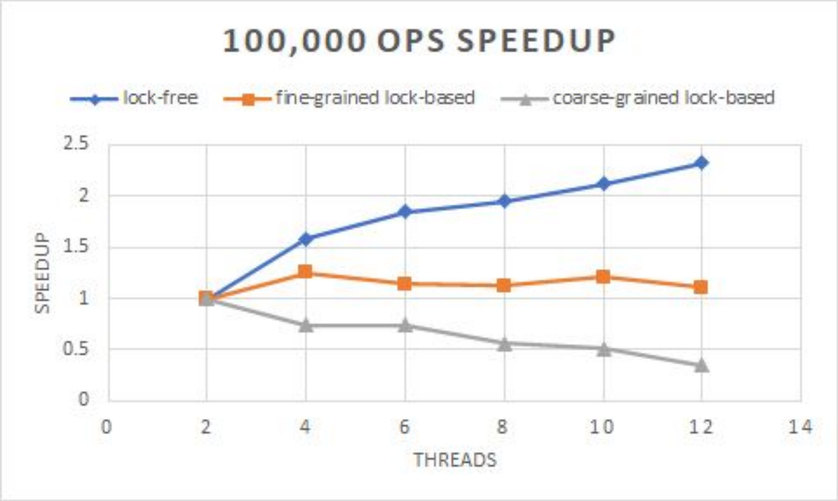


[1] Shalev, Ori, and Nir Shavit. "Split-ordered lists: Lock-free extensible hash tables." *Journal of the ACM (JACM)* 53.3 (2006): 379-405.

Performance Comparison

Machine: Mac with 12 cores.
Run on a typical workload containing 100k operations: 10% put, 88% find, and 2% remove.
Tested with the OpenMP framework

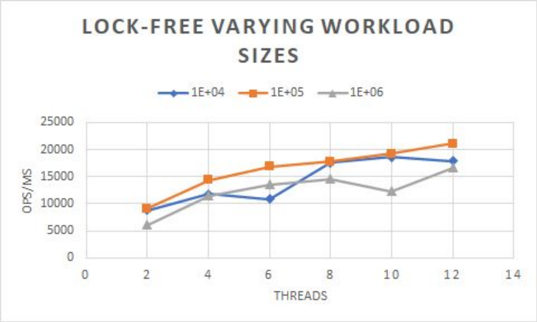


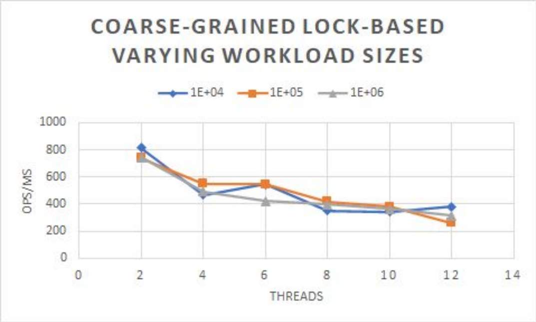


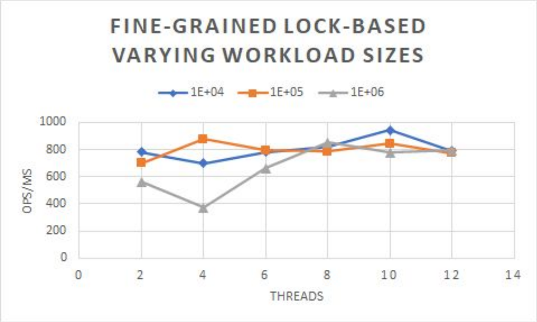
The lock-free implementation is more efficient and scalable

Varying Workload Sizes

Stable performance under different workload sizes

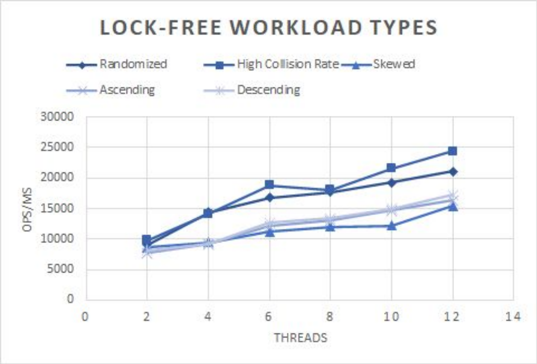


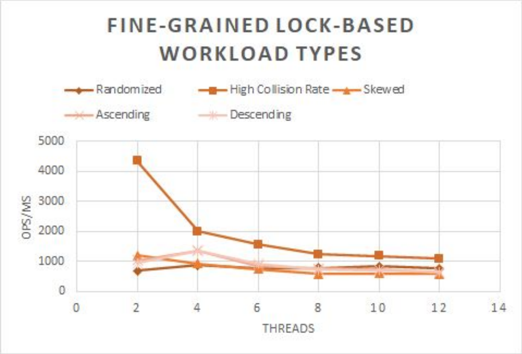


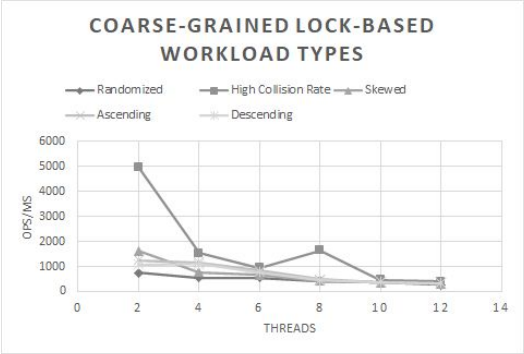


Varying Workload Types

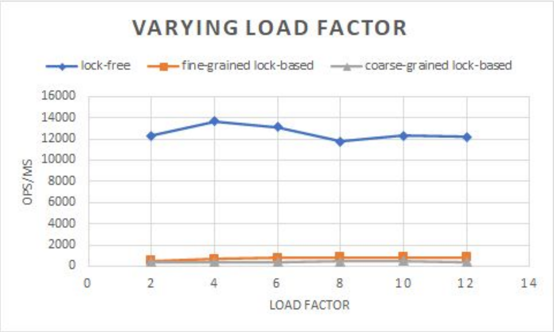
Skewed & Sorted workloads hurt performance

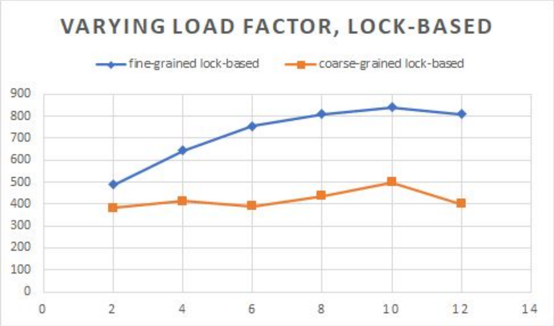






Effect of Load Factor



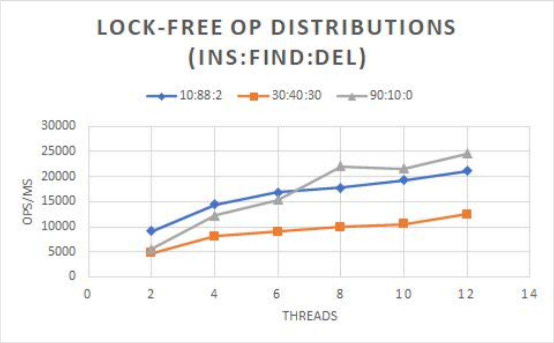


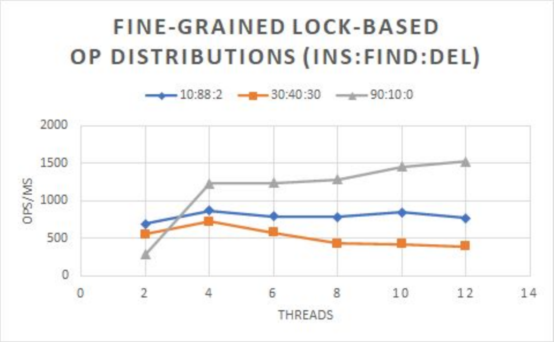
Throughputs of the lock-free and coarse-grained implementations are not affected by the load factor

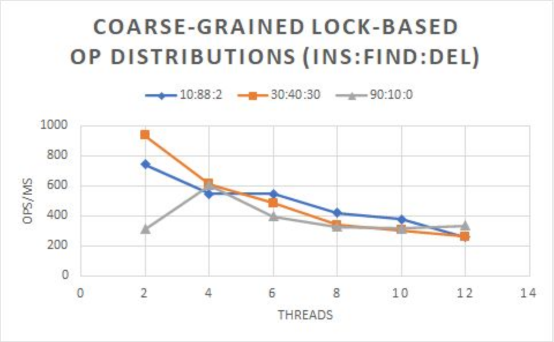
Throughput of the fine-grained lock-based one increases as load factor increases because one lock per bucket

Varying Operation Distributions

Percentage of update operations affects throughputs of fine-grained and lock-free







Conclusion

Summary

- The lock-free extensible hash table is more efficient, scalable compared with the lock-based ones
- Our lock-free implementation achieved stable performance on various workload scales
- Better performance on typical workloads than skewed and sorted ones
- Better performance when the percentage of find operations is higher

Future Work

- Based on profiling results, efficiency of the lock implementation and the one lock per bucket schema are the bottleneck of our fine-grained lock-based implementation
- Scheduling is the bottleneck of lock-free implementation
- Further improve efficiency of calculating split-order keys in the lock-free implementation