ECE/CS 5510 - Multiprocessor Programming - Fall 2018
Course Project 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++**

   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 block any 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.)

   References:

   - Interval-Based Memory Reclamation, Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott, PPoPP'18, https://www.cs.rochester.edu/~scott/papers/2018_PPoPP_IBR.pdf (also includes description for epoch-based reclamation)
   - Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation, Pedro Ramalhete and Andreia Correia, SPAA'17, http://doi.acm.org/10.1145/3087556.3087588
   - Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, Maged M. Michael, IEEE Trans. Parallel Distrib. Syst. June 2004, http://dx.doi.org/10.1109/TPDS.2004.8

2. **Evaluation of lock-free/wait-free bounded/unbounded FIFO queues**

   Both bounded FIFO queues (ring buffers) and unbounded FIFO queues are commonly used data structures in many application domains. However, achieving high scalability is challenging and a number of lock-free and wait-free solutions were proposed recently. The purpose of the project is to implement and compare Michael & Scott's FIFO queue (from the textbook) against other recent solutions. Both ABA-safe and ABA-unsafe (i.e., queues where elements cannot be safely reclaimed without a safe memory reclamation scheme) can be considered.

   References:

   - Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, Maged M. Michael and Michael L. Scott, PODC'1996, http://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf
   - A Wait-free Queue as Fast as Fetch-and-Add, Chaoran Yang and John Mellor-Crummey, PPoPP'16, http://chaoran.me/assets/pdf/wfq-ppopp16.pdf
   - A Wait-free Multi-producer Multi-consumer Ring Buffer, Steven Feldman and Damian Dechev, SIGAPP Appl. Comput. Rev. September 2015, http://doi.acm.org/10.1145/2835260.2835264

3. **Solving for Traveling Salesman Problem through a parallel implementation of a metaheuristic optimization algorithm**

   Traveling Salesman Problem (TSP) is a well-known NP-hard problem. It asks to find the shortest possible route that goes through every city exactly once and returns to the start point, given a set of cities and distance between every pair of cities. Metaheuristic optimization algorithms such as Simulated Annealing and Genetic Algorithm can be used to

provide good solutions for TSP, but they are inherently serial. Therefore, they are hard to parallelize and take a lot of time for finding a solution. Attempt a parallel implementation for either of them to solve for TSP.

References:

- A Parallel Multi-Phase Implementation of Simulated Annealing for the Traveling Salesman Problem, D.R. Mallampati, P.P. Mutalik and R.L. Wainwright, The Sixth Distributed Memory Computing Conference, 1991. Proceedings, https://ieeexplore.ieee.org/document/633303
- A Fast Parallel Genetic Algorithm for Traveling Salesman Problem, Chun-Wei Tsai, S. Tseng, M. Chiang and C. Yang, MTPP 2010: Methods and Tools of Parallel Programming Multicomputers pp 241-250, https://link.springer.com/chapter/10.1007/978-3-642-14822-4_27

4. **A parallel benchmark in Rust language and compare its performance with a corresponding C or C++ implementation**

Quoting from Wikipedia, ***Rust is a systems programming language developed by Mozilla Labs with a focus on safety, especially safe concurrency,*** *supporting functional and imperative-procedural paradigms. Rust is syntactically similar to C++, but its designers intend it to provide better memory safety while still maintaining performance.* In this project, you learn Rust (great if you know already!) and write a benchmark to gauge Rust's performance against C/C++. Through your project, you will validate Rust's claims on performance, memory safety and safe concurrency.

References:

- The Rust Programming Language, https://doc.rust-lang.org/book/
- Rust vs. C++: Fine-grained Performance, http://cantrip.org/rust-vs-c++.html [might be outdated]
- Safe Concurrency with Rust, http://www.squidarth.com/rc/rust/2018/06/04/rust-concurrency.html
- Exploring lock-free Rust 1: Locks, https://morestina.net/blog/742/exploring-lock-free-rust-1-locks

5. **Linearizability Checker in Java**

Testing concurrent systems is challenging. Concurrency and nondeterminism make it difficult to catch bugs in tests, especially when the most subtle bugs surface only under scenarios that are uncommon in regular operation. Build a linearizability checker that takes in a specification and a history and verifies if it is linearizable.

References:

- Horn, Alex, and Daniel Kroening. "Faster Linearizability Checking via P-Compositionality." Lecture Notes in Computer Science (2015): 50–65. Crossref. Web. (https://arxiv.org/abs/1504.00204)

6. **Concurrent Data Structures in Rust**

Implementing concurrent data structures in memory unmanaged languages such as C++ is hard, and is prone to concurrency-memory bugs. Rust is programming language with an important goal: to be as low-level as C++ while still striving to provide memory safety. Implement some of the concurrent data structures (hash maps and trees) from class in Rust and compare them (in terms of performance) to their Java counterparts. Finally, comment on the programming model. Rust also allows unsafe code like C++, but try your best to stick to the Rust semantics and provide an explanation when writing unsafe code.

References:

- The Rust Programming Language, https://doc.rust-lang.org/book/

7. Lock-free version of Zip Trees

Zip tree is a form of randomized binary search tree. One can view a zip tree as a treap in which priority ties are allowed and in which insertions and deletions are done by unmerging and merging paths (unzipping and zipping) rather than by doing rotations. Design a lock-free version of such a tree using ideas from other concurrent tree and skip list implementations.

References:

- https://arxiv.org/abs/1806.06726
- http://drops.dagstuhl.de/opus/volltexte/2018/9807/

8. Flat Combining Hash Maps and Search Trees

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-grain* lock-based data structure by having one thread do the work for all contenting threads: when multiple threads concurrently invoke a lock-protected operation, the thread which 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.

References:

- Flat combining and the synchronization-parallelism tradeoff, D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA 2010), pages 355—364, 2010, http://mcg.cs.tau.ac.il/papers/spaa2010-fc.pdf

9. Concurrent Trees

Design and implement concurrent AVL and/or Red-Black trees. Compare its performance against one-lock course-grained AVL/Red-Black tree and fine-grained locking (only the path and with tri state locks to increase concurrency). Measure the throughput under different read/write ratios, and contention and non-contention situations.

References:

- A Practical Concurrent Binary Search Tree, N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming, pages 257-268, 2010, http://ppl.stanford.edu/papers/ppopp207-bronson.pdf
- The Performance of Concurrent Red-Black Tree Algorithms, S. Hanke, Proceedings of the 3rd International Workshop on Algorithm Engineering, LNCS, Volume 1668/1999, 286-300, http://www.springerlink.com/content/2q914552q2515859/
- Concurrent Search and Insertion in AVL Trees, C. S. Ellis, IEEE Transactions on Computers, vol. C-29, no.9, pp.811-817, Sept. 1980.
- Relaxed AVL trees, main-memory databases and concurrency, O. Nurmi, E. S. Soininen, and D. Wood, International journal of computer mathematics, 1996, vol. 62, no1-2, pp. 23-44 (28 ref.)
- AVL trees with relaxed balance, by K. S. Larsen, Eighth International Parallel Processing Symposium, pp. 888-893, 1994.

10. Non-blocking *k*-ary search tree with manual memory management

A *k*-ary search tree generalizes a binary search tree to trees where each internal node has *k* children. Consider Brown and Helga's non-blocking *k*-ary search tree. Modify it to work without a garbage collector, by using a thread-local pool of pre-allocated nodes and recycling them.

References:

- Non-blocking *k*-ary Search Trees, T. Brown and J. Helga, Technical Report CSE-2011-04, York University, http://www.cse.yorku.ca/techreports/2011/CSE-2011-04.pdf
- Lock-free *k*-ary Search Trees, T. Brown and J. Helga, OPODIS 2011 (to appear)

11. Elimination-Combining Set

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:

- A Dynamic Elimination-Combining Stack Algorithm, H. Hendler, G. B.-Nissan and A. Suissa, OPODIS 2011 (to appear). http://www.cs.bgu.ac.il/~hendlerd/papers/DECS.pdf

12. Pre-emptible Atomic Regions (PARs)

PARs are a restricted form of software transactional memory (STM) that provides a convincing alternative to mutual exclusion monitors, for single-processor systems. A PAR is a sequence of instructions, which is guaranteed to execute atomically. If a higher-priority task is released, the effects of the PAR are undone and the high-priority task gets to execute as if the lower-priority task (with the PAR) never ran at all. Once the lower priority task is scheduled again, the PAR is transparently re-executed. The advantage of this approach is that high priority tasks get to execute quickly.

Design (or reimplement) PAR for multiprocessors that optimizes objectives such as minimize the number of deadline misses, minimize task response times, reduce task jitter, etc.

References:

- Preemptible Atomic Regions for Real-time Java (2005), J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. In 26th IEEE Real-Time Systems Symposium.

13. Parallelization of Molecular Biology Techniques

Design and implement parallel versions of well-known algorithms from molecular biology (e.g., protein folding, pattern/DNA matching/profiling), exploiting insights and intuitions from the concurrency/synchronization patterns learned in the course. The project should detail the choices made and justifications for particular approaches taken.

References:

- A parallel algorithm for pattern discovery in biological sequences, G. Mauri and G. Pavesi, Parallel computing technologies (PaCT-2001) pp. 849-854.
- Parallel tempering algorithm for conformational studies of biological molecules, U. Hansmann, Chem. Phys. Lett. 281 (1997), pp. 140–150.

- New Monte Carlo algorithms for protein folding, U. Hansmann, Y. Okamoto, Current Opinion in Structural Biology, Volume 9, Issue 2, April 1999, pp. 177-183.
- Parallel multiple sequences alignment in SMP cluster, G. Tan, S. Feng and N. Sun. Eighth International Conference on High-Performance Computing in Asia-Pacific Region, 2005. vol., no., pp.6 pp.-431, 1-1 July 2005.

14. Parallel Image Compression Algorithms

Design and implement parallel versions of JPEG or MPEG compression algorithms, exploiting the concurrency/synchronization patterns learned in the course.

References:

- Parallel embedded block coding architecture for JPEG 2000, H.-Chi Fang Y-Wei Chang, T.-Chih Wang, C.-Jr Lian, L.-Gee Chen, IEEE Transactions Circuits and Systems for Video Technology, pp 1086- 097, Volume: 15 Issue: 9, Sept. 2005.
- Open JPEG group, http://www.openjpeg.org/
- Plzip: parallel multi-threaded open-source project, http://www.nongnu.org/lzip/plzip.html
- Methods for encrypting and decrypting MPEG video data efficiently, L. Tang, Proceedings of the fourth ACM international conference on Multimedia, pp. 219 - 229, 1997.

15. Parallel Construction of Supertrees

Supertrees are phylogenies (rooted evolutionary trees) assembled from smaller phylogenies that share some but not necessarily all taxa (leaf nodes) in common. Thus, supertrees can make novel statements about relationships of taxa that do not co-occur on any single input tree while still retaining hierarchical information from the input trees. As a method of combining existing phylogenetic information, supertrees potentially solve many of the problems associated with other methods (e.g., absence of homologous characters, incompatible data types, or non-overlapping sets of taxa). In addition to helping synthesize hypotheses of relationships among larger sets of taxa, supertrees can suggest optimal strategies for taxon sampling (either for future supertree construction or for experimental design issues such as choice of outgroups), can reveal emerging patterns in the large knowledge base of phylogenies currently in the literature, and can provide useful tools for comparative biologists who frequently have information about variation across much broader sets of taxa than those found in any one tree.

Design and implement a new algorithm that exploits parallelism for the construction of supertrees.

References:

- The evolution of supertrees, O. R. P. Bininda-Emonds. Trends in Ecology & Evolution, Volume 19, Issue 6, June 2004, Pages 315-322, ISSN 0169-5347, DOI: 10.1016/j.tree.2004.03.015.
- Flipping: A supertree construction method, D. Chen, L. Diao, O. Eulenstein, D F. Baca, and M. Sanderson, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Volume 61. 2003
- Performance of Flip Supertree Construction with a Heuristic Algorithm, O. Eulenstein, D. Chen, J. G. Burleigh, D. Fernández-Baca, and M. J. Sanderson, Oxford Journals, Life SciencesS, ystematic Biology Volume53, pp. 299-308.
- The (Super) Tree of Life: Procedures, Problems, and Prospects, O. R. P. Bininda-Emonds, J. L. Gittleman, and M. A. Steel. Annual Review of Ecology and Systematics, Vol. 33, (2002), pp. 265-289
- Constructing Majority-Rule Supertrees, J. Dong, D. Fernández-Baca, and F. McMorris, LNCS, 2009, Volume 5724/2009, pp. 73-84.

Project Proposal

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

Deliverables

Project deliverables include:

- Project report as a MS Word/PDF file. 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, excepted output, etc.

Submit a compressed file that includes the deliverables to Canvas as **term_project_pid.zip** or **term_project_pid1_pid2.zip** by the announced due date.