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 1

List of Abbreviations 1

2 Related Work 2
2.1 Parallelization History . . . 2
2.2 Synchronization Methods . . . 2
2.3 Depth First Search Parallelization . . . 2

3 Background 3
3.1 Implementation Details . . . 3
3.1.1 Code Base . . . 3
3.1.2 Data Synchronization . . . 4
3.2 Environment . . . 5
3.3 Performance Measurement . . . 6
3.3.1 Benchmarking Scopes And Methods . . . 6
3.3.2 Profilers . . . 7
3.3.3 Choice . . . 7

4 Approach 8
4.1 Switching The Synchronization . . . 8
4.2 Manual Instrumentation . . . 8
4.2.1 First Iteration . . . 8
4.2.2 Second Iteration . . . 8
4.2.3 Result and Conclusion . . . 9
4.3 Profiling . . . 9
4.3.1 Approach . . . 9
4.3.2 Result and Conclusion . . . 9
4.4 Memory Allocator . . . 10
4.5 Results . . . 10
4.5.1 Profiling . . . 10
4.5.2 Macrobenchmark . . . 10
4.6 Evaluation . . . 10

5 Conclusion **11**

6 Future Work **12**

Bibliography **13**

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

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 Implementation Details

3.1.1 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.

3.1.2 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.

3.2 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

3.3 Performance Measurement

3.3.1 Benchmarking Scopes And Methods

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 metric 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 metrics that define the performance of exactly that given program or subroutine. Other metrics 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 and an overview, the last scope has to be something in between. And this holds tre for the so called meso benchmark. Perhaps the best explanation for this kind is, that it 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 almost always require to manipulate the program in some way. Which again means that the measured performance differs from the actual performance.
- microbenchmarks
- macrobenchmarks
- profiling
- "das ganze kann noch ausgedehnt werden, reicht aber erstmal"

3.3.2 Profilers

- endless possibilities
- testing all profilers is nearly impossible in the time frame
- common tools identified through internet research

3.3.2.1 gprof

3.3.2.2 valgrind

3.3.2.3 perf

3.3.2.4 operf

3.3.3 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

4 Approach

4.1 Switching The Synchronization

- pthread to mutex

4.2 Manual Instrumentation

4.2.1 First Iteration

- benchmark on own vm and ebro - PIC: vm stats - PIC: ebro stats - meassuring of time
- mesobenchmark - benchmarking inside search and insert of parallel exploration
- search loop begins there - majority of the time spent there
- 1000 philosopher net as example with a full check
- net is reasonable complex to keep machine busy for a reasonable amount of time
- net is not a to specialized case
- net is a commonly used example and therefore known to a decent part of the community
- PIC: measurements - no significant speedup
- threads distribute state expansions evenly
- idle thread times are not significant
- bottleneck is SearchAndInsert() (the store of the individual thread)

4.2.2 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

4.2.3 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

4.3.1 Approach

- 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

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

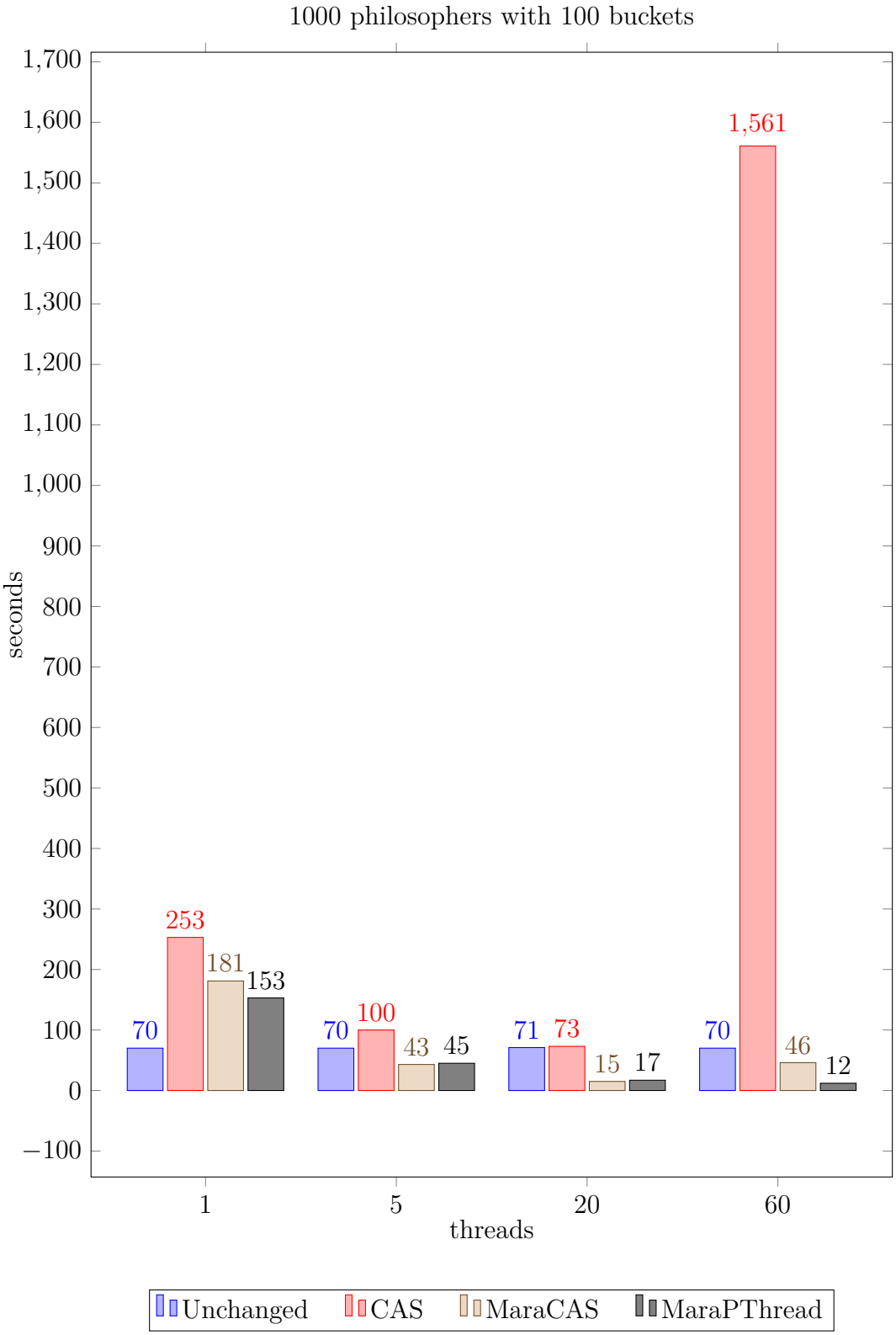
Bibliography

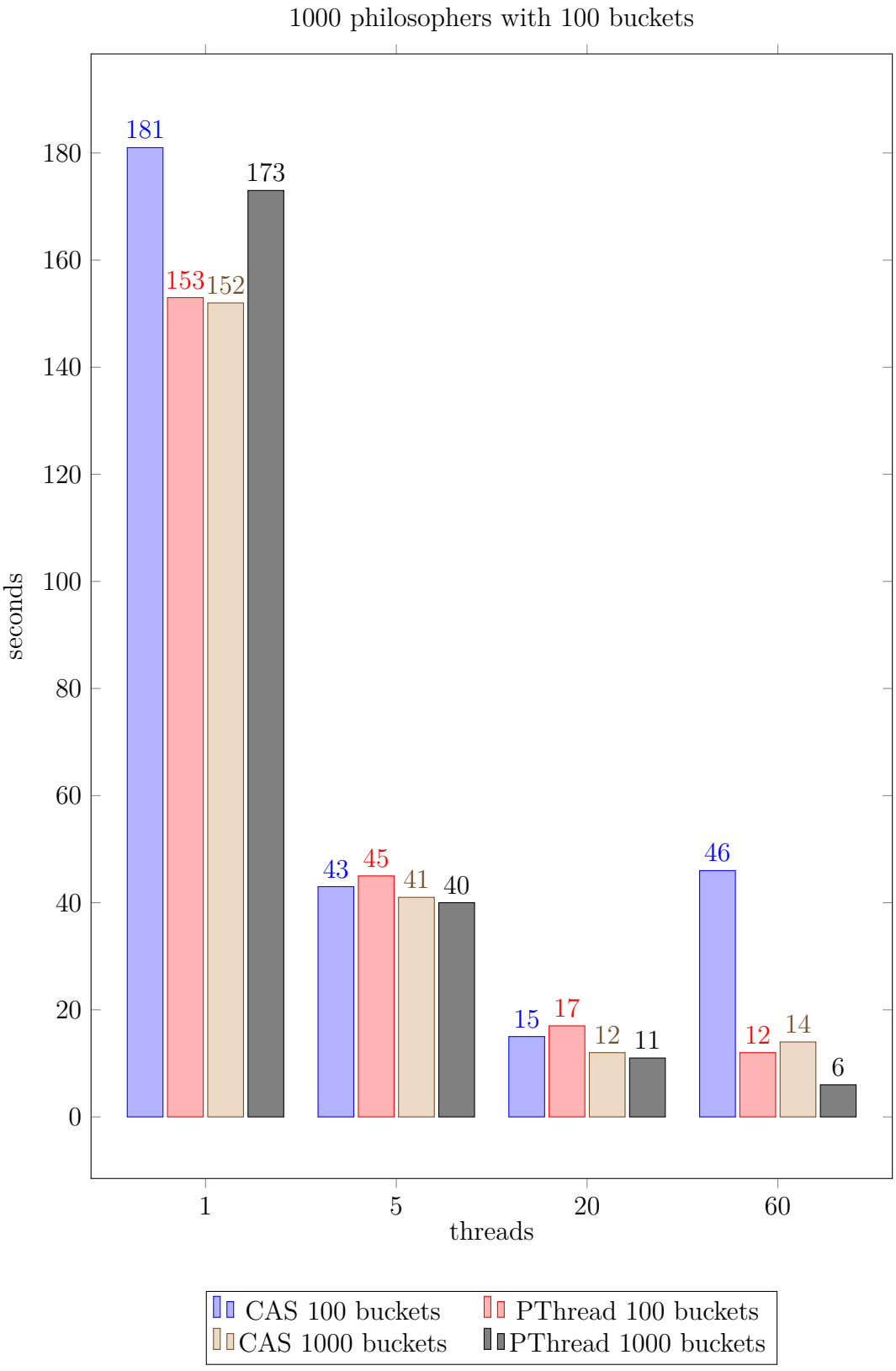
[FK17] D. Buchs F. Kordon. Model checking contest results 2017, 2017. Accessed: 2018-11-01.

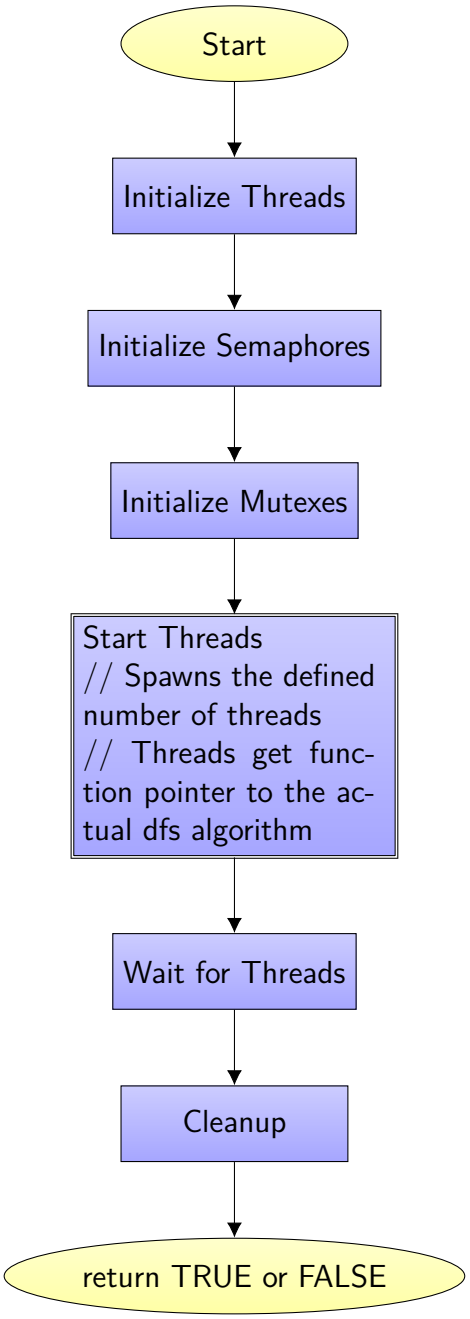
[Inc18] Advanced Micro Devices Inc. Amd opteron 6284 se specification, 2018. Accessed: 2018-15-01.

[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



ParallelExploration	
restartSemaphores	:semaphore[ ]
finished	:bool
numSuspended	:mutex
suspendedThreads	:arrayIndex
searchStacks	:SearchStack<>[ ]
threadNetStates	:NetState[ ]
goalProperties	:SimpleProperty[ ]
globalProperty	:SimpleProperty*
globalBaseFireList	:FireList*
globalStore	:Store<void>*
globalProperty	:mutex
threads	:thread[ ]
numberOfThreads	:int

ThreadArguments	
threadNumber	:int
parent	:ParallelExploration


Start
Return satisfying NetState or null
Initialization
Property satisfied?
yes
Successive cleanup
Aborting cleanup
no
Store net state
Get FireList with the current index
yes
Other thread finished?
no
yes
Transition available?
no
yes
Fire current transition
Backfire transition
yes
State exists?
no
State satisfies property?
yes
no
Safe FireList
Get new FireList
no
Can hand a transition to another thread?
yes
yes
Lock Mutex
Any thread suspended?
no
Unlock Mutex
Delete FireList
yes
Get any suspended thread
Stack is empty?
yes
no
Unlock Mutex
Pop new FireList
Copy exploration data for new thread
Backfire transition and update enabledness
Increment Semaphore
Update result
Add self to suspended threads
Other thread finished?
no
Unsafe FireList
Self is last thread?
no
Backfire and return to original state
yes
Set finished
Wake all other threads
Decrease semaphore
Is Finished?
no

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