Advanced Multiprocessor Programming
Practical Project

Project 2:
Bounded-Timestamp Register Locks

Simon Hinterseer, 09925802

Peter Holzner, 01426733

Vienna, 29.06.2020

Contents

1 Task Description

The task was to implement certain lock algorithms. The algorithms, that were implemented were

- Lamport: [1]
- Taubenfeld: [2]
- Jayanti: [3]
- Aravind: [4]

These algorithms were implemented in C++, and tested in terms of correctness and performance.

2 Program structure

2.1 Modules

The program consists of 8 modules.

- main.cpp
  contains the `main()` function that creates instances of lock objects and passes them on to test functions from tests.hpp (+ special versions used only for benchmarking on the cluster, called bm_<lock_name>.cpp)
- locks.hpp
  contains abstract lock classes as well as the Reference_Lock class
- tests.hpp
  contains testing function both for correctness and performance
- toolbox.hpp
  contains some helper functions

as well as the four modules containing the specific lock classes

- lamport.hpp
- taubenfeld.hpp
- aravind.hpp
- jayanti.hpp

2.2 Lock class structure

Two abstract classes were created to define the common interfaces between the implementations of the various lock types.

class Lock

this pure abstract class is a basic the Lock interface.

class DW_Lock

this pure abstract class is an interface for "doorway locks". These are locks where the `lock()` function can be written like this

```
public: void lock(){
      doorway();
      wait();
}
```

This is used so that fairness properties (e.g. FCFS) can be tested from the outside (i.e. without putting logging functions into the lock class).

2.3 Test structure

Tests are located in the file `tests.hpp`.

The test functions serve to test correctness properties of locks as well as measuring the throughput of a lock.

2.3.1 Correctness Properties

The tested properties are

- mutual exclusion (MUTEX)
- first come first served (FCFS)
- least recently used (LRU)

Mutual Exclusion (MUTEX)

The function `test_mutex()` is passed a lock and testing parameters about the number of threads, size of the test, workload and randomness of the workload. It will then start a parallel section using OpenMP and send threads through the lock, while writing an array that logs the acquisition and unlocking events.

It will test mutual exclusion by checking, that no two acquisition events ever occur without an unlock event between them. The function will return the number of violations of the mutex-property.

During the compilation of this document, it became apparent, that there was an oversight considering this function: there was a non-atomic shared counter variable used for this function, which required the use of a helper lock (by using a `std::atomic_flag` object) to make the reading and increasing of the shared counter atomic. This could influence the behavior of the lock and change the result. The other correctness tests, were implemented at a later stage of the project and use the more sophisticated `record_event_log()` function, that deals with this issue in a better way.

Record Event Log

This function is not a correctness, but it is crucial to the other correctness tests, which is why it is described here. The function `record_event_log()` is passed a doorway lock and testing

parameters about the number of threads, size of the test, workload and randomness of the workload.

It will record an array of events, while minimally influencing the behavior of the lock. The only change to purely sending threads through the lock is one shared atomic counter variable.

The `event_log` array will hold the sequence of one events paired with the thread number. Four events are logged:

- doorway begin
- doorway finish
- lock acquisition
- unlock

This array is sufficient to evaluate not only all of the considered correctness properties, but also allows to determine the number of contenders for each acquisition. This will be used in the throughput test, where the average number of contenders is measured.

This function is described in more detail in the annex.

first come first served (FCFS)

The function `test_fcfs()` is passed a lock and testing parameters about the number of threads, size of the test, workload and randomness of the workload.

It will call `record_event_log()`to send threads through the lock and generate an the array of events.

It will then check for each acquisition, whether it was in accordance with the FCFS property, or if one or more threads are trying to acquire the lock, that have a higher priority.

Do be more precise: it will from each acquisition event $(A, acquire)_k$ go back through the event log to find $(A, begin)_k$. For each $(B, finsh)_l$, it comes across, it needs to check, that $(B, acquire)_l \rightarrow (A, acquire)_k$. More formally:

$$\forall (A, acquire)_k \colon \forall (B, finish)_l \colon$$

$$(B, finish)_l \rightarrow (A, begin)_k \Rightarrow (B, acquire)_l \rightarrow (A, acquire)_k$$

The function will return the number of FCFS violations.

least recently used (LRU)

The function `test_lru()` is in many ways similar to `test_fcfs()`. The difference is in the evaluation of the `event_log` array.

Since Aravind does not give a formal definition of the LRU property, these definitions are being used:

Definition: "Contender of an acquisition"

$$B \in C((A, acquire)_k) \Leftrightarrow$$

$$\exists l \in \mathbb{N} \colon (B, finish)_l \rightarrow (A, acquire)_k \; and \; not((B, acquire)_l \rightarrow (A, acquire)_k)$$

in words: B has finished his doorway when A acquires, but has not had an acquisition of his own in between. Therefore B is trying to acquire the lock, when A acquires.

Definition: "LRU"

in a history LRU holds, if and only if

$$\forall (A, acquire)_k : \forall B \in C((A, acquire)_k) : (A, acquire)_{k-1} \rightarrow (B, acquire)_{l-1}$$

with l as in the definition of contender.

In words: for each acquisition, each contender's last acquisition has to be more recent, than the acquiring thread's last acquisition.

The test is then quite straight forward: for each $(A, acquire)_k$ in the event log, the set of contenders is determined. Then it is checked for each contender, if their last acquisition has been more recent than $(A, acquire)_{k-1}$.

The function will return the number of LRU violations.

2.3.2 Throughput

When the function `throughput()` is passed testing parameters like the above tests. It will like the `test_lru()` and `test_fcfs()` functions call `record_event_log()` with the corresponding parameters.

Optionally it will call it with the optional parameter `test_case=2`, which leads to the avoidance of creating the `event_log` array. It has been observed on quad core workstations, that in that case, the throughput is about 25 % higher[1], which is most likely due to not having the additional shared atomic counter object, that needs to be accessed at every event. It has been concluded, that while the reduction in accuracy when measuring the throughput is worth recording the event log, because this allows the calculation of the average number of contenders.

The function will store runtime, throughput and average number of contenders in an array, that was passed by reference.

[1] No statistical investigation of this number has been conducted. The 25 % should be considered a magnitude rather than a precise value.

3 Lock Implementations

3.1 Reference Lock

class Reference_Lock

this uses these statements

- create an instance of the lock with this statement
  `std::atomic_flag lock_stream = ATOMIC_FLAG_INIT;`
- execute `lock()` (i.e. doorway and waiting section) with this statement
  `while ( lock_stream.test_and_set() ) {}`
- execute `unlock()` with this statement
  `lock_stream.clear();`

This lock fulfills mutual exclusion and starvation freedom.

3.2 Lamport's Bakery

The lock algorithm according to Lamport [1]. In this project a variation of the lock has been implemented, that was taken from the lecture notes.

This lock uses two shared arrays (one boolean to indicate threads, that are trying to acquire the lock and one integer that holds ticket numbers) and one integer register for the size of the lock. The basic idea is, that a thread indicates, that it is trying to acquire the lock by setting his boolean element to true. Then it will draw a ticket, which will be an integer greater than all existing tickets. The thread will keep spinning on the test, if it has the lowest tickets of all threads, that are trying to acquire the lock. In case two threads have the same integer as a ticked, the thread id is used to determine the priority. Unlocking is done by resetting the boolean element, that indicates that a thread is trying to acquire the lock. The lock is displayed in Figure 1.

```
1  class Bakery implements Lock {
2    boolean[] flag;
3    Label[] label;
4    public Bakery (int n) {
5      flag = new boolean[n];
6      label = new Label[n];
7      for (int i = 0; i < n; i++) {
8        flag[i] = false; label[i] = 0;
9      }
10   }
11   public void lock() {
12     int i = ThreadID.get();
13     flag[i] = true;
14     label[i] = max(label[0], ...,label[n-1]) + 1;
15     while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16   }
17   public void unlock() {
18     flag[ThreadID.get()] = false;
19   }
20 }
```

Figure 1: Lamport's Bakery Lock as presented by Herlihy [7]

Lamport's Bakery algorithm fulfills mutual exclusion and first-come-first-served.

class Lamport_Lecture_atomic

Class `Lamport_Lecture_atomic` is a subclass of the `DW_Lock` class. It uses atomic registers for the shared arrays and a standard integer variable for the size. The implementation is very much straight forward.

- `Lamport_Lecture_atomic(int n)`
  Constructor. Takes the size of the lock as an argument. Space for arrays is allocated and variables are initialized.
- `~ Lamport_Lecture_atomic ()`
  Destructor
- `draw_ticket()`
  hands out a new ticket (line 14 in Figure 1)
- `keep_waiting(int id)`
  tests if there is a contender with higher priority (line 15 in Figure 1)
- `doorway()`
  doorway section. The thread states that it is trying to acquire the lock and draws a ticket (lines 13 and 14 in Figure 1)
- `wait()`
  waiting section. threads check if there are contenders with higher priority. They will spin on shared arrays by repeatedly calling `keep_waiting()`(line 14 in Figure 1)
- `unlock()`
  The thread states that it is no longer trying to acquire the lock. (line 18 in Figure 1)

3.3 Taubenfeld (Black/White-Bakery)

In 2004, Gadi Taubenfeld presented his modification[2] of Lamport's bakery algorithm that, first and foremost, promises to bound the size of the needed registers. In total three different versions of his algorithm are presented, where the latter two build upon the first, basic version.

All three versions can be shown to satisfy *mutual exclusion*, *deadlock freedom* and *FIFO (FCFS)*, which also implies *starvation freedom*. The other two versions add the properties *adaptive* and *local-spinning*, although we did not implement them due to time constraints. Though we also want to note, that they are significantly more complex to achieve these added properties, whereby they lose the simple elegance of Lamport's original algorithm.

The basic idea behind Taubenfeld's Black-White-Bakery (sometimes also called Color-Bakery) is to add a color to the tickets, either black or white, and then working through a block of one color before considering tickets of the other color. Tickets are colored based on the value of a shared Color bit. Tickets with a color different to the shared Color bit are considered to have higher priority than those that share the shared bits color, regardless of the tickets' numbers. In each color block, priority is given to the ticket with the lower number, and failing that, the threads' IDs are considered. A thread that has successfully acquired the lock and executed the CS will afterwards set the shared Color bit to the color different to its ticket and then reset its ticket.

In this way the finite number of registers needed can be each bound to the sizes *log(2n +2) bit* and *1 bit* respectively. In total three registers are needed, one shared register of size 1 bit for the color

class Taubenfeld

The first, basic variant presented by Taubenfeld in his paper as shown in Figure 2: Black-White Bakery Algorithm [2], implemented as a subclass of DW_Lock.

It is not mentioned in the paper what type of registers are required for the algorithm to work, so we implemented this version using just volatile int C-arrays. We then noticed during our tests that this version of the Taubenfeld lock very rarely led to mutual exclusion failures. We tried to fix this issue by using atomic registers instead, which led to the next version.

Algorithm 2. THE BLACK-WHITE BAKERY ALGORITHM: `process` i`'s code`

Shared variables:
$color$: a bit of type $\{\texttt{black}, \texttt{white}\}$
$choosing[1..n]$: boolean array
$(mycolor, number)[1..n]$: array of type $\{\texttt{black}, \texttt{white}\} \times \{0, ..., n\}$
Initially $\forall i : 1 \leq i \leq n : choosing_i = \texttt{false}$ and $number_i = 0$,
the initial values of all the other variables are immaterial.

```
1   choosing_i := true                                  /* beginning of doorway */
2   mycolor_i := color
3   number_i := 1 + max({number_j | (1 ≤ j ≤ n) ∧ (mycolor_j = mycolor_i)})
4   choosing_i := false                                 /* end of doorway */
5   for j = 1 to n do
6       await choosing_j = false
7       if mycolor_j = mycolor_i
8       then await (number_j = 0) ∨ ([number_j, j] ≥ [number_i, i]) ∨
                   (mycolor_j ≠ mycolor_i)
9       else await (number_j = 0) ∨ (mycolor_i ≠ color) ∨
                   (mycolor_j = mycolor_i) fi
10  od
11  critical section
12  if mycolor_i = black then color := white else color := black fi
13  number_i := 0
```

Figure 2: Black-White Bakery Algorithm [2]

class Taubenfeld_atomic

This class is also a subclass of DW_Lock and implements the necessary methods.

This implementation of the Black-White-Bakery uses *atomic* registers for the *color*, *choosing* and *ticket* arrays. We never encountered mutual exclusion failures with this version.

class Taubenfeld_adaptive

This class is also a subclass of DW_Lock and implements the necessary methods.

This was our attempt at an implementation of the second variant presented in the paper, the adaptive Black-White-Bakery, which uses a so-called active Set [2] (page 64). Implementing this active set in C++ proved too difficult and time consuming for us. One obstacle was the fact that the STL-*Set* in C++11 is not thread-safe. Although it "only" rarely produces mutual exclusion failures, that also means it does NOT work.

We therefore decided to focus on the basic version of Taubenfeld's lock algorithm. This class is mentioned only for completeness sake.

The structure of this class is very similar to the class Lamport_lecture_atomic. Each method has been slightly modified to adapt the necessary changes concerning the ticket coloring as described above.

3.4 Jayanti

Basic Idea

In the paper by Jayanti et al. [3] another variation to to Lamport's Bakery Lock [1] is described. The advantage of the described lock over Lamport's algorithm is, that is able to keep the token (sometimes called ticket) bounded: for a lock of size n, only $2n-1$ token values are required (i.e. $V_{token} = \{0, \dots, 2n-2\}$. The paper describes the algorithm in two steps. First an algorithm with clustered but still unbounded tokens is presented. Then this algorithm is enhanced to make the tokens bounded in value.

Algorithm with Clustered Tokens

It is shown that with Lamport's algorithm in special cases, the differences between tokens can grow without bounds. The first algorithm in the paper fixes that, by introducing an additional atomic register (variable X in Figure 3), that holds the token value of the last thread, that acquired the lock. The value stored in X is to be considered, when issuing a new token, which prevents token values to reset to their initial value. This prevents the mentioned special case, where the differences between tokens in Lamport's algorithm grow without bounds. Figure 3 shows the algorithm of Jayanti et al.

[initialize $X := 0$;
$\forall i : 1 \leq i \leq n : (gettoken_i :=$ **false**
$token_i := -1)$]

1. $gettoken_i :=$ **true**
2. $S := \{token_j | 1 \leq j \leq n\}$
3. $x := X$
4. $token_i := max(S \cup \{x\}) + 1$
5. $gettoken_i :=$ **false**
   for $j \in \{1 \cdots n\} - \{i\}$
6. **wait till** $gettoken_j =$ **false**
7. **wait till** $(token_j = -1) \vee ([token_i, i)] < [token_j, j])$
8. $X := token_i$
9. CS
10. $token_i := -1$

Figure 3: The algorithm by Jayanti et al. [3]

Algorithm with Bounded Tokens

In a second step this algorithm is enhanced such that the now clustered tokens will also be bounded in value. This is being accomplished by changing the way, a new token is issued as well as changing the order relation by which the priority between two threads is decided.

first two new binary arithmetic operations are defined:

let $a \oplus b$ denote $(a + b)\ mod\ (2n-1)$

let $a \ominus b$ denote $(a - b)\ mod\ (2n-1)$

With these operations a new way to calculate the maximum of a set of tokens S_b is defined (here x_b is the value of X read in line 3 of Figure 3):

let $T = S_b \cup \{x_b\} - \{-1\}$

then define $\max_{new}(S_b \cup \{x_b\}) = \max_{v \in T}\{v \oplus (n-1+x_b)\}$

Furthermore, the order relation from line 7 is replaced by the relation $\prec$, that is defined as follows:

$$[v_i, i] \prec [v_j, j] \Leftrightarrow [v_i \oplus (n-1-v_i), i] < [v_j \oplus (n-1-v_i), j]$$

With these replacements, the tokens are bounded.

Implementation

Two classes were implemented

- `Jayanti` - the described algorithm with unbounded tokens
- `Jayanti_BT` - the adapted algorithm with bounded tokens

Class Jayanti

Class `Jayanti` is a subclass of the `DW_Lock` class. It uses atomic registers for the shared arrays, an atomic register for the value of X and a standard integer variable for the size. The implementation is very much straight forward.

- `Jayanti(int n)`
  Constructor. Takes the size of the lock as an argument. Space for arrays is allocated and variables are initialized.
- `~Jayanti()`
  Destructor
- `get_token(int* tl_token, int tl_x)`
  hands out a new token, considering the tokens of other threads as well as the value of X. (line 4 of Figure 3). It is worth mentioning, that this function is passed a thread local array of tokens that was read beforehand as well as a thread local representation of the value of X. the reasoning behind this are the specifications of the algorithm (lines 2 and 3 of Figure 3)
- `keep_waiting_1(int j)`
  tests if a thread is drawing a token (line 6 of Figure 3)
- `keep_waiting_2(int id, int j)`
  tests if a thread has higher priority than the calling thread (line 7 of Figure 3)
- `doorway()`
  doorway section. Draws a token in the style of the original Lamport algorithm and fills the thread local array `tl_token[]` as well as `tl_x` (lines 1-5 in of Figure 3).
- `wait()`
  waiting section. threads check if there are contenders with higher priority. They will spin on shared arrays by repeatedly calling `keep_waiting_1()` and `keep_waiting_2()`.
- `unlock()`
  resets the thread's token value .

Class Jayanti_BT

Class `Jayanti_BT` is a subclass of the `Jayanti` class that implements the variation of the Jayanti algorithm that keeps tokens bounded. The changes involve definition of some functions as described in the section above:

jaya_plus()

- `jaya_plus(int a, int b)`
  implements the $\oplus$ operation. It should be stated here, that in C++ it is necessary to make sure, this returns a positive value, since $x \; mod \; y < 0$ is a possible phenomenon of the % operator, if one of the arguments is lesser than zero.
  Therefore $a \oplus b \coloneqq (a + b) mod (2n - 1)$ is calculated by first shifting $(a + b)$ in steps of $(2n - 1)$ until it is positive and then returning the remainder of the division.
- `jaya_minus(int a, int b)`
  analogous to `jaya_plus()`
- `jaya_comp(int token_a,int a,int token_b,int b)`
  compares the priority between two threads as described by the $\prec$ relation above
- `jaya_max(int* tl_token, int tl_x)`
  calculates the maximum over a set of tokens as described by the $\max_{\text{new}}$ function above.
- `get_token()` and `keep_waiting2()` override existing functions of the parent class to call the sub-class specific functions.
- a sub-class specific `wait()` function overrides the method of the parent class for debugging reasons

3.5 Aravind

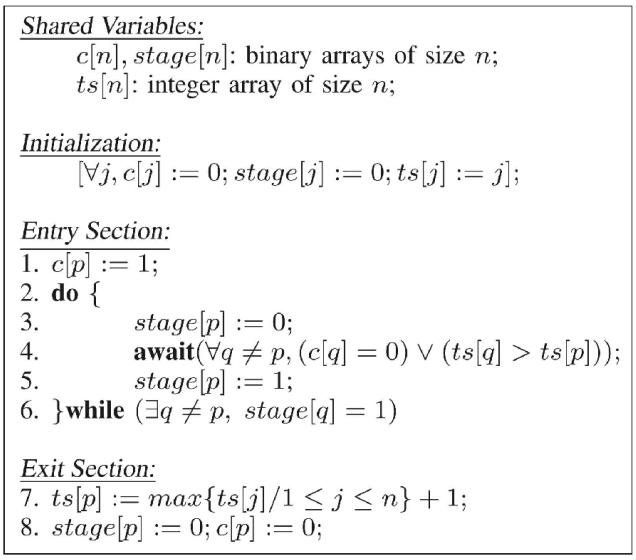
Basic Idea

The paper by Aravind [4] presents yet another variation to Lamport's Bakery Lock. This locks outstanding feature is, that instead of fulfilling the FCFS property the least-recently-used property is fulfilled. This fairness property defines different rules considering, which thread should be allowed to acquire the lock, in case an acquisition is contended: LRU demands that amongst the contenders, the thread, which has not had the lock for the longest time, should be allowed to acquire it. This is accomplished by handing out timestamps (tokens, tickets, ...) to threads when they leave the critical section and are about to unlock, rather than handing them out in the doorway section.

LRU Algorithm

As it was mentioned, the timestamps are handed out, when threads leave the critical section. This has the advantage, that no two threads can ever end up with the same timestamp as long as mutual exclusion holds. Therefore the comparison of threads in terms of which one has the higher priority to acquire the lock is simplified, since it is reduced to just comparing timestamps without the need to consider the possibility of equality. By comparing the which thread was first to finish the critical

section rather than which thread was first to go through the doorway, the lock fulfills the LRU property instead of the FCFS property. Figure 4 shows the Algorithm.



Shared Variables:
$c[n], stage[n]$: binary arrays of size n;
$ts[n]$: integer array of size n;

Initialization:
$[\forall j, c[j] := 0; stage[j] := 0; ts[j] := j];$

Entry Section:
1. $c[p] := 1;$
2. **do** {
3. $stage[p] := 0;$
4. **await**$(\forall q \neq p, (c[q] = 0) \vee (ts[q] > ts[p]));$
5. $stage[p] := 1;$
6. }**while** $(\exists q \neq p,\ stage[q] = 1)$

Exit Section:
7. $ts[p] := max\{ts[j]/1 \leq j \leq n\} + 1;$
8. $stage[p] := 0; c[p] := 0;$

Figure 4: The algorithm by Aravind [4]

The algorithm uses three arrays:

- `c[j]`: Boolean – indicates whether process *j* is trying to acquire the lock
- `stage[j]`: Boolean – indicates the stage that process *j* is in (see below)
- `ts[j]`: integer – indicates when process *j* has last acquired the lock

when process *j* tries to acquire the lock it will:

- at stage 0
    - set `c[j] = 1`
    - check all `c[k]` and `ts[k]` to check that no one else should rather get the lock. If ok, set `stage[j] = 1`; if not, wait and check again,
- at stage 1
    - check that no one else is at stage 1: if ok, proceed to CS, if not, set `stage[j] = 0` and start over with stage 0

When a process *j* unlocks, it will:

- Draw a new timestep and write it to `ts[j]`
- Set `stage[j] = 0`, `c[j] = 0`

Proceeding like this the LRU property is not strictly fulfilled as Aravind himself admits in his paper. It is possible, that a thread A with higher priority starts to try and acquire the lock, while a lower priority thread B is at line 5 of Figure 4. The fact that thread A is now a contender will not be considered by thread B who will (in many cases) be allowed to go ahead and acquire the lock.

This defect can be reduced by storing a list of contenders when executing line 4 in a thread local boolean array and checking, if the list of contenders has changed before finally acquiring the lock. This has been done in the class `Aravind_fix`. This will however still not fully remove the defect,

because the checking of the list is not done atomically, which means a higher order thread can become a contender, while the list is checked.

LRU Algorithm with bounded tokens

The paper gives an example of how the presented algorithm can ensure that the timestamps stay bounded. Once a timestamp would exceed the maximum valid value, all timestamps are reset to their initial values. This of course violates the LRU property, but it is stated in the paper, that while it is easily possible to reset the timestamps, such that the LRU property remains fully intact, but it was deliberately chosen not to do so, because priority was given to the simplicity of the algorithm.

This way of bounding the timestamps has not been implemented in this project, because it does not seem to have been the focus of Aravind's work.

Implementation

Two classes were implemented

- Aravind - the described LRU algorithm with unbounded tokens
- Aravind_fix - includes the above described change to reduce LRU violations

Class Aravind

Class `Aravind` is a subclass of the `DW_Lock` class. It uses atomic registers for the shared arrays and a standard integer variable for the size. The implementation is very much straight forward.

- `Aravind(int n)`
  Constructor. Takes the size of the lock as an argument. Space for arrays is allocated and variables are initialized.
- `~Aravind()`
  Destructor
- `new_ts()`
  hands out a new timestamp
- `keep_waiting_1(int id)`
  tests if there are contenders with higher priority (line 4 in Figure 4)
- `keep_waiting_2(int id)`
  tests if there are other contenders, who got passed acquisition stage 0 (line 6 in Figure 4)
- `doorway()`
  doorway section. set the thread's contender flag to true (line 1 in Figure 4). Nothing else to be done here, since the timestamp is drawn when unlocking.
- `wait()`
  waiting section. threads try to acquire the lock by passing through the stages. They will spin on shared arrays by repeatedly calling `keep_waiting_1()` and `keep_waiting_2()`.
- `unlock()`
  draws a new timestamp and resets the thread's stage and contender values .

Class Aravind_fix

Class `Aravind_fix` is a subclass of the `Aravind` class. The only difference is, that a thread local boolean array that indicates the contenders is used kept to reduce LRU violations by checking, whether a new contender arrived.

Changes to the `Aravind` class:

- `keep_waiting_1(int id, bool* check_c)`
  the array of contenders is passed and written when checking the priorities of the contenders
- `check_contenders(bool* check_c)`
  checks, if the shared array of contenders is the same as the thread local one
- `wait()`
  will now call `check_contenders()` before allowing a thread to acquire the lock.

4 Results

Our main performance criterium is throughput, which we measure by the amount of successful lock acquisitions per second. A higher amount of workload in the CS will then lead a lower amount of acquisitions per second, naturally. A lock algorithm that is more efficient will produce a higher amount of acquisitions per second given the same workload (in and outside the CS).

We tested three different scenarios:

Name	Workload	Workload_cs	randomness
Scenario 1	0	0	0
Scenario 2	100	1000	0.4
Scenario 3	1000	100	0.4

Workload… amount of work to be done by each thread outside of the CS
Workload_cs… amount of work to be done inside the CS
Randomness… randomization factor that scales the actual work to be done in and outside the CS

Both workload and workload_cs specify the upper limit for a for-loop. A workload of 100 with randomness of 0.4 then means, that the for-loop is executed at least 60 times and at most 100 times. Randomness gives the percentage (of the number of iterations of the respective for-loop), that is randomized each time either the outside work section or CS is called. The purpose of this setting is to make the sequence of lock attempts of the threads more random.

We tested the above scenarios for the following numbers of threads [2, 3, 4, 8, 16, 32, 64]. However, even though the Reference Lock otherwise performed well, when we used 32 and 64 threads in scenarios 1 and 2, the benchmarks were interrupted due to the time constraint of 5mins. There is probably an issue in our implementation that we could not fix. Our implementation of the reference lock worked just fine for scenario 3. Where we could not gather data due to the issue described above, they are simply omitted.

4.1 Scenario 1 (workload=0)

Scenario 0 represents the edge case scenario where threads constantly attempt to acquire the lock and are only really waiting for their turn to do work on the CS. This scenario is of course not realistic but serves as an edge case or limit for poorly parallelized code.

Figure 5 and Figure 6 show the Throughput (as defined above) with and without the logging functionality of our test function. We notice that the logging has a significant impact on performance, as was expected, but it does not impact the different locks differently and does not make the results incomparable between the different implementations. Error bars are plotted in every plot, but are often so small, that they are not visible. Lamport is the fastest of the ticket locks, whereas the Reference Lock (test_and_set) outperforms it for lower amounts of threads.

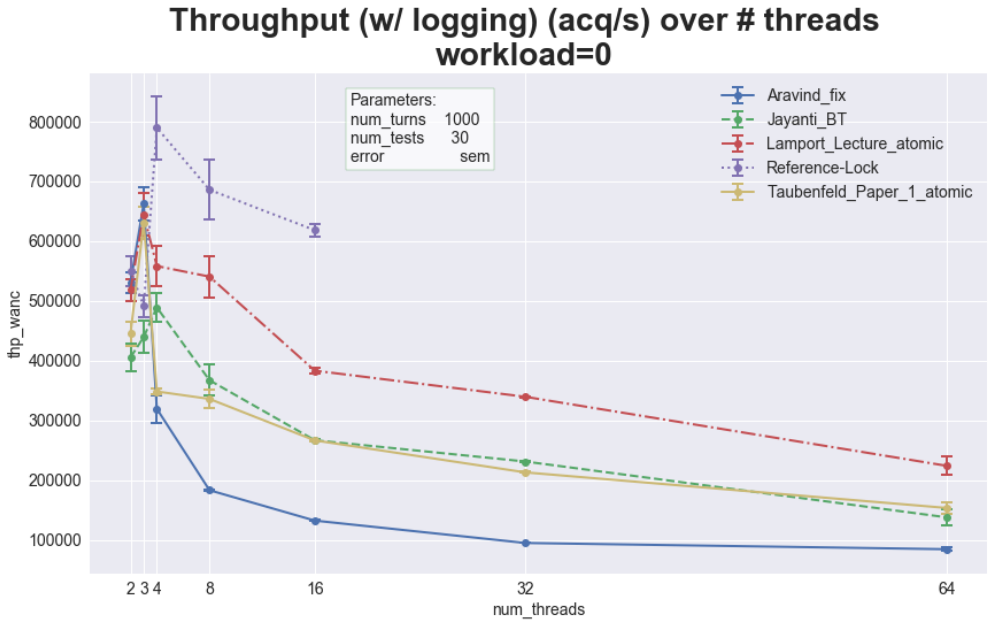


Figure 5: Scenario 1 : throughput with logging

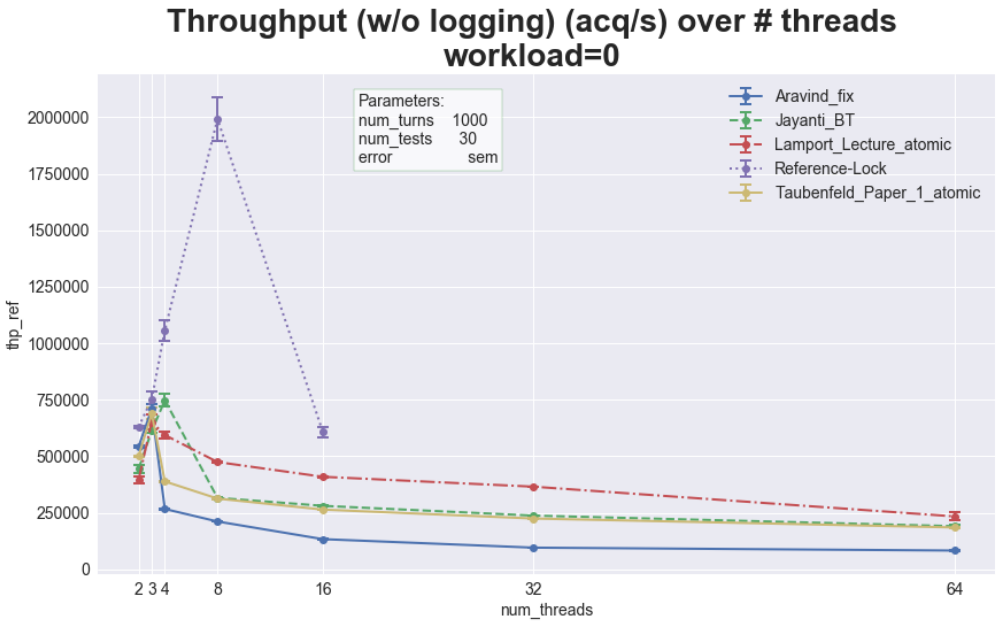


Figure 6: Scenario 1 : throughput without logging

4.2 Scenario 2 (workload=100)

This test scenario is similar to the first. It represents a scenario where performance is mainly restricted by a CS and the workload outside is comparatively small (workload outside : workload CS = 0.1), but is a more realistic scenario than 1 as we take vary (randomize) the workload in and outside the CS.

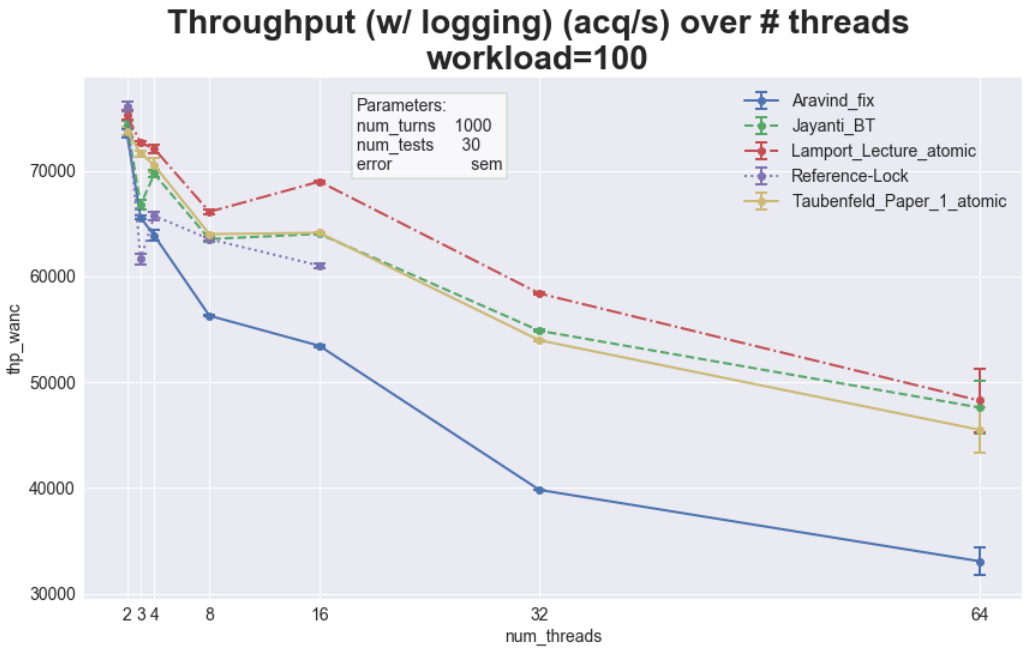


Figure 7: Scenario 2: throughput with logging

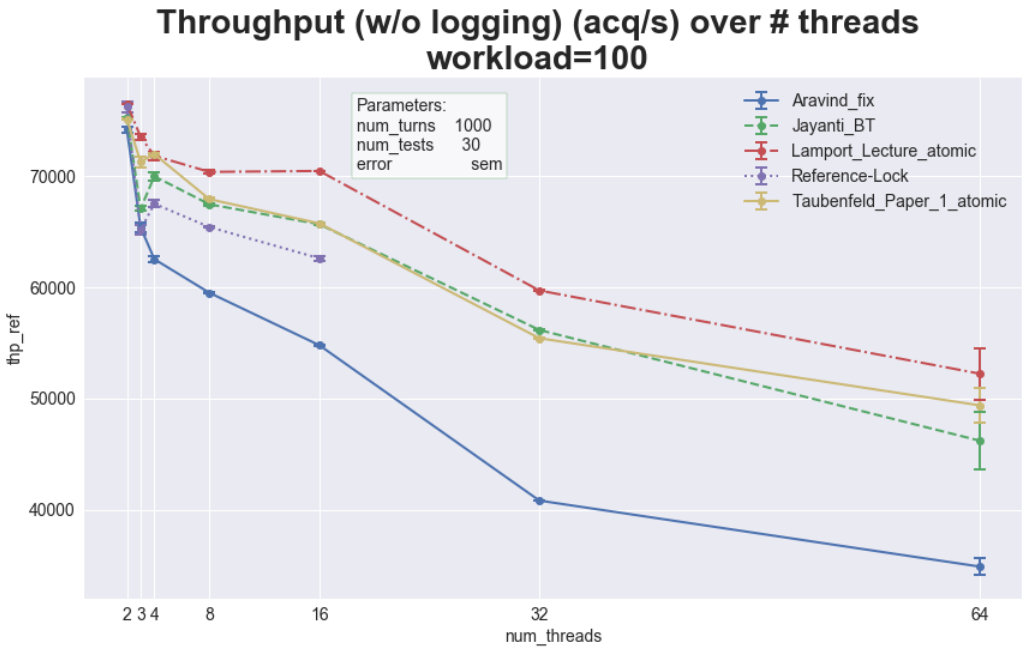


Figure 8: Scenario 2: throughput without logging

Figure 7 and Figure 8 shows the results for the throughput tests for this scenario. This time Lamport's Bakery performs the best across the board and the Reference Lock is even slower than both Jayanti_BT and the Taubenfeld Black-White-Bakery lock. Aravind again performs the worst out of all.

4.3 Scenario 3 (workload=1000)

Here, we test the locks for scenarios where the ratio of workload/workload_CS=10, so where a program is parallelized to a higher degree than before.

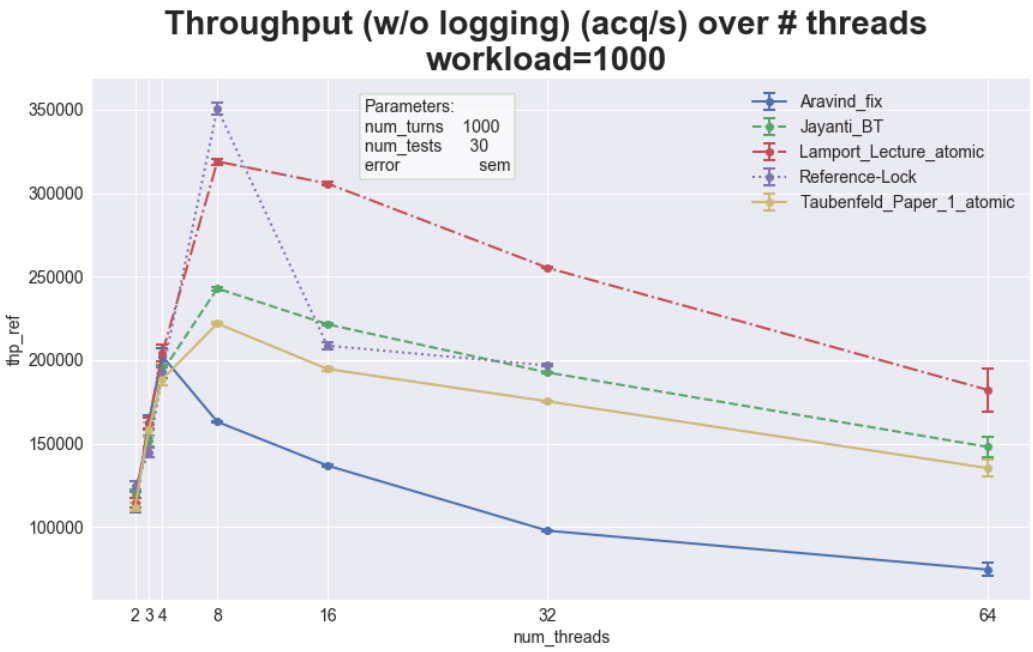


Figure 9: Scenario 3 : throughput with logging

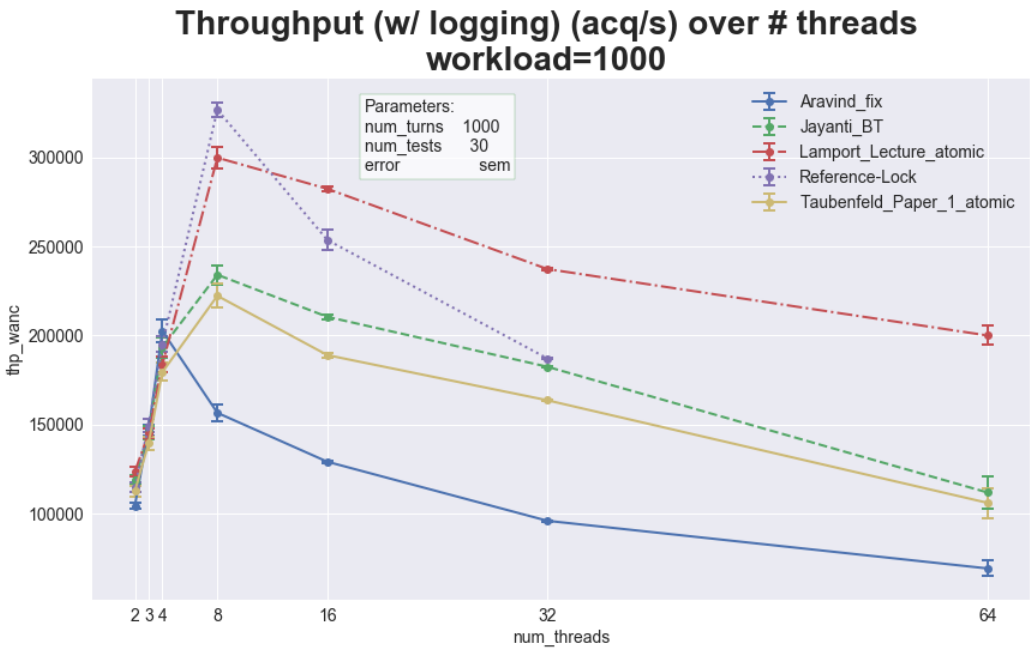


Figure 10: Scenario 3 : throughput without logging

In Figure 9 and Figure 10, we can see the throughput for this scenario. Again, Lamport's Bakery performs the best across the board, while the Reference lock (test_and_set) performs the best for low thread numbers but performs similar to the Jayanti_BT and Taubenfeld locks. Aravind is again the worst out of all.

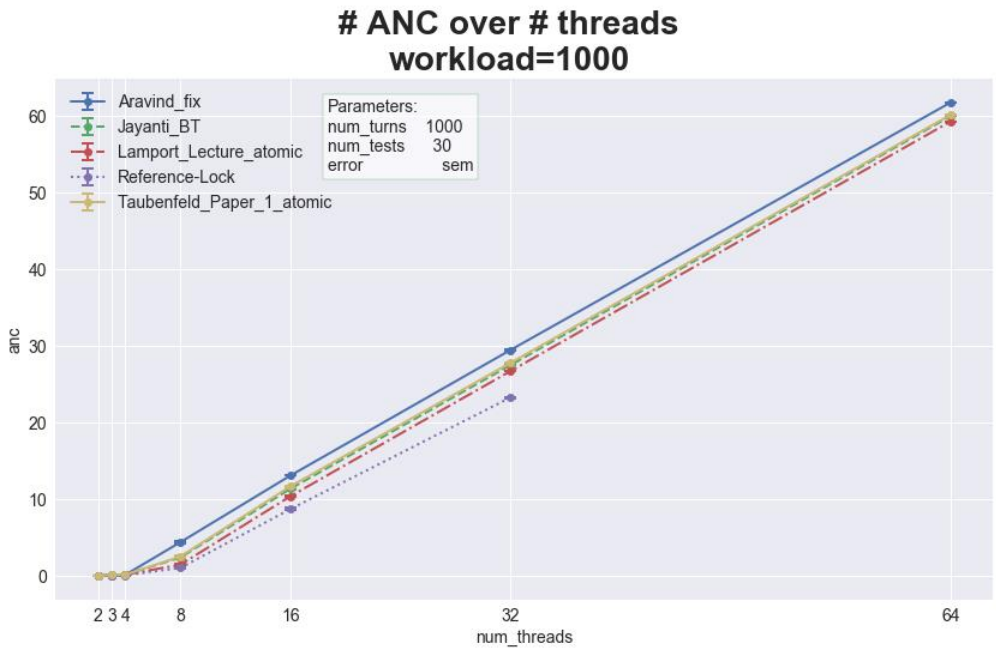


Figure 11: ANC (Average Number of Contenders)

Figure 11 shows the average number of contenders for acquisition of the lock. For the ticket locks, the appear in the same ordering as in the throughput graphs. A lower ANC would indicate a more efficient lock algorithm as that would mean that less threads are . This holds true at least for the ticket locks. However, the Reference lock (test_and_set) does not adhere to this.

5 References

[1] L. Lamport, “A New Solution of Dijkstra’s Concurrent Programming Problem,” *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[2] G. Taubenfeld, “The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3274, no. 4, pp. 56–70, 2004.

[3] P. Jayanti, K. Tan, G. Friedland, and A. Katz, “Bounding Lamport ’ s Bakery Algorithm,” pp. 261–270, 2001.

[4] A. A. Aravind, “Yet another simple solution for the concurrent programming control problem,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 1056–1063, 2011.

[5] B. K. Szymanski, “A simple solution to Lamport’s concurrent programming problem with linear wait,” *Proc. Int. Conf. Supercomput.*, vol. Part F1301, pp. 621–626, 1988.

[6] V. Gramoli, “More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms,” *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program. PPOPP*, vol. 2015-Janua, pp. 1–10, 2015.

[7] M. Herlihy, *The art of multiprocessor programming*. 2006.

6 Annex - Record Event Log

6.1 Function Description

Recording the `event_log` array is done by performing these steps:

- create seeds for thread local random number generators (RNG)
- start a parallel section using OpenMP
- have threads:
    - create a `std::mt19937` object as a thread local RNG
    - acquire the lock for a number of times
    - log events in a thread local array only sharing the atomic counter
    - perform a specified random workload inside and outside the critical section
    - measure runtime (master thread only)
    - (after the lock test) write their thread_local event logs to one global event_log array.

6.2 Inaccuracies

There is a certain inaccuracy, when recording these events. For example, in between a thread, finishing a doorway section and atomically reading and increasing the shared counter to document, when the finishing of the doorway took place, other threads take over and record their events more quickly even if the events actually took place later. This can cause inaccuracies in the recorded sequence of events. In the this section the consequences of these inaccuracies are being discussed.

The method of this discussion:

There is a condition based on the sequence of events.

There is a noted sequence of events and a real sequence of events.

The inaccuracy causes a difference between these sequences.

In this section it is discussed, whether the condition becomes stronger or weaker. If it becomes stronger this can cause false positives when checking for violations of the property. If the condition becomes weaker it is possible that violations of the property are not found.

(1) begin

A thread A notes, that it is about to begin its doorway. Other threads can cause events between a thread noting, that it will now begin its doorway, and it actually beginning the doorway. So $(A, begin)_k$ has actually happened later than what was noted.

In the real sequence $(A, begin)_k$ can be later than in the recorded sequence. ("recorded begins appear earlier than they are").

Relevant in fcfs: condition $(A, finish) \rightarrow (B, begin) \Rightarrow (A, acquire) \rightarrow (B, acquire)$

In the real sequence $(A, finish) \rightarrow (B, begin)$ is will be more often satisfied than in the recorded sequence. Therefore the inaccuracy makes the condition weaker because $X \Rightarrow Y$ is hard to satisfy if X is easy to satisfy.

(2) finish

A thread A notes, that has just finished its doorway. Other threads can cause events between a thread finishing its doorway and logging it. So $(A, finish)_k$ has actually happened earlier than what was logged. ("logged finishes appear later than they are")

Relevant in fcfs: condition $(A, finish) \rightarrow (B, begin) \Rightarrow (A, acquire) \rightarrow (B, acquire)$

Once again: In the real sequence $(A, finish) \rightarrow (B, begin)$ is will be more often satisfied than in the recorded sequence. Therefore the inaccuracy makes the condition weaker.

(3) acquire

A thread A notes, that has just acquired the lock. Other threads can cause events between a thread acquiring the lock and logging the acquisition. Assuming mutual exclusion for the lock, this cannot be acquisition or unlock events. So $(A, acquire)_k$ has actually happened earlier than what was logged. ("logged acquisitions appear later than they are")

- Relevant for mutual exclusion: condition $(A, unlock) \rightarrow (B, acquire)$

Acquisitions appearing later than they are, can cause satisfaction of the condition where it was actually violated.

- Relevant for FCFS: condition $(A, finish) \rightarrow (B, begin) \Rightarrow (A, acquire) \rightarrow (B, acquire)$

Acquisitions appearing later than they are can cause both wrongful satisfaction of the condition as well as false positives when checking the log.

- Relevant for LRU: condition

$$\underbrace{(A, acquire)_{(k-1)} \rightarrow (B, acquire)_{(l-1)}}_{A\ has\ less\ recently\ used}\ and\ \underbrace{(A, finish)_k \rightarrow (B, acquire)_l}_{A\ is\ a\ contender}$$
$$\Rightarrow \underbrace{(A, acquire)_k \rightarrow (B, acquire)_l}_{A\ should\ go\ first}$$

If mutual exclusion is assumed, the order of the acquisitions cannot be changed by this inaccuracy. Therefore only $(A, finish)_k \rightarrow (B, acquire)_l$ is affected, which is harder to satisfy due to the inaccuracy. Therefore the LRU condition is easier to satisfy. Therefore there might be violations of the LRU property that are not registered.

(4) unlock

A thread A notes, that it is about to unlock. Other threads can cause events between a thread logging the unlock and the actual unlock. Assuming mutual exclusion for the lock, this cannot be acquisition or unlock events. So $(A, unlock)_k$ has actually happened later than what was logged. ("logged unlocks appear earlier than they are")

- Relevant for mutual exclusion: condition $(A, unlock) \rightarrow (B, acquire)$

Unlock events appearing earlier than they are, can cause satisfaction of the condition when it was actually violated.