The Journal of Open Source Software

Synch: A framework for concurrent data-structures and benchmarks

Nikolaos D. Kallimanis[1]

1 Institute of Computer Science - Foundation for Research and Technology-Hellas (FORTH-ICS)

DOI: 10.21105/joss.03143

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @hkaiser
- @jarrah42

Submitted: 22 March 2021
Published: 02 April 2021

Summary

The recent advancements in multicore machines highlight the need to simplify concurrent programming in order to leverage their computational power. One way to achieve this is by designing efficient concurrent data structures (e.g. stacks, queues, hash-tables, etc.) and synchronization techniques (e.g. locks, combining techniques, etc.) that perform well in machines with large amounts of cores. In contrast to ordinary, sequential data-structures, the concurrent data-structures allow multiple threads to simultaneously access and/or modify them.

Synch is an open-source framework that not only provides some common high-performant concurrent data-structures, but it also provides researchers with the tools for designing and benchmarking high performant concurrent data-structures. The Synch framework contains a substantial set of concurrent data-structures such as queues, stacks, combining-objects, hash-tables, locks, etc. and it provides a user-friendly runtime for developing and benchmarking concurrent data-structures. Among other features, the provided runtime provides functionality for creating threads easily (both POSIX and user-level threads), tools for measuring performance, etc. Moreover, the provided concurrent data-structures and the runtime are highly optimized for contemporary NUMA multiprocessors such as AMD Epyc and Intel Xeon.

Statement of need

The Synch framework aims to provide researchers with the appropriate tools for implementing and evaluating state-of-the-art concurrent objects and synchronization mechanisms. Moreover, the Synch framework provides a substantial set of concurrent data-structures giving researchers/developers the ability not only to implement their own concurrent data-structures, but to compare with some state-of-the-art data-structures. The Synch framework has been extensively used for implementing and evaluating concurrent data-structures and synchronization techniques in papers, such as (Agathos et al., 2012; Fatourou et al., 2018; Fatourou & Kallimanis, 2011, 2012, 2018; Fatourou & Kallimanis, 2014).

Provided concurrent data-structures

The current version of the Synch framework provides a large set of high-performant concurrent data-structures, such as combining-objects, concurrent queues and stacks, concurrent hash-tables and locks. The cornerstone of the Synch framework are the combining objects. A Combining object is a concurrent object/data-structure that is able to simulate any other concurrent object, e.g. stacks, queues, atomic counters, barriers, etc. The Synch framework provides the PSim wait-free combining object (Fatourou & Kallimanis, 2011; Fatourou & Kallimanis, 2014), the blocking combining objects CC-Synch, DSM-Synch and H-Synch (Fatourou

& Kallimanis, 2012), and the blocking combining object based on the technique presented in (Oyama et al., 1999). Moreover, the Synch framework provides the Osci blocking, combining technique (Fatourou & Kallimanis, 2018) that achieves good performance using user-level threads.

In terms of concurrent queues, the Synch framework provides the SimQueue (Fatourou & Kallimanis, 2011; Fatourou & Kallimanis, 2014) wait-free queue implementation that is based on the PSim combining object, the CC-Queue, DSM-Queue and H-Queue (Fatourou & Kallimanis, 2012) blocking queue implementations based on the CC-Synch, DSM-Synch and H-Synch combining objects. A blocking queue implementation based on the CLH locks (Craig, 1993; Magnusson et al., 1994) and the lock-free implementation presented in (Michael & Scott, 1996) are also provided. In terms of concurrent stacks, the Synch framework provides the SimStack (Fatourou & Kallimanis, 2011; Fatourou & Kallimanis, 2014) wait-free stack implementation that is based on the PSim combining object, the CC-Stack, DSM-Stack and H-Stack (Fatourou & Kallimanis, 2012) blocking stack implementations based on the CC-Synch, DSM-Synch and H-Synch combining objects. Moreover, the lock-free stack implementation of (Treiber, 1986) and the blocking implementation based on the CLH locks (Craig, 1993; Magnusson et al., 1994) are provided. The Synch framework also provides concurrent queue and stacks implementations (i.e. OsciQueue and OsciStack implementations) that achieve very high performance using user-level threads (Fatourou & Kallimanis, 2018).

Furthermore, the Synch framework provides a few scalable lock implementations, i.e. the MCS queue-lock presented in (Mellor-Crummey & Scott, 1991) and the CLH queue-lock presented in (Craig, 1993; Magnusson et al., 1994). Finally, the Synch framework provides two example-implementations of concurrent hash-tables. More specifically, it provides a simple implementation based on CLH queue-locks (Craig, 1993; Magnusson et al., 1994) and an implementation based on the DSM-Synch (Fatourou & Kallimanis, 2012) combining technique.

The following table presents a summary of the concurrent data-structures offered by the Synch framework.

Concurrent Object	Provided Implementations
Combining Objects	CC-Synch, DSM-Synch and H-Synch (Fatourou & Kallimanis, 2012)
	PSim (Fatourou & Kallimanis, 2011; Fatourou & Kallimanis, 2014)
	Osci (Fatourou & Kallimanis, 2018)
	Oyama (Oyama et al., 1999)
Concurrent Queues	CC-Queue, DSM-Queue and H-Queue (Fatourou & Kallimanis, 2012)
	SimQueue (Fatourou & Kallimanis, 2011; Fatourou & Kallimanis, 2014)
	OsciQueue (Fatourou & Kallimanis, 2018)
	CLH-Queue (Craig, 1993; Magnusson et al., 1994)
	MS-Queue (Michael & Scott, 1996)
Concurrent Stacks	CC-Stack, DSM-Stack and H-Stack (Fatourou & Kallimanis, 2012)
	SimStack (Fatourou & Kallimanis, 2011; Fatourou & Kallimanis, 2014)
	OsciStack (Fatourou & Kallimanis, 2018)
	CLH-Stack (Craig, 1993; Magnusson et al., 1994)
	LF-Stack (Treiber, 1986)
Locks	CLH (Craig, 1993; Magnusson et al., 1994)
	MCS (Mellor-Crummey & Scott, 1991)
Hash Tables	CLH-Hash (Craig, 1993; Magnusson et al., 1994)
	A hash-table based on DSM-Synch (Fatourou & Kallimanis, 2012)

Benchmarks and performance optimizations

For almost every concurrent data-structure, Synch provides at least one benchmark for evaluating its performance. The provided benchmarks allow users to assess the performance of concurrent data-structures, as well as to perform some basic correctness tests on them. All the provided benchmarks offer a great variety of command-line options for controlling the duration of the benchmark, the amount of processing cores and/or threads to be used, the contention, the type of threads (i.e. user-level or POSIX), etc.

Source code structure

The Synch framework (Figure 1) consists of 3 main parts, i.e. the Runtime/Primitives, the Concurrent library and the Benchmarks. The Runtime/Primitives part provides some basic functionality for creating and managing threads, functionality for basic atomic primitives (e.g. Compare&Swap, Fetch&Add, fences, simple synchronization barriers, etc.), mechanisms for memory allocation/management (e.g. memory pools, etc.), functionality for measuring time, reporting CPU counters, etc. Furthermore, the Runtime/Primitives provides a simple and lightweight library of user level-threads (Fatourou & Kallimanis, 2018) that can be used in order to evaluate the provided data-structures and algorithms. The Concurrent library utilizes the building blocks of the Runtime/Primitives layer in order to provide all the concurrent data-structures (e.g. combining objects, queues, stacks, etc.). For almost every concurrent data-structure or synchronization mechanism, Synch provides at least one benchmark for evaluating its performance.

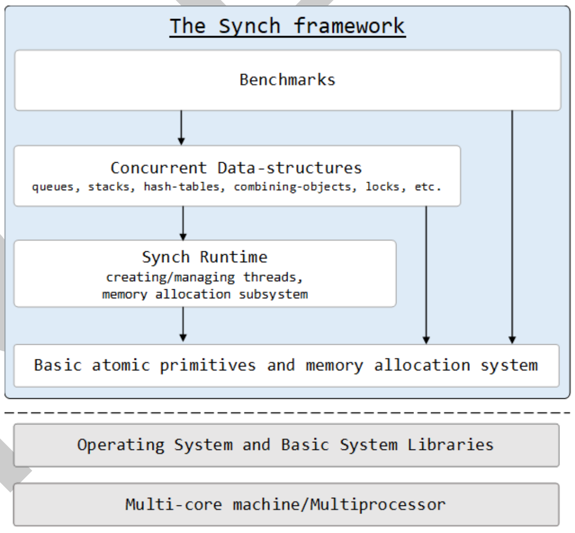
The Synch framework
Benchmarks
Concurrent Data-structures
queues, stacks, hash-tables, combining-objects, locks, etc.
Synch Runtime
creating/managing threads,
memory allocation subsystem
Basic atomic primitives and memory allocation system
Operating System and Basic System Libraries
Multi-core machine/Multiprocessor

Figure 1: Code-structure of the Synch framework.

Requirements

- A modern 64-bit multi-core machine. Currently, 32-bit architectures are not supported. The current version of this code is optimized for the x86_64 machine architecture, but the code is also successfully tested in other machine architectures, such as ARM-V8 and RISC-V. Some of the benchmarks perform much better in architectures that natively support Fetch&Add instructions (e.g., x86_64, etc.). For the case of x86_64 architecture, the code has been evaluated in numerous Intel and AMD multicore machines. In the case of ARM-V8 architecture, the code has been successfully evaluated in a Trenz Zynq UltraScale+ board (4 A53 Cortex cores) and in a Raspberry Pi 3 board(4 Cortex A53 cores). For the RISC-V architecture, the code has been evaluated in a SiFive HiFive Unleashed (4 U54 RISC-V cores) respectively.
- As a compiler, gcc of version 4.3 or greater is recommended, but the framework has been successfully built with icx and clang.

- Building requires the following development packages:
    - `libpapi` in the case that the user wants to measure performance using CPU performance counters.
    - `libnuma`

Acknowledgments

This work was partially supported by the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the "European Processor Initiative: Specific Grant Agreement 1" (Grant Agreement Nr 826647).

Many thanks to Panagiota Fatourou for all the fruitful discussions and her significant contribution on the concurrent data-structures implementations presented in (Fatourou & Kallimanis, 2011, 2012, 2018; Fatourou & Kallimanis, 2014).

Thanks also to Spiros Agathos for his feedback on the paper and committing some valuable patches to the repository. Many thanks also to Eftychia Datsika for her feedback on the paper.

References

Agathos, S. N., Kallimanis, N. D., & Dimakopoulos, V. V. (2012). Speeding up OpenMP tasking. *Euro-Par 2012 Parallel Processing*, 650–661. https://doi.org/10.1007/978-3-642-32820-6_64

Craig, T. S. (1993). *Building FIFO and priority queuing spin locks from atomic swap* (No. 93-02-02). Computer Science Department, University of Washington.

Fatourou, P., & Kallimanis, N. D. (2011). A highly-efficient wait-free universal construction. *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 325–334. https://doi.org/10.1145/1989493.1989549

Fatourou, P., & Kallimanis, N. D. (2018). Lock Oscillation: Boosting the Performance of Concurrent Data Structures. In J. Aspnes, A. Bessani, P. Felber, & J. Leitão (Eds.), *21st international conference on principles of distributed systems (OPODIS 2017)* (Vol. 95, pp. 8:1–8:17). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.OPODIS.2017.8

Fatourou, P., & Kallimanis, N. D. (2012). Revisiting the combining synchronization technique. *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 257–266. https://doi.org/10.1145/2145816.2145849

Fatourou, P., & Kallimanis, N. D. (2014). Highly-efficient wait-free synchronization. *Theory of Computing Systems*, *55*(3), 475–520. https://doi.org/10.1007/s00224-013-9491-y

Fatourou, P., Kallimanis, N. D., & Ropars, T. (2018). An efficient wait-free resizable hash table. *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 111–120. https://doi.org/10.1145/3210377.3210408

Magnusson, P., Landin, A., & Hagersten, E. (1994). Queue locks on cache coherent multiprocessors. *Proceedings of 8th International Parallel Processing Symposium*, 165–171. https://doi.org/10.1109/IPPS.1994.288305

Mellor-Crummey, J. M., & Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, *9*(1), 21–65. https://doi.org/10.1145/103727.103729

Michael, M. M., & Scott, M. L. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 267–275. https://doi.org/10.1145/248052.248106

Oyama, Y., Taura, K., & Yonezawa, A. (1999). Executing parallel programs with synchronization bottlenecks efficiently. *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, *16*, 95. https://doi.org/10.1142/4278

Treiber, R. K. (1986). *Systems programming: Coping with parallelism.* (RJ-5118). International Business Machines Incorporated, Thomas J. Watson Research.