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

Abstract

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
- locks

2.3 Depth First Search Parallelization

3 Approach

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 Relevant Code

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 the single threaded algorithm in the `DFSExploration` class, since the basic structure is is similar and several lines of code and comments are ...Deckungsgleich?...

- code was handed over from previous developers
- was Based on the existing single threaded exploration
- subclass of a general exploration class
- search class is determent by the call switches of lola

- perpose: get performance boost with multiple threads
- challenge: sync overhead and load distribution

- threads get local and shared data
- shared data has to be thread safe
- ?which data?
- parallel exploration is mainly sync
- actual work is done in a thread local single threaded search

- another challenge: searchandinsert inside global data
- threadsadety accomplished with netstate has and threadsafe buckets

3.1.2 Swapping the Synchronization

- swap pthread with compare and swap
- get a fitting CAS implementation
- find all mutexes and semaphores
- replace all pthread calls with CAS calls

3.2 Environment

- ebro
- vm

3.3 Benchmark

3.3.1 Benchmark Types

- microbenchmarks
- macrobenchmarks
- profiling

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

3.4 Manual Instrumentation

3.4.1 First Iteration

- benchmark on own vm and ebro - PIC: vm stats - PIC: ebro stats - 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 SearechAndInsert() (the store of the individual thread)

3.4.2 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

3.4.3 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

3.5 Profiling

3.5.1 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)

3.5.2 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

3.6 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

3.7 Results

3.7.1 Profiling

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

3.7.2 Macrobenchmark

- knowing a major bottleneck is gone, execution and scaling of the whole program is of interest
- simple measuremnt of execution time is sufficiant
- 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

3.8 Evaluation

4 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

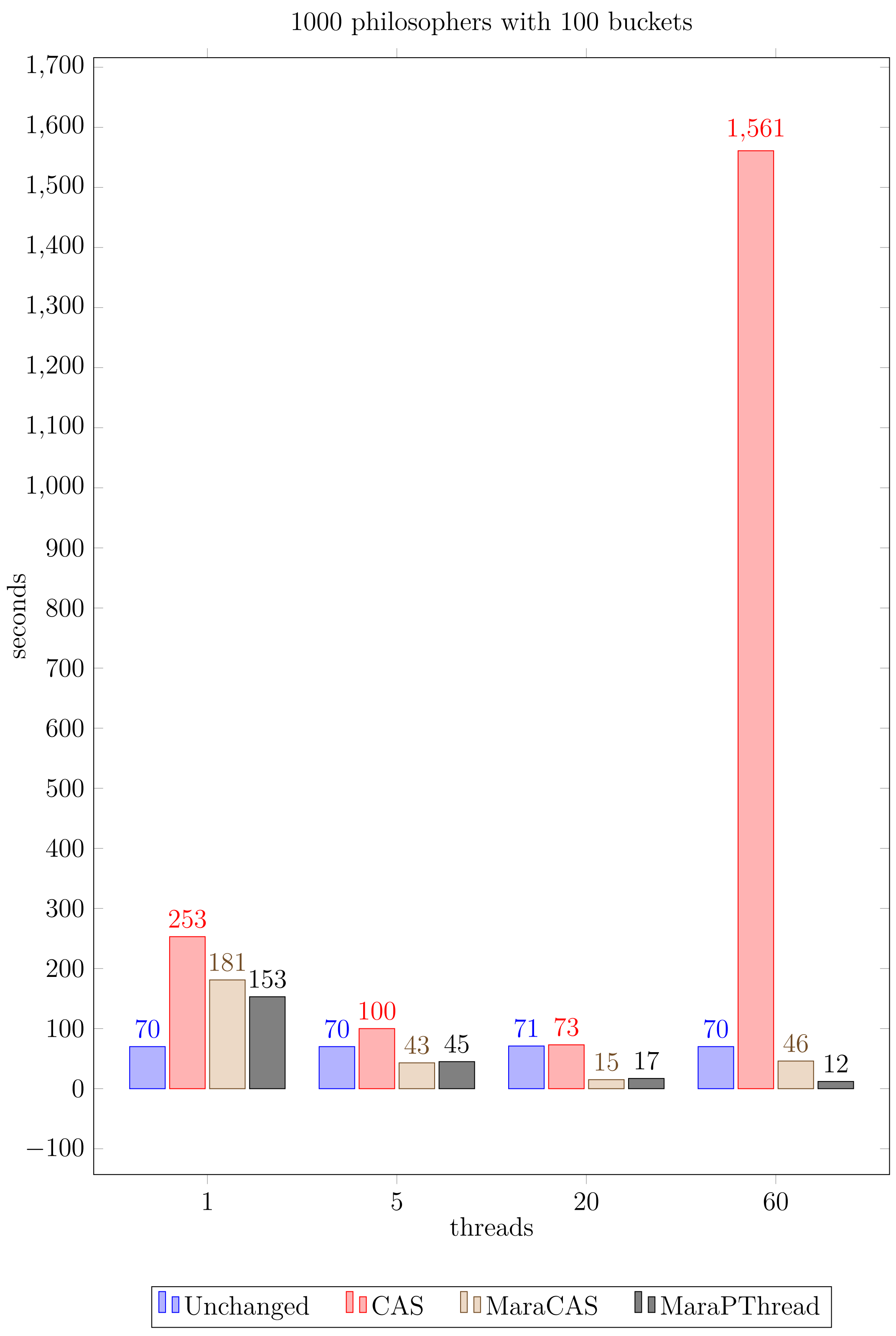
5 Future Work

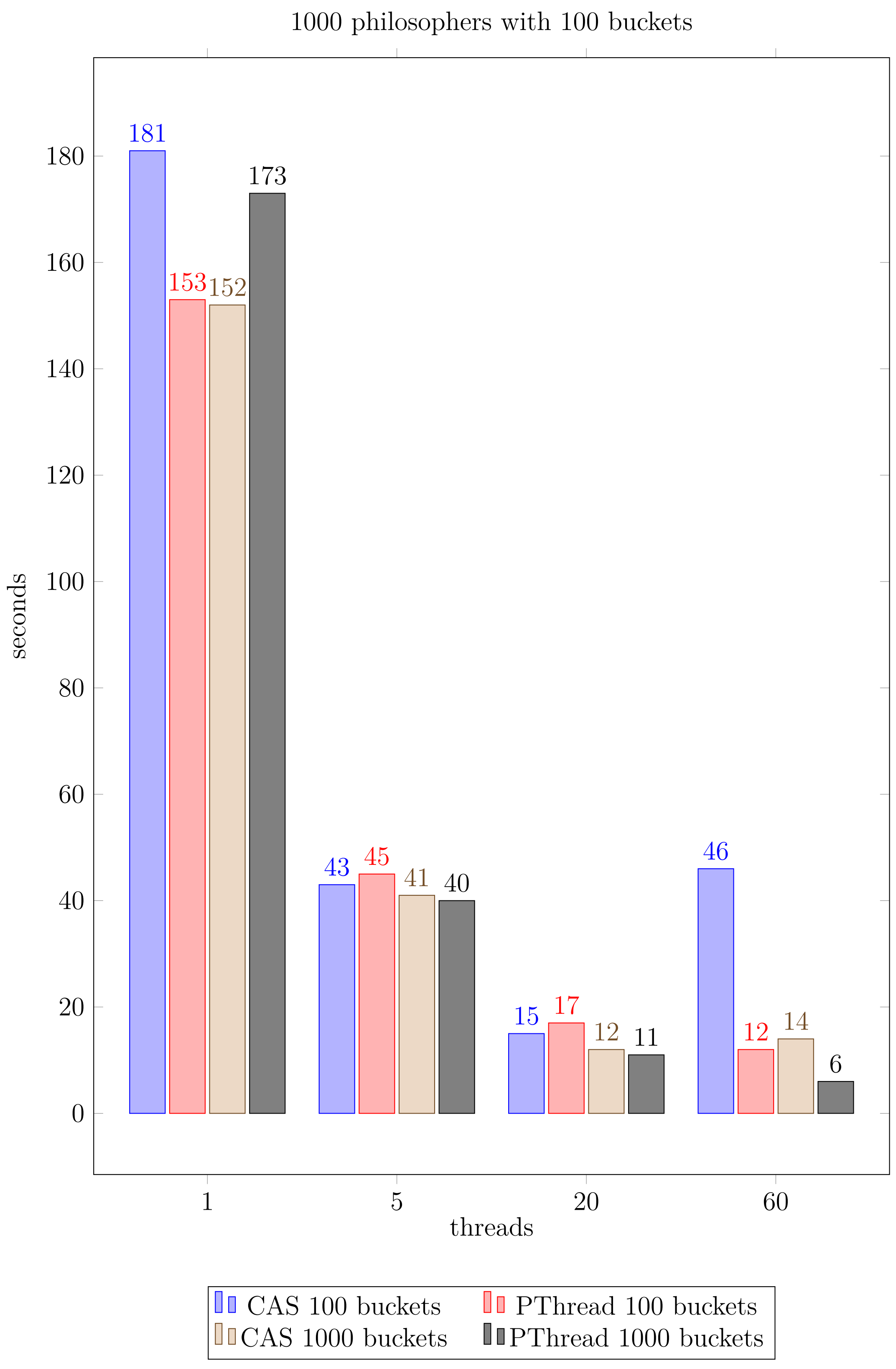
- messure performance from searchAndInsert
- find the limiting parts which hinder parallelization from using all threads efficiently
- optimize at least one implementation from search and insert for parallel use
- if the bottleneck returns to the exploration after optimizing:
- find a reasonable heuristic 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

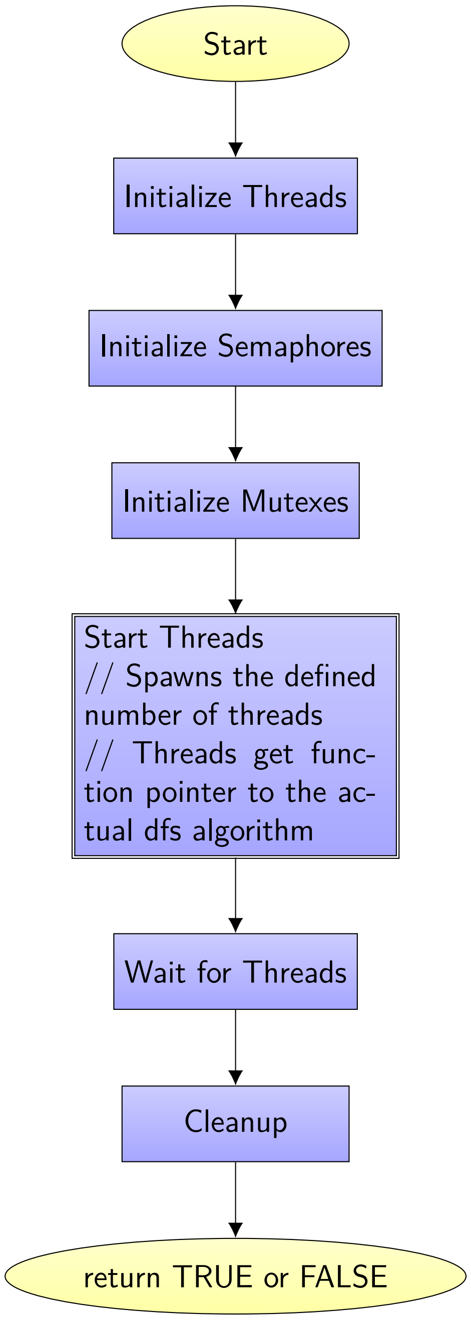
Bibliography

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

[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
threads
0
20
40
60
80
100
120
140
160
180
1
5
20
60
181
153
152
173
43
45
41
40
15
17
12
11
46
12
14
6
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
Can hand a transition to another thread?
no
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