Milestone Report:

Concurrent Hash Table Lookup And Acceleration

Ying Jiang(yingj2), Yinyue Wu(yinyuew)
https://yinyuewu.github.io/pp_proj/

Adjusted Schedule

Time	Plan
11/7 ~ 11/13	(Done) Research and Proposal
11/14 ~ 11/20	(Done) Implement versions of coarse-grained/fine-grained locking with two storage alternatives and do validation
11/21 ~ 11/27	(Adjusted) Implement lock-free version ~~with storage alternatives **(not feasible)**~~ and do validation (Done) Build SIMD implementation for different alternatives
11/28 ~ 11/30	(Done) Preliminary comparison of SIMD implementations, storage alternatives with basic CPU implementation (Done) Milestone Report
11/30 ~ 12/2	All: Optimize current versions of code;
12/3 ~ 12/6	All: Add storage format support: storing pointers to original KV pairs for tagged fine-grained locks and lock-free versions, w/ and w/o SIMD
12/7 ~ 12/11	All: GPU implementation
12/12 ~ 12/15	All: Finish profiling, scalability analysis, and different experiments.
12/15 ~ 12/18	All: 125% goals of exploring lock-free versions with hashtag + ptr stored together, resizing / migration, etc. Create poster

Finished Goals

We've finished implementing coarse-grained/fine-grained locking with two storage alternatives (storing key-value (KV) pairs together / key tags together for fast lookup) and finished correctness validation.

We also finished the lock-free version with no tomb reusing (the reason stated in the proposal: conflicts when multiple threads operate on the same key) and finished correctness validation. However, storage alternatives introduce inconsistency between key hash tags and KV pairs: it requires multiple dependent CAS operations, which makes the algorithm blocking unless we have double-CAS operations. So we plan to give up this experiment (illustrated below).

We've finished SIMD versions based on the fine-grained version's two storage alternatives and lock-free version and finished correctness validation.

We also produced preliminary outcomes as graphs below on GHC machines. They're not very promising in read-write settings, and we'll continue to optimize them (illustrated below).

Unexpected Outcomes

1. It seems that we cannot maintain KV pairs and separate key hashes to be consistent in the lock-free version unless we have double CAS. If we use two separate CAS, between the two operations, the program would be blocked.

   To make up for this, we're planning to explore storing tags and pointers as pairs for each KV pair for this lock-free case, not sure if this would work out.

   > For example, suppose both the insert and the delete operations are done in this way: they first CAS on the KV pair, then on the hash tags. Suppose different threads are operating on the same key X with an invalid tag, and invalid KV pair, (i, i).
   >
   > Thread A inserts kv, then he stalls. Thread B sees an invalid tag but a valid key pair here. However, this should be the dedicated slot for X (according to the no-tomb assumption in the proposal). Should B keep retrying? Then blocking operation if thread A failed at this point, with no one making progress.
   >
   > If B force updates this value to be consistent (v,v) or (i, i), then maybe after a long period of time, the slot becomes (invalid tag, invalid KV) again. Then thread A wakes up, and he continues and makes the tag valid. So the state ends up being (valid tag, invalid data), similar to the ABA problem. A ended up returning success to the user, and we're in an inconsistent state again. How can we look A up correctly in the future?
   >
   > Even if we use counters here to solve ABA to eliminate inconsistencies, this is equivalent to keep everyone retrying. The algorithm is still blocking because the action is split into 2 steps, each being a CAS with the counter.
   >
   > No matter what the order of operations is, we perform insert / delete to the hash table, (first modify tag then KV pair / first KV pair then tag), we would run into an inconsistent state when an operation is half-done, e.g., invalid tag + valid KV pair,

or the opposite. If we spin-loop with this state, then the algorithm is blocking. If we force the updates to happen, then we lose correctness.

2. Our attempts at lock-free / SIMD / storage optimization optimizations have not demonstrated a big advantage yet. More details are below in Preliminary Results. They only work well on read-only cases. Although this might be due to the low thread count / we have not further optimized the code yet.

Adjusted Goals

75% -- Finish exploring different locking mechanisms

For goal 2 of storage alternatives: Implement alternative techniques of storing k-v pairs only / continuous key storage for fast lookups for ~~**both fine-grained and CAS versions**~~ **only the blocking versions.**

a. Store keys only once. Just store keys and values pairs as entries
b. Continuous storage of key hashtags for fast lookup. Tradeoff between extra space vs. fast key lookups by storing all keys/hashes consecutively for better locality and SIMD efficiency.

100% -- Acceleration

Switch 125%'s **GPU implementation** here, replacing "implementing pointers as values for storage"

Plans For Poster Session

At the poster session, we plan to show different graphs to illustrate the evaluation of all implementations.

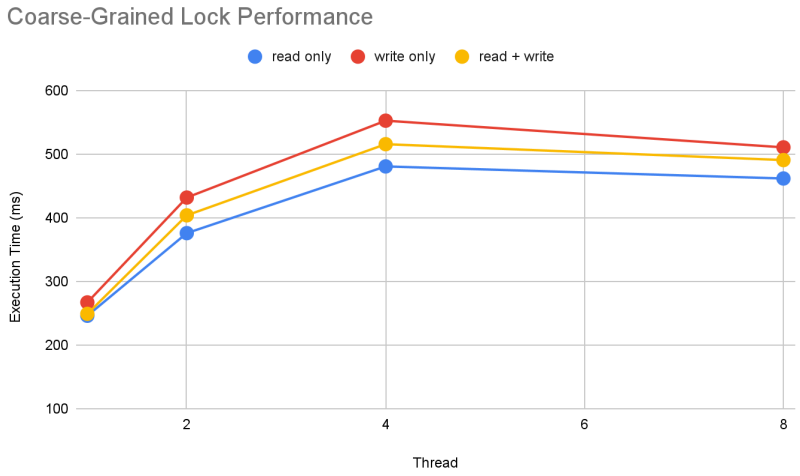
We will show the comparison of the scalability and performance of each implementation on different workload benchmarks. We are also going to show the experiment outcomes about how the performance varies with the changes in the number of threads, different thresholds, and different sizes of allocated space/keys/hash tags. We'll find out the best version to achieve cache awareness.

Preliminary Results

Here’re some results on GHC machines with thread count up to 8. Our optimizations work well on read-only cases (SIMD / storage optimization). A higher thread count would help, but the benefit is very small.

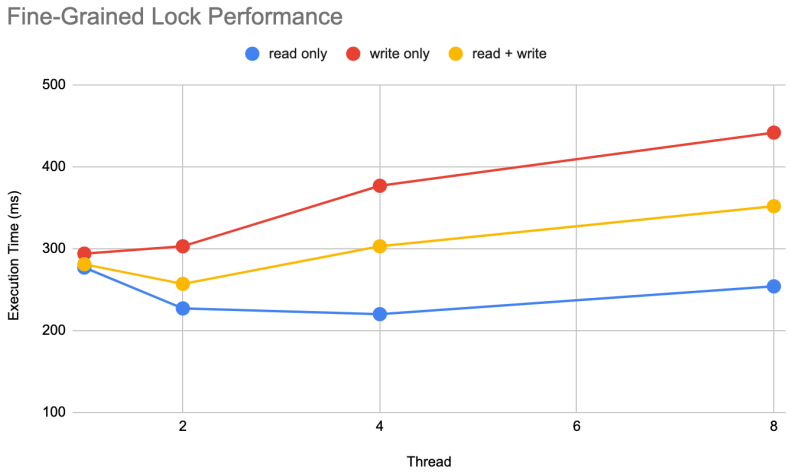
Lock-free did not achieve the expected speedup, probably due to: 1. Not enough contention, but more retries 2. Atomic operations are too expensive: aside from CAS, we’re also using atomic.load() for all **reads** of KV pairs to eliminate torn reads: the KV pair being changed between reading its key and its value. However, when there are only <=8 threads, blocking versions still do better.

Coarse-Grained Lock Performance			
Thread	Read Only	Write Only	Read + Write
1	246 ms	267 ms	249 ms
2	376 ms	432 ms	404 ms
4	481 ms	553 ms	516 ms
8	462 ms	511 ms	491 ms

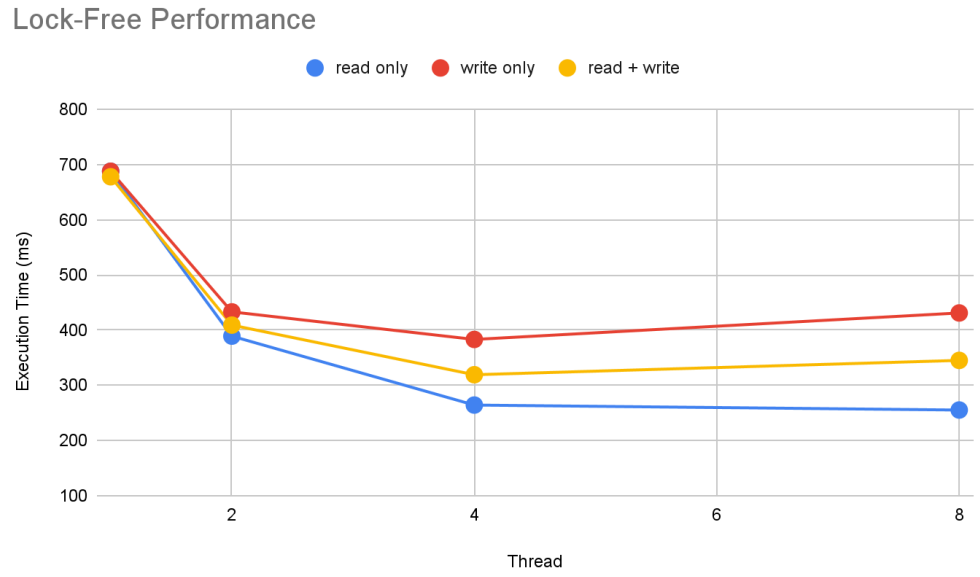


Fine-Grained Lock Performance			
Thread	Read Only	Write Only	Read + Write
1	277 ms	294 ms	281 ms
2	227 ms	303 ms	257 ms
4	220 ms	377 ms	303 ms

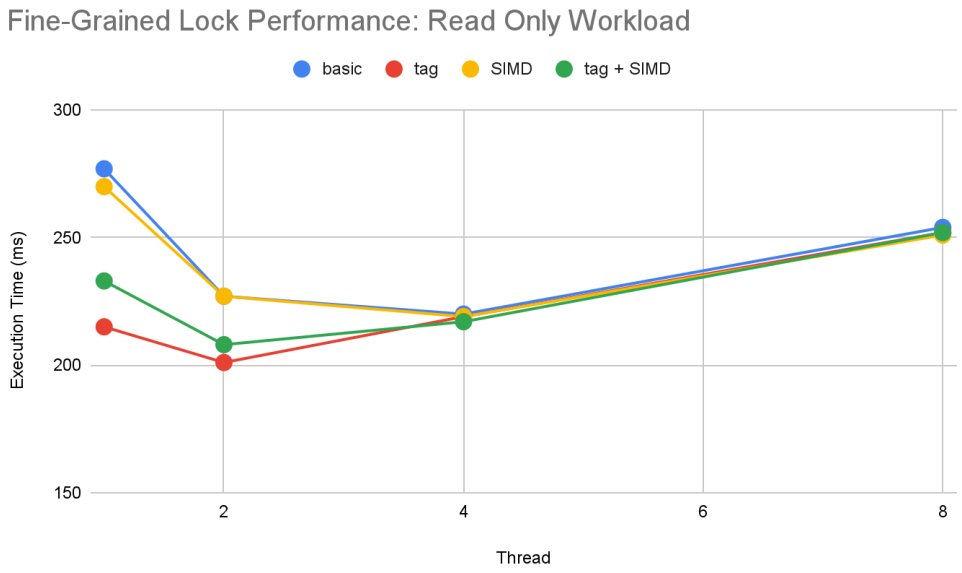
8	254 ms	442 ms	352 ms



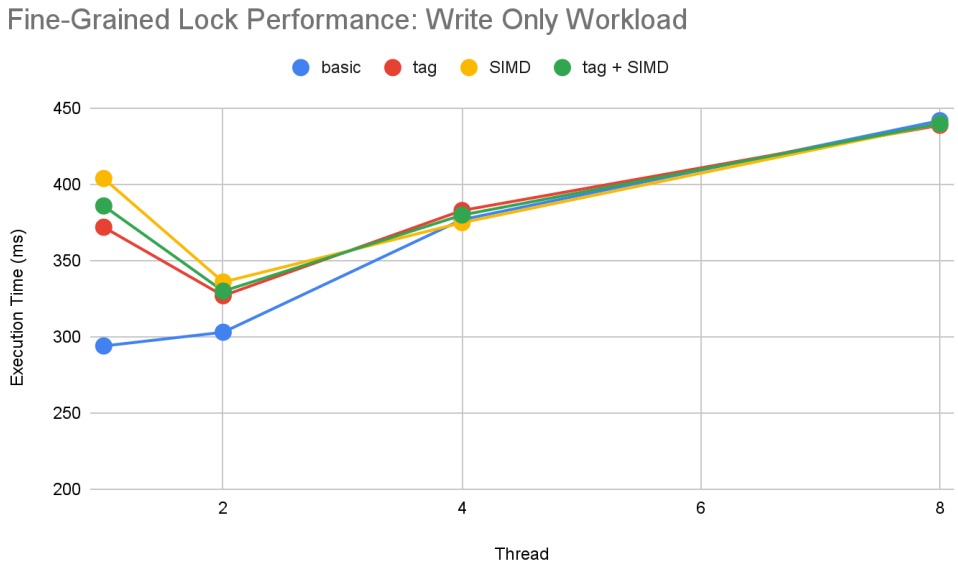
Lock-Free Performance			
Thread	Read Only	Write Only	Read + Write
1	688 ms	688 ms	678 ms
2	389 ms	433 ms	409 ms
4	264 ms	383 ms	319 ms
8	255 ms	431 ms	345 ms



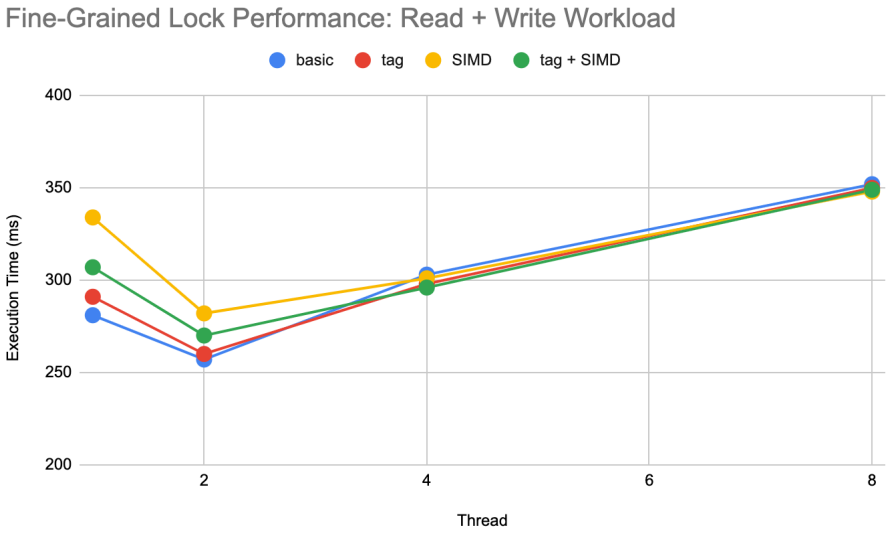
Fine-Grained Performance: Read Only Workload				
Thread	Basic Version	With Tag	With SIMD	Tag + SIMD
1	277 ms	**215 ms**	270 ms	233 ms
2	227 ms	**201 ms**	227 ms	208 ms
4	220 ms	219 ms	219 ms	**217 ms**
8	254 ms	252 ms	**251 ms**	252 ms



Fine-Grained Performance: Write Only Workload				
Thread	Basic Version	With Tag	With SIMD	Tag + SIMD
1	**294 ms**	372 ms	404 ms	386 ms
2	**303 ms**	327 ms	336 ms	330 ms
4	**377 ms**	383 ms	375 ms	380 ms
8	442 ms	**439 ms**	440 ms	440 ms



Fine-Grained Performance: Read + Write Workload				
Thread	Basic Version	With Tag	With SIMD	Tag + SIMD
1	**281 ms**	291 ms	334 ms	307 ms
2	**257 ms**	260 ms	282 ms	270 ms
4	303 ms	298 ms	301 ms	**296 ms**
8	352 ms	350 ms	**348 ms**	349 ms



Remaining Unknowns & Concerns

1. Our SIMD / storage optimization optimizations only work well on read-only cases, not sure whether further optimization would work out.
2. Lock-free tagged optimization requires Double-CAS. Is it unsolvable?
3. We're planning to explore storing tags and pointers as pairs for each KV pair for this lock-free case, not sure if this would help.