TU WIEN Informatics

# INSTITUTE OF COMPUTER ENGINEERING

## AMP - Project 6

Member:

*Christian* Gollmann, 01435044
*Alexander* Leitner, 01525882

Submission: June 9, 2020

# Contents

# 1 Introduction

Mutual exclusion ensures that at no point it is possible for more than one thread to be in the Critical Section. In order to practically realize mutual exclusion, locks are used. Locks can be divided into two groups based on their behaviour while waiting for lock acquisition. Let's say a thread is currently waiting for the lock to become free. When it is known that the lock will be used for a long time, it would be a waste of computing resources if the thread were just to repeatedly test for the lock to become free. In this case it is far more efficient to suspend the thread and let the operating system's scheduler pick up another thread to do work and come back to the original thread at a later point. This is called "blocking". Switching between threads is expensive in terms of cycles and therefore only an option if the lock is expected to be occupied for quite some time. If on the other hand the lock delay is expected to be short, it is best to make the waiting thread repeatedly test if the lock is free. This method is known as "spinning".

In the following, it is our aim to implement and present the following four different spin locks and examine them on their performance

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

These locks share another property, being that they are so called queue locks. What is so special about those kind of locks we will derive later. First let's ask the following question:

Why are there differences in performance and how is performance for a lock defined anyway?

The latter can be answered in a rather short manner. Imagine we have n processors competing for a lock. All of them call the lock-function, only one actually makes it into the Critical Section. What are the remaining threads now doing? They repeatedly test if the lock is free. This testing of course comes with a cost in computation and clock cycles. So it can happen that the threads stall each other and despite the lock being free again, no thread enters the Cricitcal Section because they are all busy spinning. The goal therefore must be to reduce contention over resources to a minimum. The better this is achieved, the better the lock's performance, meaning more repetitions of the Critical Section can be done in the same amount of time.

For this report we used two sources:

- "The Art of Multiprocessor Programming" from Herlihy and Shavit
- The lecture notes of this course

So most of what follows, except the experimental data and its interpretation, can be found in those two sources.

## 1.1 Setup for the benchmarks

In the following we want to describe how we set up our performance test. We benchmark two things:

- how long does it take to execute a Critical Section a certain number of times when n threads are contending for the lock
- how fair is this contention

Our base setups for the parallel region and the Critical Section look like this:

Listing 1: Parallel region

```
auto start = std::chrono::high_resolution_clock::now();
auto end = std::chrono::high_resolution_clock::now();
#pragma omp parallel private(tid) shared(counter,start,end)
{
    tid = omp_get_thread_num();
    #pragma omp barrier
    if (tid==0)
        start = std::chrono::high_resolution_clock::now();
    while(counter < iterations)
    {
        mylock.lock();
        counter = CS(counter,iterations,turns,tid);
        mylock.unlock();
    }
    if (tid==0)
        end = std::chrono::high_resolution_clock::now();
}
double runtime = std::chrono::duration_cast<std::chrono::microseconds> \
        (end - start).count();
```

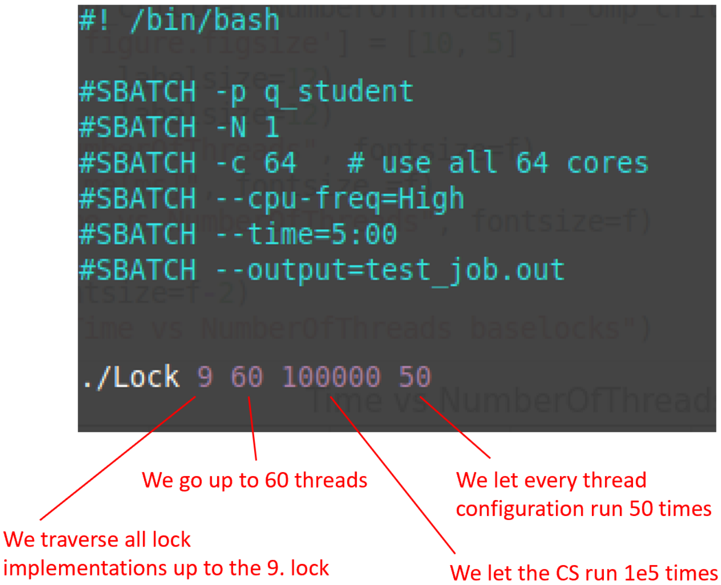
Listing 2: Critical Section

```
long int CS(long int counter, int iterations, double *turns,int tid)
{
    double k = 0;
    try {
            if(counter < iterations)
            {
                counter++;
                turns[tid*8]++;
            }
        }
        catch (int j) {
            std::cout << "Some error occured while in CS" << std::endl;
        }
    return counter;
}
```

Our intention was to keep the CS short, so we can be sure the runtime really only origins from acquiring the lock. In our tests, we set ***iterations*** to 1e5, let each lock execute the parallel region 50 times and take then the average for our comparison. We also count how many times a specific thread entered the CS and evaluate the lock's fairness by calculating the standard deviation between the threads. The counting is done in the ***turns*** array. In order to avoid false sharing, we pad the array ( ***turns*** is of type double, so ***tid*8*** corresponds to 64byte, the size of a cache line).

We also wanted to keep everything as simple as possible for the test. Therefore all the code we used is collected in one single file.
Before running our file, we compiled it at the nebula headnode using the compiler provided there.

On the computenode we used the submit file described in figure 1.



```
#! /bin/bash

#SBATCH -p q_student
#SBATCH -N 1
#SBATCH -c 64   # use all 64 cores
#SBATCH --cpu-freq=High
#SBATCH --time=5:00
#SBATCH --output=test_job.out


./Lock 9 60 100000 50
```

We go up to 60 threads

We let every thread configuration run 50 times

We traverse all lock implementations up to the 9. lock

We let the CS run 1e5 times

Figure 1: submit file

### 1.2 Base lock performances

In order to get a better feeling for how different implementations affect the performance, we will look at the TAS and TTAS lock. Those are no queue locks, but we will later take them as base comparison for our own implementations.

Listing 3: TAS lock

```
class TAS_lock {
    private:
    std::atomic<bool> state;

    public:
    TAS_lock(){
        state = false;
    }
    void lock(){
        while (state.exchange(true))
        {}
    }
    void unlock(){
        state.exchange(false);
    }
};
```

Listing 4: TTAS lock

```
class TTAS_lock {
    private:
    std::atomic<bool> state;

    public:
    TTAS_lock(){
        state = false;
    }
    void lock(){
        while (true) {
            while (state)
            {}
            if (!state.exchange(true))
                return;
        }
    }
    void unlock(){
        state.exchange(false);
    }
};
```

Looking at the implementation of the TAS and TTAS lock, one can convince himself that both of them guarantee mutual exclusion. However, we tested both on the computenode and arrived at the results presented in figure 2. While both locks look pretty similar, they have a major difference. In the TAS lock's while-loop in the lock-method, threads spin all on the same variable. That means whenever a thread calls ***state.exchange(true)***, it alters the ***state*** variable, what leads to an invalidation of all the other threads' cached copies. So they all have to get the new value from memory, what results in heavy traffic on the memory bus which therefore leads to delays in execution.

The TTAS lock on the other hand just repeatedly tests the ***state*** variable without resetting it, so as long as no threads alter ***state***, cached copies of the variable stay valid and it therefore comes to less traffic on the memory bus. Only when the lock appears to be free, all threads contend for the lock.
This subtle difference in the locks' architectures leads to the difference in runtime we see in figure 2.

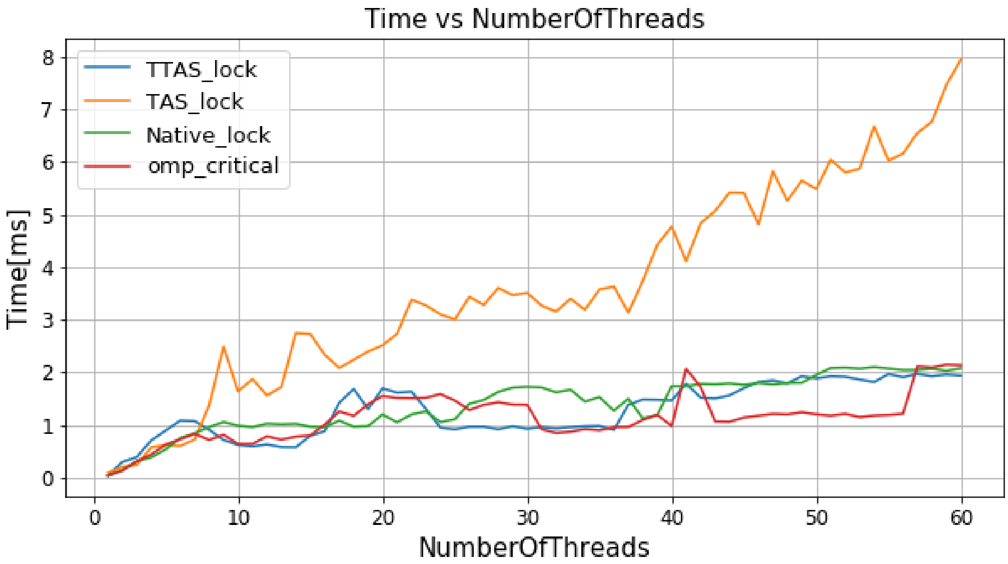


Figure 2: performance of the base locks

So it appears that the TTAS Lock is actually a pretty good implementation of a Lock. We have to admit we expected it to follow the TAS' curve in a weaker form.
We also compared with the omp Native lock and the omp critical section implementation which look like this

Listing 5: omp Native lock

```
omp_lock_t mylock;
omp_init_lock(&mylock);
...
...
...
#pragma omp parallel private(tid) shared(counter)
{
    tid = omp_get_thread_num();
    while(counter < iterations)
    {
        omp_set_lock(&mylock);
        counter = CS(counter,iterations,turns,tid);
        omp_unset_lock(&mylock);
    }
}
```

Listing 6: omp critical Section

```
#pragma omp parallel private(tid) shared(counter)
{
    tid = omp_get_thread_num();
    while(counter < iterations)
    {
        # pragma omp critical
        {
            counter = CS(counter,iterations,turns,tid);
        }
    }
}
```

They also show some scaling with respect to the number of threads even though this scaling is not as strong as with the TAS lock.

We also look at the locks' fairness. We count how often a thread has executed the CS and then calculate the standard deviation $s$ of all locks according to (1).

$$s = \sqrt{\frac{\sum_i (\overline{x} - x_i)^2}{i}} \tag{1}$$

For the base locks we arrive at the following result, presented in figure 3. It is obvious that those base locks are not fair (the theoretical biggest Standard Deviation represents execution when only one thread enters the CS all the time).

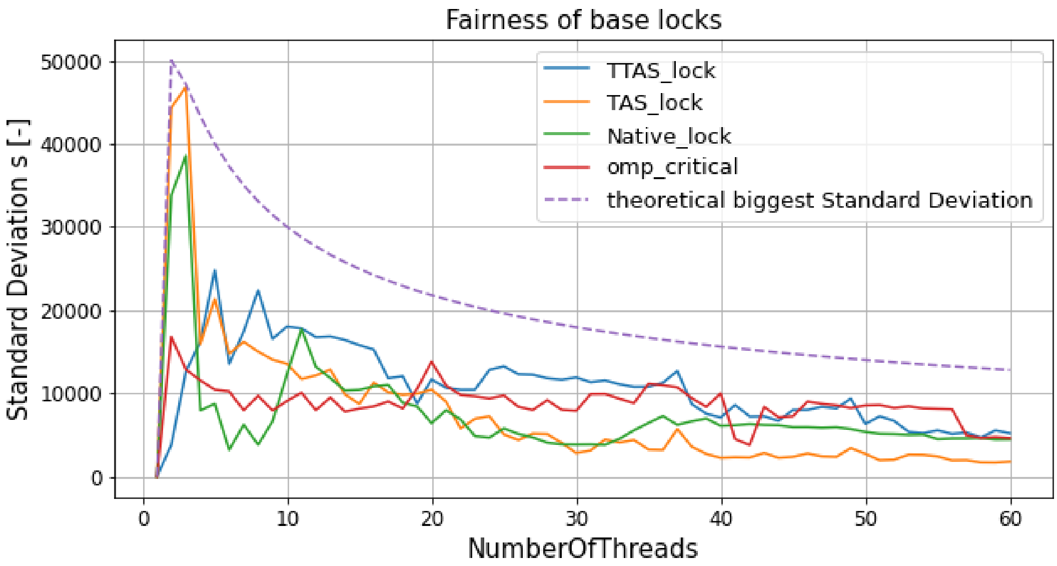


Figure 3: fairness of the base locks

## 2 Ticket Lock

The Ticket lock is a very simple but fair implementation of a lock. There are basically two global variables in it; ***ticket*** and ***served***. When a thread tries to acquire the lock, it draws a number by atomically fetching and increasing ***ticket***. This atomic operation actually marks the linearization point of the lock. The thread then waits until it's its turn by comparing its drawn number with ***served***. Whenever a thread releases the lock, it increases ***served*** by one.

Listing 7: Ticket lock

```cpp
class Ticket_lock
{
    private:
    std::atomic<int> ticket;
    volatile int served;

    public:
    Ticket_lock() {
        ticket = 0;
        served = 0;
    }
    void lock()
    {
        int next = ticket.fetch_add(1);
        while (served < next)
        {}
    }
    void unlock()
    {
        served++;
    }
};
```

The Ticket lock has the following properties:

1. It is space efficient with O(1). That means no matter how many threads try to use the lock, the memory needed to implement it stays the same.

2. The ***ticket*** and ***served*** variables grow without a limit. Of course, if we suppose 64bit, there is a lot of room, but still this is a problem that at least shouldn't be neglected.

3. The Ticket lock is not fault-tolerant. That means if a any thread that has drawn a number crashes or is delayed, it stalls all other threads.

4. The lock is starvation-free, that means every thread trying to get the lock succeeds at some point. Starvation freedom also implies Deadlock freedom.

5. The lock is fair, meaning after some initialization phase, every thread gets the lock as often as the other threads.

We tested the performance of the lock and compare it to the TTAS implementation.

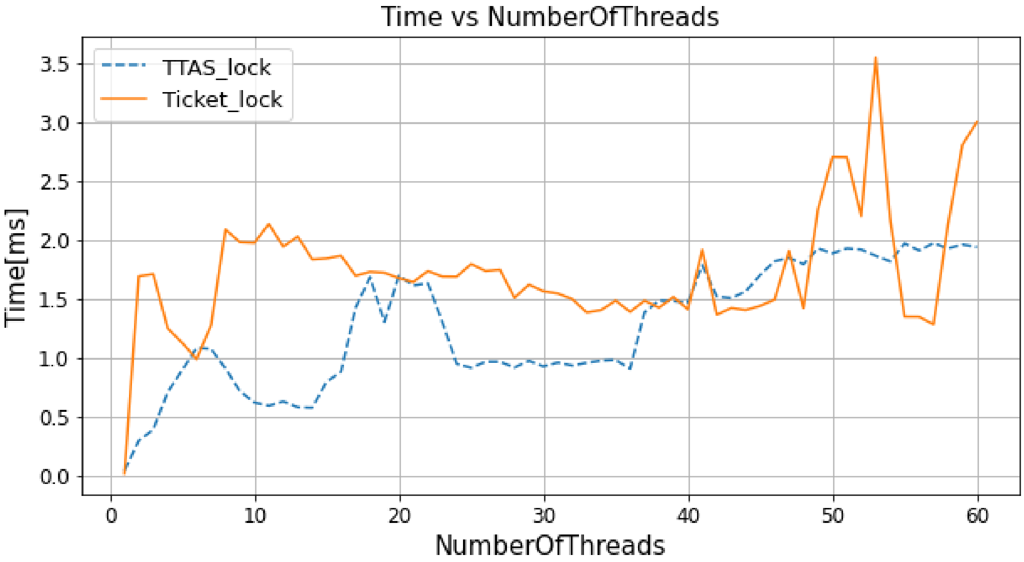


Figure 4: Runtime of the Ticket lock compared with the TTAS lock

One can see that the runtimes for a higher number of threads are on average the same. This is not surprising, since the Ticket Lock basically operates in the same way as the TTAS Lock. They both spin on variables in cache which get only invalidated when a thread calls ***unlock()***. We were though surprised that the Ticket Lock seems to perform worse than TTAS in the beginning.

The fairness of the Ticket lock is rather strong, just take in mind the curve for the theoretically biggest standard deviation from figure 3. The few peaks probably stem from an initialization phase at the beginning of every execution.

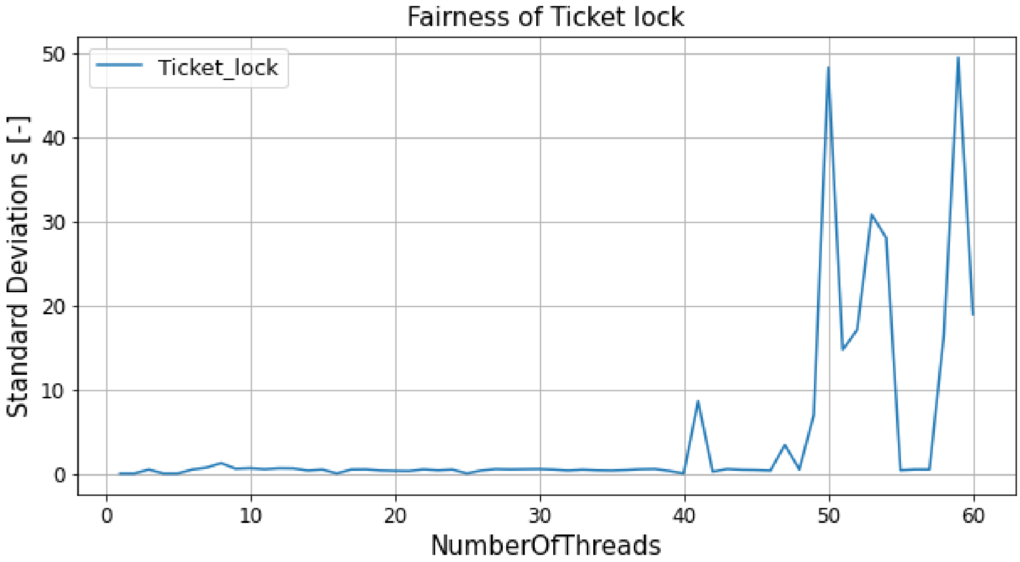


Figure 5: fairness of the Ticket Lock

## 3 Array Lock

The Ticket Lock does a good job in trying to minimize traffic on the memory bus. But still, everytime ***served*** gets incremented, every thread experiences an invalidation of its cache line. They all have to fetch the new value from memory which results in high contention on the memory bus. The Array Lock tries to solve this problem.

Similarly to the Ticket Lock, a thread that wants to participate in the lock contention draws a number by atomically increasing the ***tail*** variable and stores it in ***mySlot***, which is thread local, meaning every thread has its own copy of ***mySlot*** with a different value. ***mySlot*** is the index in a boolean array ***flags***, which' values are initially set to false. A waiting thread now spins as long on ***flags[mySlot]*** until its predecessor sets its ***flags***-field to true after it is done with the CS. The difference to the Ticket lock now is that the thread's action in the ***unlock()*** only invalidates the cache line of a single other thread, unlike in the Ticket Lock, where an update of ***served*** invalidated the cacheline of all other threads.

Listing 8: Array lock

```
class Array_lock
{
    private:
    volatile bool* flag;
    std::atomic<int> tail;
    int numthreads;

    public:
    Array_lock(int n) : flag(new volatile bool[n])
    {
        for (int i = 0; i < n; ++i)
            flag[i] = false;
        flag[0] = true;
        tail = 0;
        numthreads = n;
    }
    void lock(int* mySlot) {
        *mySlot = tail.fetch_add(1)%numthreads;
        while (!flag[*mySlot])
        {}
    }
    void unlock(int* mySlot) {
        flag[*mySlot] = false;
        flag[(*mySlot+1)%numthreads] = true;
    }
};
```

There is yet another way to increase the lock's efficiency. As it is right now, all fields of ***flags*** lie close to each other in memory. This means they will also lie close to each other in a cache line. Because of that, false sharing will occur. That means a thread spinning on ***flags[x]*** might experience an invalidation of its cache line because ***flags[x-1]*** got updated for example. In order to bypass this issue, one introduces padding of the ***flags*** array. ***flags*** therefore now has more fields than there

[0] C++ only allows to create static thread local variables inside classes. We therefore decided to pass on ***mySlot*** when calling ***lock()*** and ***unlock()*** as it was suggested in the lecture notes.

are threads but the fields lie far enough from each other so that they end up on different cache lines.

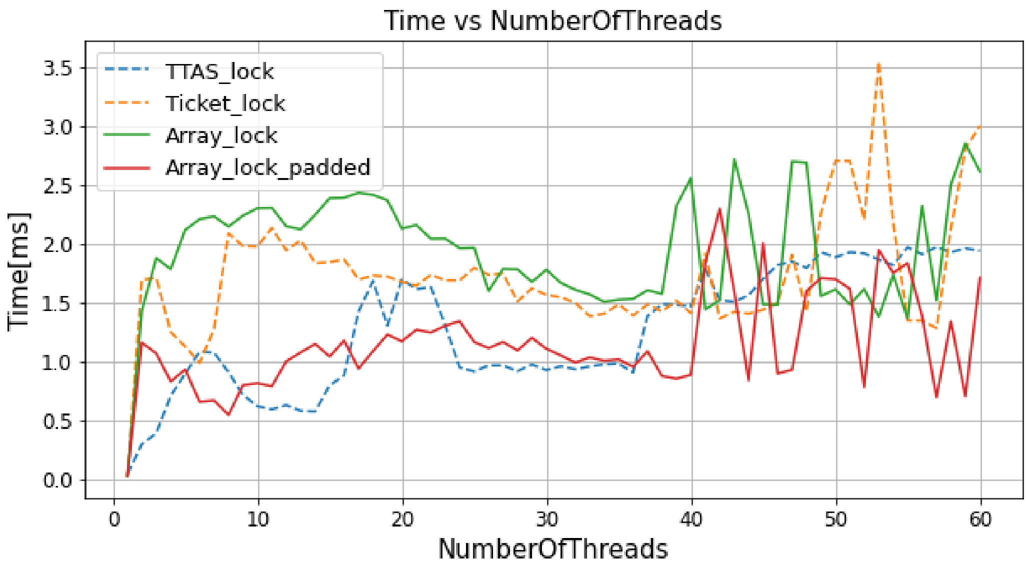


Figure 6: Runtime of the Array Lock compared with other locks

We see that the Array Lock ranks around the Ticket Lock. But as expected, the padded version does a little better than the not padded one.

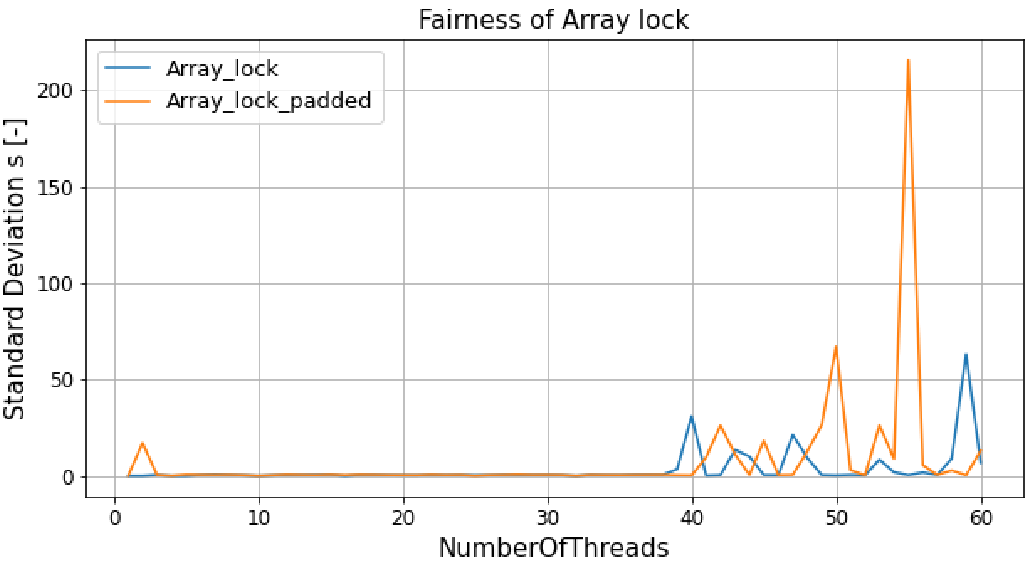


Figure 7: Fairness of the Array Lock

An issue that the Array Lock has though, is its space inefficiency. It allocates O(n) space per lock, even more in the padded version. In addition to that, it has to be known beforehand how many threads will participate in the lock contention. If for some reason more threads than anticipated take part, it could happen that two threads end up in the CS simultaneously.

# 4 CLH Lock

The CLH Lock is an improvement of the Array Lock and tackles two of its issues - its space inefficiency and the fact that for the Array Lock, the number of threads participating in the lock contention must be known beforehand.

Simply said, the CLH Lock works like this:

1. A thread that wants to acquire the lock picks an object QNode.
2. This QNode object has two fields: a pointer ***pred*** that points to another QNode object and a boolean field ***locked***.
3. If a thread wants to acquire the lock, it sets its QNode's ***locked*** field to true.
4. The CLH lock has an atomic field of type pointer ***tail*** that points to the QNode object of the thread that most recently enqueued for the lock. After setting ***locked*** to true, a thread atomically puts his QNode to the end of the tail and stores a pointer to its predecessor in its QNode object.
5. While this predecessor has the lock, the thread spins on its predecessor's QNode's ***locked*** field waiting for its predecessor to set it to true.
6. A thread releases the lock by deleting its predecessor node (avoid memory leak) and setting its own ***locked*** field to true.

Listing 9: CLH Lock

```
class QNode
{
    public:
    std::atomic<bool> locked;
    QNode* pred;

    QNode() {
        locked = false;
        pred = nullptr;
    }
};

class CLH_lock
{
    private:
    std::atomic<QNode*> tail;

    public:
    CLH_lock() {
        tail = new QNode;
    }
    void lock(QNode** pointerToNode) {
        QNode* node = new QNode;
        *pointerToNode = node;
        node->locked = true;
        node->pred = std::atomic_exchange(&tail,node);
        while (node->pred->locked)
        {}
    }
    void unlock(QNode* node) {
        delete node->pred;
        node->locked = false;
    }
};
```

In the CLH Lock, it is not necessary to know how many threads will contend for the lock before it is initialized. The space requirement therefore is O(L) with L being the number of threads trying to get the lock right now in the moment, as opposed to the Array lock, whose memory grows with O(n). There is also no unbounded growing of any variable, as opposed to the Ticket and Array Lock.

Even though the CLH Lock is space efficient, we did not expect any speedup compared to the padded Array Lock because in both locks, the threads spin on separate locations. In figure 8 we draw a comparison between them.

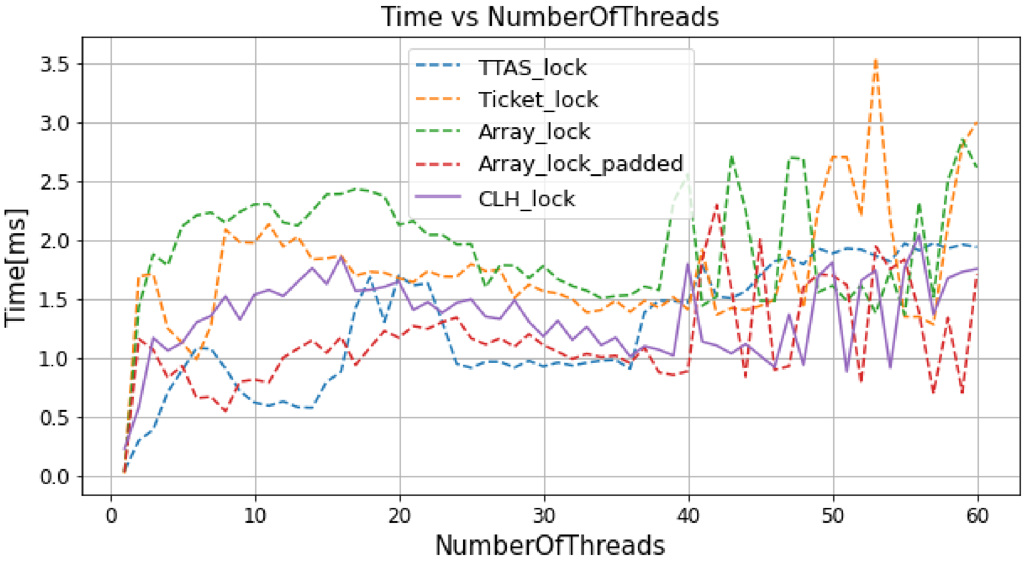


Figure 8: Runtime of the CLH Lock compared with other locks

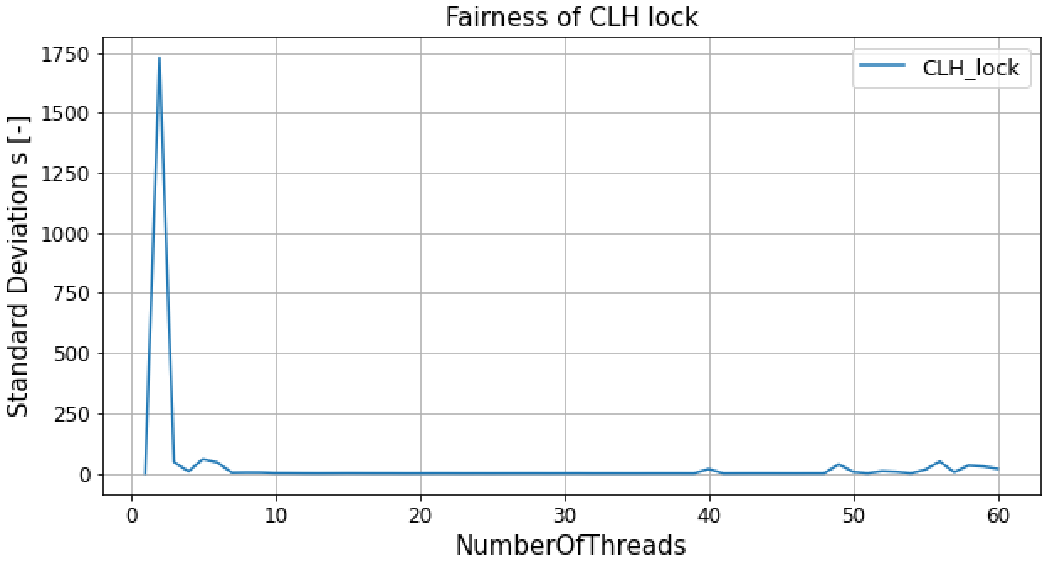


Figure 9: Fairness of the CLH Lock

A disadvantage the CLH Lock could have is that it performs not so well on cache-less NUMA architectures. In figure 10, which we got from the book ”The Art of Multiprocessor Programming”, we compare two different processor architectures. On the right, one can see a symmetric multiprocessing unit (SMP). In this architecture, every processor has its own cache and they are all linked to memory over a common interconnect called the bus. If too many processors want to read/write memory at once, the bus gets overloaded and threads get delayed. Though, this architecture is widely used today because of its simplicity to build.
On the left we see a nonuniform memory access unit (NUMA). Here every processor has its own memory and processors are linked over a network. That means that it takes longer for a processor to access another processors’ memory than it takes for it to access its own.
In the CLH Lock, threads spin on their predecessor’s QNode object. If this object is far away in a NUMA architecture, accessing it may take some time if there is no cache available where it could reside.

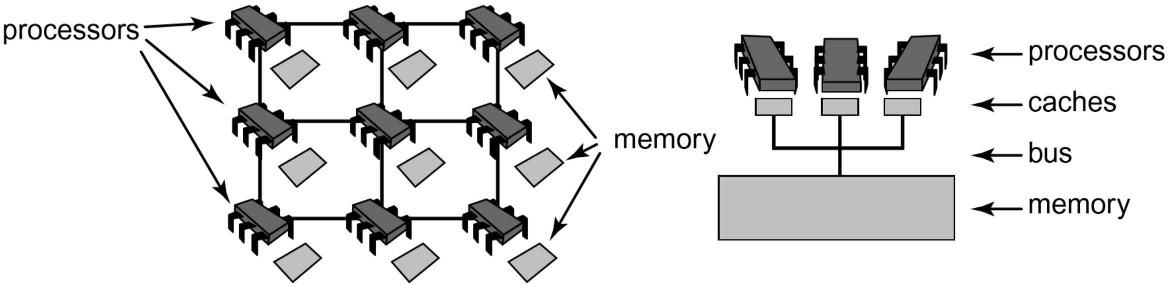


Figure 10: NUMA (left) and SMP (right) architecture

## 5 MCS Lock

The MCS Lock is a further development of the CLH Lock and combines the advantages of Array and CLH Lock. To put it shortly, MCS works very similar to CLH with the difference that in the MCS implementation, a thread does not spin on its predecessors Node but its predecessor updates the thread's Node's ***locked*** field, similar to the Array Lock.

In more detail, the MCS Lock works like this:

1. A thread that wants to acquire the lock picks up its thread local Node object.
2. It then enqueues its Node to the end of ***tail***, an atomic field of type pointer within the lock class and gets a pointer to its predecessor in return.
3. The thread then sets the ***locked*** field of its Node to true, sets the ***next*** pointer of its predecessor pointing to itself and spins on its own Node's ***locked*** field waiting for its predecessor to set it to false.
4. To release the lock, a thread sets its successor's Node's ***locked*** field to false and discards its ***next*** pointer.
5. If by the time a thread wants to release the lock, there is no successor yet, the thread checks whether there really is no other thread trying to acquire the lock, or if the contending thread is just slow. In the latter case, the thread will wait for its successor to finish the necessary steps. From this behaviour arises a property that we haven't seen so far: the ***unlock()*** is no longer wait free. That means a thread wanting to release the lock, can be blocked indefinitely by another thread.

Listing 10: MCS Lock

```
class Node
{
    private:
     std::atomic<bool> locked;
     std::atomic<Node*> next;

    public:
    Node() {
        next = nullptr;
        locked = false;
    }
    void setLocked(bool val) {
        this->locked = val;
    }
    void setNext(Node* val) {
        this->next = val;
    }
    bool getLocked() {
        return this->locked;
    }
    Node* getNext() {
        return this->next;
    }
};

class MCS_lock
```

```cpp
{
    public:
    std::atomic<Node*> tail;

    MCS_lock() {
        tail = nullptr;
    }
    void lock(Node* node) {
        Node* my = node;
        Node* pred = tail.exchange(my, std::memory_order_acquire);
        if (pred != nullptr) {
            my->setLocked(true);
            pred->setNext(my);
            while (my->getLocked())
            {}
        }
    }
    void unlock(Node* node) {
        Node* my = node;
        if (my->getNext() == nullptr) {
            Node* p = my;
            if (tail.compare_exchange_strong(p, nullptr, std::memory_order_release,
            std::memory_order_relaxed)) {
                return;
            }
            while (my->getNext() == nullptr)
            {}
        }
        my->getNext()->setLocked(false);
        my->setNext(nullptr);
    }
};
```

We now see why the MCS Lock can be considered as a combination of Array and CLH Lock and has space complexity O(L) as well. This implementation should work better on cache-less NUMA architectures than CLH because each thread spins on a location "close" to it.

However, one disadvantage of the MCS is that realeasing the lock is no longer wait free, but this should not lead to an overall delay anyway. We expected it to have a performance similar to the CLH Lock which can be seen if figure 11.

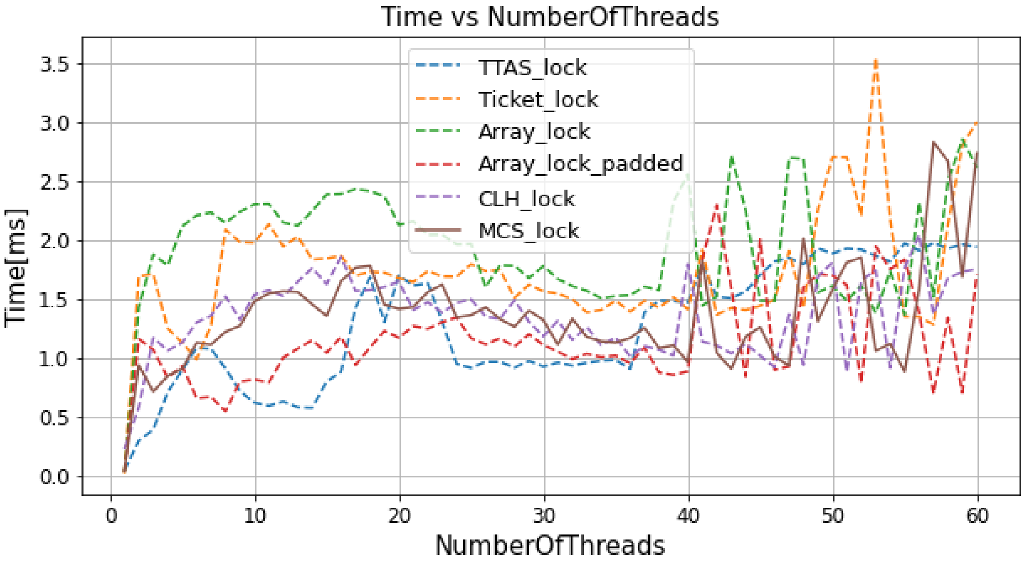


Figure 11: Runtime of the MCS Lock compared with other locks

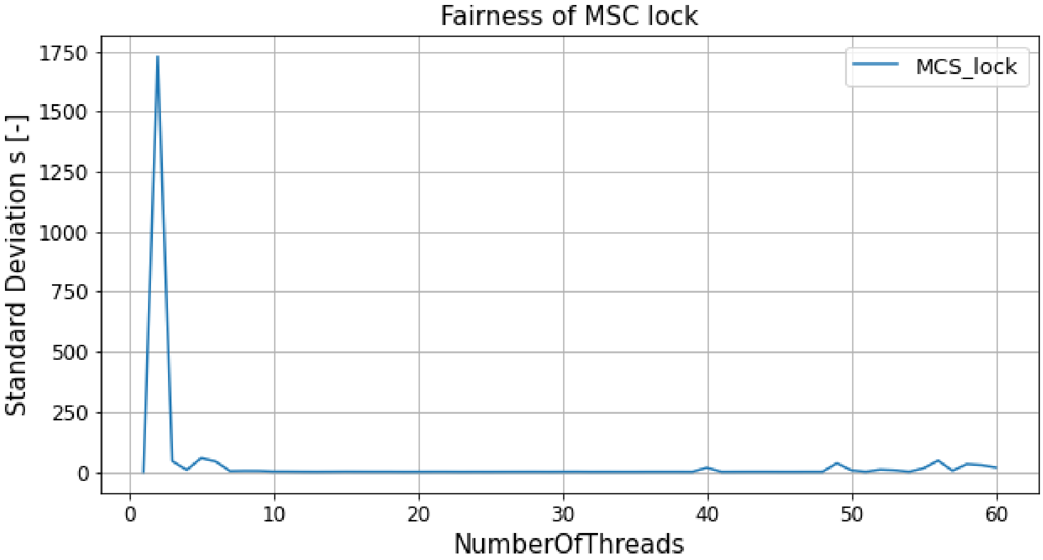


Figure 12: Fairness of the MCS Lock

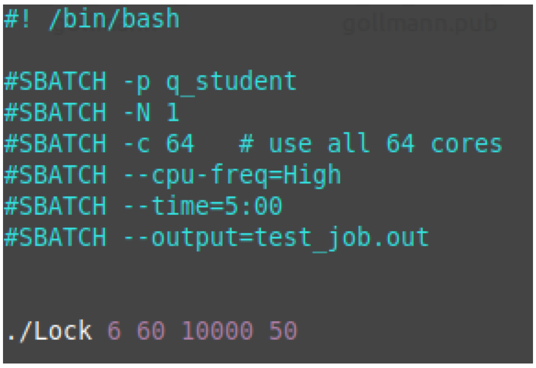
## 6 Benchmark with longer CS

In our first benchmark, we kept the Critical Section very short to make sure that all the time spent came from the lock acquisitions. But now we also want to test with a longer CS. We modified it in the following way:

Listing 11: CS long

```cpp
long int CS(long int counter, int iterations, double *turns,int tid)
{
    double k = 0;
    try {
            if(counter < iterations)
            {
                counter++;
                turns[tid*8]++;
                for (int i = 0; i < 120000;i++)
                {
                    k = log(i);
                }
            }
        }
        catch (int j) {
            std::cout << "Some error occured while in CS" << std::endl;
        }
return counter;
}
```

Since the execution took much longer now, we had to change the setup of our test to the following, depicted in figure 13:



```bash
#! /bin/bash

#SBATCH -p q_student
#SBATCH -N 1
#SBATCH -c 64    # use all 64 cores
#SBATCH --cpu-freq=High
#SBATCH --time=5:00
#SBATCH --output=test_job.out


./Lock 6 60 10000 50
```

Figure 13: submit file for longer CS

The results in figure 15 were quite surprising. To compare it better, we also added the results from the first benchmark in figure 14.

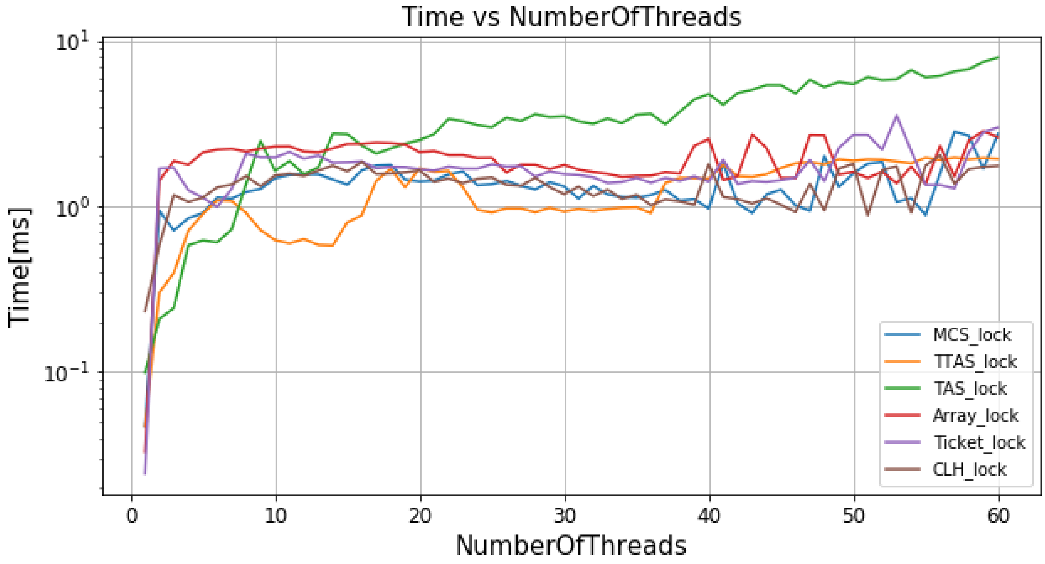


Figure 14: lock performance from first benchmark

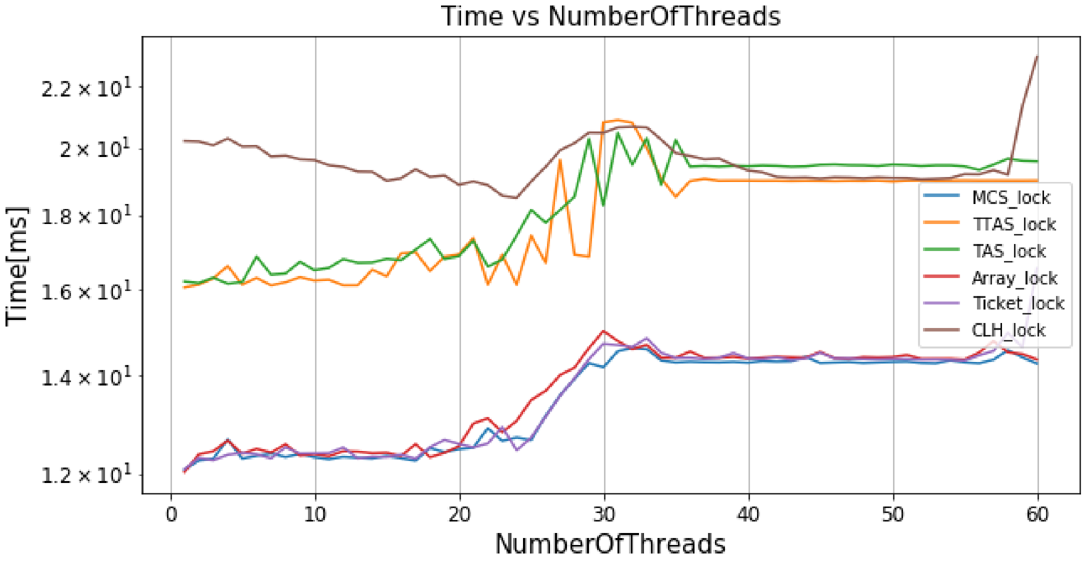


Figure 15: lock performance from second benchmark

There are some obvious differences. In the setup of the first benchmark, shown in figure 14, we see that the runtimes all ramp up during the first few thread variations. There it is visible that more threads contending for the lock cause a delay. This effect, however, seems to even out after the 10 threads mark, and except for the TAS Lock, all locks seem to scale roughly with O(1) after that point. That probably means that the computenode's interconnect is ”strong enough” to handle the traffic caused by the threads. It could also be that during the first few threads, we operate inside the L1 cache, and after this cache gets too small for more threads, we have to move to L2 cache which is big enough to hold the data for up to 60 threads.

In the setup of the second benchmark, shown in figure 15, we do not see any ramp up phase. That probably is because the runtime now is governed by the length of the CS (keep in mind that

the runtimes are now higher by a factor 10 and and the benchmarks only ran for 1e4 iterations). What's striking now is that the Array, MCS and Ticket Lock all have nearly identical performance, whereas the same seems to hold for TAS and TTAS Lock. Especially that TAS and TTAS Lock are so similar, is an indicator that now, time spent for execution is governed by the length of the CS.
What really surprised us, is that the CLH Lock now differs so much from the MCS Lock and we dont't really have an explanation for that.
What all locks seem to have in common though, is that their runtimes ramp up between the 20 and 30 threads mark. Maybe this comes from a change to L1 to L2 cache as well. But this would not correspond to the observation we made regarding the cash in the first benchmark. So we are not completely sure what causes this behaviour.

# 7 Conclusion

We have to admit, we expected the locks' runtimes all to follow a kind of exponential curve. The fact that only the TAS Lock seemed to do this surprised us, but was still an indicator that our setup was done in the right way. That the runtimes scale more or less with O(1) probably origins from the fact that the computenode's bus can handle all the traffic caused by the threads.

What we learn from that is that in reality, many factors play into the overall performance when it comes to locking. So if someone has to design a time crucial implementation with locks, it's best to do some tests to find out which lock suits their needs the best.