Advanced Multiprocessor Programming
Project Topics and Requirements

Jesper Larsson Träff

TU Wien

April 25th, 2022

Projects

Goal: Get practical, own experience with concurrent algorithms and data structures, including their performance, and obstacles to obtaining the performance that may naively be expected. Learn something new (all of us).

- Projects done in two- (or one-)person groups
  - Content and effort of the project independent of group size
- Each group selects (only!) one project from the list
- Implementation of material from the lecture and beyond
  - But allowed to be creative, and bring in own ideas
- Implementation in C, C++ with OpenMP (or native threads or `pthreads`)
- Or: Implementation in Java with automatic benchmarking and verification with the BLT framework.

Allowed (and encouraged) to study and use additional papers (as well as internet information), but state clearly your sources!

Timeline

- Today: Online announcement of project topics and rules
- Commit next week, 2.5 (check in TUWEL, register for a group) $\rightarrow$ Extended until 9.5.
- We can have regular Q&A meetings (Thursday morning slot)
- Deadline for hand-in: Monday, 20.6.2022, at midnight
- Exam from June 27th to July 1nd (sign up in TISS, TBA)

All projects

- Test for correctness first, use assertions where possible, start sequentially, then gradually increase the number of threads
- Define a good benchmark to measure: latency (time per operation), throughput (number of operations in some given time slot), fairness; as function of the number of threads
- Compare to well-chosen baseline
- Use good experimental practice to be able to make well-founded claims that some implementation is better than another (see HPC lecture). Repeat experiment a large number of times (> 30), report averages with confidence intervals
- State properties of the algorithms and give worst-case bounds where possible. Proofs not required

Projects based on papers beyond the lecture material

Some (many) of the projects are based on research papers with pretty advanced results and many algorithms.

It is definitely not required to and expected that you can (in the time available) implement all proposed solutions. The task is to destill out and understand the basic algorithms, and implement these in a sensible way.

The projects should not be impossible, so apply good judgement on what to implement (explain in the report). But do not trivialize your task!

Memory management

A major problem with many lock- and wait-free algorithms is safe reclamation and reuse of dynamically allocated memory. The problem is often circumvented (Java, the book) by relying on garbage collection to magically solve the (difficult!) problem.

Many of the projects will have issues with memory reclamation, and the papers often take substantial effort to solve the problem (or employ known solutions like Hazard pointers, only touched upon in the lecture).

It is acceptable to "ignore the problem" and let memory leak. This may of course make very long or intensive benchmarks impossible, so be careful.

But, in general, you will not be required to implemt a concurrent memory reclamation scheme.

Benchmarking

Many papers use throughput as the measure of performance. Throughput is hoped/expected to scale (linearly?) with the number of threads/cores.

For some benchmarks and ideas, see

> Vincent Gramoli: More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. PPOPP 2015: 1-10

For complex data structures: Think about the mix of operations, describe clearly, benchmark different scenarios.

Benchmarking

To substantiate the analysis and claims about the implementations, invent and use meaningful performance counters, e.g., number of iterations of important loops, number of successful/unsuccessful CAS operations, ...

Think about tests for fairness and other properties claimed for the data structure. Not easy.

Be careful not to introduce false sharing: Keep performance counters in local per thread variables, summarize at the end.

Performance counters in arrays, e.g., `event[i]` for thread `i`, will be on the same cacheline, heavy updating can result in harmful performance degradation.

Expectations and requirements

- Efficient and correct (!) implementation
- State theoretical properties (formal proofs not required, these are in the papers/lectures)

  (Mention, e.g., invariants, linearizability, progress guarantees)
- Good benchmark analysis
- Short document (English or German)
  - 6-10 pages excluding plots and source code
  - Statement of problem and expectations
  - Description of data structure
  - Main properties
- Benchmark (results, how obtained)
- Code must be available, part of hand-in (explain how to compile and run)
- Project status presentation, short, all
- Final project presentation and examination (individual)

Projects in Java: The BLT Framework

Some of the projects (where marked) can be done in Java.

This year, we (again) use a new framework, BLT, by Klaus Kraßnitzer, for automatically checking data structures for linearizability and for performing Java benchmarking.

Projects in Java are required to use this tool!

Documentation will be available via TUWEL and homepage, a separate introduction-video will follow.

Implementations in C/C++ with atomics and OpenMP

Projects done in C or C++ with C/C++ atomics can use OpenMP to manage thread creation and otherwise support the implementation. For a simple example, see the code for the paper

> Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457-459

which is available via
`https://github.com/parlab-tuwien/lockfree-linked-list`

Machines

For benchmarks, use TU Wien Parallel Computing group system `nebula` (64-core AMD EPYC). Accounts created in late April (Exercise 0: 4K ssh key uploaded via TUWEL).

An ARM-based server will also be available (on request; has different memory system)

Develop gradually, start with own system at home if possible

Good practice so that others can reproduce findings: State properties of machine, compiler, environment (required!). E.g., `gcc version 8.3.0 (Debian 8.3.0-6)`

What to hand in

Your solution/hand-in consists of

- the report describing problems and solutions and benchmark analysis with plots/tables, and
- the source code, including `Makefile`, `README` and other things necessary to compile and run the code. Either report or `README` should give instructions for compiling and running

The hand-in must be uploaded in TUWEL as a single `.zip`-, `.tgz`-, or `.tar.gz`-file. Name the file clearly: your names followed by `_amp_project` and the number of the project you choose.

LaTeX template to be provided.

Project 1: Register Locks (C/C++ only)

Implement:

- Filter-lock (generalized Peterson)
- Tournament tree of 2-thread Peterson locks (see exercise)
- Lamport Bakery, Herlihy-Shavit version (see lecture)
- Lamport Bakery, Lamport's original version
- Boulangerie

Which is better? For baseline performance, compare to the following locks: `pthreads` or native C11 locks (or OpenMP locks), simple test-and-set lock, simple test-and-test-and-set lock

Challenge: Memory behavior. Ensure that memory (register) updates become visible in required order! Explain what happens if not (Peterson).

Register Locks

Boulangerie is another slight variation of Lamport's Bakery, see

> Yoram Moses, Katia Patkin: Mutual exclusion as a matter of priority. Theor. Comput. Sci. 751: 46-60 (2018) 2015
> Yoram Moses, Katia Patkin: Under the Hood of the Bakery Algorithm: Mutual Exclusion as a Matter of Priority. SIROCCO 2015: 399-413

What does this algorithm try to achieve?

Project 2: Bounded-timestamp register locks (C/C++)

Implement Taubenfeld, Lamport, and two out of the other three:

- Szymanski's solution (Boleslaw K. Szymanski: A simple solution to Lamport's concurrent programming problem with linear wait. ICS 1988: 621-626)
- Jayanti et al.'s solution (Prasad Jayanti, King Tan, Gregory Friedland, Amir Katz: Bounding Lamport's Bakery Algorithm. SOFSEM 2001: 261-270)
- Aravind's solution (Alex A. Aravind: Yet Another Simple Solution for the Concurrent Programming Control Problem. IEEE Trans. Parallel Distrib. Syst. 22(6): 1056-1063, 2011)
- Black-white Bakery (Gadi Taubenfeld: The Black-White Bakery Algorithm and Related Bounded-Space, Adaptive, Local-Spinning and FIFO Algorithms. DISC 2004: 56-70)
- Lamport's Bakery (lecture version or original)

Bounded-timestamp register locks

Implement Taubenfeld, Lamport, and two out of the other three (previous slide); verify (make plausible) with performance counters and assertions that time stamps are within bounds.

Which is better? For baseline performance, compare to the following locks: `pthreads` or native C11 locks, simple test-and-set lock, simple test-and-test-and-set lock

Challenge: Memory behavior. Ensure that memory (register) updates become visible in required order!

Project 3: Snapshots (C or Java)

Implement:

- The MRSW wait-free snapshot from the lecture
- The MRMW wait-free snapshot extension from Damien Imbs, Michel Raynal: Help when needed, but no more: Efficient read/write partial snapshot. J. Parallel Distrib. Comput. 72(1): 1-12 (2012)

Benchmark for throughput, with different mixes of update and scan operations. Try to benchmark for correctness (linearizability) by defining good sequences of operations. How does the throughput scale with number of threads?

Project 4: Multi-word CAS (I) (C/C++ only)

The universal hardware instruction CAS (compare-and-swap) can be used to implement generalized CAS operations that work on several locations atomically. Implement an n-CAS operation from either:

- Timothy L. Harris, Keir Fraser, Ian A. Pratt: A Practical Multi-word Compare-and-Swap Operation. DISC 2002: 265-279
- Maya Arbel-Raviv, Trevor Brown: Reuse, Don't Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors. DISC 2017: 4:1-4:16

or combinations thereof.

Compare performance to the baseline CAS instruction of your machine. How does the performance change with increasing n, and increasing number of threads?

Project 5: Multi-word CAS (II) (C/C++ only)

The universal hardware instruction CAS (compare-and-swap) can be used to implement generalized CAS operations that work on several locations atomically. Implement the n-CAS operation from:

- Håkan Sundell: Wait-Free Multi-Word Compare-and-Swap Using Greedy Helping and Grabbing. International Journal of Parallel Programming 39(6): 694-716 (2011)

Compare performance to the baseline CAS instruction of your machine. How does the performance change with increasing n, and increasing number of threads?

Project 6: Multi-word CAS (III) (C/C++ only)

The universal hardware instruction CAS (compare-and-swap) can be used to implement generalized CAS operations that work on several locations atomically. Implement the $\mathfrak{n}$-CAS operation (volatile only) from:

- Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, Igor Zablotchi: Efficient Multi-Word Compare and Swap. DISC 2020: 4:1-4:19

Compare performance to the baseline CAS instruction of your machine. How does the performance change with increasing $\mathfrak{n}$, and increasing number of threads?

Project 7: Queue locks (C/C++ only)

Implement:

- Ticket lock
- Array lock
- CLH Lock
- MCS Lock

from the lecture (use literature as needed).

Which is better? For baseline performance, compare to the following locks: `pthreads` or native C11 locks (or OpenMP locks), simple test-and-set lock, simple test-and-test-and-set lock.

Challenge: Memory management! Explain the problems (how much memory can be wasted?) and how you have solved/circumvented the problems.

Project 8: Readers and Writers Locks (C/C++ only)

Readers-and-writers locks were only touched upon in the lecture, but are important for many applications. Implement (basic algorithms) readers-and-writers locks as described in

> Yossi Lev, Victor Luchangco, Marek Olszewski: Scalable reader-writer locks. SPAA 2009: 101-110
> Irina Calciu, David Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, Nir Shavit: NUMA-aware reader-writer locks. PPOPP 2013: 157-166

Compare, if possible, to readers and writers locks in a threading library like `pthreads`.

Project 9: Try Locks (C/C++ only)

Locks with a try-lock operation were used for the multi-queue in the lecture. Implement (basic algorithms) a lock with a try-locks as described in

> Andreia Correia, Pedro Ramalhete: Strong trylocks for reader-writer locks. PPOPP 2018: 387-388

Compare, if possible, to try-locks in a threading library like `pthreads`.

Project 10: List-based set (I) (C or Java)

Implement the improved, singly-linked list from

> Mikhail Fomitchev, Eric Ruppert: Lock-free linked lists and skip lists. PODC 2004: 50-59

Describe the improvements, and your expectations compared to the lock-free imlementation from the lecture. Compare to the lock-free implementation from the lecture. Compare to baseline implementation with global lock on all set operations. Which are better, for which operations? Discuss possible experimental designs.

Challenge: Memory management!

Project 11: List-based set (II) (C or Java)

Implement the improved, singly-linked list from

> Anastasia Braginsky, Erez Petrank: Locality-Conscious Lock-Free Linked Lists. ICDCN 2011: 107-118

Describe the improvements, and your expectations compared to the lock-free imlementation from the lecture. Compare to the lock-free implementation from the lecture. Compare to baseline implementation with global lock on all set operations. Which are better, for which operations? Discuss possible experimental designs.

Challenge: Memory management!

Project 12: List-based set (III) (C or Java)

Implement the wait-free list from

> Shahar Timnat, Anastasia Braginsky, Alex Kogan, Erez Petrank: Wait-Free Linked-Lists. OPODIS 2012: 330-344

Describe the additional effort to achieve wait-freeness. Compare to the lock-free implementation from the lecture. Compare to baseline implementation with global lock on all set operations (time permitting, to the simple, lock-free implementation from the lecture). Which are better, for which operations? Discuss possible experimental designs.

Challenge: Memory management!

Project 13: List-based set (IV) (C or Java)

Implement:

- List-based set with fine-grained locks
- Binary tree (or better) with coarse- and/or fine-grained locks (with wait-free `contains`?) (develop your own algorithm, be creative)

Compare to baseline implementation with global lock, and efficient, sequential implementation. Compare this baseline also to the performance of the list-based set. Is there an advantage to using a good, sequential data structure?

Look in the literature for fine-grained or lock-free algorithms for tree-data structures. Bonus: Implement what is useful or looks interesting.

Project 14: Binary search tree with fine-grained locking (C or Java)

Implement the binary search tree with fine-grained locks from

- Tyler Crain, Vincent Gramoli, Michel Raynal: A Contention-Friendly Binary Search Tree. Euro-Par 2013: 229-240
- Tyler Crain, Vincent Gramoli, Michel Raynal: A Fast Contention-Friendly Binary Search Tree. Parallel Process. Lett. 26(3): 1650015:1-1650015:17 (2016)

Compre to same binary tree with global locks. Are there memory management issues?

Project 15: Concurrent trie with snapshot (C or Java)

Implement the concurrent trie data structure from

> Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, Martin Odersky: Concurrent tries with efficient non-blocking snapshots. PPOPP 2012: 151-160

For baseline performance, compare against a sequential implementation with locks on all operations.
Implement also the snapshotting operations, if time permits. Are there memory management issues with this data structure?

Project 16: Queues and stacks (C or Java)

Implement:

- Unbounded, lock-free queue
- Unbounded, lock-free stack
- Elimination back-off stack
- Creative alternative: Lock-free queue from
  Adam Morrison, Yehuda Afek: Fast concurrent queues for x86 processors. PPOPP 2013: 103-112

from the lecture (use literature as needed). Try to find good use-cases for concurrent stacks (and queues).

Compare to baseline-implementations with global lock (or more fine-grained locking, if you can come up with a good implementation; for the queue use the two-lock implementation from the lecture).

Challenge: memory management!

Project 17: A lock-free queue (C or Java)

Implement the queue-variation of

> Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, Erez Petrank: BQ: A Lock-Free Queue with Batching. SPAA 2018: 99-109

Compare to the simple, lock-free queue from the lecture. Design benchmark with different mixes and distributions of enqueue and dequeue operations.

Challenge: memory management!

Project 18: Queue-based BFS: lock-free? (C or Java)

This is a creative project. Implement a lock-free queue (for instance, the one from the lecture), and use this to implement a standard, level-based, concurrent/parallel BFS (Breadth-First Search) algorithm. Input is a graph in adjacency list representation and a start vertex, output a BFS tree/labeling of the vertices.

Can/will your implementation be lock-free? What is needed in order to ensure correctness? Are additional synchronization mechanisms (barrier) needed?

Time permitting: How does this compare with a standard, level-by-level, barrier-synchronized implementation in, say, OpenMP?

Challenge: memory management!

Project 19: Wait-free stack (C or Java)

Data structure/algorithm not from the lecture. Implement:

> Yaqiong Peng, Zhiyu Hao: FA-Stack: A Fast Array-Based Stack with Wait-Free Progress Guarantee. IEEE Trans. Parallel Distrib. Syst. 29(4): 843-857 (2018)

Compare to the simple unbounded, lock-free (Treiber) stack from the lecture, and a global lock baseline. Does the algorithm have an ABA problem? What atomic operations are used?

Project 20: Skip-list (I) (C or Java)

Implement:

- Lock-based lazy skip-list
- Lock-free skip-list

from the lecture (use literature as needed).

For baseline performance, compare to sequential skip-list (own implementation! And/or implementation from some standard C/C++ library) with global locks on all set operations.

Challenge: Memory management!

Skip-list (I)

Can the improvements suggested in

> Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457-459

be applied here?

Project 21: Skip-list (II) (C or Java)

Implement the skip list from

> Tyler Crain, Vincent Gramoli, Michel Raynal: No Hot Spot Non-blocking Skip List. ICDCS 2013: 196-205

For baseline performance, compare to sequential skip-list (own implementation! And/or implementation from some standard C/C++ library or from JDK) with global locks on all set operations.

Are there memory management problems?

Project 22: Work-stealing queue (C or Java)

Implement the dynamic work-stealing queue (special case queue, see lecture) from:

> David Chase, Yossi Lev: Dynamic circular work-stealing deque. SPAA 2005: 21-28
> Danny Hendler, Yossi Lev, Mark Moir, Nir Shavit: A dynamic-sized nonblocking work stealing deque. Distributed Comput. 18(3): 189-207 (2006)

For baseline performance, use the array-based, bounded work-stealing queue from the lecture (for a number of operations, or with optimistic wrap-around; how often do operations fail?), and a simple lock-based dynamic queue.

Project 23: Concurrent bags (C or Java)

Study and implement the bag data structure from

> Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, Philippas Tsigas: A lock-free algorithm for concurrent bags. SPAA 2011: 335-344

For baseline performance, compare against a simple (own) sequential implementation with locks on all operations.

Project 24: Wait-free queues (C or Java)

Study and implement the wait-free queue from

> Alex Kogan, Erez Petrank: Wait-free queues with multiple enqueuers and dequeuers. PPOPP 2011: 223-234

Compare against a sequential baseline with global locks, and the standard lock-free queue from the lecture. Are there ABA or memory reclamation problems?

Project 25: Doubly-linked list (C or Java)

Implement the double-linked list from

> Håkan Sundell, Philippas Tsigas: Lock-free deques and doubly linked lists. J. Parallel Distributed Comput. 68(7): 1008-1020 (2008)

Compare to a baseline implementation with global lock on all list operations. Which are better, for which operations? Discuss possible experimental designs.

Challenge: Memory management!

Project 26: FIFO queue (C or Java)

A data structure (enqueue/dequeue operation) only touched upon in the lecture. Implement the FIFO queue from

> Ruslan Nikolaev: A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. DISC 2019: 28:1-28:16

Compare to a baseline implementation with locks on both operations (and if possible the implementation with two locks as on the lecture slides). Which are better, for which operations? Discuss possible experimental designs.

Project 27: Priority queue (C or Java)

A data structure (enqueue/dequeue operation) only touched upon in the lecture. Implement the priority queue from

> Anastasia Braginsky, Nachshon Cohen, Erez Petrank:
> CBPQ: High Performance Lock-Free Priority Queue.
> Euro-Par 2016: 460-474

Compare to a baseline simple priority queue (heap?) with locks. Which is better, for which operations? Discuss possible experimental designs.

Project 28: Multi-queue (C or Java)

The relaxed priority queue data structure (enqueue/dequeue operation) described in the lecture. Implement and benchmark the basic, relaxed priority queue from

> Hamza Rihani, Peter Sanders, Roman Dementiev: Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. SPAA 2015: 80-82
> Marvin Williams, Peter Sanders, Roman Dementiev: Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. CoRR abs/2107.01350 (2021)

Design an experiment that allows to measure how far a deleted minimum can be from the (linearized), "real" minimum.

Compare to a baseline simple priority queue (heap?) with locks. Which is better, for which operations? Discuss possible experimental designs.

Project 29: Hash table (I) (C or Java)

Implement the extensible hash-table (see lecture) originally from

> Ori Shalev, Nir Shavit: Split-ordered lists: Lock-free extensible hash tables. J. ACM 53(3): 379-405 (2006)

Find a good mix of operations for throughput benchmark. As baseline, use lock-based efficient hash-table from standard library (or own implementation).

Can the improvements suggested in

> Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457-459

be applied here?