Advanced Multiprocessor Programming
Project Topics and Requirements

Jesper Larsson Träff

TU Wien

April 17th, 2023

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 three-(or two- or one-)person groups
  - Content and effort of the project independent of group size
- Each group selects (only!) one project from the list
  - Same project may be done by several groups: Let's compare
- 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`)

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, 24.4 (check in TUWEL, register for a group)
- We can have regular Q&A meetings (Thursday morning slot)
- Deadline for hand-in: Monday, 19.6.2023, at midnight
- Exam from June 26th to June 30th (sign up in TISS)

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)

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: 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 as far as 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

See more detailed instructions and hints on separate sheet.

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

Implement:

- Filter-lock (generalized Peterson)
- Tournament tree of 2-thread Peterson locks (as in the exercises)
- Block-Woo filter-lock
- Alagarsamy filter-lock

Which is better (what is a good benchmark, which scenarios)?

Introduce performance counters to verify that the the claimed bounds on overtaking are not violated. Benchmark for fairness.

For baseline performance, compare to `pthreads` or native C11 locks (or OpenMP locks).

Project 1: Register Locks (cont'd)

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

The additional lock algorithms are presented in the paper:

> K. Alagarsamy: A mutual exclusion algorithm with optimally bounded bypasses. Inf. Process. Lett. 96(1): 36-40 (2005)

Project 2: Hardware supported locks

Implement and compare different hardware-supported locks:

- Test-and-set, test-and-test-and-set lock
- Ticket lock
- Array lock
- CLH lock
- MCS lock
- Hemlock

Hemlock is from

> Dave Dice, Alex Kogan: Hemlock: Compact and Scalable Mutual Exclusion. SPAA 2021: 173-183

Benchmark throughput and lock latency under various scenarious (much contention, litte contention). Benchmark for fairness. Use OpenMP or `pthread` mutex'es as baseline for comparison.
Challenge: Memory management.

Project 3: Snapshots

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: Concurrent Queue

Implement the concurrent, multi-producer, multi-consumer queue with dedicated memory reclamation proposed in:

> Oliver Giersch, Jörg Nolte: Fast and Portable Concurrent FIFO Queues With Deterministic Memory Reclamation. IEEE Trans. Parallel Distributed Syst. 33(3): 604-616 (2022)

Implement the basic idea from the pseudo-code of the paper, compare to the simple Michael-Scott FIFO queue from the lecture. Extend to handle memory reclamation. Do not rely on the code of the authors!

Project 5: Skip-list

Implement:

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

from the lecture (use literature as needed).

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

Challenge: Memory management!

Project 5: Skip-list (con't)

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 (to an extend, they can - what difference do they make?)?

Project 6: Work-stealing queue

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 7: Concurrent bags

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 8: Priority queue

Implement the lock-free priority queue (only touched upon in the lecture) 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 9: Hash table

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?