Register locks

Advanced Multiprocessor Programming, SS2022

2022-06-17

Tobias Florian
01526529
tobias.florian@tuwien.ac.at

Lukas Heisler
12038042
e12038042@student.tuwien.ac.at

Contents

1 Introduction

The aim of this project is the implementation and investigation of below listed register locks. The locks are tested for correctness and later benchmarked with respect to latency, throughput and fairness. As a baseline, the implementations are compared to native C11 and OpenMP locks, a simple test-and-set lock and a test-and-test-and-set lock.

- Filter-lock (Generalized Peterson)
- Tournament-tree of 2 thread Peterson locks
- Lamport Bakery, Lamport's original version
- Lamport Bakery, Herlihy-Shavit version
- Boulangerie, Yoram Moses and Katia Patkin version

2 Lock implementation and theoretical properties

In this section, the general ideas and the C++ implementations of the locks are shortly discussed. The code can be found in the attached *locks.hpp* and *locks.hpp* files. All of the presented register locks are supposed to work for n greater than 2 threads and are proven to be mutually exclusive, starvation free and deadlock free, in the literature.

2.1 Filter-lock (Generalized Peterson)

The first lock to visit is the Filter lock, that generalizes the two-thread Peterson lock. The basic idea of the Filter lock is the construction of n-1 levels that each thread has to pass before entering the critical section. This is illustrated by the figure bellow.

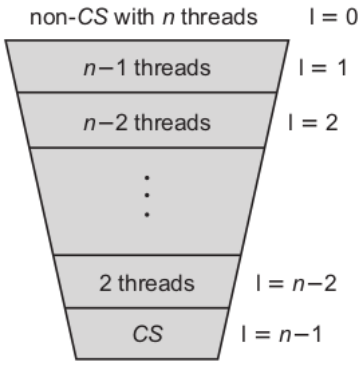

Figure 1: Filter lock sketch

Each of the levels feature that if more than one thread is trying to reach the next level, at least one thread has to wait and at least one thread succeeds entering the next level. In terms of code, the protocol uses an n-element integer array level[] for which for a thread A, the value level[A] reveals the highest level thread A is trying to enter. Additionally, each level uses an n-element victim[] array to find out which thread may not enter the next level. In other words; the n-1 levels prevent all the threads, except one, to enter the critical section. Entering the critical section is equivalent to being at level 1 and not being the victim.

2.2 Peterson lock Tournament-tree

A further way to generalize the two-thread Peterson lock is to set up a Peterson lock Tournament-tree for n locks. For that, consider n as a power of two building up a binary tree with n/2 leaves, where each thread is assigned a leaf. Each leaf and node represent a two-thread Peterson lock that treats one thread as thread 0 and the other one as thread 1. As in the figure below, threads start at the leaves and move upwards when they acquire the lock at a node. When a thread finally receives the lock at the root node, it can enter the critical section. If this thread again leaves the critical section, it releases the locks of all the nodes that it has acquired.

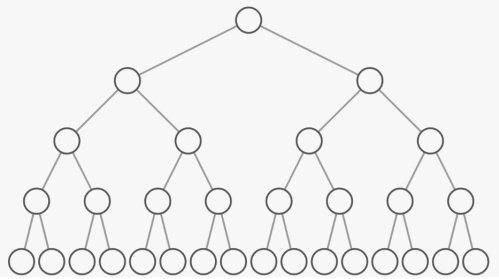

Figure 2: Binary tree

2.3 Lamport Bakery, Lamport's original version - Herlihy-Shavit version

Lamports Bakery algorithm for mutual exclusion is a very influential and early solution to the mutual exclusion problem. The original algorithm is correct under the weak assumption, that registers are safe. Meaning a read operation on a safe register, that overlaps a write to the same register can return an arbitrary value in the registers range including a value that has never been written to the register. The Bakery algorithm obtains its name from the ordering scheme of some shops, where each customer who enters the shop will draw a number from an increasing number sequence and the customer with the smallest number is served. In the algorithm, for two threads that are both far enough along to enter the critical section, the one with the smaller number has priority over the other. Ties are broken according to the process IDs.
With respect to the code, the following happens: Having n threads, one n-element boolean flag[] array and one n-element label[] array are needed, with the label[] array holding the ticket numbers of the threads. If a thread i wants to get into the critical section it first shows its interest by setting the flag[i] to true and then finds the maximum number in the label[] array and sets the label[i] value to maximum increased by one. This is called the doorway. Subsequently, it has to maintain in the waiting room until no other thread j has a lower ticket (label[j]). And, as already mentioned, if i and j have the same ticket, the one with the lower thread ID has priority and can enter the critical section. In this work, two slightly differing versions are implemented, i.s. the original version of Lamport's bakery and secondly, the version of Herlihy and Shavit. The difference is, that in the original version, a thread i sets its flag[i] to false immediately after drawing the ticket, before entering the while loop (waiting room). Further, as the thread leaves the critical section, it sets its ticket number (label[i]) to zero again. Whereas, in the other version (Herlihy-Shavit), flag[i] stays true when entering the while loop (waiting room). When leaving the critical section, flag[i] is then set to false, letting the next thread into the critical section. Subsequently, the while loop looks slightly different. This can be seen in the source code.

2.4 Boulangerie lock

The Boulangerie lock is another variation of the original Lamport's Bakery lock. The basic idea is to avoid thread blocking at some parts in order to improve performance - in detail, there are two optimizations. Speaking about the first one, recall that a thread i sets its label[i] to zero when it leaves the critical section and unlocks the lock. Thus, in a program of low contention, it might occur that all the labels are zero. However, imagine thread i draws its label[i] to be 1. Subsequently, only threads j with lower thread IDs can have priority over thread i. Therefore, it is not necessary to compare the labels for j $>$ i in the for loop (waiting room). In order to illustrate this in terms of the code, please refer to the pseudo codes below. In order to point out the difference to the bakery lock, both are visualized.

The Bakery Algorithm for process i.

```
0  Initialize: number[i] = 0;  choosing[i] = false;
1  while true do
2      noncritical section;
3      choosing[i] ← true;
4      number[i] ← 1 + max{number[1], ..., number[N]};
5      choosing[i] ← false;
6      forall the j ≤ N s.t. j ≠ i do
7          await choosing[j] = false ;
8          await number[j] = 0 ∨ ⟨number[i], i⟩ <L ⟨number[j], j⟩ ;
9      critical section;
10     number[i] ← 0;
```

Figure 3: Lamport Bakery Original: difference in namings -> number = label; choosing = flag

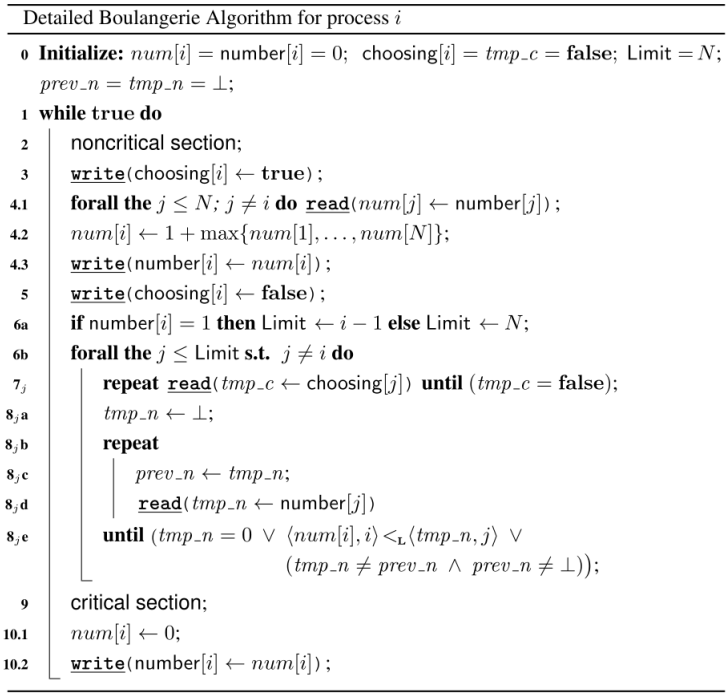

Figure 4: Boulangerie lock

Herein, we see that if thread i draws ticket number 1, the following for loop in line 6b breaks after one loop iteration and avoids further blocking. This improvement comes into play when thread contention is low, but does not really take place when contention is high.

For the second improvement, the Boulangerie algorithm takes advantage of inconsistent reads. We consider the blocking imposed by line 8_j. Here process i will block until it reads a value that contradicts the fact that j has a better ticket. Suppose for example $number[i] = 10$ and i reads the value 5 in $number[j]$. This confirms the fact that j has a better ticket. On a consecutive read on $number[j]$ the read operation could return a different value in the range of the register, because it is only a safe register. Suppose on the second read, $number[j]$ returns 4, which still states that j has a better ticket than i. But since j is already in the Bakery, it performs no writes on $number[j]$ thus $number[j]$ is stable and all reads must return the same value. Hence if i returns two different reads on $number[j]$ while blocking on 8_j it has proof that j was outside at least during one of the reads. This means $i \triangleleft j$ is true and i can continue blocking on 8_j on another process. The unnecessary blocking can be avoided by comparing the previous reads on the $number$ register.

2.5 Baseline locks

In order to compare the previously discussed algorithms the following shortly presented locks should serve as a baseline.

2.5.1 Standard C11 lock and OpenMP lock

The C11 std::mutex class and the OpenMP methods omp_init_lock, omp_set_lock and omp_unset_lock are already implemented and considered as the fast baseline for comparison. Both locks are programmed into custom classes resembling the class structure of the above mentioned classes. That is in the constructor the number of threads is passed but not used and in the *.lock(int thread_id)* and *.unlock(int thread_id)* member functions, the thread id is passed but not used. This is just to comply with the code and avoid unnecessary checks.

2.5.2 Test-And-Set and Test-And-Test-And-Set locks

The simplest lock, the Test-And-Set lock, only needs a single shared memory that shows whether a lock is free or not. The test-and-set method atomically writes 1 to this single memory and immediately returns its old value. The lock can be build, such that if the method returns 0, it can enter the critical section and sets the variable to 1. When unlocking, it sets the variable to 0 again. If the method returns 1, it must spin, as long

as it eventually returns 0. This protocol does not serve FIFO and is not fair. This algorithm fulfills mutual exclusion but can lead to resource contention, since all the threads try to acquire the lock at the same time. A slight modification to Test-And-Test-And-Set lock can help out. Thereby, the first step of the lock method is to test if the lock variable is 0 (lock is free) or 1. Only if it is free, the thread tries to acquire the lock like in the Test-And-Set lock. This improvement is expected to enhance performance.

2.6 General remarks for locks

The first 5 locks, Filter, Tournament tree, and the three ticket

3 Benchmarks

After benchmarking on two local machines, all benchmarks are conducted on the nebula cluster. System info about the cluster can be found in Appendix A. The benchmark programs as well as the locks are compiled with gcc, using the following optimization flags:

```
CXX=gcc
CXXFLAGS := $(CXXFLAGS) -std=c++14 -fopenmp -Wall -pedantic -march=native -fconcepts -O3
```

Refer to the full program code in the following files: *bm_correctness.cpp*, *bm_fairness_throughput.cpp* and *bm_latency.cpp* as well as the *Makefile*.

3.1 Correctness

First of all, each lock is tested with respect to correctness. Therefore, n threads try to acquire the lock m times. Inside the critical section, a shared variable control is set to the id of the thread that is inside the critical section. For all threads with uneven id, a computational intensive task is to be performed.

```
if(id%2)
{
    const int max = 100000;
    std::vector<std::size_t> a(max);

    for(std::size_t i = 1; i != a.size(); ++i) {
        for (std::size_t j = 2 * i; j < a.size(); j += i) {
            a[j] += i;
        }
    }
    for (std::size_t i = 1; i != a.size(); ++i) {
        if(a[i] == i) {
            //std::cout << i << " is a perfect number!" << std::endl;
        }
    }
}
```

This will increase the chance of another thread entering the CS while a preceding thread is still inside the CS, should the lock not work correctly. After the computationally intensive task the thread in CS will read the value of the control variable and compare it with its own id. Should the two values not match because a subsequent thread changed the variables value, then a message is written to console, informing the user about the erroneous lock. If no other thread can enter CS while it is occupied, the program will finish without writing an error message to console.
An exemplary output of the correctness benchmark can be found in Appendix B. According to this benchmark, all locks are implemented correctly.

3.2 Fairness-Throughput

Next, the locks are benchmarked in terms of fairness and throughput. Thereby, a prescribed amount of time is simulated where the threads try to acquire the lock as often as they can. Correctness is measured as how the lock acquisitions are distributed among the threads and throughput is measured as how many acquisitions are achieved in the simulation time. The time is measured by one thread, who will set a shared flag to false as soon as the time is over. All other locks are trying to acquire the lock as long as the flag is set to true. Consequently, there is a time interval where threads are still trying to acquire the lock even after the timing has stopped. In the worst case, all threads will wait for the critical section one more time. The simulation time is set to one second and thus orders of magnitudes greater than the worst case latency for one acquisition, justifying this approach. Below, the OpenMP parallel section is visible.

```
#pragma omp parallel private(id) shared(lock, _continue) reduction(min:minV) reduction(+:counter)
    reduction(max:maxV)
{
    id = omp_get_thread_num();

```

```
    if(id == 0)
    {
        auto start = std::chrono::high_resolution_clock::now(); //start time measurement
        while(_continue)
        {
            auto finish = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double, std::micro> elapsed = finish - start;
            if(elapsed.count() > std::chrono::microseconds(sec).count())
            {
                _continue = false;
            }
        }
    }
    else
    {
        while(_continue) //during timeframe, try to obtain lock as often as possible
        {
            lock.lock(id);
            lock.unlock(id);
            counter++;
        }
        minV = counter;
        maxV = counter;
    }

}
```

The benchmark is conducted on the nebula cluster (see details in Appendix A) for different number of threads. Each thread configuration is executed 30 times and the confidence intervals are calculated from here. Since the tournament lock only works for power of 2 thread configurations, this lock is excluded in all configurations that do not comply with this condition. As it is visible above, no work is conducted in the CS, so the benchmark focuses on the lock acquisition overhead and fairness measures alone.

Regarding the total throughput visible in Figure 5, the lock implementations differ by more than two orders of magnitude. Unsurprisingly, the native C11 lock scales best, not only in the total number of lock acquisitions but also in the invariance towards the number of threads. The Test-And-Test-And-Set lock (TATAS) and the OpenMP lock perform equally well. At lower thread count, both locks perform extremely well with around the same number of acquisitions as the baseline C11 lock. With increasing threads, both locks stale slightly. At 64 threads, around $2 \cdot 10^6$ locks are acquired in one second. The simple Test-And-Set lock performs equal with the TATAS lock in terms of throughput up until 16 threads. For more threads, there is a steep decline in throughput. As mentioned in Section 2, the TATAS shows its advantage here.
All implemented locks except the Filter lock perform consistent at around $5 \cdot 10^5 \frac{1}{sec}$ to $8 \cdot 10^5 \frac{1}{sec}$ for larger number of threads after declining from $1 \cdot 10^6 \frac{1}{sec}$ at a two thread configuration. The Filter lock has the lowest throughput for the compared locks and shows a high dependency on the number of threads trying to enter the CS even for 32 or 64 threads.

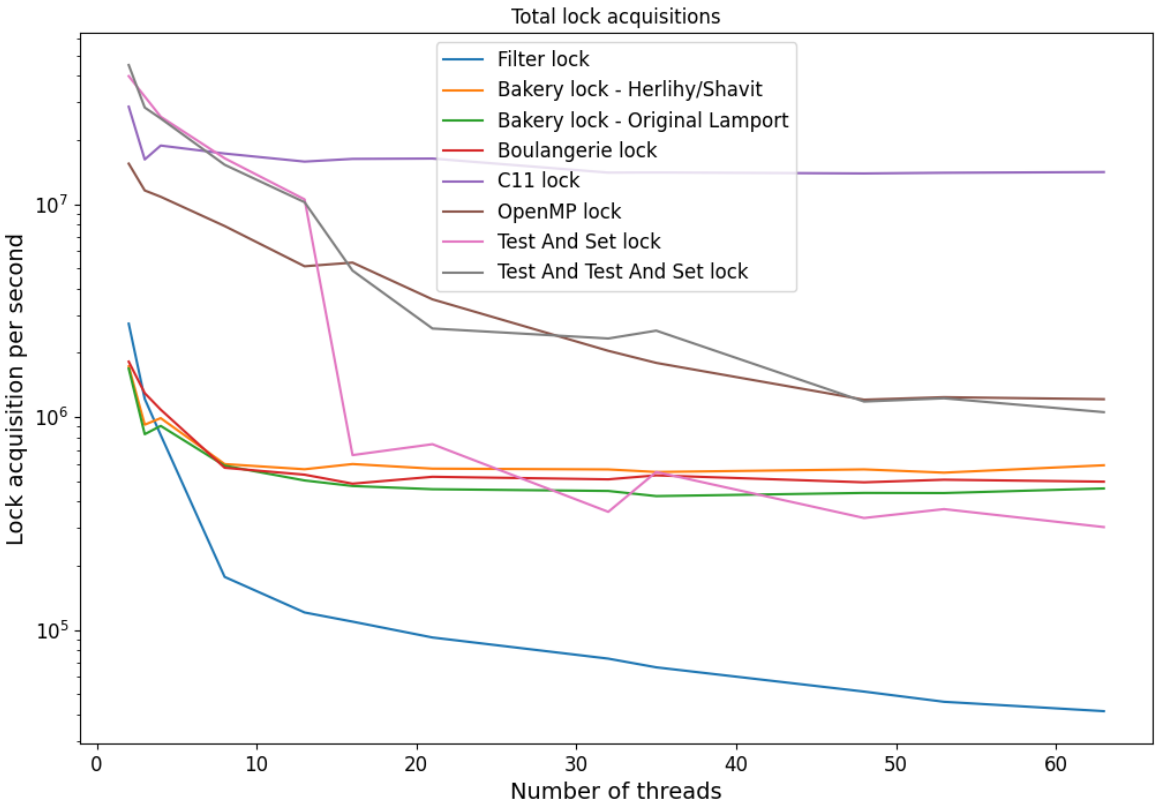

Figure 5: Total number of lock acquisitions in one second

The fairness of a lock describes the difference in the number lock acquisitions when all threads want to acquire the lock simultaneously. A perfectly fair lock would result in the minimum and the maximum number of acquisitions over all locks to be roughly equal. This could be achieved by a ticketing measure that orders the entrance to the CS by the time a thread took a ticket, allowing the thread with the smallest ticket number into the CS, since that thread waited the longest. Naturally, there is a uncertainty associated with this benchmark. A thread could stall for no apparent reason before taking a ticket, resulting in a lower number of acquisitions compared to a ”faster” thread. Only in combination with a latency benchmark, a complete picture about a lock can be obtained.

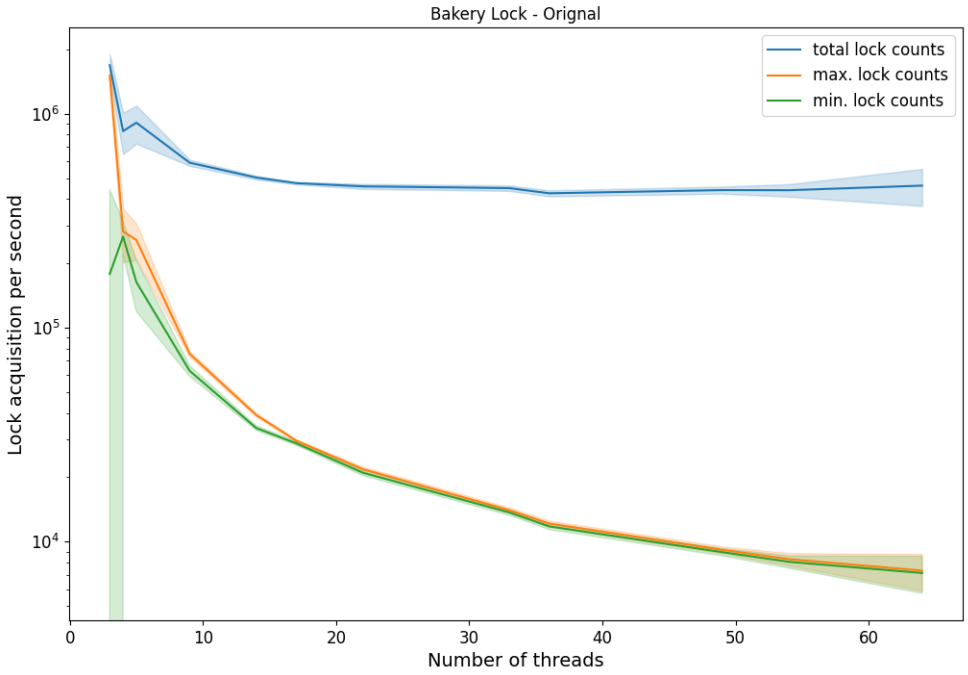

Figure 6: Lamport Bakery Original version throughput and fairness

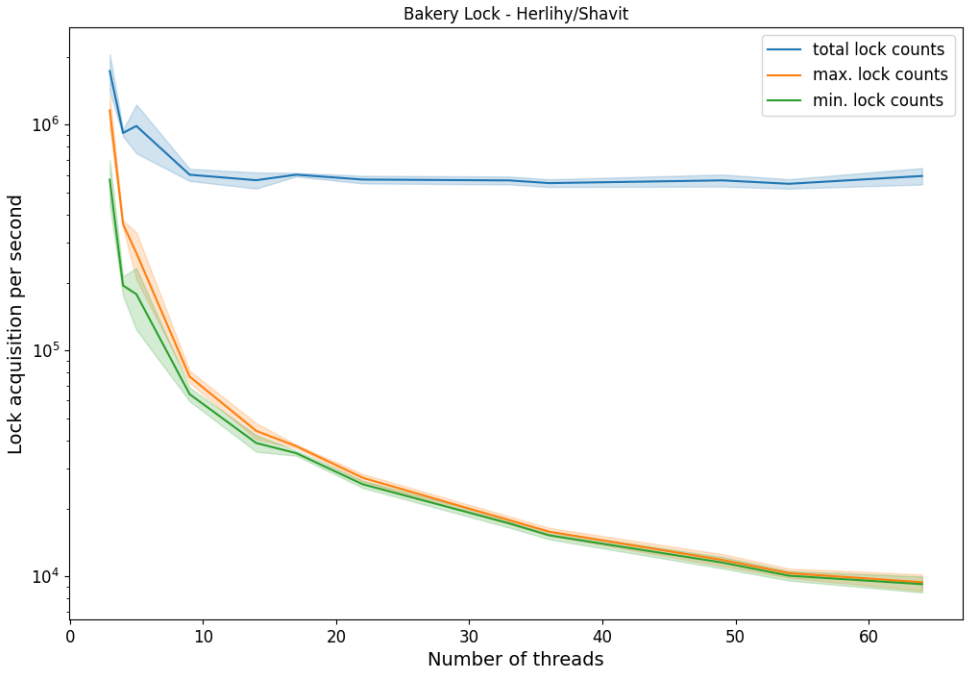

Figure 7: Lamport Bakery Herlihy version throughput and fairness

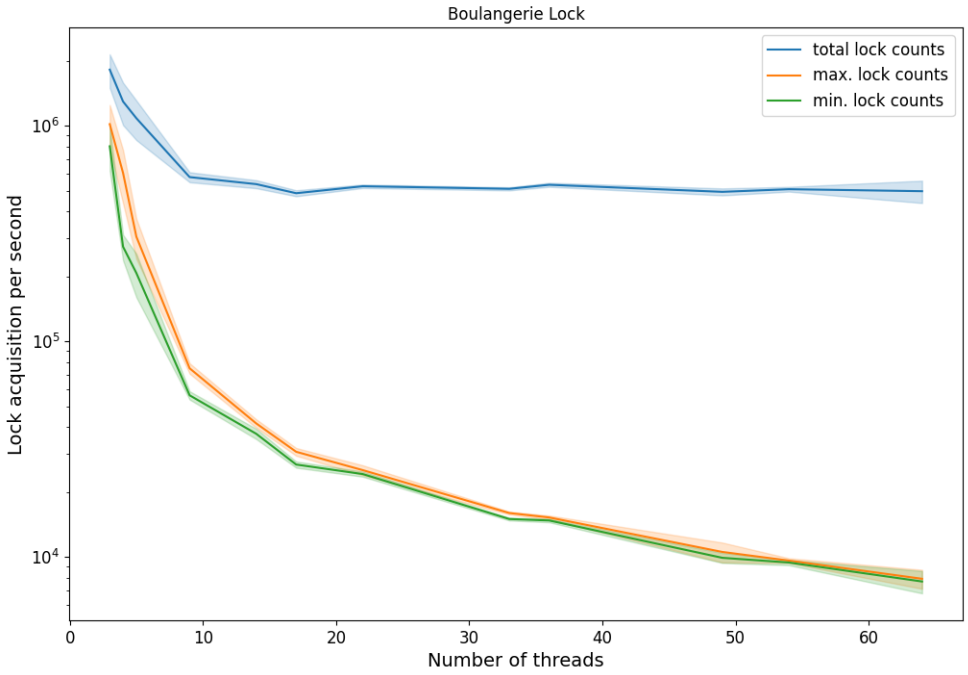

Figure 8: Boulangerie lock throughput and fairness

As described in Section 2, the Bakery algorithms as well as the improved Boulangerie version all rely on a ticketing system for the acquisition process of a lock. This shows clearly in the throughput-fairness plots visible in the Figures 6, 7 and 8. The maximum and minimum lock count are reduced after the parallel section by the OpenMP *min* and *max* reduce operators. For all locks, the two numbers are almost identical, proving the fairness of the ticket based locks. Compared to a lock that does not serve FIFO like the TAS or TATAS lock, a huge advantage of the FIFO based locks is apparent. Figure 9 shows the TAS lock displaying a huge difference in the maximum and minimum number of lock acquisitions. The large confidence interval around the minimum lock count shows, that there are instances where ”slower” threads manage to obtain the lock only once or twice while other threads are able to obtain in 10^5 or 10^6 times more often, making up for almost all of the total acquisitions.
The baseline C11 lock shows a more fair approach compared to TAS and TATAS, as can be seen in Figure 10. The implementation of the lock seems to be using a ticketing approach with an optimization towards performance. The min and max count diverge slightly but are always in the same neighborhood.

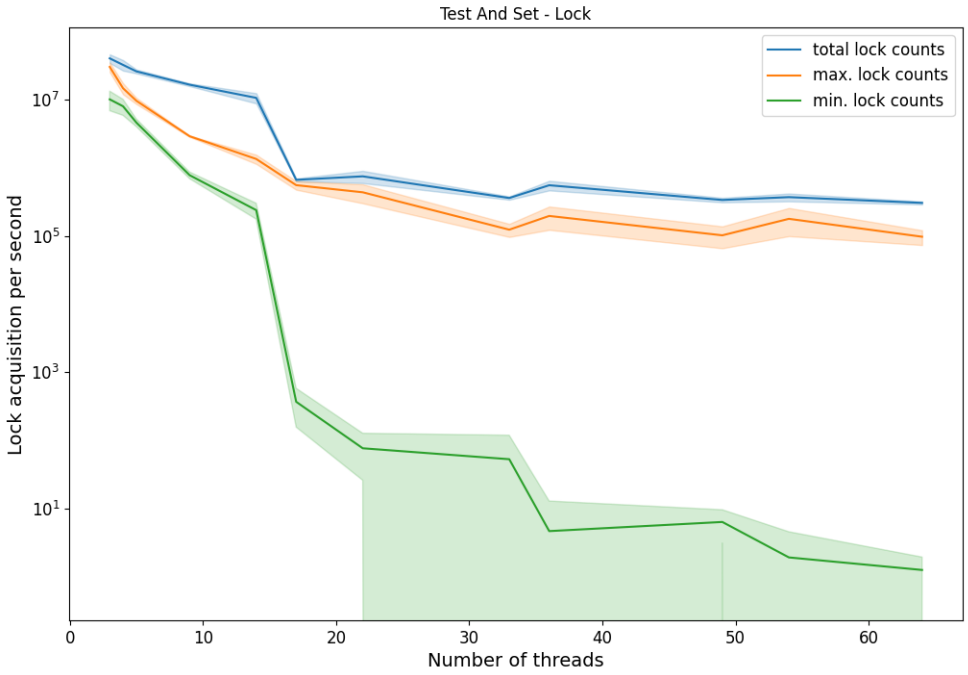

Figure 9: TAS lock throughput and fairness

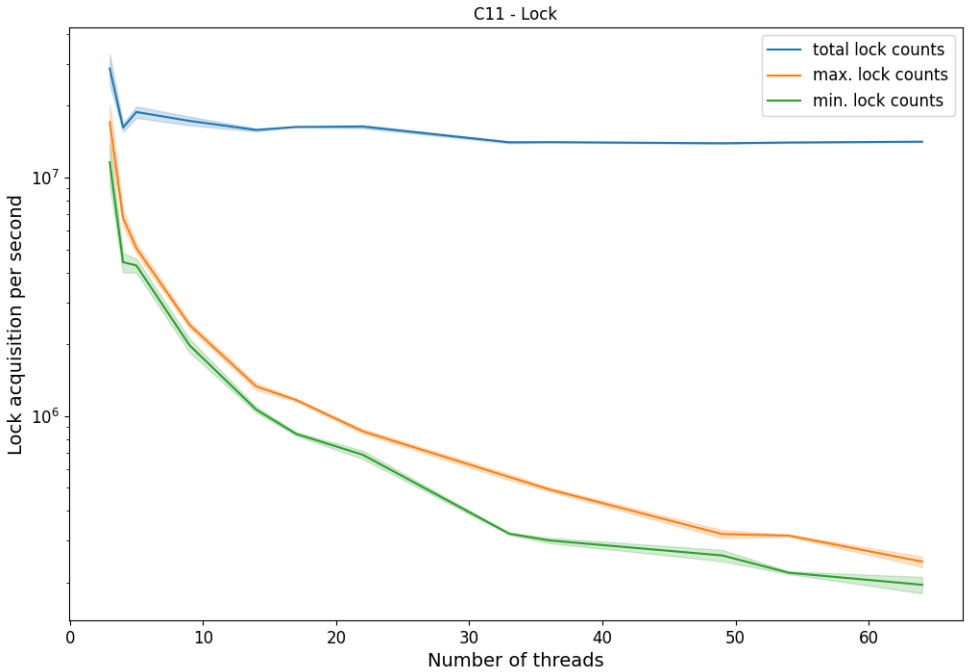

Figure 10: Native C11 lock throughput and fairness

Comparing all the locks in terms of fairness can be done by calculating a fairness quotient from $\frac{minCount}{maxCount}$, representing the fraction of the ”slowest” threads acquisitions of the ”fastest” threads acquisitions. A perfectly fair lock has a fairness quotient of 1, meaning the ”slowest” and the ”fastest” thread entered the CS exactly equally often in the timeframe. We can compare the fairness quotient of all investigated locks in Figure 11. Here the natural strength of the ticket-based locks like Bakery (both versions), Boulangerie and the Filter lock are visible. All four locks perform better in terms of fairness than the baseline C11 lock. As stated above, this is probably due to the C11 lock sacrificing perfect fairness for higher throughput and lower latency. The

confidence intervals are extremely small, supporting this property.
The OpenMP lock is still quite fair compared to some of the worse performing locks with a fairness quotient of 10% for lower thread configurations and around 0.1% for larger thread configurations. Hence the fairness is dependant on the number of threads but scales relatively good, while the ticket-based locks are very fair, no matter what thread configuration. Compared to the TATAS lock, it is much fairer. As will become visible later, this is the only big difference to the TATAS implementation.
The Tournament tree is very unfair, with the fairness quotient decreasing exponentially when the number of threads is increased up to 16 threads. After that, it scales better with the fairness decreasing not as fast. At 32 threads, the ”slowest” thread will acquire the lock only 0.00005 times as often as the ”fastest” thread.
The TAS and TATAS are unsurprisingly very unfair, since the lock acquisition is not FIFO based. TATAS is more unfair that TAS, with its fairness quotient being 10 to 100 times smaller than TAS fairness quotient.

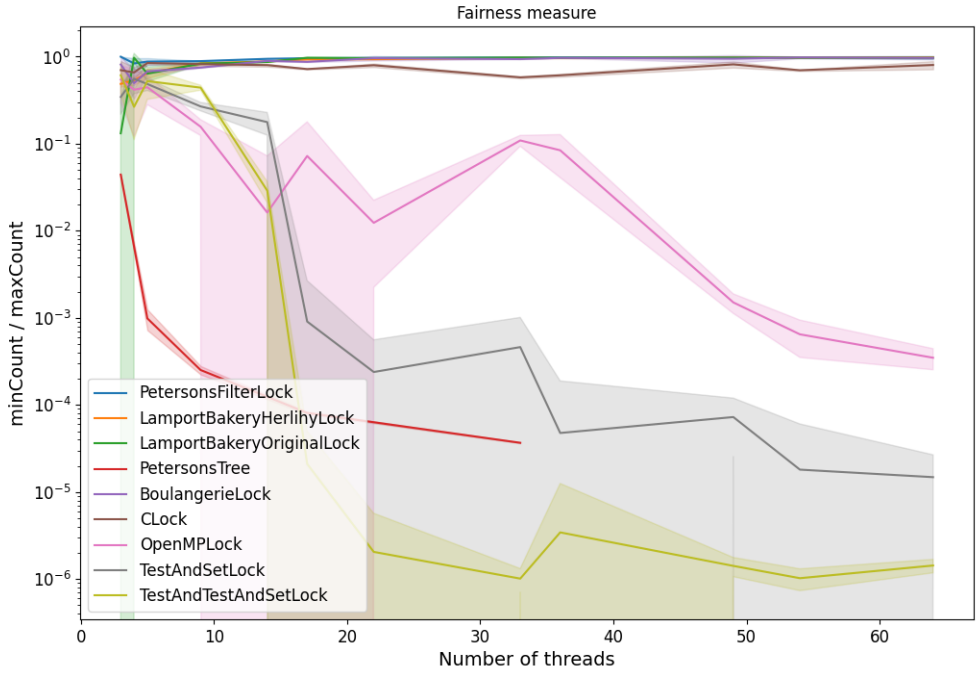

Figure 11: Fairness benchmark for all locks

3.3 Latency

The third benchmark refers to latency. Here, the time from trying to acquire the lock to actually receiving the lock and unlocking it is measured. Each thread will acquire the lock 1000 times. Before the thread enters the doorway section, a time measurement is started to measure the interval up until the lock is released. Again as in the throughput benchmark no work is performed inside the critical section. The maximum acquisition is stored per thread over all 1000 acquisitions. After the parallel section, the total elapsed time is summed up over all threads and averaged and the maximum acquisition time is reduced via the OpenMP *max* operator. This benchmark is conducted 30 times per thread configuration to allow the calculation of confidence intervals.

```
#pragma omp parallel private(id) shared(lock) reduction(+:total) reduction(max:maximum)
{
    id = omp_get_thread_num();

    for(int i = 0; i < OPERATIONS; i++)
    {
        auto start = std::chrono::high_resolution_clock::now(); //start time measurement
        lock.lock(id);
        lock.unlock(id);
        auto finish = std::chrono::high_resolution_clock::now(); //stop time measurement

        double elapsed = std::chrono::duration_cast<std::chrono::duration<double> >(finish - start)
            .count();
        maximum = std::max(maximum, elapsed);
        total = total + elapsed;
    }
    //average the latency over operations
    total = total / OPERATIONS;
}
```

The mean latency averaged over 30 benchmarks per thread configuration can be found in Figure 12. The baseline C11 lock performs best again, showing the smallest latency for any thread configuration. The OpenMP, TAS and TATAS locks are just slightly worse. At 64 threads, the C11 lock takes on average $8 \cdot 10^{-6}s$ to acquire, TATAS and OpenMP are almost as fast with $2 \cdot 10^{-5}s$ per lock.
The Bakery based locks perform equally well. Naturally, the average latency increases with an increasing number of threads trying to acquire one lock at the same time. The slope is comparable to the slope of the OpenMP lock in its steepness, scaling better than the Filter lock. From the latency and throughput benchmark combined, only a slight improvement of Herlihy's version compared to the original version of Lamport can be found. The Boulangerie version is not really better but it has to be mentioned, that the improvement only really shows when thread contention is low, which is not benchmarked here.
The Tournament lock tree has a slightly higher latency compared to the Bakery based locks, finishing with around $2 \cdot 10^{-4}s$ per lock at a 64 thread configuration.
Worse by one order of magnitude scales the Filter lock. Especially with many threads trying to acquire the lock at the same time, the latency of the Filter lock gets really high.

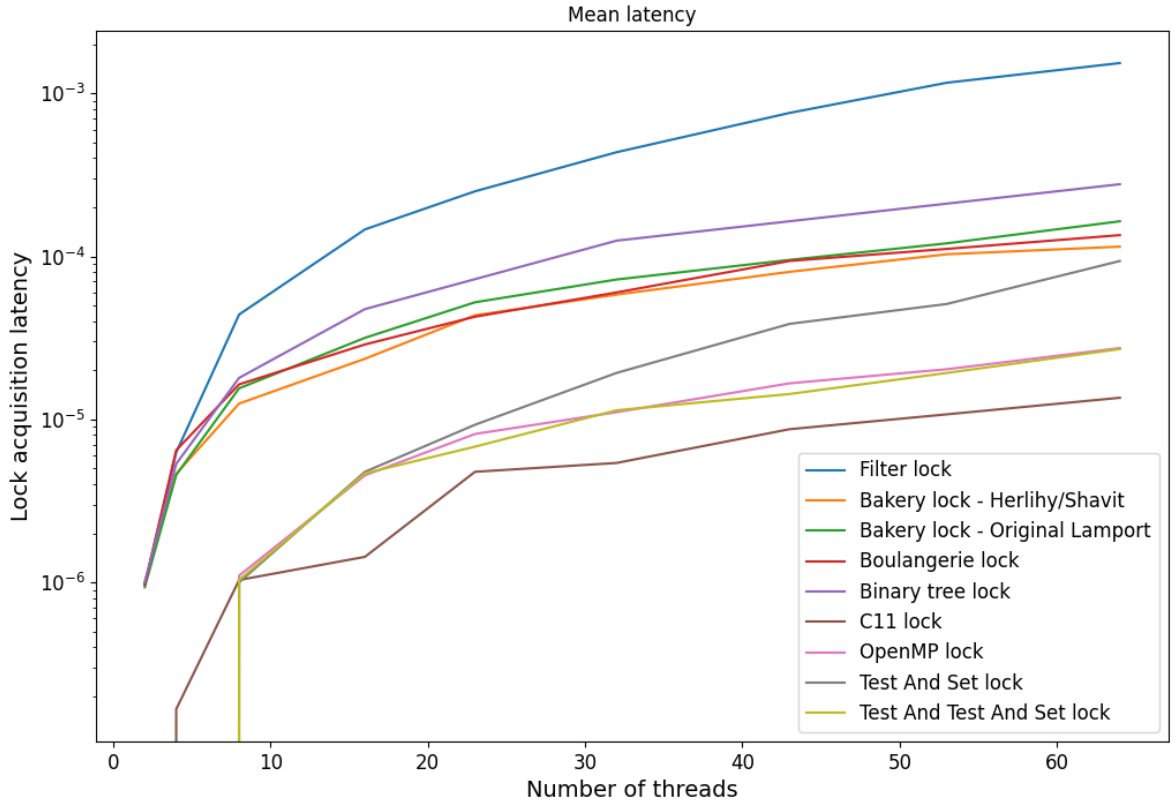

Figure 12: Mean latency over all locks at various thread configurations

The maximum latency shows whether some threads had to wait extremely long to acquire the lock. This is an important measure, as it levels the alleged performance of some locks. As it is visible in Figure 13, the TAS, TATAS and OpenMP lock score the worst for most of the thread configurations. Although all three locks report a very high throughput, they are not fair. With an average latency scoring only slightly worse than the baseline C11 lock, this comes at the cost of some threads being held up extremely long in the lock acquisition process.

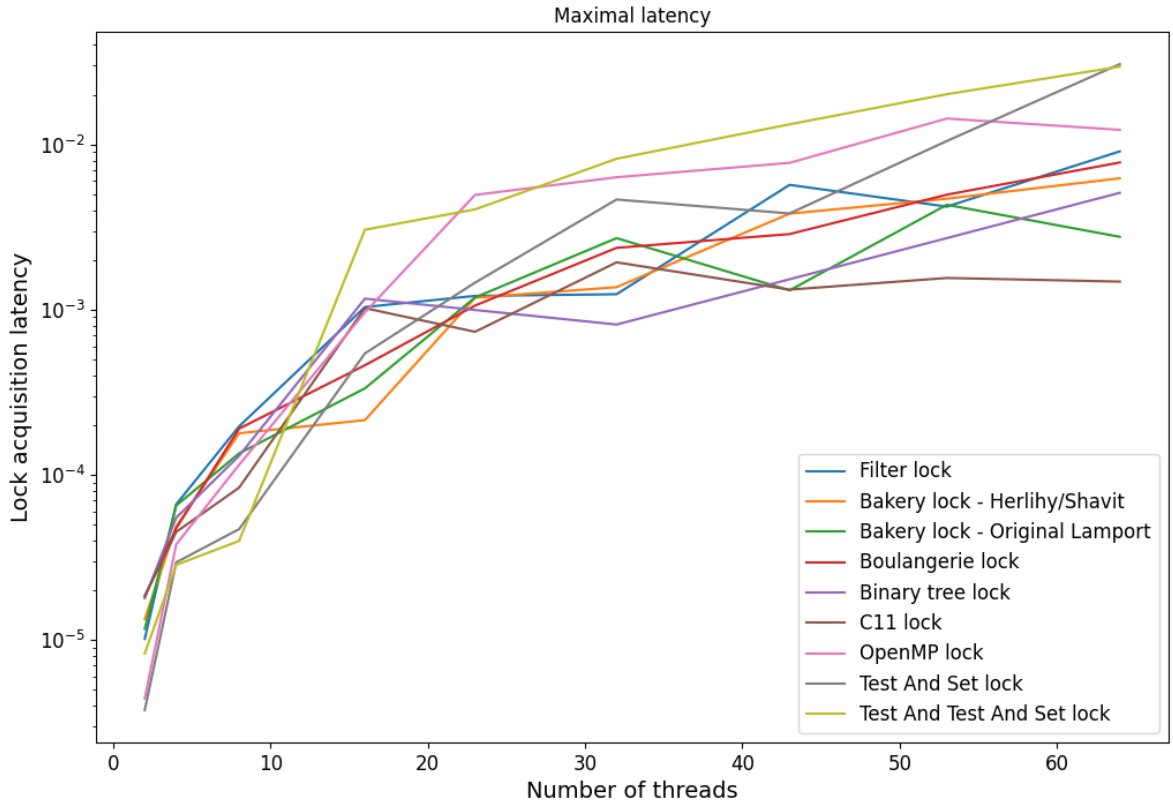

Figure 13: Max latency over all locks at various thread configurations

4 Summary

Although there are slight differences in the benchmark results, the OpenMP and TATAS lock obtained surprisingly similar benchmarking results. As can be seen in Figure 5, 12 and 13 the two locks perform so similar, that it can be assumed a Test-And-Test-And-Set approach was used in the OpenMP lock implementation. This makes for a lightweight, high throughput low latency lock at the cost of being highly unfair. It must be mentioned that the OpenMP lock is not as unfair in the min-max count distribution as the TATAS lock, probably due to a more fair TATAS approach in the source code. Another surprising observation is that the C11 lock performs significantly better than the OpenMP lock using more threads ($<$16). Furthermore, the self implemented locks cannot compete at all with the standard C11 lock - meaning the C11 lock must be highly optimized.

Overall, the ticket based locks like the Bakery (both versions), Boulangerie or Filter lock perform worse in terms of latency and throughput compared to C11, OpenMP or TAS/TATAS. But considering the fairness aspect, these lock implementations perform slightly better than C11 and orders of magnitude better than the other mentioned. It thus depends on the use case, whether such a lock makes sense. If the performance forfeit is less costly than the benefit of a perfectly fair lock, these implementations make perfectly sense.
It is no wonder of course, that the native C11 lock proves to be the best overall pick regarding all benchmarked properties, as it is clearly the most fine-tuned implementation.

The Filter lock proved to be quite inefficient in general. The The Tournament lock did not perform better and was very restrictive in its thread configurations due to the Power-Of-2 condition necessary for its construction.

TAS and TATAS are simple and low overhead locks that can have a low latency and high throughput. Yet they are very unfair. Especially the TAS has high cache coherence traffic as all threads spin on the same memory location. On the other hand, these locks are fault tolerant, as a stalled thread not having the lock is not preventing other threads from obtaining the lock.

A HPC system specification

The benchmark was conducted on the nebula HPC cluster of TU Wiens Parallel Computing institute.
The compute nodes are Dell PowerEdge R6415 server racks interconnected via Gigabit LAN, which provides up to 10 Gb/s. In the following, you can see the system specification of the compute nodes:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	46 bits physical, 48 bits virtual
CPU(s):	16
On-line CPU(s) list:	0-15
Thread(s) per core:	1
Core(s) per socket:	16
Socket(s):	1
NUMA node(s):	4
Vendor ID:	AuthenticAMD
CPU family:	23
Model:	1
Model name:	AMD EPYC 7351P 16-Core Processor
Stepping:	2
CPU MHz:	1196.540
CPU max MHz:	2400,0000
CPU min MHz:	1200,0000
BogoMIPS:	4790.88
Virtualization:	AMD-V
L1d cache:	32K
L1i cache:	64K
L2 cache:	512K
L3 cache:	8192K

B Output Correctness Benchmark

The following output of the correctness benchmark was created on the nebula cluster using the following slurm script. All scripts can also be found in the submission folder.

```
#!/bin/bash
#SBATCH -p q_student
#SBATCH --job-name=amp_proj_heisler       # Job name
#SBATCH -N 1
#SBATCH -c 16
#SBATCH --cpu-freq=High
#SBATCH --time=00:04:50              # Wall time limit (days-hrs:min:sec)
#SBATCH --output=test_job.out

export OMP_NUM_THREADS=16
procs=( 2 4 8 16 )

for proc in "${procs[@]"
do
    ./bm_correctness $proc
done
```

Here, only the respective output for 16 OpenMP threads is displayed.

```
Correctnes benchmark started! Measure if lock works
Maximum number of threads for this system: 16
Benchmark started! Number of threads: 16

Testing correctness of Petersons Filter lock...
-----------------------------------------------------------------

Testing correctness of Lamport Bakery Herlihy lock...
-----------------------------------------------------------------

Testing correctness of Lamport Bakery original lock...
-----------------------------------------------------------------

Testing correctness of Petersons Tournament tree lock...
-----------------------------------------------------------------

Testing correctness of Boulangerie lock...
-----------------------------------------------------------------

Testing correctness of C11 mutex lock...
```

```
---------------------------------------------------------------------

Testing correctness of OpenMP lock...
---------------------------------------------------------------------

Testing correctness of test and set lock...
---------------------------------------------------------------------

Testing correctness of test and test and set lock...
---------------------------------------------------------------------
```