# Advanced Multiprocessor Programming
# Project Topics and Requirements

Jesper Larsson Träff

TU Wien

April 8th, 2019

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

## 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 one- or two-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
  - But allowed to be creative, and bring in own ideas
- Implementation in C, C++ with OpenMP (or native threads or `pthreads`)
  - But if you choose other, you are on your own

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

## Timeline

- Today: Announcement of project topics and rules
- Commit after Easter (informally)
- We can have regular Q&A (Mondays, Thursdays)
- Project status presentation, all groups: Thursday 6.6.2019
- Deadline for hand-in: Monday, 17.6.2019, at midnight
- Exam from June 21th to July 5th (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

## 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 updates can result in harmful performance degradation.

## Expectations and requirements

- Efficient and correct (!) implementation
- Good theoretical analysis (not necessarily formal proofs)

  (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
  - Theoretical analysis
- Benchmark (results, how obtained)
- Code must be available, part of hand-in (explain how to compile and run)
- Project status presentation, short, all (on Thursday 6.6.2019)
- Final project presentation and examination (individual)

## Machines

For benchmarks, use TU Wien Parallel Computing group system `nebula` (64-core AMD EPYC); alternatively `hydra`. Possibly: `ceres` (64-core, 8-way multithreading = 512 thread Fujitsu/Sparc). Accounts after easter, we will need 4K ssh key uploaded via TUWEL.

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 4.7.2 (Debian 4.7.2-5)`

## 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, if necessary, `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.

## Project 1: Register Locks

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

## Project 2: Bounded-timestamp register locks

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 (I)

Implement Aravind, Lamport, and two out of the other three (previous slide); verify (make plausible) with performance counters and asertions 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

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)

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)

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:

- Håkan Sundell: Wait-Free Multi-Word Compare-and-Swap Using Greedy Helping and Grabbing. International Journal of Parallel Programming 39(6): 694-716 (2011)
- Steven D. Feldman, Pierre LaBorde, Damian Dechev: A Wait-Free Multi-Word Compare-and-Swap Operation. International Journal of Parallel Programming 43(4): 572-596 (2015)

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: Queue locks

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!

## Project 7: List-based set

Implement:

- List-based set with fine-grained locks
- List-based set with optimistic synchronization
- List-based set with lazy synchronization
- Lock-free list based-set

from the lecture (use literature as needed).

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 8: Queues and stacks

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 9: Wait-free stack

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 10: Wait-free vector

Data structure/algorithm not from the lecture. Implement:

> Steven D. Feldman, Carlos Valera-Leon, Damian Dechev: An Efficient Wait-Free Vector. IEEE Trans. Parallel Distrib. Syst. 27(3): 654-667 (2016)

Compare to baseline-implementations with global lock (or more fine-grained locking, if you can come up with a good implementation)

## Project 11: Efficient FAA queue

Implement an efficient queue using fetch-and-add from

> Chaoran Yang, John M. Mellor-Crummey: A wait-free queue as fast as fetch-and-add. PPOPP 2016: 16:1-16:13

Compare to baseline-implementations with global lock or more fine-grained locking.

## Project 12: Skip-lists

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!

## Project 13: 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

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 14: Lock-free priority queue

Data structure/algorithm not from the lecture. Implement:

> Deli Zhang, Damian Dechev: A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists. IEEE Trans. Parallel Distrib. Syst. 27(3): 613-626 (2016)

Compare to baseline-implementations with global lock using a good (heap-based) sequential implementation.

## Project 15: Hash table (I)

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).

## Project 16: Hash table (II)

Implement lock-based Hopscotch hashing as proposed in:

> Maurice Herlihy, Nir Shavit, Moran Tzafrir: Hopscotch Hashing. DISC 2008: 350-364

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

## Project 17: Hash-table (III)

Study and implement the wait-free hash map proposed in:

> Pierre LaBorde, Steven D. Feldman, Damian Dechev: A Wait-Free Hash Map. International Journal of Parallel Programming 45(3): 421-448 (2017)

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