Project Proposal

Projects are to be done by a team of no more than 2 students. Email the instructor with a brief 1-paragraph description of your project and the team members by the announced due date for approval. One email per each team is required.

Deliverables

Project deliverables (due for project submission) include:

- A presentation at the end of the semester. Details will be announced soon.
- A project report with at most 5 pages of content including text, figures, tables, and references. For better presentation, we recommend that you use either the IEEE or the ACM formatting templates. The report must describe the following:
  - Aims and objectives of the project, the problem being proposed to solve, and overview of your solution approach
  - Past and related efforts on the proposed problem space
  - Detailed description of the problem being solved
  - Detailed description of the solution approach, and the language and system choices on which this solution was implemented. Discuss the concurrency and synchronization techniques that you learned during the course and used to solve the problem. Justify their usage
  - Description of any properties of your solution (typically established through analysis)
  - Description of experimental study, comparing your solution against competing solutions. Specify the experiments conducted, why these experiments were chosen, and what the results imply
  - Conclusion and your recommendations for future work
- Source code and executable. Include a README file that explains how to run your project, command line parameters, expected output, etc.

Topical List

Following is a list of possible course projects. This list is only intended as a starting (or departure) point, and is by no means exhaustive. Students are strongly encouraged to come up with their own project ideas, likely intersecting with their ongoing MS/PhD thesis research, and discuss them with the instructor.

1. **Evaluation of different safe memory reclamation schemes in C or C++** [12, 14]

   Programming languages with manual memory management such as C and C++ typically need extra care when reclaiming memory allocated by lock-free data structures. Specifically, a memory block cannot be deallocated until after all threads are guaranteed to not access this memory blockany more. Although garbage collectors can solve this problem, they are not commonly used with C or C++. Not to mention that building a fully lock-free garbage collector can be even more challenging than a safe memory reclamation scheme. The goal of the project is to study and implement different memory reclamation schemes (2-4 depending on what algorithms are chosen) and evaluate them with real lock-free data structure(s) (e.g., a lock-free list, hashmap, etc.)

2. **Read-Log-Update** [7, 11]

   Read Log Update is a novel extension of the popular read-copy-update (RCU) synchronization mechanism that supports scalability of concurrent code by allowing unsynchronized sequences of reads to execute concurrently with updates. Explore the feasibility of implementing RLU and MV-RLU in the userspace in a managed memory programming language such as Java, Golang, etc. Demonstrate its effectiveness by building some data structures using the RLU primitives (e.g. Queues, Stacks, Skiplists, Hashmaps and Trees) [1] and evaluating them against other concurrent versions.

3. **Machine Learning Computation on Modern Hardware**

   It has been shown in [10] that the performance of Stochastic gradient descent (SGD), the most popular optimization method for model training, is not always better on GPUs than CPUs. The authors show that the performance of computing the SGD depends on various factors including computing architecture (multi-core CPU or GPU), synchronous or asynchronous model updates, and data sparsity. For this project, perform a similar analysis for some other machine learning task such as matrix factorization or an optimization technique such as adam optimizer (`https://pytorch.org/docs/stable/optim.html`).

4. **High Performance Scalable SkipTrie**

   A trie is a tree-like data structure whose nodes store the letters of an alphabet. By structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree. It is possible to represent Tries as Skiplists [13]. The objective for this project is to design, implement, and evaluate a NUMA-aware Scalable SkipTrie implementation based on [4].

5. **Concurrent LSM Trees** [15, 5]

   Log-structured data stores (LSM-DSs) are widely accepted as the state-of-the-art implementation of key-value stores. They replace random disk writes with sequential I/O, by accumulating large batches of updates in an in-memory data structure and merging it with the on-disk store in the background. While LSM-DS implementations proved to be highly successful at masking the I/O bottleneck, scaling them up on multicore CPUs remains a challenge. This is nontrivial due to their often rich APIs, as well as the need to coordinate the RAM access with the background I/O. Design, implement, and evaluate concurrent lock-free Log-Structed Merge Trees in Java.

6. **Finding Concurrency Bugs** [9]

   Concurrency bugs are hard to find, reproduce, and debug. They often escape rigorous in-house testing, but result in large-scale outages in production. Evaluate the ability of TSVD [9] to find bugs by injecting concurrency bugs in production, open source software. You may also find existing bugs in software not already covered by the publication.

7. **Black-box Concurrent Datastructures for NUMA Architectures** [3]

   Node Replication (NR) [3] is a black-box approach to obtaining efficient concurrent data structures that are aware of the NUMA performance artifacts. You may choose one of the two ideas below:

(a) Implement concurrent data structures (e.g. trees, hashmaps, skiplists, etc.) using the NR technique. Through evaluation, show the advantages and disadvantages of this technique in terms of what data structures benefit/don't benfit from this new technique. You should aim to achieve better performance than Lock-free counterparts.

(b) Apply this technique in one production system such as BadgerKV, RocksDB, etc. that was already not done in [3].

8. **Multi-core nested depth-first search** [8]

   Sequential NDFS is a memory-efficient algorithm with optimal linear time complexity for detecting cycles in a graph. It relies on the so-called post-order of the depth first search for its correctness, which makes it inherently sequential and challenging to parallelize. Implement a multi-core nested depth-first search algorithm (MC-NDFS) as described in the paper. In addition, implement an improved version of the algorithm. Do not try to improve on the algorithm itself, but rather investigate more clever ways to access shared data, or other ways to improve the efficiency of the concurrent versions.

9. **Flat Combining Hash Maps and Search Trees** [6]

   Flat combining is a synchronization technique recently proposed by Hendler et. al. (It is based on the idea of software combining.) The key idea is to improve the parallelism of a coarse-grained lock-based data structure by having one thread do the work for all contending threads: when multiple threads concurrently invoke a lock-protected operation, the thread that acquires the lock, called combiner, iteratively does the operation for all threads who are waiting for the lock. Once done, the combiner notifies the waiting threads by writing to (respective) thread-local fields on which the waiting threads (locally) spin. Interestingly, the synchronization overhead of this technique is significantly low. Studies show that, the technique yields higher throughput than fine-grained lock and lock-free versions of the data structure.

   Design and implement a hash map or a search tree (e.g., binary search tree, k-ary tree, red black tree, etc.) that uses flat combining. Compare the throughput with fine-grain and lock-free versions of the data structure.

10. **Elimination-Combining Set** [2]

    Design and implement a set interface that combines two key synchronization ideas: elimination and software combining. Elimination means using operations with opposite semantics (e.g., a stack's push and pop) to directly exchange elements, instead of synchronizing at a central location (e.g., elimination back-off stack). This has been shown to be effective for symmetric workloads. Software combining means having one thread iteratively do the work of multiple operations with identical semantics (e.g., push) while other threads wait, instead of all threads synchronizing at a central location. This has been shown to be effective for asymmetric workloads.

References

[1] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM.

[2] Gal Bar-Nissan, Danny Hendler, and Adi Suissa. A dynamic elimination-combining stack algorithm. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, pages 544–561, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for numa architectures. *SIGPLAN Not.*, 52(4):207–221, April 2017.

[4] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 32:1–32:14, New York, NY, USA, 2015. ACM.

[6] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[7] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 779–792, New York, NY, USA, 2019. ACM.

[8] Alfons Laarman, Rom Langerak, Jaco Van De Pol, Michael Weber, and Anton Wijs. Multi-core nested depth-first search. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, pages 321–335, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 162–180, New York, NY, USA, 2019. ACM.

[10] Y. Ma, F. Rusu, and M. Torres. Stochastic gradient descent on modern hardware: Multi-core cpu or gpu? synchronous or asynchronous? In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1063–1072, May 2019.

[11] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015. ACM.

[12] Ruslan Nikolaev and Binoy Ravindran. Hyaline&#58; fast and transparent lock-free memory reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 419–421, New York, NY, USA, 2019. ACM.

[13] Rotem Oshman and Nir Shavit. The skiptrie: Low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 23–32, New York, NY, USA, 2013. ACM.

[14] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 1–13, New York, NY, USA, 2018. ACM.

[15] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 277–278, New York, NY, USA, 2015. ACM.