Scalable Multi-core Model Checking: Technology & Applications of Brute Force
Day I: Reachability

Jaco van de Pol
30, 31 October 2014

VTSA 2014, Luxembourg
# Table of Contents

## 1 Introduction
- The case for high-performance model checking
- LTSmin tool architecture and PINS interface
- Course Overview

## 2 Multi-core Reachability
- Shared hash table
- Parallel state compression
The Reachability Problem

Reachability Problem – Instances:

- Find assertion violations in multi-core software
- Find safety risks in Railway Interlockings
- Find solutions to games/puzzles, e.g. Sokoban

The Reachability Problem in general graphs

- Given a graph $G = (V, R)$ (nodes, edges)
- Initial states $I \subseteq V$ and goal/error states $F \subseteq V$
- Check: is there a path in $G$ from $I$ to $F$? i.e. is $F$ reachable?

- Typically, the graph is given implicitly, as the state space of a program or a specification.
Reasons for State Space Explosion

Concurrency: *exponential* growth

- System of $n$ components, each can be in $m$ states
- The total state space may consist of $m^n$ states.
- Example: Railway safety systems (signals, points, tracks)

Data variables: *exponential* growth

- Given $n$ different variables, each may take $m$ values
- Potential number of different state vectors: $m^n$
- Example: model checking software, rather than models

How to handle $> 10^{100}$ states??

- **Partial Order Reduction**: Avoid certain states systematically
- **Symbolic model checking**: Treat sets of states simultaneously
- **Focus of my lectures**: Brute force parallel computation
Solution to State Space Explosion?

- Model checking suffers from the state space explosion, therefore it is very time and memory intensive.
- Reaching the memory bound is an immediate show stopper, but also excessive waiting times put a bound on applicability.
- Why not simply throw more computer power at the problem?

Will this help in practice? Is this scientifically interesting?

- Is the problem embarrassingly parallel?
- No: Graph algorithms are not easy to parallelize efficiently, so clever algorithm engineering is necessary.
- But: only linear improvement for an exponential problem...
- Yes, orthogonal to clever reduction techniques: start simple...
Various possibilities regarding underlying hardware

**Distributed computing:**
- network of workstations, clusters, Grid - cheap
- this allows accumulation of available memory
- **But:** limited *bandwidth*, high *latency*

**Parallel computing (shared memory):**
- Multi-core, supercomputers - expensive, but price dropping
- 64-bit machines, > 120GB RAM, 8-64 cores: quite popular
- **But:** Scalability is imperfect, heterogeneous (so distributed?)

**Several alternatives are under investigation:**
- Use hard disk as substitute for RAM
- CUDA (GPU), Cell processors, FPGA, cloud, map/reduce

**In all cases:** algorithms must be fundamentally revised!
Table of Contents

1 Introduction
   - The case for high-performance model checking
   - LTSmin tool architecture and PINS interface
   - Course Overview

2 Multi-core Reachability
   - Shared hash table
   - Parallel state compression
# Model Checking made Practical and Widespread?

## Main obstacles

- **Scalability**
  - parallel components
  - data, buffers, ...
- **Modeling effort**
  - many languages
  - avoid modeling?
- **Complex tools**
  - algorithms, heuristics
  - low-level details

## Algorithmic solutions (combinatorics: locality)

- on-the-fly model checking
- symbolic model checking
- bounded model checking
- partial-order reduction
- symmetry reduction
- parallel model checking

### Problem: algorithms are often tied to specification languages

- No particular technique suits all applications / models
- A user needs to rewrite his model into different languages
Solution Direction

Where to draw the line?

- Separate languages and algorithms via a clean interface (API)
- API should be simple: allow many different languages
- API should be rich: expose locality structure to algorithms

**PINS interface of LTSmin toolset:**

- Frontends provide on-the-fly access to a state space
- Backend algorithms determine the verification strategy
Introduction

Multi-core Reachability

High-performance Model Checking for the Masses

Specification Languages

<table>
<thead>
<tr>
<th>mCRL2</th>
<th>Promela</th>
<th>DVE</th>
<th>UPPAAL</th>
</tr>
</thead>
</table>

Reachability Tools

<table>
<thead>
<tr>
<th>Distributed</th>
<th>Multi-core</th>
<th>Symbolic</th>
</tr>
</thead>
</table>

Advantages of tool and interface (LTSmin / PINS)

- General and flexible: support for arbitrary state/edge labels
  - Also: LLVM, parity games, Markov Automata, C-code, B||CSP
  - Indirectly: GSPN, xUML, Signalling Networks in Biology
- On-the-fly API: next-state function to pull the implicit graph
- Efficiency: models expose locality in a dependency matrix
LTSmin architecture and PINS interface

Blom, van de Pol, Weber [CAV’10], Laarman, van de Pol, Weber [NFM’11]
http://fmt.cs.utwente.nl/tools/ltsmin/

Specification Languages
- mCRL2
- Promela
- DVE
- UPPAAL

PINS
- Transition caching
- Variable reordering
- Transition grouping
- Partial-order reduction

Pins2pins Wrappers

Reachability Tools
- Distributed
- Multi-core
- Symbolic

Analysis Algorithms
- Bisimulation reduction / lumping
- LTL
- mu-calculus
1 Introduction
   - The case for high-performance model checking
   - LTSmin tool architecture and PINS interface
   - Course Overview

2 Multi-core Reachability
   - Shared hash table
   - Parallel state compression
# Lecture on High-performance Model Checking

## High-level Goals

- Investigate high-performance model checking algorithms
- Applications to complex man-made and natural systems

## Ingredients

- Basic multi-core datastructures for Reachability
- Checking liveness properties – LTL, multi-core Nested DFS
- Symbolic representation: LTL for Timed Automata
- Symbolic representation: Multi-core Decision Diagrams
- Application to Biological Signaling Pathways
- Application to xUML diagrams for Railway Safety
Synthesizing a medicine could be a reachability problem...
Table of Contents

1 Introduction
   - The case for high-performance model checking
   - LTSmin tool architecture and PINS interface
   - Course Overview

2 Multi-core Reachability
   - Shared hash table
   - Parallel state compression
Which architecture suits Multi-core Model Checking?

Static partitioning
- Distributed memory solution
- Communication: $W^2$ queues
- (Relaxed) BFS only

Shared hash table
- (Pseudo) DFS & BFS
- Communication: shared hash table
- Load balancing
**Algorithm: parallel reachability**

**Data:** Global set $V = \emptyset$, Local sets $S_0 = I, S_1 = \cdots = S_{N-1} = \emptyset$

for $0 \leq id < N$ do in parallel

while $\text{LoadBalance}(S_{id})$ do

while some work to do and no timeout do

state $\leftarrow S_{id}.\text{Get}()$

$\text{count} \leftarrow 0$

check invariants on state

for $s \in \text{NextState}(state)$ do

increment $\text{count}$

if not $V.\text{FindOrPut}(s)$ then

$S_{id}.\text{Put}(s)$

if $\text{count} = 0$ then report deadlock

(1) “Open” set $S$ influences search order (e.g.: BFS, DFS)

(2) Shared-Memory synchronization point

- Locking the hashtable is not an option
### Lockless Hash Table: Design

**Alfons Laarman, van de Pol, Weber [fmcad10]**

<table>
<thead>
<tr>
<th>Main bottlenecks for scalable implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ State storage: requires concurrent access (lock contention)</td>
</tr>
<tr>
<td>▶ Graph traversal: random memory access (bandwidth)</td>
</tr>
<tr>
<td>▶ Computer architecture: shared L2 caches (false sharing)</td>
</tr>
</tbody>
</table>

### Design: keep it simple

- Open addressing
- Hash memoization: read less data
- Separate hash and data
- On collision: **Walking the Line**
- In-situ locking (1 bit per bucket)
- Bucket operations require CAS
- Not strictly wait-free
Algorithm: multi-core **FindOrPut**

**Input**: state  
**Output**: true if seen, false otherwise  
**Data**: size, Bucket[size], Data[size]

1. \( h \leftarrow \text{Hash}(\text{state}); \ \text{index} \leftarrow h \mod \text{size} \)
2. for \( i \) in WalkTheLineFrom(index) do
   3. if \( \text{empty} = \text{Bucket}[i] \) then
      4. if \( \text{CompareAndSwap}(\text{Bucket}[i], \text{empty}, \langle h, \text{write} \rangle) \) then
         5. \( \text{Data}[i] \leftarrow \text{state} \)
         6. \( \text{Bucket}[i] \leftarrow \langle h, \text{done} \rangle \)
         7. return false
   8. if \( \langle h, ? \rangle = \text{Bucket}[i] \) then
      9. while \( \langle ?, \text{write} \rangle = \text{Bucket}[i] \) do . . . wait . . .
     10. if \( \text{Data}[i] = \text{state} \) then return true
Scalability Experiments from 2010 (BEEM database)

Barnat (2007)

- “our shared hash tables do not scale beyond 8 cores”
- “could not investigate lockless hash table solution”
- “haven’t found the cause of the scalability issues”

SPIN 5.2.4 (NASA/JPL)

DiVinE 2.2 (Brno, CZ)

LTSmin (U Twente, NL)
Table of Contents

1. Introduction
   - The case for high-performance model checking
   - LTSmin tool architecture and PINS interface
   - Course Overview

2. Multi-core Reachability
   - Shared hash table
   - Parallel state compression
State space compression

Where is the bottleneck for parallel reachability?

- In every step: read and write long state vectors
- Memory: puts an upper limit to the state space
- Time: memory bus becomes the bottleneck for speedup

Exploit locality

- Due to locality: subsequent state vectors have a lot of overlap
- The set of state vectors can be greatly compressed
- Requirement: quick check if a state has been visited
- (otherwise the specification is a very good compression)
Recursive indexing (Tree Compression)
Blom, Lisser, van de Pol, Weber [PDMC’07, JLC’09]

\[ H_K \]

\[(K - 1) \times H_2 \]

Analysis

- Locality \(\implies\) balanced tree \((N + 2\sqrt{N} + 4\frac{4}{4}(N) \cdots \approx N)\)
  Compresses states of length \(K\) to almost 2 (!)
- Hard to parallelize:
  - Sequential operation on tree of tables
  - Many small (variable size) hash tables
**Solution**

- **Reuse lockless hash table**: merge tree of tables into one
- **Incremental updates**: use the Dependency Matrix
  - \((K - 1) \rightarrow \log_2(K - 1)\) lookups

\[
\begin{array}{c|c|c|c|c|c}
4 & 1 & & & & \\
6 & 5 & & & & \\
1 & 3 & & & & \\
3 & 5 & & & & \\
2 & 4 & & & & \\
\end{array}
\]

\[
\langle 3, 5, 5, 4, 1, 3 \rangle \rightarrow \langle 3, 5, 9, 4, 1, 3 \rangle
\]

\[
\begin{array}{c}
\langle 3, 5, 5 \rangle \\
6 & 5 \\
1 & 3 \\
3 & 5 \\
4 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
\langle 4, 1, 3 \rangle \\
2 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
\langle 3, 5 \rangle \\
\langle 4, 1 \rangle \\
\end{array}
\]

\[
\begin{array}{c}
? & 4 \\
6 & 9 \\
\end{array}
\]

\[
\langle 3, 5 \rangle
\]

\[
\langle 4, 1 \rangle
\]
Exploiting locality once more

Dependency Matrix $D_{M \times N}$ predicts changing state parts:

- Incremental tree insertions:
  - Traverse only the changing paths in the Tree of Tables
  
  $H_x \oplus Z_{g,1} \oplus Z_{f,3} = H_y$

- Even further compression:
  - J.G. Cleary (1984): infer part of hash value from its address
  - Vegt/Laarman (2012): Parallel Compact Hash Table
- Can now compress $2^{35} = 3.4 \cdot 10^{10}$ states into 160GB
Compression Experiments from 2011 [BEEM database]
Laarman, van de Pol, Weber [spin11]

- Tree compression is a recursive variant of SPIN’s COLLAPSE (’97)
- Exploit combinatorial structure:
  - State vectors are highly similar
  - Impressive compression ratios
- Extreme case: firewire_tree
  Uncompressed: 14 GB
  Tree Compression: 96 MB
- Compression comes for free
  - Arithmetic intensity increases
  - Less memory-bus traffic
Literature on LTSmin (reachability)

LTSmin toolset

- Stefan Blom, Jaco van de Pol, Michael Weber, LTSmin: Distributed and Symbolic Reachability ............... (CAV 2010)
- Alfons Laarman, Jaco van de Pol, Michael Weber, Multi-Core LTSmin: Marrying Modularity and Scalability .... (NFM 2011)

Reachability and State Compression

- Alfons Laarman, Jaco van de Pol and Michael Weber, ... (FMCAD 2010) Boosting Multi-Core Reachability Performance with Shared Hash Tables
- Alfons Laarman, Jaco van de Pol, Michael Weber, Parallel Recursive State Compression for Free ............... (SPIN 2011)
- Steven van der Vegt, Alfons Laarman, A Parallel Compact Hash Table ......................... (MEMICS 2011)