15618 final project milestone report

jiaqidon, yizhenw

Jiaqi Dong, Yizhen Wu

Project link:

Website: https://alicewyz.github.io/parallel-linear-hashing/
Github Link: https://github.com/amstqq/15618-final/tree/main

Detailed Schedule:

Nov 30 - Dec 4:

1. Design and implement more comprehensive tests and analysis for lock based linear hashing table (Jackie)
2. Finish primitive design for lock-free linear hashing. Study how Split Ordered List can be applied to Linear Hashing. (Alice)

Dec 5 - Dec 8:

1. Study C++ concurrency primitives (Both)
2. Implement lock-free linked list and write tests. (Alice)

Dec 9 - Dec 11:

1. Use lock-free linked list to implement a lock-free hash table (Jackie)
2. Optimize performance (Alice)

Dec 12 - Dec 15:

1. Performance analysis and benchmarking (Both)

Dec 15 - Dec 17:

1. Final report (Both)
2. Final Poster (Both)

Work Completed:

We finished implementing a coarse-grain locked hash table using Linear Hashing algorithm and with support of C++'s linked list and vector library. The coarse-grained locking algorithm takes a write lock for all functions in the hashtable class. We designed single-threaded and concurrent test cases for insert and get functions to test the performance and correctness of our coarse-locked hash table.

Next we implemented a fine-grain locked hash table of Linear Hashing where we have reader-writer lock for the directory of the hash table, and one reader-writer lock for each bucket in the directory. For all operations, a read lock is first acquired to maximize performance. The

read lock is updated to a write lock when modifications of internal structures of the hashtable are necessary.

We further investigated how Split Ordered List is applied in achieving lock free Extendible Hashing algorithm. We analyzed various ways to utilize the algorithm to achieve lock free Linear Hashing.

Analysis:

INSERT	GET	Total Item	Bucket Size	Thread num	Coarse-Locked (ms)	Fine-Locked (ms)
100%	0%	10000	4	50	18	30
		10000	600	50	187	65
		10000	1600	50	145	27
100%	0%	10000	4	100	19	30
		10000	600	100	62	30
		10000	1600	100	139	47
50%	50%	10000	4	50	9	14
		10000	600	50	65	17
		10000	1600	50	78	27
50%	50%	10000	4	100	10	13
		10000	600	100	42	14
		10000	1600	100	88	23

In the above table, the performance of a fine-locked table performs better when bucket size changes from 4 to 1600 compared to a coarse-locked table. This is because when the bucket size is small, the directory of the table will expand dramatically, and the fine-locked table will need to keep track of more locks than the coarse-locked table, which creates much more overhead that eliminates the benefit of fine-locking. However when the bucket size changes from small to large, the sequential seek in a coarse-locked table will create much more overhead than a fine-locked table, which allows multiple threads to read from buckets when no insert is happening in the currently read buckets.

Goals and Deliverables:

We are on track to complete the fine-grained and coarse-grained locking implementations of Linear Hashing. However, adapting Split Ordered List to Linear Hashing is a more difficult task since we have to re-design many aspects of the algorithm to suit Linear Hashing. We plan to implement a working lock-free implementation and test its performance extensively. If we can identify performance bottlenecks, then we can further improve the algorithm. If time permits, we would like to experiment with shadow paging implementation of lock free Linear Hashing.

Plan to Achieve

- Complete basic coarse-grained and fine-grained locking linear hashing.
- Develop a working and correct lock-free implementation that can beat the speedup of coarse-grained and fine-grained locking implementation.
- Complete basic test cases that ensure correctness and scale tests that compare performance of lock-based vs. lock-free implementations.

Hope to Achieve

- Improve and optimize the lock-free algorithm through experimenting with different ideas from literature.
- Try out shadow paging technique, where a temporary bucket is created and a pointer to it is atomically swapped into the directory. Compare its performance against Split Ordered List.

Deliverables

Execution time and speedup of each algorithm under different threads and different workloads. Demonstrate the ideal scenario for lock free hash table vs. lock-based hashtable.

Current challenge:

Linear hashing uses a different algorithm for splitting bucket number from that of extendible hashing. When splitting a bucket, lock-free extendible hashing only needs to compare and swap one variable, which is the size of the current buckets. However, for lock-free linear hashing, we need to consider atomically updating four elements:

1. Next bucket to split
2. Current round (starting from 0)
3. Number of current buckets
4. Total size of elements

And the updating condition is critical: when we are updating these four elements, if we all update them at the same time and pause other threads for accessing them, our get_bucket function and insert function will quickly become a bottleneck, every thread will try to successfully compare and swap these four elements, but however for some operations, number of current buckets is not needed. However, if we do not ask the number of buckets to be updated together with the other three elements, when thread A is performing an insert and thread B tries to perform the insert as well, thread B may incorrectly update the number of current buckets while waiting for thread A to release other elements.

Poster Session

Performance comparison of all three implementations of Linear Hashing on different workloads (read vs. write heavy). Identify bottlenecks of performance using tools like perf and analyze sync, busy, and compute time.