Universität Rostock

Traditio et Innovatio

Bachelorarbeit

Parallele Zustandsraumsuche

Vorgelegt von:

Tom Meyer

Matrikel-Nr.: 8200839

Eingereicht am:

02. März 2018

Betreuer:

Prof. Dr. rer. nat. habil. Karsten Wolf

UNIVERSITÄT ROSTOCK | FAKULTÄT FÜR INFORMATIK UND ELEKTROTECHNIK

Institut für Theoretische Informatik

Albert-Einstein-Straße 22, 18059 Rostock | Tel: +49 381 498 7641

Contents

1 Introduction

A thesis statement should do the following:

Explain the readers how you interpret the subject of the research Tell the readers what to expect from your paper Answer the question you were asked Present your claim which other people may want to dispute

Make sure your thesis is strong.

- What is paralelization
- What are petri nets
- What should be done in lola
- How should it be done
— profiling
- What was done
- What can be done in the future

2 Related Work

2.1 Parallelization History

2.2 Synchronization Methods

- mutexe
- semaphores
- compare and swap
- spin locks

2.3 Depth First Search Parallelization

- what is a graph

2.4 petri nets

-reachability

3 Background

The purpose of this work is to efficiently make use of multiple threads with the LowLevelAnalyzer (lola).

Lolas development startet in 1998. It was aimed to be used by third party tools to check properties of petri nets [Sch00]. Since then it was steadily updated to compete with other state of the art tools. Recurring prizes in a model checking contest with focus on petri nets suggest a success in this attempt [FK17]. The internal property evaluation however, is still most performant single threaded. To evaluate a property of a petri net, lola searches all necessary net states that can be reached from the initial one. The search is a depth first search on an undirected graph. The graph is discovered during the search itself.

Parallelization of depth first search is a difficult matter. Although rather general algorithms exist (some of them mentioned in the previous chapter), they usually make use of an assumption which cannot be made with lola.

For the quite specialized search in lola, a previous attempt of parallelization exists. Unfortunately the performance is usually worse then the single threaded approach. It even gets worse the more threads are used.

This is the staring point of this work. There is an algorithm which is not performing as expected. The most important task, is to find the bottle neck of this algorithm. After that it is desired to remove that bottleneck. The improved version should hopefully outperform the single threaded algorithm with use of a reasonable amount of threads. Additionally a desirable result would show a linear (or better) scaling of the performance with the amount of threads.

3.1 Performance Measurement

3.1.1 A distinction between different views on software

The performance of a program is a property which is always of interest somehow. Even if it is not a critical aspect in the working environment. But as reachable the concept of performance might look at first, the complicated it can get.

Just like that, what is called a performant application is highly dependent on its environment. The possibly most obvious characteristic would be the time it takes to execute a program or a subroutine. But in a lot of szenarios, the time consumption actually does not matter that much. A user might not even notice a factor of 1000

between routines that operate in an order of micro seconds. At the same time he might worry that the program can access enough memory to finish its task. That means before measuring the performance of a program or subroutine, one must define the characteristics that define the performance of exactly that given program or subroutine. Other characteristics beside execution time and memory consumption can be network latency, energy consumption or responsibility. Ultimately it is everything that increases the useability of a program.

After knowing what will be measured the reasonable next concern is how it will be measured. The act of measuring is typically called benchmark and the approach here is influenced by the scope of the measurement. Oaks describes a general distinction between macro, meso and micro benchmarks [Oak14]. Each type addresses a different scope of a programm.

Macro benchmarks give an overview over the entire execution. The data collection is often less complicated since it is done over the whole application runtime and hardly requires any modifications to it. Measuring the runtime of a program is as simple as it gets: the difference between the end and the start of the program. The memory allocation can often also be measured with simple tools that the operating system or other third party software provide. The downside is that a lot of software cannot be measured at all in this scope. So it is a property of event driven programs that they do not terminate at a known point in time and therefore the execution time is of limited use.

Micro benchmarks consider the most narrow aspects of an application. They messure the execution of single instructions or atomic routines. Results of micro benchmarks have to be taken with a grain of salt because compile and hardware optimization strongly influences the results. Assumptions that work for the single code snippet might become false after the compiler integrates that part into the complete program or vise versa. It is a good advise to use a benchmarking tool which is built for this task. Overwise there is a risk to be trapped in false results. Like compilers that are optimizing away whole loops (which in turn often be the benchmark itself), because the result can be evaluated at compile time. The advantages of micro benchmarks however, is that different implementations of basic algorithms can be compared directly. Provided they are performing equivalent in the macro scope of the program.

After the previous scopes described a detailed view and an overview, the last scope has to be something in between. And this holds true for the so called meso benchmark. Perhaps the best explanation for this kind is, that a meso benchmark is what is not macro nor micro benchmark. Meso benchmarks deal with parts of a program that get more complex. Like Modules or further reaching subroutines. These can be the most interesting benchmarks since they can narrow down bottle necks to specific parts of the software. Unfortunately they often require to manipulate the program in some way. Which again means that the measured performance differs from the actual performance.

3.1.2 Methods of measurement

For the actual measurements, two basic approaches can be distinguished: manual implementation or usage of a tool.

The quickest and most precise approach is to make own measurements. Self written, and therefore known software can be easily extended e.g. by counters for variables or measurements of time periods. All measurements can be tailored to the individual use case and results can be exported in the most convenient format. On the over hand this approach can come with several downsides. Some of them might be that the maintenance of the measurements can get increasingly complex the more numerous they get, added code will influence the performance of the software itself, and own mistakes might hide behind satisfying results. And there are probably other reasons that can be thought of, but short, temporary measurements of this kind might give o quick overview the own software.

Probably the safer approach is the usage of a tool. There exist a variety of tools which are build for the inherent purpose to measure application performance with common metrics and use cases. Since the metrics reflect some characteristics of an application - a profile - these tools are typically called profilers. Using established profilers for performance measurement can help avoiding common traps and mistakes especially when they were designed by experienced developers who are skilled in this area. But it also means to invest time and effort to learn how to use it and how to interpret its results.

Profilers aid to do the things automatically that otherwise would have to be implemented manually. An often used method to figure out bottle necks is to take samples of the call stack during program execution. The calls that are most frequently sampled will correlate with the program parts where the most time was spent in. Another method is to use code injections. Here, code is inserted automatically into the source or binary code at relevant spots. The injections are then used to count events or measure time intervals.

With the general techniques in mind we can proceed to the actual software and its characteristics that will be examined in this work.

3.2 Implementation Details

3.2.1 Code Base

This work is based on a previous attempt by Gregor Behnke to use multiple threads for lola. For this reason, his code will be taken as base implementation for the parallel search. In the lola project structure, this would correspond to the `ParallelExploration` class in src/Exploration.

This class itself seems to be based on a single threaded algorithm in the `DFSExploration` class, since the basic structure and several lines of code as well as comments are equivalent. The base algorithm was then altered by Behnke to make data manipulation thread

safe. This implies that no specialized Depth First Search algorithm was used, but the existing algorithm was extended.

The purpose of the exploration class family in lola, is to keep track of the **paths** that where discovered during the search. The actual net **states** that are the vertices of the path, are managed by another family of classes: the store classes. In more detail this means that the exploration class will store the edges that lead to the current net state and choses which edges of the graph will be used next to discover another net state. The discovered net states will be handed to the store class. It will tell the exploration class if this net state was already discovered earlier. If that is not the case the net state will be stored permanently

Behnke tried to implement multi threading (inside the exploration class) by exploring a different path with each thread. He therefore introduced thread local variables that keep track of the path, determine which edge will be expanded next and what the current net state is. The store is globally shared. Thread safe searching and adding to the store must be implemented by the store itself for this approach. The crucial work of Behnke inside the exploration algorithm, was to distribute the work over all threads and keep the data synchronized.

Load distribution is done with a simple approach. If a thread can expand multiple edges and an idle thread exists, one edge will be discovered by the current thread and another will be discovered by the next possible idle thread. The data will be synchronized, so that the woken up thread will copy the previously discovered path of the waken. Cases like: what should be done if a thread has no more work (accessible edges) left or what to do if the current net state satisfies the asked property, are also handled.

The store which is used by lola in the default case (`PluginStore`) does not provide any thread safety. The thread number is completely ignored within the relevant parts and so the behavior of this store when using multiple threads is undefined. But there is another store implementation which implements several buckets to store the states in: the `HashingWrapperStore`. The buckets should only be accessible by at most one thread. The net states that should be stored are now given a hash value and every bucket has a range of hash values associated with them. With this solution multiple searches can be issued, as long as the hash value of the searched net states differ.

The actual classes which are used at runtime to expand the search tree, are determent by the call switches which are handed to the lola executable. The relevant switches for the parallel exploration are `--threads=[threadCount]` and `--bucketing=[bucketCount]`. A thread count greater then 1 will tell lola to use the `ParallelExploration` class and the bucketing switch will signal to use a hashing store.

3.2.2 Data Synchronization

To keep global data consistent, a method is needed to synchronize this kind of data between threads. A common way to achieve this, is to restrict the access to one thread at a time. for this the concept of mutexes and semaphores are often used as locks for

access control. The downside of this approach is a bottleneck at this global data. If access is granted to at most one thread, every calculation that is done while accessing, is also done at most with the speed that a single thread can provide. Therefore accessing this regions (including locking and unlocking them) should be done as infrequent and as quick as possible.

Since lola computes a lot of short and low level instructions, locking and unlocking data segments might sum up to a significant amount of time. So the locking mechanism used by lola might also be a good starting point to search for a reason for that unexpected performance that lola shows.

Behnke used locks from the pthread library [**?**]. A possible substitute would be an own implementation with a Compare And Swap instruction as spin lock. This has the potential to keep the overhead for the locks at a bare minimum.

3.3 Environment

The used development environment consist of two machines. The general development is done on a virtual machine (vm). It runs with 4 threads on a Intel(R) Core(TM) i7-4770S CPU @ 3.10GHz with 4 physical cores. Each core supports hyper threading which makes a sum of 8 threads on the host of the vm. A total of 17.4 GB of physical memory is available for use inside the vm. Ubuntu 4.10 is running as operating system.

As a performant testsystem a powerful server (ebro) was used. Ebro has 4 AMD Opteron(tm) 6284 SE processors. Each with 16 cores and 2.7GHz base clock with a max boost of 3.4GHz [Inc18]. This sums up to 64 physical cpu cores (and threads) in total. As physical memory 995.2 GHz are installed. The operating system is CentOS Linux 7 Core (collected with hostnamectl command).

If no other source is provided the data was collected with the proc filesystem provided by the linux distributions.

Spec Name	**VM**	**Ebro**
Max. Clock per Thread	3.1GHz	3.4GHz
Available Threads	4	64
Memory	17.4 GB	995.2 GB
Operating System	Ubuntu 4.10	CentOS 7 Core

Table 3.1: Specifications of the two development systems

4 Approach

4.1 Switching The Synchronization

The first step taken was switching from the pthread library to a compare and swap (CAS) algorithm. Since C++ 11 there is an equivalent implementation in the standard library called `bool std::atomic::compare_exchange_weak(T& expected, T val)` (or `bool std::atomic::compare_exchange_strong(T& expected, T val)`) that was used for this task. This function compares the current value of an `std::atomic` with `expected` and replace it with `val` if the comparison returns true. If it returns false it replaces `expected` with the actual value of the atomic. The weak version is allowed to return false in favor for a general performance gain, even if the compared values are actually equal.

To represent a lock that can either be locked or unlocked, a boolean as value is sufficient. To shape a spin lock like mutexes and semaphores with a CAS function, they have to loop until the expected value compares equal. There are two reasonable modes to lock: the first is to just wait until an observed lock is unlocked and the second is to wait for an unlock with an immediate locking. The first approach can be helpful to suspend the execution of the current thread until an external thread is signalling a continuation. The second approach can be used to block access to a resource until all preceeding manipulations are completed. At last there are functions necessary that can block or release the resource in a privileged manner. The resulting implementation is shown in listing 4.1.

Swapping the old pthread implementation with the new CAS implementation now remains a matter of search and replace. The equivalent of the mutexes `pthread_mutex_lock()` is `waitAndLock()`. `Pthread_mutex_unlock()` corresponds to `unlock()`. The previous `sem_wait()` correspond to a `waitForUnlock()` call after a `lock()`. And `sem_post()` corresponds to `unlock()`.
However, this is a very specialized replacement which acts as a proof of concept. Other parts of the code might have to be replaced in another way, depending on the semantics of the part. Additionally the CAS implementation is as short as possible. The pthread library comes with some importend features like a mechanism to reduces the risk of dead locks and different modes for the semaphores.

```
1 #include <atomic>

enum LOCK{
    LOCKED,
```

```
    UNLOCKED
};

static inline void waitAndLock(std::atomic<bool>* lock){
    bool expected = UNLOCKED;
    while (!lock->compare_exchange_weak(expected,LOCKED)){
        expected = UNLOCKED;
    }
}

static inline void waitForUnlock(std::atomic<bool>* lock){
    while (lock->load() != UNLOCKED){
        continue;
    }
}

static inline void lock(std::atomic<bool>* lock){
    lock->store(LOCKED);
}

static inline void unlock(std::atomic<bool>* lock){
    lock->store(UNLOCKED);
}
```

Listing 4.1: Basic implementation of a spin lock with compare and swap

4.2 Finding the bottleneck

4.2.1 Benchmark characteristics

We have now an application with a bottleneck and a possible solution to fix that bottleneck. Next we will compare the performances.

In this case, performance means execution time. The expectation is that the execution time decreases with an increased amount of threads (computing power). The time is therefore the characteristic of interest.

As test systems the both machines described in 3.3 where used.

As test data a predefined petrinet of the dining philosophers was used. It is a common concept in theoretical computer science to illustrate problems and risks of parallel processes and was originally introduced by Dijkstra [Dij71]. The philosophers count can be scaled to an arbitrary amount. This is useful to increase the complexity (and therefore the execution time) of the net to a convenient level. The reachability graph is also reasonably branched to allow a parallel discovery. The actual used net is a version with one thousand philosophers.

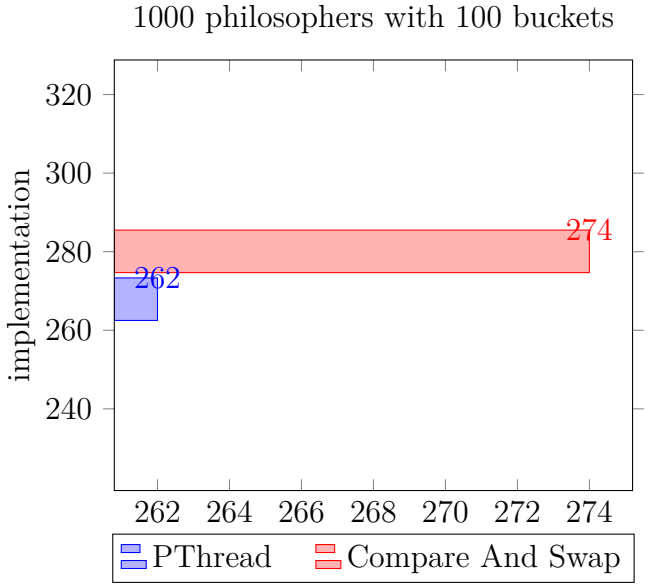
LoLA is executed with the `--threads=[threadCount]` and `--check=full` switches. The first switch simply sets the amount of threads that should be used for the state

exploration. The second switch cause LoLA to explore the whole state space without exploring a property. This ensures that the application will terminate after all states have been discovered and no varying discovery paths can influence the execution time.

4.2.2 A general performance survey

LoLA will spend most of the time inside the depth first search of the state exploration. For this reason a simple execution with the given parameters will uncover if the different synchronization implementations have an impact on the performance. This would correspond to the macro benchmark scope and can be achieved without any additional modification or tools because LoLA already measures its own execution time.

Unfortunately a simple testrun on the VM reveals that there is no performance gain. Figure TODO:ref shows that the new implementation is actually slower than the previous. But as stated in TODO:ref, the performance of the original implementation always stayed behind the sequential algorithm. This means that a relevant speedup would divide the execution time by an order of the used threads. Taking this expectation as base, both implementations still can be considered in the same order of performance. Thus, we can conclude that we either missed the cause of the bottleneck, or the new implementation faces similar challenges.



PThread	Compare And Swap
262s	274s

Table 4.1: Macro benchmark comparison between the original (PThread) implementation and the substitute (Compare and swap)
LoLA call: ”lola –check=full –threads=4 ../tests/testfiles/phils1000.lola”

The benchmark result shows no significant change in performance. Therefore an exhaustive evaluation with multiple testruns is dropped in favor of a detailed bottle neck analysis. In the next section we will use a more precise benchmark method to inspect the search characteristics.

4.2.3 Searching for inefficient application components

To develop a deeper understanding of the internal processes, we have to examine the individual program parts. The most important one remains the search for net states which is done in the `ParallelExploration` class.

LoLA will call the `depth_first()` method of the `ParallelExploration` to initiate the search. There, all search threads will be initialized, started and destroyed (after they finished their work). Starting and destroying the threads takes constant time per thread. Since we will work with very few threads in relation to the amount of states we will handle, these parts are insignificant for the performance measurement and can be ignored. But each thread will execute the `threadedExploration` method. This method is the algorithm for the actual search. It will loop until a given predicate is satisfied by any thread. For these reasons we will focus our measurements on this method.

We now know a precise code section that we want to inspect. Next we have to choose a new measurement method. We can decide between two general approaches: we can use an existing tool that can hopefully inspect the application parts we are interested in, or we can extend our code manually. Both approaches have their up and downsides. In our case we decided to use the manual approach. First because the interesting code is relatively short and clear and second because it can be done right away, without spending much time for learning a third party software.

In the next step measurements have to be inserted at the relevant sections. However, what section is relevant cannot be known before measuring. We have to choose parts that seem to be most likely. This is an inherently subjective process, but some factors will be influential. For example, elemental assignments like `x = 5;` will take an insignificant amount of time, whereas function calls and loops can take arbitrarily long times to be executed. The cost of a bad selection is very low, since the measurements can be easily changed in the short method.

The time was measured with the functions provided by `std::chrono` from the c++ standard library. The exact code is considered trivial and will not be discussed further.

Table TODO:ref is listing the performance of the code pieces that are considered relevant. Here is an explanation of the values:

- Total Thread Time - The cumulative time spend in each thread is called total thread time.
- Synchronization Time - Time spend to synchronize the threads with mutexes.

- Store Search Time - Time spend inside the `searchAndInsert` call. A method to safe the discovered states.
- Idle Time - The Time each thread spent for the `restartSemaphore` to be unlocked

	Trhead 0	**Trhead 1**	**Trhead 2**	**Trhead 3**
Total Thread Time	256s	256s	256s	256s
Synchronization Time	0.00000004s	0.00000004s	0,00000005s	0.00000005s
Store Search Time	58s	58s	56s	58s
Idle Time	0.22s	0.22s	0.22s	0.22s

Table 4.2: Manual meso Benchmark of the `ParallelExploration`

- bottleneck is SearchAndInsert() (the store of the individual thread)

Second Iteration - therefore new benchmark inside the thread local search and insert
- PIC: measurements
- PIC: CodeSnippet
- times seem inkonsistent
- time spent in thread local space is ?longer? then the bracket around the function call
- time inside the own functioncalls is marginal
- additional unnamed measurements did not improve the reliability of the measured values

4.2.4 Result and Conclusion

- time measurements inside threads can be tricky
- measurements of wall clock time vs ?process? time ?CITATION?
- measurement method might not be appropriate
- moving on to another method

4.3 Profiling

4.3.0.1 gprof

4.3.0.2 valgrind

4.3.0.3 perf

4.3.0.4 operf

4.3.1 Choice

- dfs is the main time consumption
- therefore simple program runs are sufficient to messure differences between parallel and sequential implementations (macrobenchmark)
- to find bottlenecks inside the implementation, a more precise approach is necessary
- profilers have limitations
- interpreting the results is not trivial and differ from profiler to profiler
- code to benchmark is fairly compact
- manual instrumentation is time efficient
- manual instrumentation is precise
- manual instrumentation can messure custom metadata like state expansions of each thread

- use of perf ?tools?
- works out of the box
- no significant slowdown
- readable data
- other tested profilers did not work as well or not at all
- kernel feature from linux
- impossible to test every profiler in the time constraints
- profiling on vm as start
- dropping profiling on ebro since no rights and strong hint on bottleneck found like stated below
- measurement of the whole lola execution or a sample inside the search phase
- PIC: CodeSnippet
- snippet explanation
- sample measurements is sufficient data
- measurements create large amount of data (GBs)

4.3.2 Result and Conclusion

- most of the time spent in libcalloc
- PIC: TestData
- memory allocation in OS scope
- function names point to os mutex lock and wait
- to many allocation calls which blocks each other
- custom allocator is needed

4.4 Memory Allocator

- luckily there is a project where author was involved
- mara allocates stack and gives no interface to free allocated memory
- gets chunks of memory from malloc
- simple and fast pointer arithmetic is used to manage the memory stack
- each bucket can have own instance of mara so that it becomes thread safe (since the buckets are already thread safe)
- freeing memory on program termination
- the only data created is needed for the search
- terminating the search is the begin of terminating lola
- thus this method is sufficient for this perpose

4.5 Results

4.5.1 Profiling

- same environment like the previous prof env
- PIC: data
- libcalloc calls are gone -¿ under significance threshold
- mem... calls and fire list calls are now the most significant values
- meaning the greatest bottleneck seems to be gone

4.5.2 Macrobenchmark

- knowing a major bottleneck is gone, execution and scaling of the whole program is of interest
- simple measurement of execution time is sufficient
- since multiple threads will influence time consumption directly

- lola can log that itself
- all implementations are benchmarked (master, CAS, MaraCas, MaraPthread)
- significant values are: single thread performance, scaling with threads, scaling with bucket count
- PIC: measurements

4.6 Evaluation

5 Conclusion

- parallel dfs is not trivial
- messure the performance bottlenecks before fixing the performance!
- location of bottlenecks quickly become a matter of faith
- do not base actions on faith but on evidence, time and effort can be saved
- use tools which address the own context
- lola has potential in data representations in the context of parallelizations

- extension of single threaded class means that it is unlikely to beat the former single thread performance

- using tools can be frightening might also be worth it

6 Future Work

- messure performance from searchAndInsert
- find the limiting parts which hinder parallelization from using all threads efficiently
- optimize at least one implementation from search and insert for parallel use
- if the bottleneck returns to the exploration after optimizing:
- find a reasonable heuristic to share work between threads
- maybe don't let every single thread look for other idle ones, instead consider implementing a scheduler thread, or a data structure with possible tasks

- the synchronization method should be considered in the parallel design of the algorithm (dont emulate mutexe if using compare and swap)

Bibliography

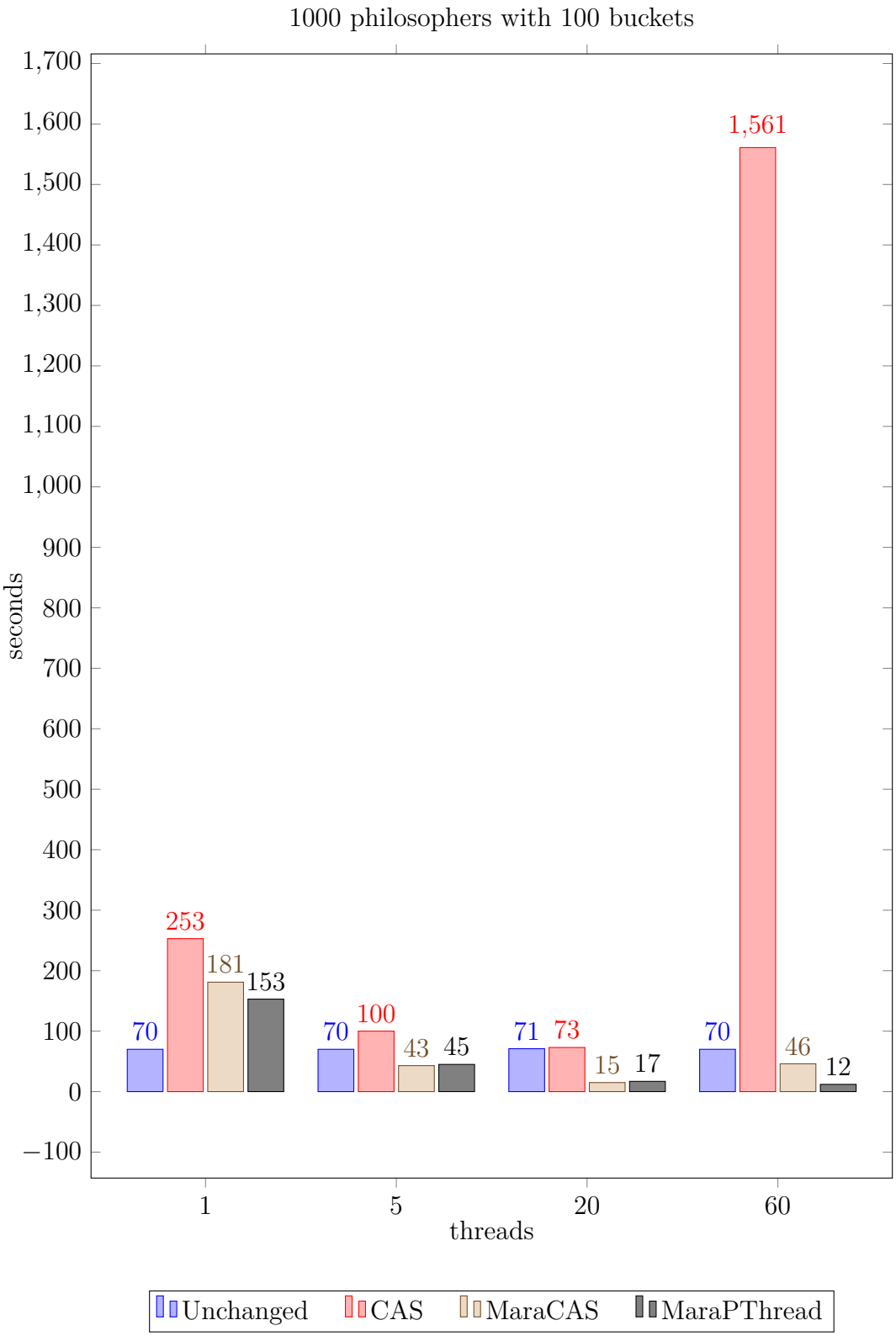
[Dij71] Edsger W Dijkstra. Hierarchical ordering of sequential processes. In The origin of concurrent programming, pages 198–227. Springer, 1971.

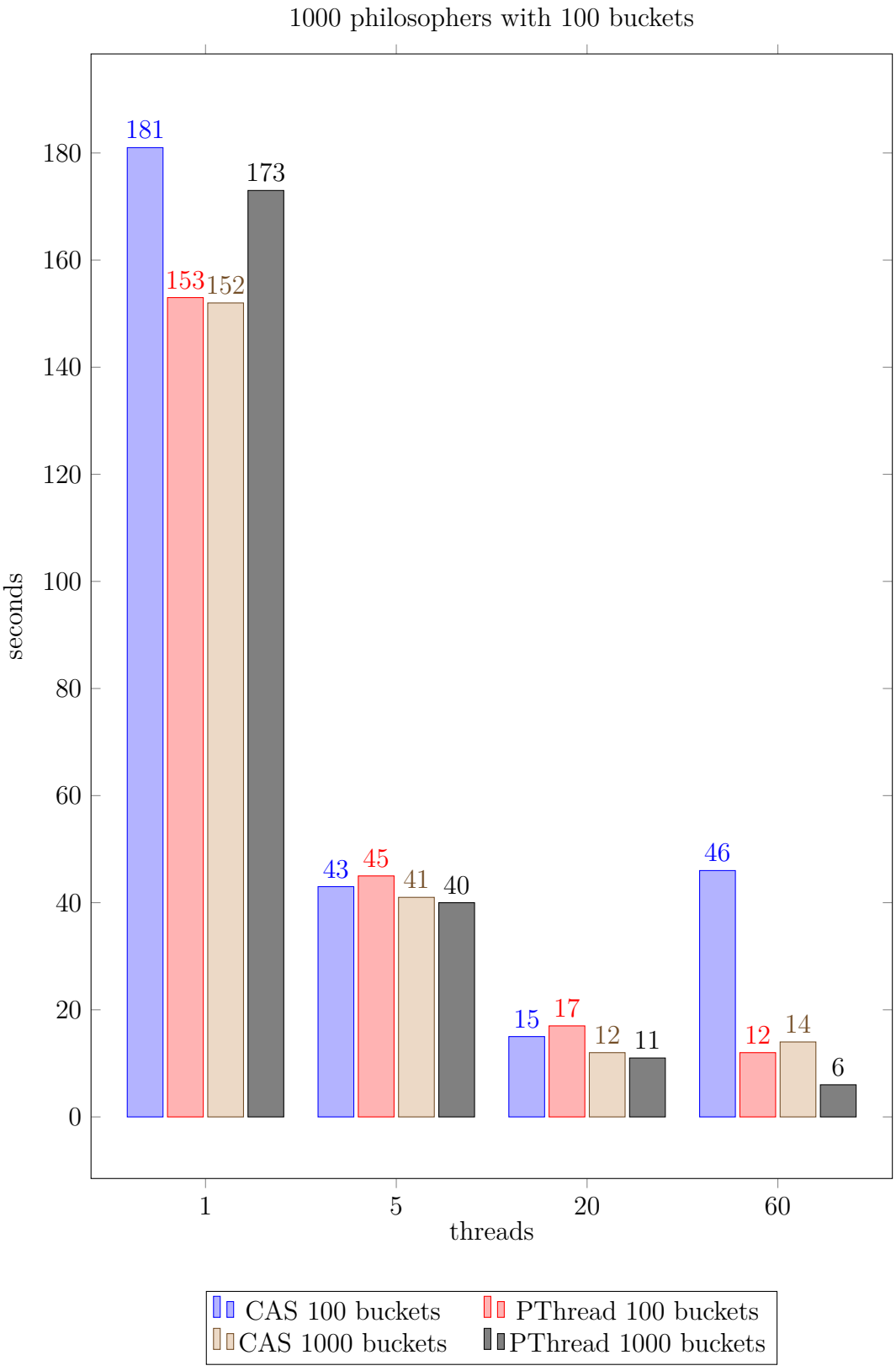
[FK17] D. Buchs F. Kordon. Model checking contest results 2017, 2017. Accessed: 2018-11-01.

[Inc18] Advanced Micro Devices Inc. Amd opteron 6284 se specification, 2018. Accessed: 2018-15-01.

[Oak14] Scott Oaks. Java Performance: The Definitive Guide: Getting the Most Out of Your Code. " O'Reilly Media, Inc.", 2014.

[Sch00] Karsten Schmidt. Lola a low level analyser. Application and Theory of Petri Nets 2000, pages 465–474, 2000.


1000 philosophers with 100 buckets
seconds
threads
1,700
1,600
1,500
1,400
1,300
1,200
1,100
1,000
900
800
700
600
500
400
300
200
100
0
−100
1
5
20
60
70
253
181
153
70
100
43
45
71
73
15
17
70
1,561
46
12
Unchanged
CAS
MaraCAS
MaraPThread


1000 philosophers with 100 buckets
seconds
0
20
40
60
80
100
120
140
160
180
181
153
152
173
43
45
41
40
15
17
12
11
46
12
14
6
1
5
20
60
threads
CAS 100 buckets
PThread 100 buckets
CAS 1000 buckets
PThread 1000 buckets

Eidesstattliche Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Rostock, 02. März 2018

_____

Tom Meyer