Advanced Multiprocessor Programming

Project 1

Enrico Coluccia, 12005483
Aron Wussler, 11717702

1 Introduction

Mutual exclusion is perhaps the most prevalent form of coordination in multiprocessor programming. In this report we analyze some mutual exclusion algorithms that work by reading and writing shared memory, called registers lock. In particular we present the results obtained on a x86_64 AMD Epyc 64-thread Cluster provided by the Technical University of Vienna (Nebula) related to the following register locks:

- Filter Lock
- Tournament tree of 2-thread Peterson locks
- Herlihy-Shavit Bakery Lock
- Lamport Bakery Lock
- Boulangerie Lock

The results are compared with three baseline provided by the test-and-set and, test-and-test-and-set and OpenMP locks: an analysis of performance and fairness is proposed.

2 Implementation

2.1 Setup

We created a class `BaseLock` that encapsulates the behaviour of all the locks with a simple interface, in order to write a framework that could test all locks re-using the same structure and giving a constant overhead.

```
class BaseLock {
  public:
    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual const char *get_name() = 0;
};
```

All the implemented locks, extensions of the `BaseLock` class, are procedurally tested from the `runLock` function. Here, we instantiate the parallelization: we define a parallel part using the `#pragma omp parallel` instruction, then sync all threads with a barrier. We then start measuring the time in each thread, since it might not be synchronised. The last thread of the measured loops (i.e. when `counter == lock_iterations`) will write the total time required for the loop to a shared variable, to then compute the lock performance.

Furthermore in the critical section we count how many locks has each thread using the `thread_counter` variable, to then compute the fairness.

In the CS we also execute a constant load defined by the variable `cs_iterations` that will run an empty `while` loop for the required amount of cycles.

All these shared variables are then logged to a CSV outside of the lock execution since file write might be a non constant bottleneck.

```
int run_lock (
        std::ofstream& dataCollector,
        BaseLock *lock,
        int iteration,
        int nthreads,
        int lock_iterations,
        int cs_iterations
) {
        int counter = 0, duration;
        int thread_counter[THREAD_MAX] = {0};
        omp_set_num_threads(nthreads);

        #pragma omp parallel shared(counter, thread_counter, duration,
        ↪   lock_iterations)
        {
                int tid = omp_get_thread_num();
                std::chrono::time_point<std::chrono::high_resolution_clock>
                ↪   start, end;

                #pragma omp barrier
                start = std::chrono::high_resolution_clock::now();

                while(counter <= lock_iterations) {
                        lock->lock();
                        counter++;

                        if (counter == lock_iterations) {
                                end =
                                ↪   std::chrono::high_resolution_clock::now();
                                duration =
                                ↪   std::chrono::duration_cast<std::chrono::microseconds>
                                ↪   (end - start).count();
                        }

                        if (counter >= lock_iterations) {
                                lock->unlock();
                                break;
                        }

                        for(int cs = 0; cs < cs_iterations; cs++)
                        ↪   {/*wait*/}

                        thread_counter[tid]++;

                        lock->unlock();
                }
        }

        dataCollector
                << "\"" << lock->get_name() << "\","
                << iteration << ","
                << nthreads << ","
                << cs_iterations << ","
                << duration;

        for (int i = 0; i < nthreads; i++) {
                dataCollector << "," << thread_counter[i];
        }

        dataCollector << std::endl << std::flush;

        return EXIT_SUCCESS;
}
```

The main function handles the file output and repeatedly runs this procedure, in order to log the performance results for all the locks with a varying amount of threads.

The data collected in the CSV is finally analysed using R where we extract the statistics and produce the plots in this report.

2.2 Filter Lock

The Filter lock is a generalized version of the Peterson lock, it creates $n-1$ “waiting rooms”, or levels, that a thread must traverse before acquiring the lock. Levels satisfy two important properties:

- At least one thread trying to enter level succeeds.
- If more than one thread is trying to enter level, then at least one is blocked

The Filter lock uses an n-element integer array (called `level[]`) to indicate the highest level that a thread is trying to enter. Each thread must pass $n-1$ levels to enter in the critical section and each level has its own victim to filter out one thread and excluding it from the next level.

The Filter lock satisfies mutual exclusion, deadlock freedom and starvation freedom properties but is not guaranteed to be fair since the threads can overtake others an arbitrary number of times (see exercise 12 from the book). The space complexity is $O(n)$ where n is the number of threads.

```
#include "filterLock.hpp"

FilterLock::FilterLock (int n) {
        level = new int[n];
        victim = new int[n];
        this -> n = n;
}

FilterLock::~FilterLock () {
        delete[] level;
        delete[] victim;
}

void FilterLock::lock() {
        int me = omp_get_thread_num();
        for (int i = 1; i < n; i++) {
                level[me] = i;
                victim[i] = me;
                for( int j = 0; j < n; j++) {
                        while ((j != me) && (level[j] >= i && victim[i] ==
                        ↪  me)) {}
                }
        }
}

void FilterLock::unlock() {
        int me = omp_get_thread_num();
        level[me] = 0;
}
```

2.3 Tournament tree of 2-thread Peterson locks

Since the Peterson Lock works for a 2-thread implementation, another way to generalize it is to arrange a number of 2-thread locks in a binary tree. Suppose n is a power of two. Each thread is assigned a leaf lock which it shares with one other thread. Each lock treats one thread as thread 0 and the other as thread 1.

In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the tree-lock unlocks each of the 2-thread Peterson locks that thread has acquired, from the root back to its leaf.

The Tournament tree of 2-thread Peterson locks satisfies mutual exclusion, deadlock freedom and starvation freedom properties but it is not guaranteed to be fair since a thread can be delayed and overtaken an arbitrary number of times (see exercise 13 from the book). In particular the more are the threads the less the algorithm is fair (see later results). Note that in order to run this algorithm the number of thread must be a power of 2.

```
#include "petersonLock.hpp"

PetersonLock::PetersonLock(int n){
        numOfThreads = n;
        root = new PetersonNode(NULL, numOfThreads);

        std::vector<PetersonNode*> initList;
        initList.push_back(root);

        leaves = createTree(initList);
}

PetersonLock::~PetersonLock(){
}

void PetersonLock::lock() {
        int i = omp_get_thread_num();
        PetersonNode* currentNode = getLeaf(i);

        while (currentNode != NULL) {
                currentNode->lock();
                currentNode = currentNode->parent;
        }
}

void PetersonLock::unlock() {
        int i = omp_get_thread_num();
        PetersonNode* currentNode = getLeaf(i);

        while (currentNode != NULL) {
                currentNode->unlock();
                currentNode = currentNode->parent;
        }
}

// get leaf for a specific thread id, each lock shares two threads.
PetersonNode* PetersonLock::getLeaf(int num) {
        return leaves.at(num / 2);
}

std::vector<PetersonNode*>
↪   PetersonLock::createTree(std::vector<PetersonNode*> nodes) {
        if ((int)nodes.size() == numOfThreads / 2)
                return nodes;

        std::vector<PetersonNode*> currentLeaves;

        for (PetersonNode* node : nodes) {
                node->leftChild = new PetersonNode(node, numOfThreads);
                node->rightChild = new PetersonNode(node, numOfThreads);

                currentLeaves.push_back(node->leftChild);
                currentLeaves.push_back(node->rightChild);
        }
        return createTree(currentLeaves);
}
```

In the peterson node we forced the compiler to ignore optimizations by adding the keyword `volatile` to the victim flag, since this can be changed from the other thread. Given that on Nebula is an x86_64 with Total Store Order (TSO) guarantees we assume that the `volatile` property to avoid re-ordering and enforcing fetching from memory is enough to ensure mutual exclusion, therefore correctness. Without this guarantee, it could happen that:

- The compiler "optimizes the loop away", i.e. the while statement

  `while(flagCheck() && victim == i) { /* wait */ }`

  might be (wrongly) optimised into

  `if(victim == i) { while(flagCheck()) { /* wait */ }}`

  therefore making the lock incorrect.

- Both locks enter the CS if the update is not seen in the right order, but this would require a very weak memory model since thread 0 would have not to see its own update to memory so that `victim != i`.

```
class PetersonNode{
  private:
    bool* flags;
    int volatile victim;
    bool flagCheck();
    int nThreads;

  public:
    PetersonNode(PetersonNode* node, int n);
    ~PetersonNode ();
    void lock();
    void unlock();
    PetersonNode* leftChild;
    PetersonNode* rightChild;
    PetersonNode* parent;

};

PetersonNode::PetersonNode(PetersonNode* node, int n) {
        flags = new bool[n];
        for (int i = 0; i < n; i ++) {
          flags[i] = false;
        }
        parent = node;
        nThreads = n;
}

PetersonNode::~PetersonNode() {
        delete[] flags;
}

void PetersonNode::lock () {
        int i = omp_get_thread_num();
        flags[i] = true;
        victim = i;
        while(flagCheck() && victim == i) { /* wait */ }
}


void PetersonNode::unlock () {
        flags[omp_get_thread_num()] = false;
}

bool PetersonNode::flagCheck() {
        int me = omp_get_thread_num();
        for (int i = 0; i < nThreads; i++) {
                if (flags[i] && (i != me))
                        return true;
        }
        return false;
}
```

2.4 Herlihy-Shavit Bakery Lock

The Herlihy-Shavit Bakery Lock is a modified version of the original Lamport's Bakery Lock implementation. In this lock, `flag[i]` is a Boolean flag indicating whether i wants to enter the critical section, and `label[i]` is an integer that indicates the thread's order when entering the "bakery", for each thread i.

Each time a thread acquires a lock, it generates a new `label[]` in two steps. First, it reads all the other threads' labels in any order. Second, it reads all the other threads' labels one after the other (this can be done in some arbitrary order) and generates a label greater by one than the maximal label it read. We call the code from the raising of the flag (Line 19) to the writing of the new `label[]` (Line 30) the doorway. The lock establishes that thread's order with respect to the other threads trying to acquire the lock. If two threads execute their doorways concurrently, they may read the same maximal label and pick the same new label. To break this symmetry, the algorithm uses a lexicographical ordering on pairs of `label[]` and thread ids at line 36, each thread read the labels one after the other in some arbitrary order until it determines that no thread with a raised flag has a lexicographically smaller label/id pair, then it enters the critical section.

The Herlihy-Shavit Bakery Lock satisfies mutual exclusion, deadlock freedom and starvation freedom properties and it is also fair (first-come-first-serve, see Lemma 2.6.2 from the Book). The space complexity is $O(n)$ where n is the number of threads since we have 2n shared integer variables. One drawback is that the labels grow without bounds.

```
1  #include "bakeryLock.hpp"
2
3  BakeryLock::BakeryLock (int num) {
4          flag = new bool[num];
5          label = new long long[num];
6          for (int i = 0; i < num; i ++) {
7                  flag[i] = false;
8                  label[i] = 0;
9          }
10         n = num;
11 }
12 BakeryLock::~BakeryLock () {
13         delete[] flag;
14         delete[] label;
15 }
16
17 void BakeryLock::lock () {
18         int i = omp_get_thread_num();
19         flag[i] = true;
20         long long max = label[0];
21         for (int j = 1; j < n; j ++) {
22                 if (label[j] > max) {
23                         max = label[j];
24                 }
25         }
26         if (max == LLONG_MAX) {
27                 std::cout << "ERROR: Label Value Overflow" << std::endl;
28                 exit (-1);
29         }
30         label[i] = max + 1;
31
32         for (int j = 0; j < n; j++) {
33                 if (i == j) {
34                         continue;
35                 }
```

```
                while (flag[j] && ((label[j] < label[i]) ||( label[i] ==
                ↪  label[j] && j < i))) {};
        }
}

void BakeryLock::unlock () {
        flag[omp_get_thread_num()] = false;
}
```

2.5 Lamport Bakery Lock

This is the original version of the Lamport's Bakery Lock[1].

Each time a thread acquires a lock, it generates a new `number[]` by reading all the other threads' labels one after the other (this can be done in some arbitrary order) and generates a number greater by one than the maximal number it read. We call the code from the raising of the choosing flag (Line 20) to the writing of the new `number[]` (Line 22) the doorway section.

It establishes that thread's order with respect to the other threads trying to acquire the lock. If two threads execute their doorways concurrently, they may read the same maximal label and pick the same new label. To break this symmetry, the algorithm uses a lexicographical ordering on pairs of `number[]` as already explained in the previous lock, then it enters the critical section.

The Lamport Bakery Lock satisfies mutual exclusion, deadlock freedom and starvation freedom properties and it is also fair, since if thread A executes the doorway before thread B then B is locked out while `number[A]` is greater than 0. As in Herlihy-Shavit Bakery Lock the space complexity is again $O(n)$ with n the number of threads and one drawback is that the labels grow without bounds.

```
1   #include "lamportLock.hpp"
2
3   LamportLock::LamportLock (int num) {
4           choosing = new bool[num];
5           number = new int[num];
6           for (int i = 0; i < num; i ++) {
7                   choosing[i] = false;
8                   number[i] = 0;
9           }
10          n = num;
11  }
12
13  LamportLock::~LamportLock () {
14          delete[] choosing;
15          delete[] number;
```

[1]`http://lamport.azurewebsites.net/pubs/bakery.pdf`

```
}

void LamportLock::lock () {
        int i = omp_get_thread_num();
        choosing[i] = true;
        number[i] = findMax() + 1;
        choosing[i] = false;

        for (int j = 0; j < n; j++) {
                if (j == i)
                        continue;

                while (choosing[j]) {}

                while (number[j] != 0 && (number[i] > number[j] ||
                ↪  (number[i] == number[j] && i > j))) {}
        }
}

void LamportLock::unlock () {
        number[omp_get_thread_num()] = false;
}

int LamportLock::findMax() {
        int m = number[0];
        for (int i=1; i <n; ++i) {
                if (number[i] > m)
                m = number[i];
        }
        return m;
}
```

2.6 Boulangerie Lock

The Boulangerie Lock [1] is a modified version of the Lamport Bakery Lock that applies two improvements:

- Optimizing for low contention: if the thread i has obtained `number[i] = 1` then the only processes j that can ever have a better ticket are ones whose tid is smaller than i. It follows that when `number[i] = 1`, there is no need to perform the spinning section for values $j > i$. To avoid this form of unnecessary blocking, we add the control lines 36-41.
- Taking advantage of inconsistent reads: let's consider two threads i and j, and that we perform read/write operation on safe registers. As long as j is in the bakery it performs no writes on `number[j]` thus, `number[j]` is stable and all reads to it must return the same value. If i reads two different values for `number[j]` while blocking during the spinning, it has proof that j was on the outside at least during one of these reads. Since i is the bakery section at that point, it follows that $i < j$ is true, and i

can stop blocking on j and move on to test the next process. To take advantage of incosistent reads we introduce the lines 53-56.

The Boulangerie Lock satisfies mutual exclusion, deadlock freedom and starvation freedom properties and it is also fair, since if thread A executes the doorway before thread B then B is locked out while `number[A]` is greater than 0. We have the same properties/drawbacks already presented for the Lamport and the Herlihy-Shavit Bakery Lock.

```
1  #include "boulangerieLock.hpp"
2
3  BoulangerieLock::BoulangerieLock (int numThreads) {
4          choosing = new bool[numThreads];
5          number = new int[numThreads];
6          num = new int[numThreads];
7          for (int i = 0; i < numThreads; i ++) {
8                  choosing[i] = false;
9                  number[i] = 0;
10                 num[i] = 0;
11         }
12         n = numThreads;
13 }
14
15 BoulangerieLock::~BoulangerieLock () {
16         delete[] choosing;
17         delete[] number;
18         delete[] num;
19 }
20
21 void BoulangerieLock::lock () {
22         bool tmp_c = false;
23         int prev_n = -1;
24         int tmp_n = -1;
25         int limit = n;
26         int i = omp_get_thread_num();
27
28         choosing[i] = true;
29         for(int j=0; j<n; j++){
30                 num[i]=number[j];
31         }
32         num[i] = findMax() + 1;
33         number[i] = num[i];
34         choosing[i] = false;
35
36         if(number[i]==1){
37                 limit = i;
38         }
39         else{
40                 limit = n;
41         }
42
43         for (int j = 0; j < limit; j++) {
44                 if (j == i)
45                         continue;
46
47                 do{
```

```
                        tmp_c = choosing[j];
                } while(tmp_c);

                tmp_n = -1;

                do{
                        prev_n = tmp_n;
                        tmp_n = number[j];
                } while (tmp_n != 0 && (num[i] > tmp_n || (num[i] == tmp_n
                ↪  && i > j)) && (tmp_n == prev_n || prev_n == -1));
        }
}

void BoulangerieLock::unlock () {
        int i = omp_get_thread_num();
        num[i] = false;
        number[i] = false;
}

int BoulangerieLock::findMax() {
        int m = num[0];
        for (int k=1; k<n; ++k) {
                if (num[k] > m)
                m = num[k];
        }
        return m;
}
```

2.7 Base Locks

For a baseline performance we consider three additional locks:

- Test-and-Set Lock: has a single flag field per lock, the thread acquire lock by changing flag from false to true and it locks on success. To unlock it resets the flag. We know from the theory (see slides) that the performance of this lock is bad, due to the high memory contention. The lock is fault tolerant.
- Test-and-Test-Set Lock: we test and set only if there is a chance of success. It has a better performance than TAS but memory contention and cache deletion problems are still present. The lock is fault tolerant.
- Native OpenMP locks

The baseline locks satisfies mutual exclusion. Furthermore, the TAS and TTAS locks are unfair, not starvation free and space efficient $O(1)$ even for an infinite number of threads. One drawback of these locks is the memory contention since all threads spin on a single memory location.

```cpp
#include "tas.hpp"

TestAndSetLock::TestAndSetLock(int n){
        state=false;
};

TestAndSetLock::~TestAndSetLock(){}

void TestAndSetLock::lock(){
        while(state.exchange(true)){}
};

void TestAndSetLock::unlock(){
        state.exchange(false);
};
```

```cpp
#include "ttas.hpp"

TestAndTestAndSetLock::TestAndTestAndSetLock(int n){
        state=false;
};

TestAndTestAndSetLock::~TestAndTestAndSetLock(){}

void TestAndTestAndSetLock::lock(){
        while(true){
                while(state){}
                if(!state.exchange(true))
                return;
        }
};

void TestAndTestAndSetLock::unlock(){
        state.exchange(false);
};
```

```cpp
#include "nativeOmpLock.hpp"

NativeOmpLock::NativeOmpLock (int n) {
        omp_init_lock(&lck);
}

NativeOmpLock::~NativeOmpLock () {
        omp_destroy_lock(&lck);
}

void NativeOmpLock::lock () {
        omp_set_lock(&lck);
}
```

```

void NativeOmpLock::unlock () {
        omp_unset_lock(&lck);
}
```

3 Fairness

In order to compare the fairness across locks and different sizes we decided to create a scale from 0 to 1 where 0 is a perfectly fair lock and 1 is a lock where only one thread has repeatedly acquired the lock. To avoid creating a biased measure, we reuse the concept of standard deviation from statistics, that we re-scale for our purposes. In particular, we call our measure U for *Unfairness*:

$$U = \frac{\sqrt{n}}{\sum_{i=1}^{n} k_i} \operatorname{sd}(k_i) = \frac{\sqrt{n}}{\sqrt{n-1}} \frac{\sqrt{\sum_{i=1}^{n}(k_i - \bar{k})^2}}{\sum_{i=1}^{n} k_i}$$

where n is the number of threads and k_i is the number of locks acquired by thread i.

Using this definition we inherit the following properties from the standard deviation:

- When all threads acquire the same amount of locks then $k_i = \bar{k} \forall i$, then $\operatorname{sd}(k_i) = 0$ and the result is zero.
- When counting elements, the standard deviation is maximal when only one element is maximal. We can show that the maximum of U is 1, by setting $k_i = 0 \forall i \neq j$ and $k_j \neq 0$:

$$\begin{aligned} U &= \frac{\sqrt{n}}{\sqrt{n-1}} \frac{\sqrt{(n-1)(\bar{k})^2 + (k_j - \bar{k})^2}}{k_j} \\ &= \frac{\sqrt{n}}{\sqrt{n-1}} \frac{\sqrt{(n-1)(k_j/n)^2 + (k_j - k_j/n)^2}}{k_j} \\ &= \frac{1}{k_j} \sqrt{\frac{n}{n-1} \left[(n-1)\frac{k_j^2}{n^2} + \frac{k_j^2}{n^2}(n-1)^2 \right]} \\ &= \frac{1}{k_j} \sqrt{n \left[\frac{k_j^2}{n^2} + \frac{k_j^2}{n^2}(n-1) \right]} \\ &= \frac{1}{k_j} \sqrt{n^2 \frac{k_j^2}{n^2}} = 1. \end{aligned}$$

This measure allows to meaningfully compare the (un)fairness across various implementations and with different sizes in figure 1.

We notice immediately that the TAS, TTAS, and OpenMP locks are highly unfair. In the collected data most of the executions are characterised by one or two threads mostly getting the lock. We can then exclude them from the plot to distinguish and better visualize the custom locks in figure 2. Note that now the y-axis value are very close to zero.

Between these, we have that the Bakery and derivatives behave very similarly, with an extremely low unfairness, while the Peterson Tree and Filter lock present some peaks. In particular we can observe a positive correlation between unfairness and size of the cluster for the Peterson lock.

This is in line with the theory, in fact the TAS and TTAS locks are extremely opportunistic and all threads wait in the same queue, no matter if they have just exited the CS or have been trying to access for a long while. The Filter and Peterson Tree Lock do not provide fairness guarantees, but still implement a queue, that prevent such behaviour from happening, and this is reflected in a slightly uneven thread acquisition distribution. Bakery, Lamport, and Boulangerie are instead FIFO, and this is reflected in their extremely fair behaviour. The fluctuation that is seen at the lower end of the graph is due to integer division: by doing only 1024 iterations some set of threads will get one iteration more if 1024 is not divisible by n, the number of threads.

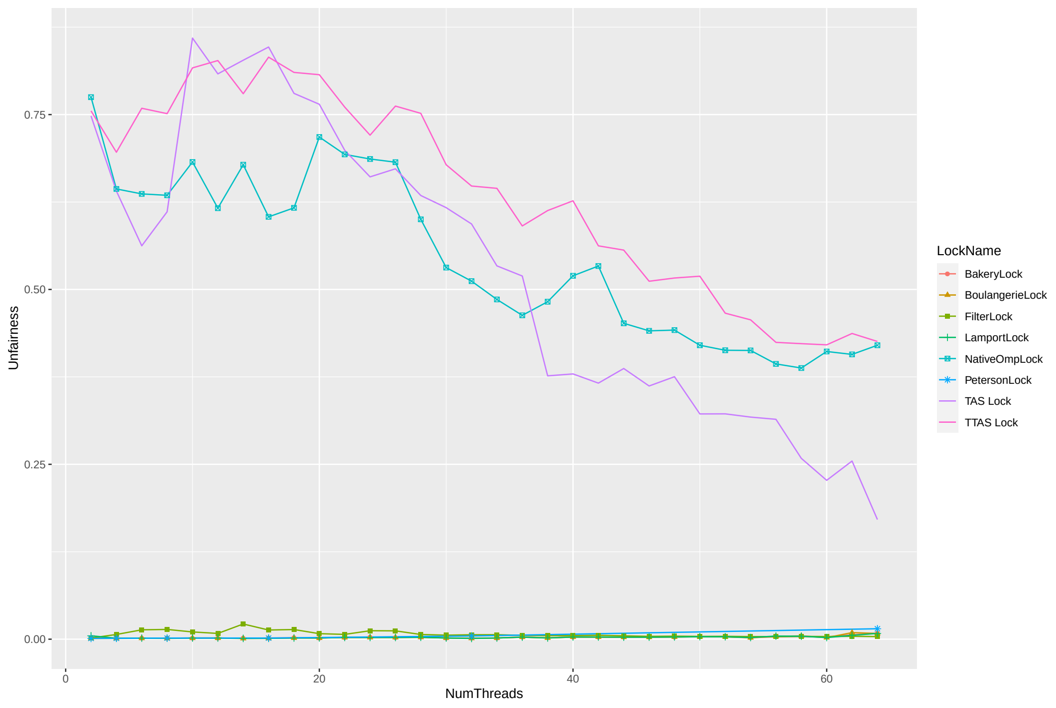

Figure 1: Fairness for all implemented locks

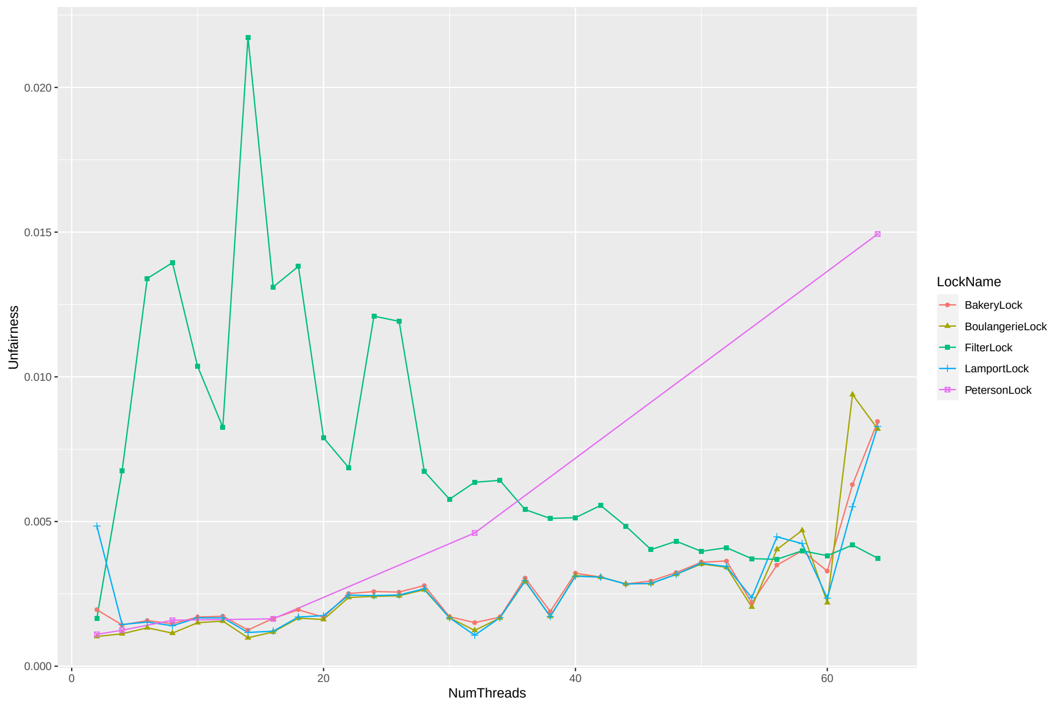

Figure 2: Fairness for custom locks

4 Performance

In order to measure the performance of the locks we measure the time to acquire $m = 1024$ locks in a row from n number of threads. Every acquisition loop is repeated 128 times, to ensure that CPU scheduling is not biasing our data. Finally, we repeat this procedure for increasing sizes in every lock, for a total of slightly over 30 million lock acquisitions.

In order to determine these thresholds we ran the algorithm on various sizes and settled with the first one where no more significant oscillation happened, in order to remain in the 5 minutes limit imposed on the Nebula back-end.

A further parameter we explored is the size of the critical section. In all the iterations this is a fixed parameter, where we perform 2, 16, 128, and 1024 while iterations, to emulate a load that un-trivializes the CS.

Finally, to evaluate the data we collected the time of each acquisition loop in a CSV with the lock type and size. We then computed for each lock a 5%-trimmed mean[2], a robust estimator of the central tendency. Due to CPU scheduling in each execution we obtained some outliers that rendered the data inconsistent. Using the this robust estimator and collecting a lot of data a very stable result across runs is ensured: by running the program again, the plots will vary just slightly, but the main features analysed will persist.

We opted for this measure because it is almost identical to the latency: our preliminary studies revealed that the behaviour of the m iterations corresponds to the single lock acquisition time. This also corresponds to the theory, since the average time to lock is computed using the sum of the individual lock times, that is the total lock time divided by m. Therefore, we obtained two plots scaled by a factor m.

Furthermore, this measure is the inverse of throughput, therefore sums up the performance very concisely.

In figure 3 we can observe the performance for all locks, given CS size 2. In particular the Filter lock will have a consistently much lower performance regardless of the CS size, and will be therefore excluded from the following plots in order to make the data readable.

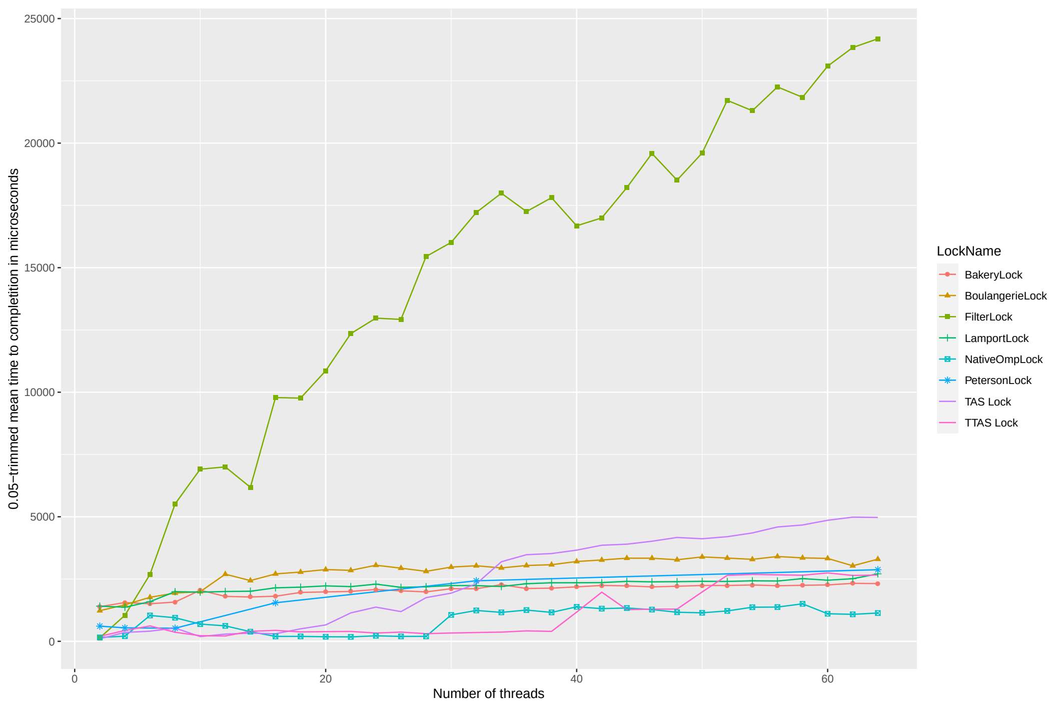

Figure 3: 5%-trimmed mean of exec. time of all locks for CS load 2

[2]α-trimmed mean is defined as a compromise between a median and a mean, where we discard the α portion of the outer values then compute the sample mean on the remaining ones.

The other locks, see figure 4 are instead in a similar range of performance, with the native OpenMP lock and TTAS leading, that maximise performance at the expense of fairness. The TAS lock is performant for lower amount of threads, but high contention will make it the worst lock (except the Filter Lock). Bakery, Lamport and Boulangerie all behave similarly, with a slow increase in latency with increasing number of thread, but in a smooth way. Finally, the Peterson lock offers a very good performance for small locks, i.e. up to size 8, being also quite fair.

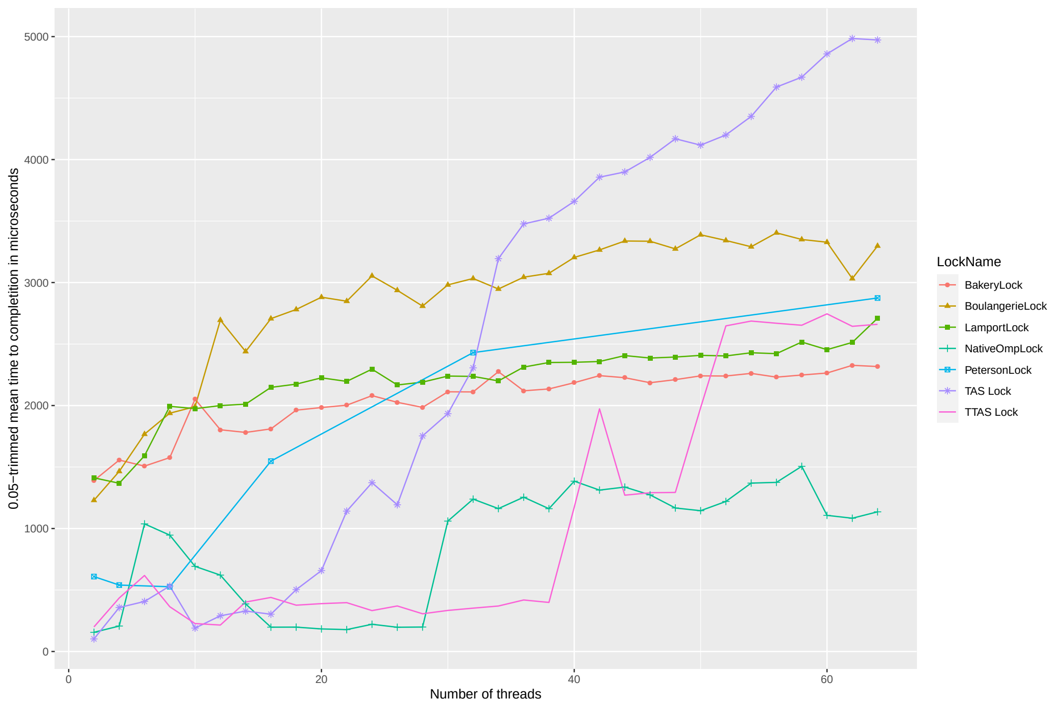

Figure 4: 5%-trimmed mean of exec. time of all but Filter lock for CS load 2

We now introduce some load in the CS with parameters with CS size 128 (figure 5) and 1024 (figure 6). It is apparent how the TAS lock performs poorly for the high contention and that the native OpenMP lock consistently performs best, alongside the TTAS.

The Peterson Lock retains its behaviour, with a smooth log-like increase in latency and the Bakery-based algorithm all perform as a cluster. It is interesting to notice that the Boulangerie, that should be an improvement over the Lamport Lock is almost in all cases performing slightly worse. It can be that on this instruction set (x86_64) the extra local computation is actually worsening the algorithm. The lock was implemented sticking strictly to the paper.

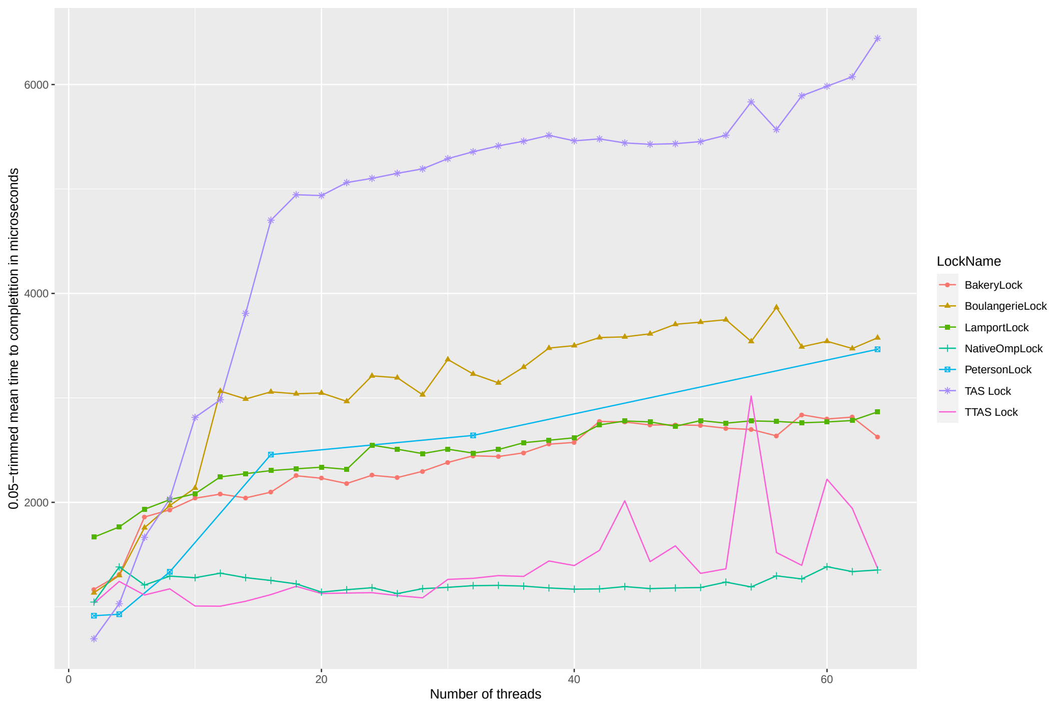

Figure 5: 5%-trimmed mean of exec. time of all but Filter lock for CS load 128

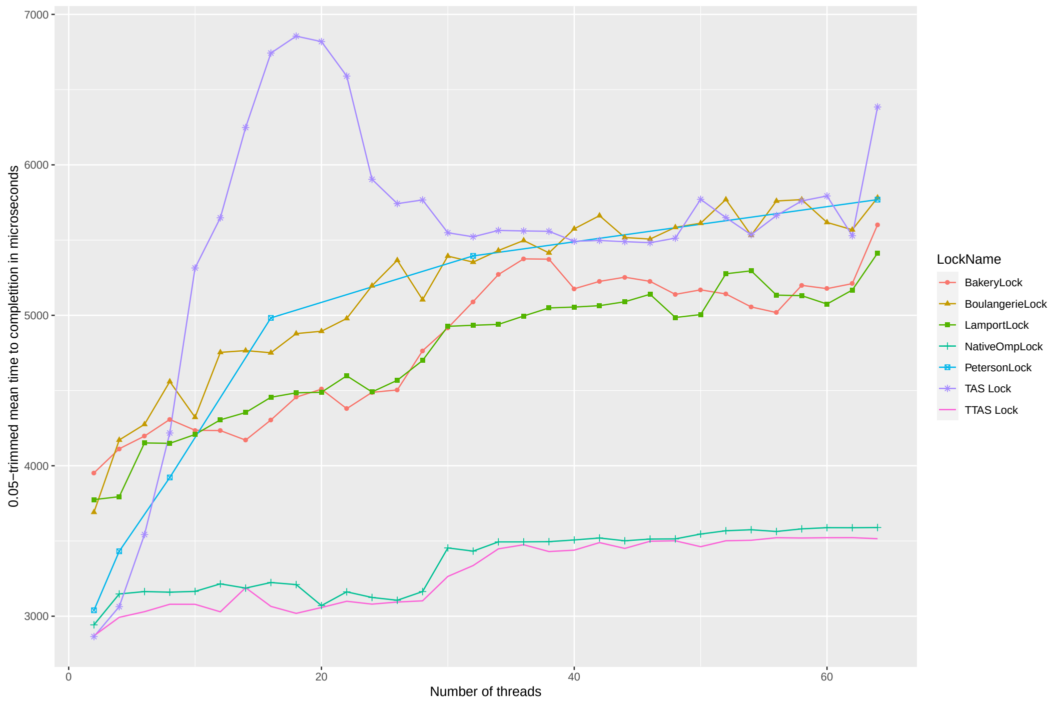

Figure 6: 5%-trimmed mean of exec. time of all but Filter lock for CS load 1024

5 Conclusion

We can easily conclude that our results reflect what we know from the theory. Furthermore, it is clear that the proposed locks has quite different performances, also affected by the number of inner iterations and the critical section size. On the other hand, it is not possible to state which lock is the best if we do not know for which kind of application the lock is used for. This because there is a trade-off between the latency (or the total running time) and the fairness.

References

[1] Yoram Moses and Katia Patkin. “Mutual exclusion as a matter of priority”. In: *Theoretical Computer Science* 751 (2018). Structural Information and Communication Complexity, pp. 46–60. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2016.12.015`. URL: `https://www.sciencedirect.com/science/article/pii/S0304397516307435`.