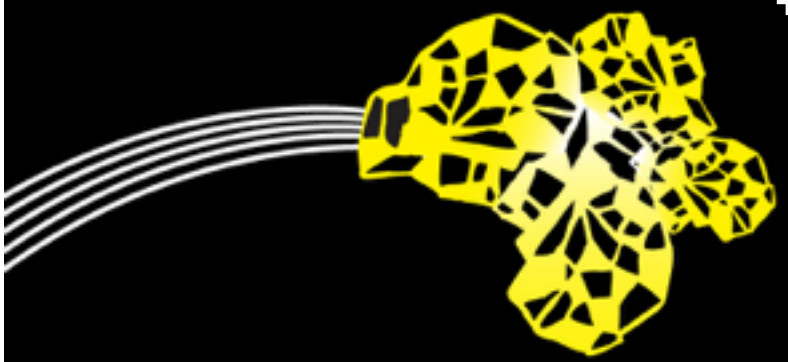
UNIVERSITY OF TWENTE.

HIGH-PERFORMANCE SYMBOLIC MODEL CHECKING
– 10 YEARS OF LTSmin –

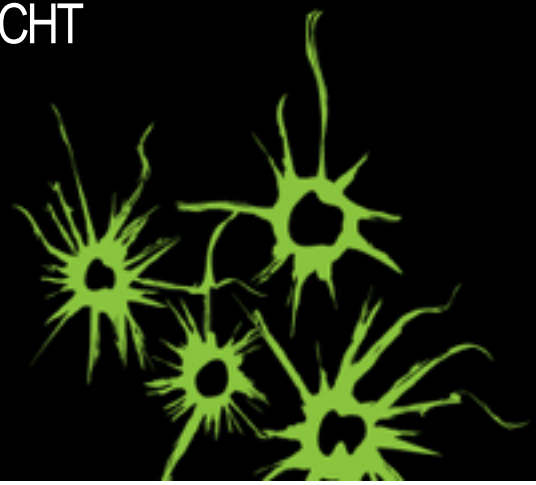
JACO VAN DE POL

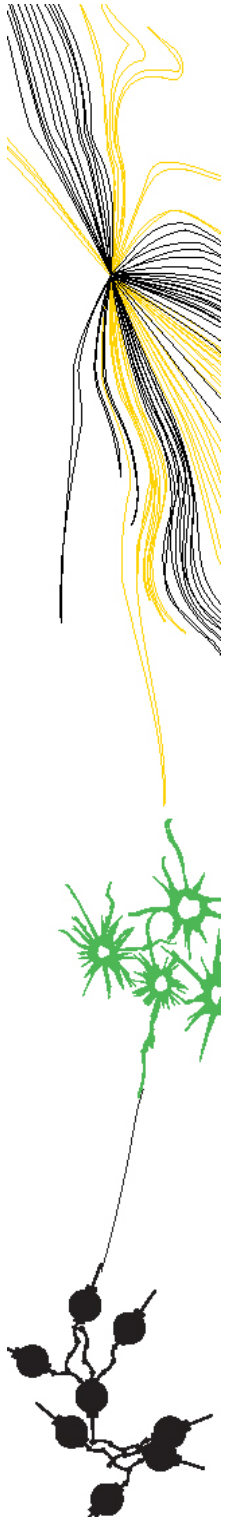
VINCENT BLOEMEN, STEFAN BLOM, TOM VAN DIJK, GIJS KANT, JEROEN KETEMA, ALFONS LAARMAN, JEROEN MEIJER, WYTSE OORTWIJN, MICHAEL WEBER

+ MANY MSC AND EVEN A FEW BSC STUDENTS



DMCD 2018, UTRECHT





OVERVIEW

HIGH-PERFORMANCE SYMBOLIC MODEL CHECKING

1. Introduction: High-performance Model Checking – 10 years LTSmin
   - Motivation of a research line
   - LTSmin overview
2. Multi-core and Distributed Decision Diagrams
   - Sylvan: multi-core BDDs, MDDs, MT-DDs
   - Distributed Hardware Architecture
3. Applications in Symbolic Verification
   - State Space Generation: Symbolic Reachability
   - Symbolic Bisimulation Reduction
   - Solving Parity Games Symbolically

MAIN CHALLENGE: STATE SPACE EXPLOSION

DOES PARALLEL MODEL CHECKING MAKE SENSE?

Many techniques to combat state space explosion:

- Abstraction and Refinement (CEGAR)
- Partial-Order Reduction (POR)
- State Space Reductions (bisimulation, symmetry)
- Symbolic Methods (BDD, SAT, SMT)

Here: **Parallel and Distributed Computing**

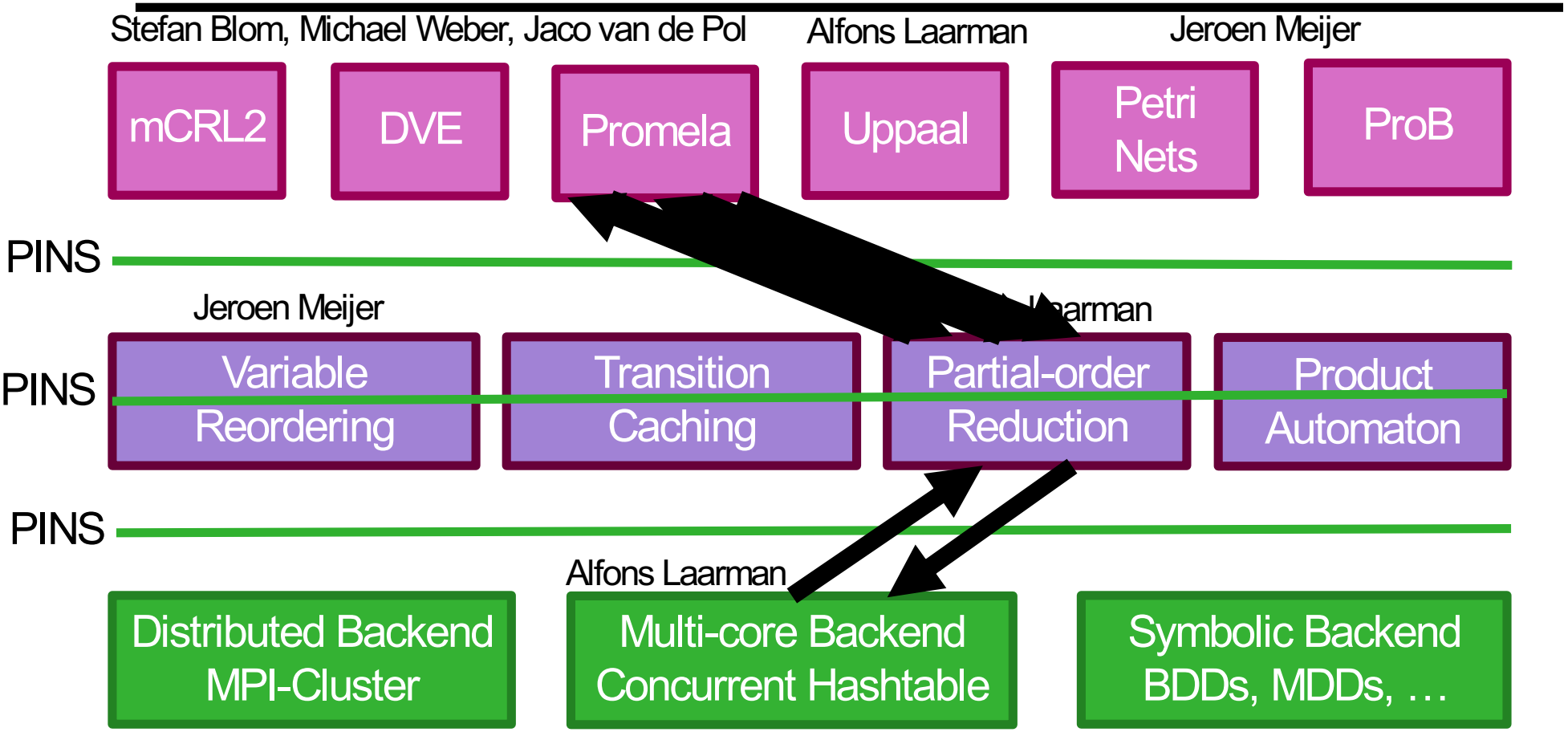
Why does this make sense?

10 YEARS OF LTSMIN: ACCORDING TO GIT LOG

- Fixed Makefile
  - Author: Stefan Blom <sccblom@gmail.com>
  - Date: Sun Sep 14 10:46:57 2008 +0200
- First step in using MPI_IO
  - Author: Stefan Blom <sccblom@gmail.com>
  - Date: Sun Sep 14 21:23:12 2008 +0200
- En documentatie
  - Author: Stefan Blom <sccblom@gmail.com>
  - Date: Fri Sep 19 18:14:48 2008 +0200
- MacOS X port
  - Author: Michael Weber <michaelw@foldr.org>
  - Date: Thu Oct 16 17:31:48 2008 +0200

LTSMIN – PARALLEL MODEL CHECKER

PINS: A PARTITIONED INTERFACE FOR THE NEXT-STATE FUNCTION



ltsmin.utwente.nl github.com/fmt-utwente/ltsmin

PROPERTY CHECKING WITH LTSMIN

STATE- AND ACTION-BASED, TIMED, PROBABILITIES

- Reachability Analysis (all backends)

```
--deadlock
--action="assert"
--invariant="p>0"
```

- Explicit multi-core LTL model-checking:

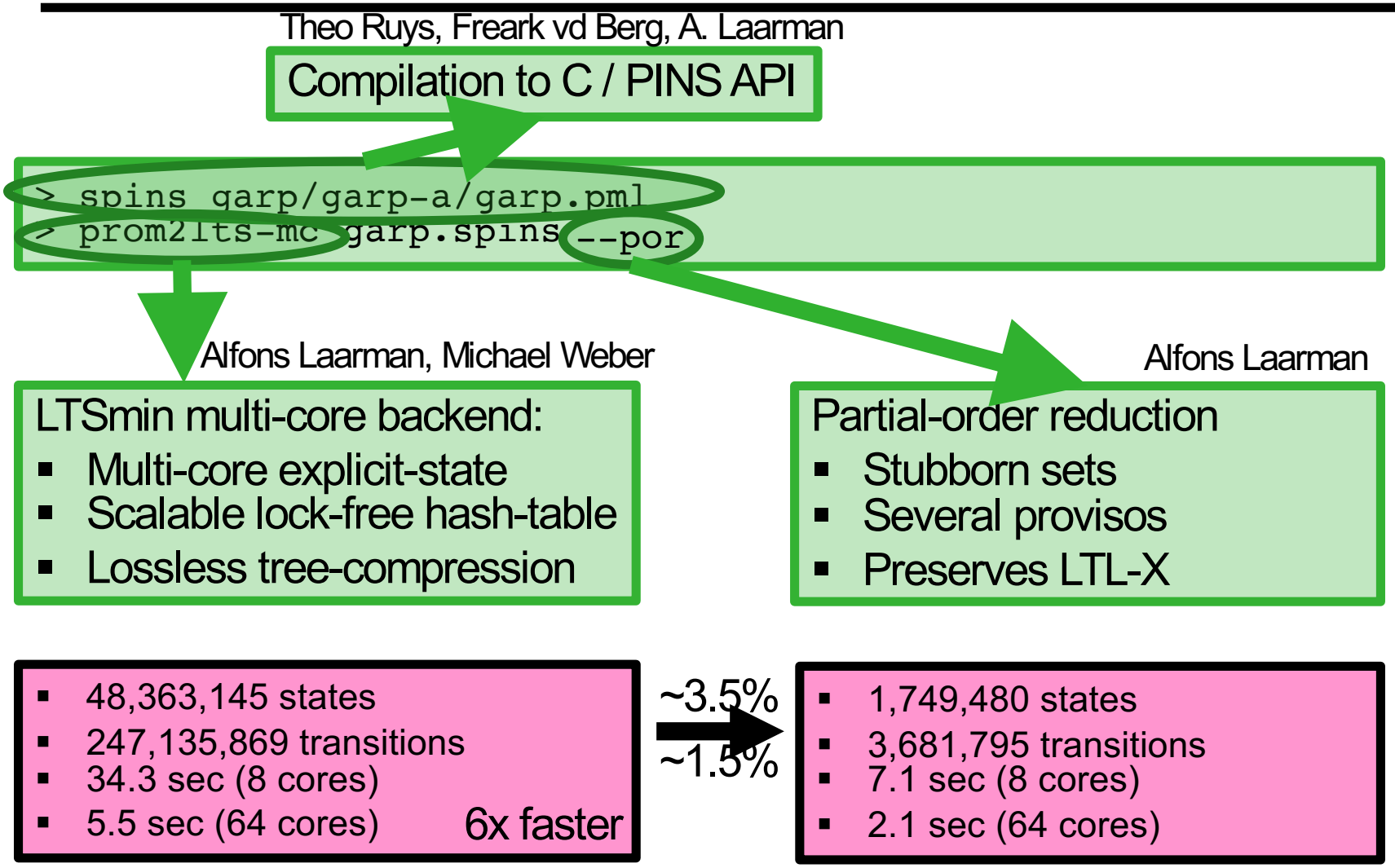
```
--ltl='[](p -> q U (r && <>s))'
--strategy=[cndfs|ufscc|...]
--por-proviso=[stack|closed-set|...]
```

- Symbolic multi-core CTL* / mu-calculus model-checking:

```
--mu='mu Z0.nu Z1.((p && []Z0)||(q && <>Z1))'
```

LTSMIN: MULTI-CORE EXPLICIT-STATE EXPLORATION

BASIC GARP MODEL



COMPATIBILITY WITH MCRL2

- mcrl22lps example.mcrl2 example.lps
- lps2lts-dist example.lps example.gcf
- ltsmin-reduce -b example.gcf example.b.[aut | fsm | bcg]

[suppressed some detailed options]

Symbolic deadlock checking:

- mcrl22lps example.mcrl2 example.lps
- lps2lts-sym –d example.mcrl2 –trace counter.gcf

RESULTS WITH LTSMIN

- Many conference papers. Tool papers: CAV 2010, TACAS 2015
- Winning Competitions
  - RERS 2012, 2013: 1st prize all categories (C-code)
  - RERS 2016: 1st prize parallel category
  - MCC 2016: gold in LTL category (Petri-Nets)
- Applications
  - Verification of EU Interlockings in mCRL2 (FP7 INESS)
  - Verification of CERN's LHC software in mCRL2

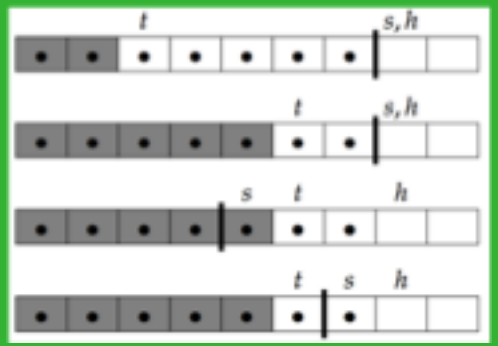
10 YEARS LTSMIN – CONCURRENT DATA STRUCTURES

- Multi-core shared-memory algorithms – states are in memory
- Require correct & efficient concurrent data structures
- **Main challenge:** Overcome memory-bandwidth limitations
  - Aware of hardware (L2-caches, NUMA, …)
  - No locking, use CAS instead (Compare-and-Swap)

Hash Tables

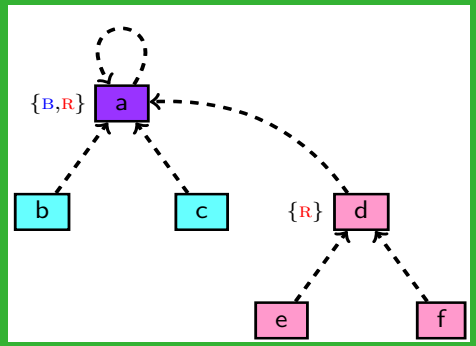
FMCAD 2010
Alfons Laarman

Split Deque



EuroPar WS 2014
Tom van Dijk

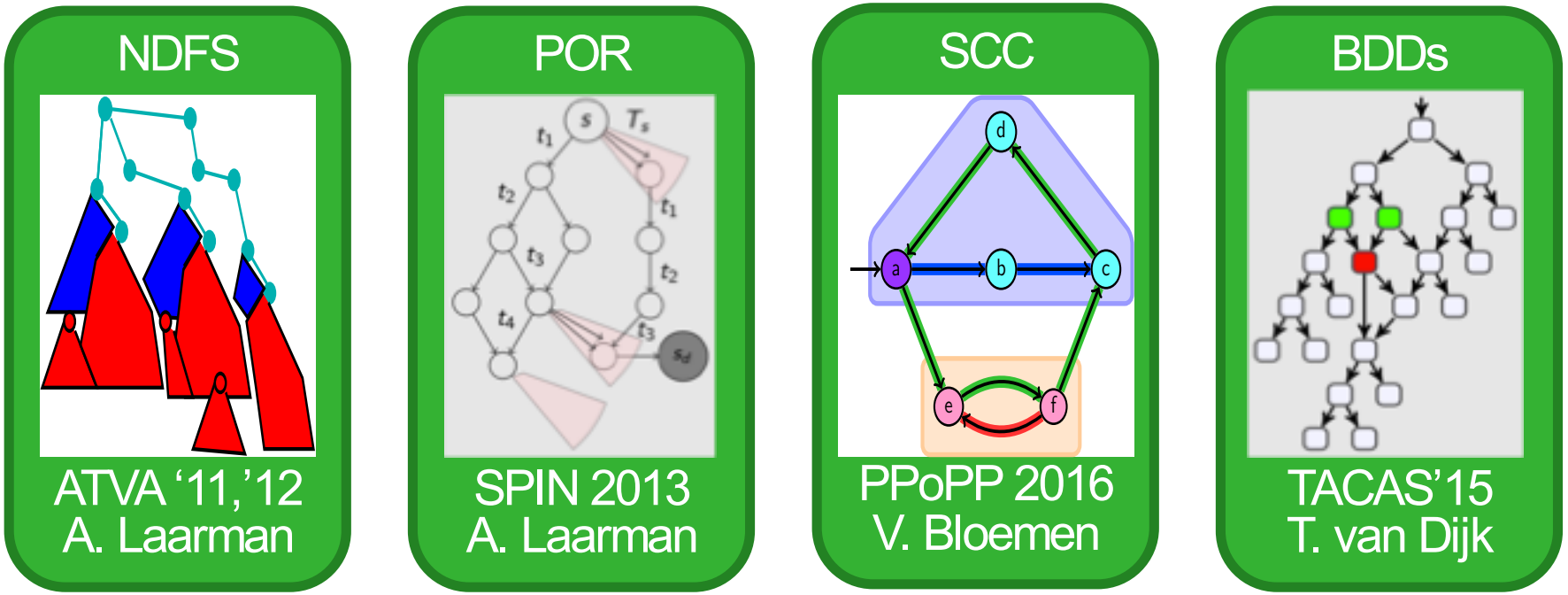
Union-Find

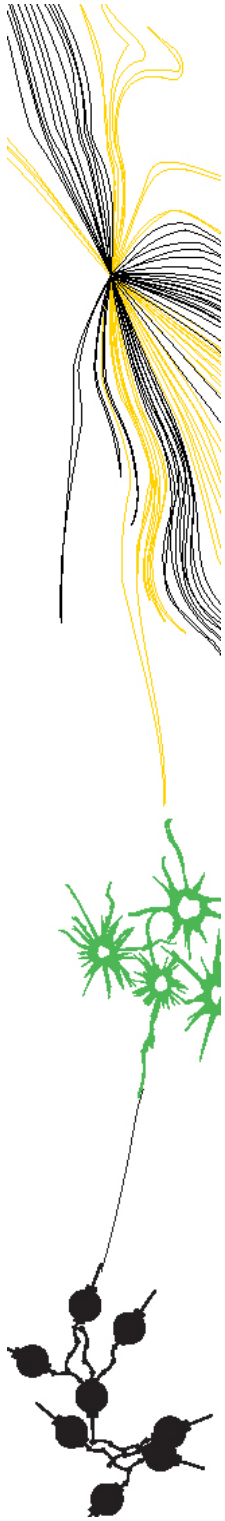


PPoPP 2016
Vincent Bloemen

10 YEARS LTSMIN – PARALLEL ALGORITHMS

- Need parallel graph algorithms – best sequential based on DFS
- Strategy: Independent workers, shared memory, load balancing





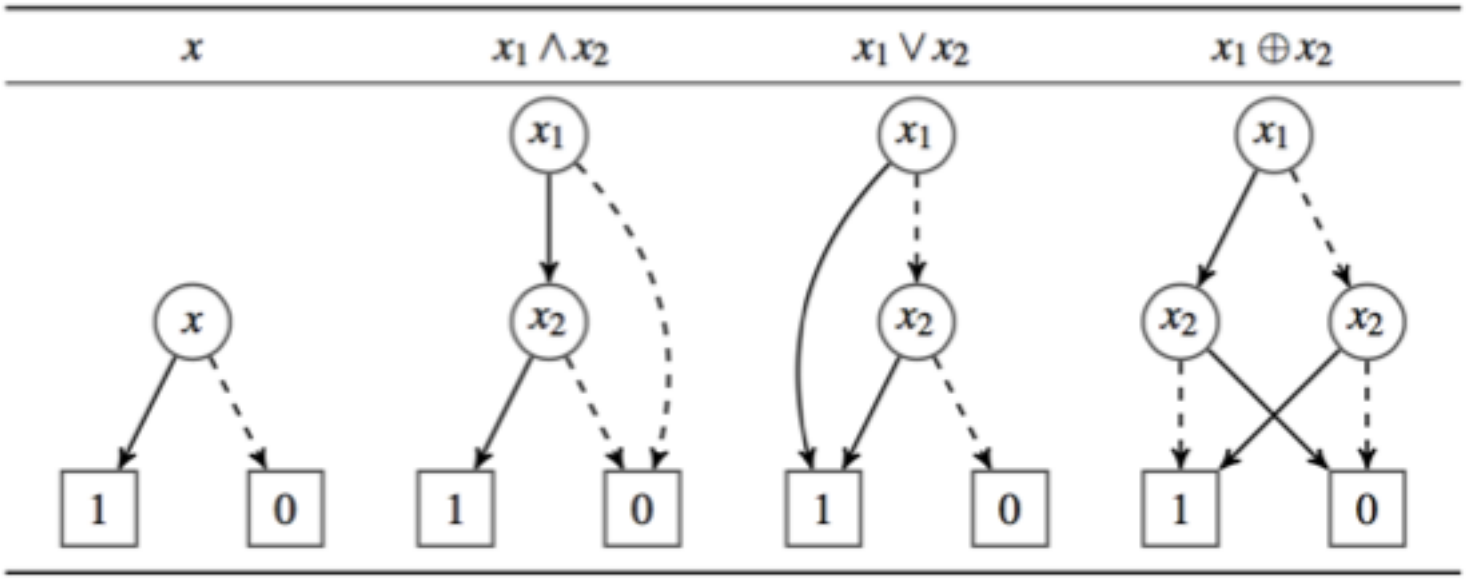
OVERVIEW

HIGH-PERFORMANCE SYMBOLIC MODEL CHECKING

1. Introduction: High-performance Model Checking – 10 years LTSmin
   - Motivation of a research line
   - LTSmin overview
2. Multi-core and Distributed Decision Diagrams
   - Sylvan: multi-core BDDs, MDDs, MT-DDs
   - Distributed Hardware Architecture
3. Applications in Symbolic Verification
   - State Space Generation: Symbolic Reachability
   - Symbolic Bisimulation Reduction
   - Solving Parity Games Symbolically

BINARY DECISION DIAGRAMS

- BDDs are directed acyclic graphs, ordered tests, maximal sharing
- Canonical representation of a set of state vectors (x) – relations (x,x')
- Often very concise (but depends on variable ordering, static)
- Symbolic Model Checking: operations directly expressed on BDDs



IMPLEMENTATION OF BDDS

- Datastructure: Unique Table – implement maximal sharing of nodes

  `(x,low,high)` ←→ `N` (hashmap)

  `MakeNode(x,low,high)` → `N` (reuse or create)

- Algorithm: `Apply(op,B1,B2)`

```
Apply( op, leaf1, leaf2) = op(leaf1,leaf2)

Apply( op, B1=(x1,low1,high1), B2=(x2,low2,high2) ) =
  let z = min(x1,x2)
      L = Apply( op, B1|z=0, B2|z=0)
      H = Apply( op, B1|z=1, B2|z=1)
  in MakeNode(z,L,H)
```

OPERATIONS CACHE

POLYNOMIAL APPLY OPERATOR

- Naïve apply is worst-case exponential in the number of nodes
- Data structure: (lossy) Operations Cache
- Improved Algorithm: Apply(op, B_1, B_2) – polynomial time

```
Apply( op, leaf1, leaf2) = op(leaf1,leaf2)
Apply( op, B1, B2) = if (op,B1,B2)→R in cache, return R
Apply( op, B1=(x1,low1,high1), B2=(x2,low2,high2) ) =
  let z = min(x1,x2)
      L = Apply( op, B1|z=0, B2|z=0)
      H = Apply( op, B1|z=1, B2|z=1)
      R = MakeNode(z,L,H)
      store (op,B1,B2) → R in cache
  in R
```

SYLVAN: MULTI-CORE BDD-OPERATIONS

TOM VAN DIJK' THESIS (TACAS 2015, STTT 2016, HANDBOOK 2017)

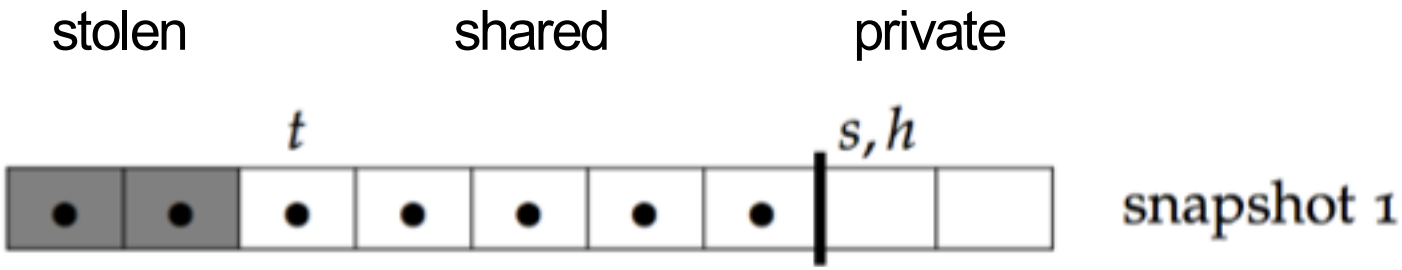
- Scalable concurrent Unique Table (lock-free, using CAS)
- Scalable concurrent Operations Cache (gives up on races)
- Very fine-grained Work-stealing framework:

  Double-Ended Queue with dynamic split-point

```
Apply( op, leaf1, leaf2) = op(leaf1,leaf2)
  Apply( op, B1=(x1,low1,high1), B2=(x2,low2,high2) ) =
    let z = min(x1,x2)
        L = spawn Apply( op, B1|z=0, B2|z=0)
        H = spawn Apply( op, B1|z=1, B2|z=1)
    in MakeNode(z, sync L, sync H)
```

SHARED DEQUE WITH DYNAMIC SPLIT POINT

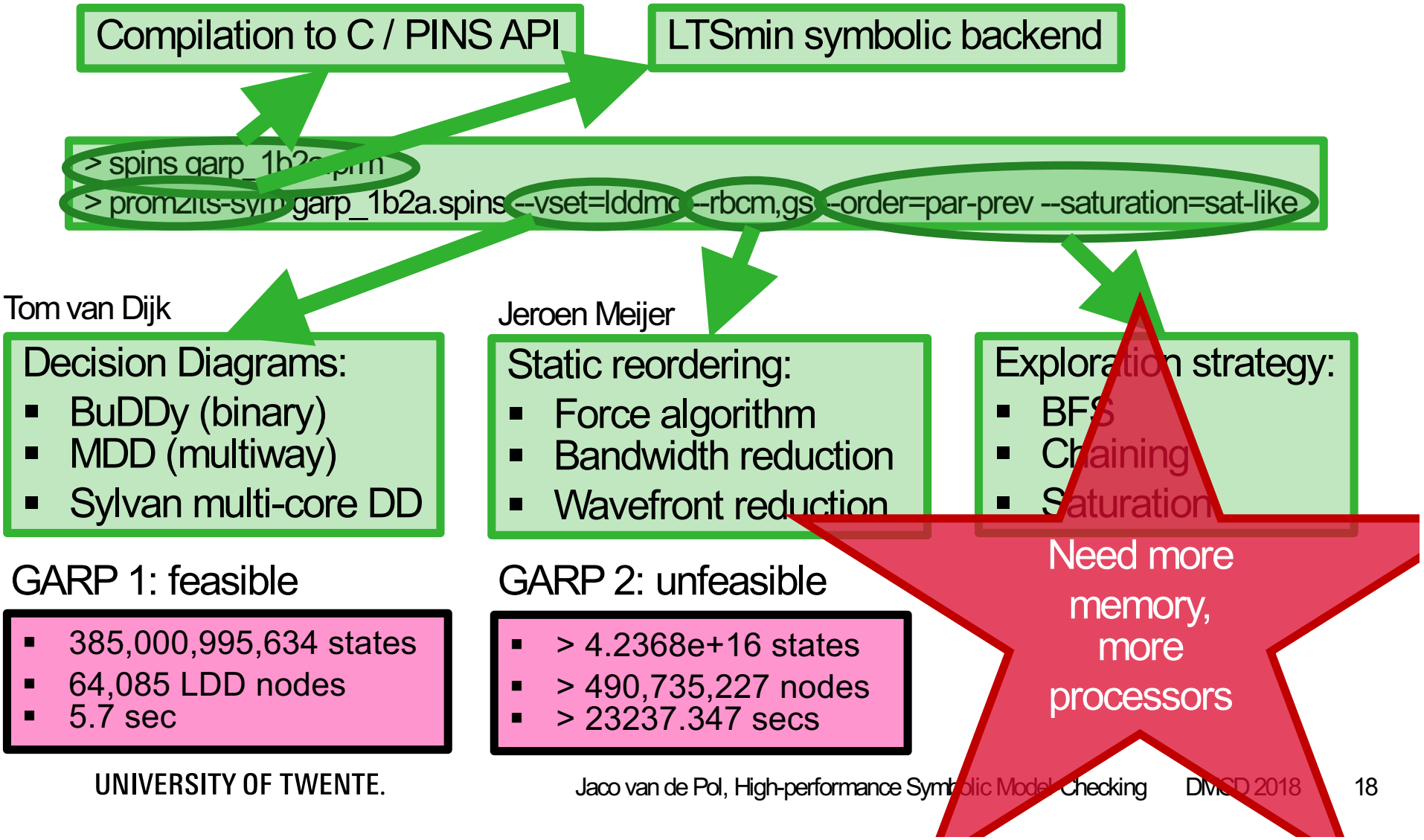
TOM VAN DIJK'S THESIS, EUROPAR WS 2014



t = tail, h = head, s = split

LTSMIN: MULTI-CORE SYMBOLIC MODEL CHECKING

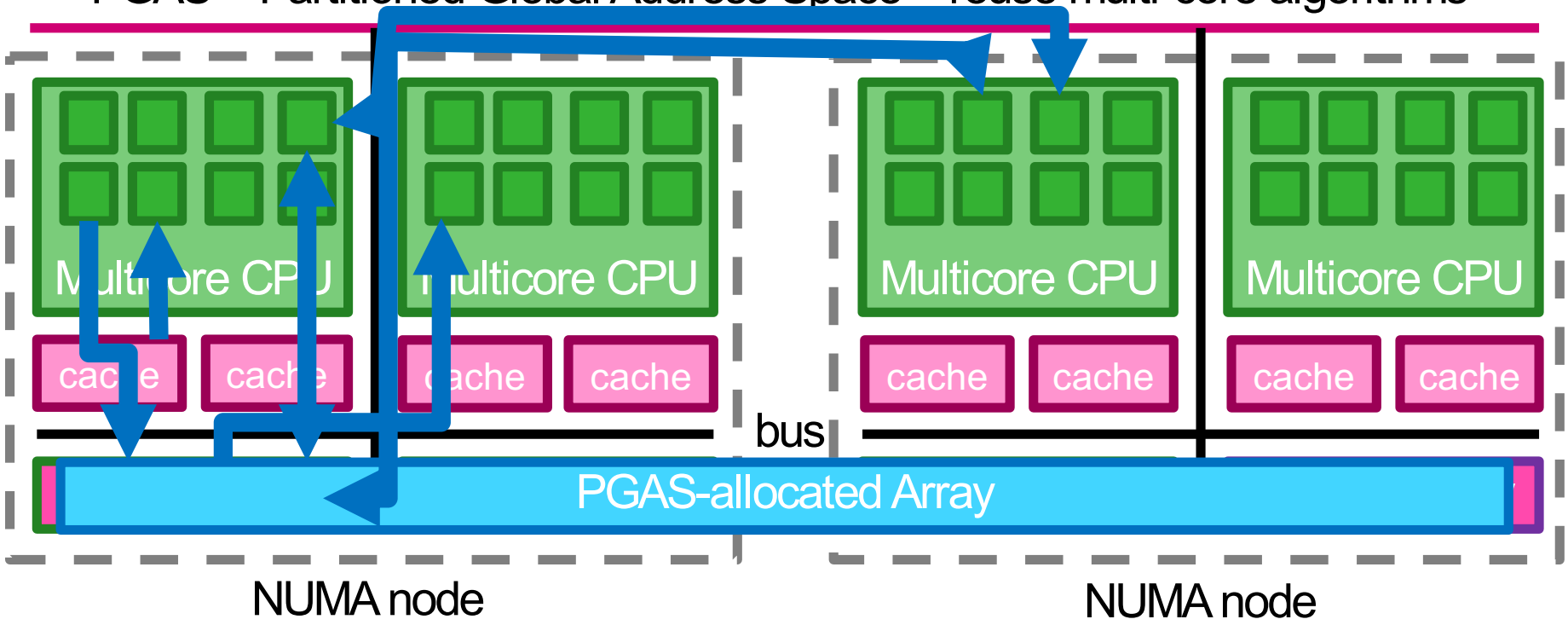
EXTENDED GARP MODELS BY IGOR KONNOV



DISTRIBUTED HARDWARE ARCHITECTURE (CLUSTER)

HETEROGENEOUS MEMORY HIERARCHY

- Multi-core CPU, with caches (L1, L2, L3)
- NUMA-node = Non-Uniform Memory Access (multiple CPUs)
- InfiniBand: high throughput, low latency network within a cluster (MPI)
- RDMA = Remote Direct Memory Access – bypass remote CPU & OS (!)
- PGAS = Partitioned Global Address Space – reuse multi-core algorithms



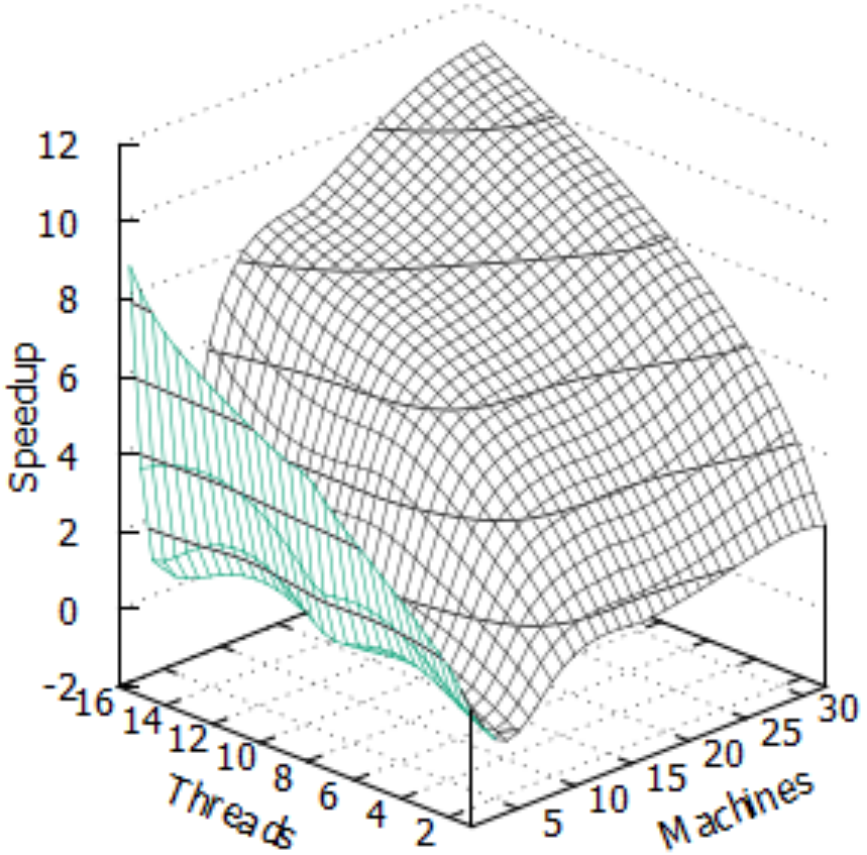
ALGORITHM RE-DESIGN FOR DISTRIBUTED BDDS

WYTSE OORTWIJN

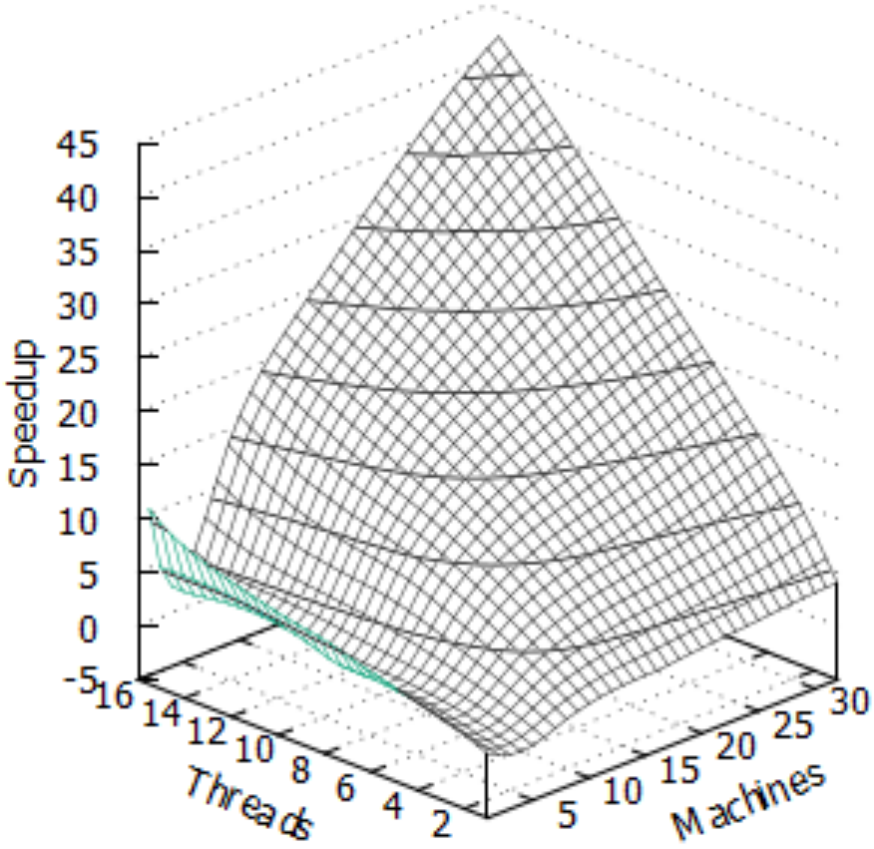
- Goal: minimize the number of roundtrips over the network
- Lock-less Distributed Hashtable
  - Uses one-sided RDMA operations
  - Linear probing: fetch range of buckets (adjust to load-factor)
- Lock-less Work Stealing
  - Victim selection is aware of the memory-hierarchy
  - Private deque: completely lockfree
  - Assistance required from victims → scheduled during waiting time

EXPERIMENTAL SCALABILITY

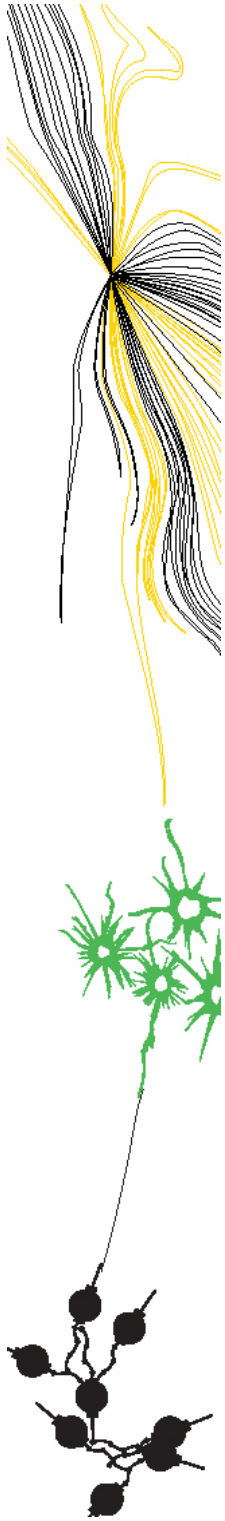
NUMBER OF NODES VERSUS NUMBER OF THREADS



BEEM: anderson.8



BEEM: adding.5



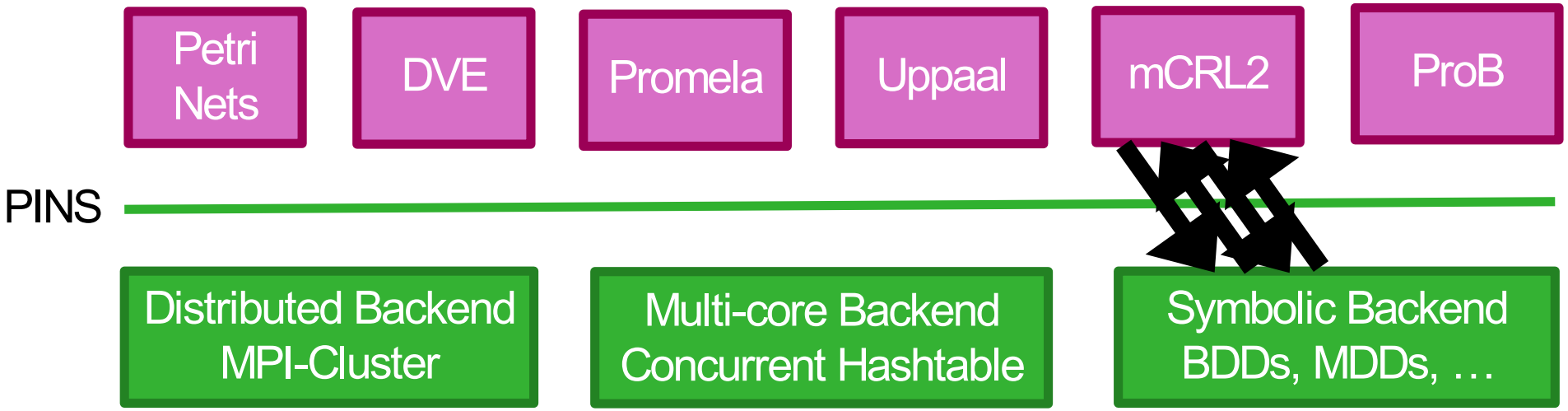
OVERVIEW

HIGH-PERFORMANCE SYMBOLIC MODEL CHECKING

1. Introduction: High-performance Model Checking – 10 years LTSmin
   - Motivation of a research line
   - LTSmin overview
2. Multi-core and Distributed Decision Diagrams
   - Sylvan: multi-core BDDs, MDDs, MT-DDs
   - Distributed Hardware Architecture
3. Applications in Symbolic Verification
   - State Space Generation: Symbolic Reachability
   - Symbolic Bisimulation Reduction
   - Solving Parity Games Symbolically

LTSMIN – SYMBOLIC REACHABILITY THROUGH PINS

PINS: A PARTITIONED INTERFACE FOR THE NEXT-STATE FUNCTION



- PINS should provide more than a function for initial state and next state
- Read-Write Dependency Matrix:
  - Exposes locality through PINS
  - States are vectors of fixed length
  - Structural transitions (summands)

	S_1	S_2	S_3	S_4
T_1	R	W	–	--
T_2	–	R	W	--
T_3	–	–	–	RW

SYMBOLIC REACHABILITY

- **Symbolic Reachability:**
  - $S_k(x_1 \ldots x_n)$: set of states after k iterations, stored as BDD
  - $T_{t,k}(x_a, x'_a, \ldots, x_b, x'_b)$:
    - Transtions of short vectors for transition T_t: R(t) → W(t)
    - Learned on-the-fly, here T_t in k-th iteration

- **Procedure:**
  - $T_{k+1} = T_k \vee \{ (x_a, x_b) \text{ –PINS→ } (x'_a, x'_b) \}$

  - $S_{k+1} = \exists x_a x_b \, (S(x_1..x_n) \wedge T_{k+1}(x_a, x'_a, x_b, x'_b))$

SYMBOLIC BISIMULATION MINIMISATION

TOM VAN DIJK, TACAS 2016, STTT 2018

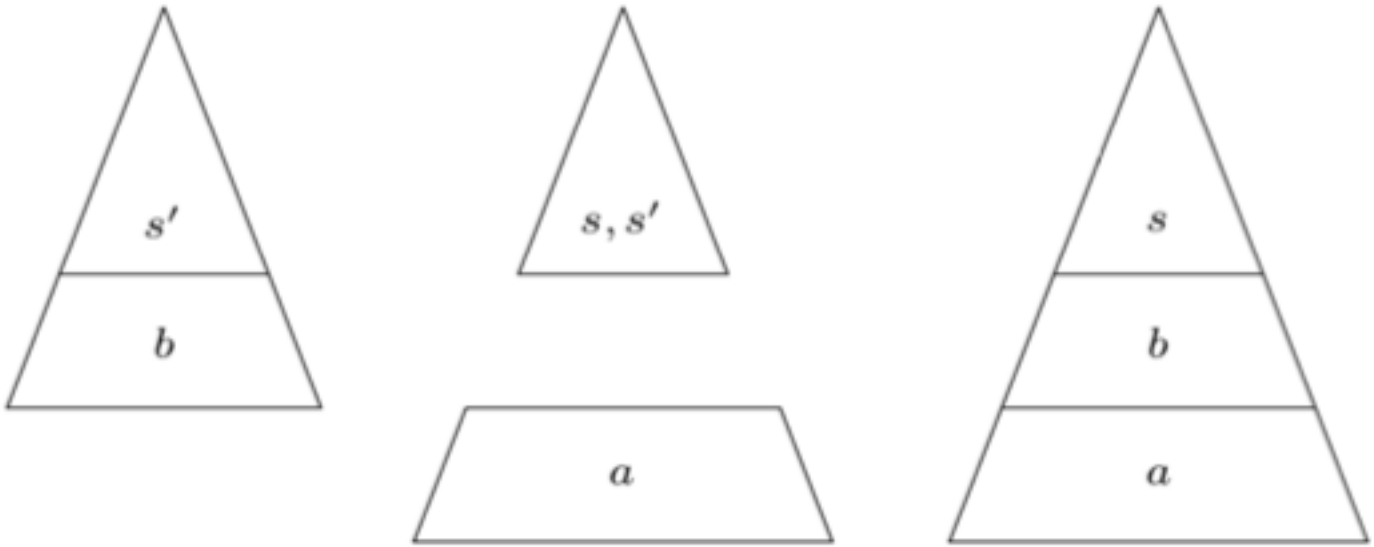
- Based on signature refinement, all operations on BDDs
- Implemented for Strong and Branching Bisimulation
- Works on LTS, CTMC, IMC (Interactive Markov Chains)

- **Partition Refinement:**
  - $P_0 = \{ S \}$
  - $P_{n+1} = \text{refine}(P_n, \text{sig})$

- **Signature:**
  - Characterizes a particular equivalence relation
  - For strong bisimulation: $\text{sig}(P,s) = \{ (a,P[s']) \mid s \xrightarrow{a} s' \}$

SYMBOLIC COMPUTATION OF SIGNATURE

- Transition Relation T(s,s',a), where a represents an action
- Partition P(s',b), where b represents a block
- Signature: ∃s′: T(s,s′,a) ∧ P(s′,b), s can do an a-step to block b



OTHER INGREDIENTS

- Refine operator: Given $P_n()$ and Sig(), compute $P_{n+1}()$
  - *Essential trick:* stable block gets same number in next iteration

- "Inert" computation needed for branching bisimulation:
  - compute τ-steps within the same block
  - *Essential trick:* compute $T_\tau(s,s') \wedge \exists b: (P(s,b) \wedge P(s',b))$ in one go

- Quotient: Given T() and final P(), compute reduced $T_s()$ or $T_b()$
  - *Essential trick:* new state will be state vector of some representant

EXPERIMENTAL RESULTS: LTS AND CTMC

LTS models (branching)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
kanban04	16020316	2785	8.47	0.52	0.24	16.39×	2.11×	34.60×
kanban05	16772032	7366	34.11	1.48	0.43	22.98×	3.47×	79.81×
kanban06	264515056	17010	118.19	3.87	0.83	30.55×	4.65×	142.20×
kanban07	268430272	35456	387.16	8.83	1.66	43.86×	5.31×	232.71×
kanban08	4224876912	68217	1091.67	17.91	2.98	60.96×	6.02×	366.72×
kanban09	4293193072	123070	3186.48	34.23	5.51	93.10×	6.21×	578.59×

CTMC models			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
cycling-4	431101	282943	220.23	26.72	2.60	8.24×	10.29×	84.84×
cycling-5	2326666	1424914	1249.23	170.28	19.42	7.34×	8.77×	64.34×
fgf	80616	38639	71.62	8.86	0.88	8.08×	10.04×	81.20×
p2p-5-6	2^{30}	336	750.29	26.96	2.99	27.83×	9.03×	251.24×
p2p-6-5	2^{30}	266	248.17	9.49	1.21	26.15×	7.82×	204.47×
p2p-7-5	2^{35}	336	2280.76	24.01	2.97	94.99×	8.08×	767.12×

State space of 2^{35} could be reduced to 336 states only

Sequential: up to 95x faster than previous work

Parallel: up to 10x speedup on 48 cores

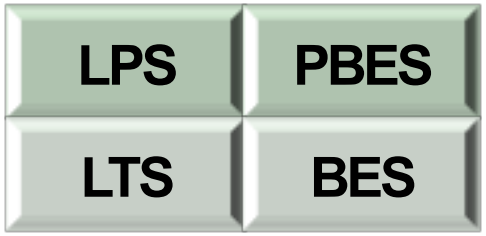
Up to 767x improvement

EXPERIMENTAL RESULTS: LTS FROM MCRL2

LTS model (branching)			signature refinement			quotient (`block`)		
Model	States	Blocks	T_1	T_{48}	Sp.	T_1	T_{48}	Sp.
brp-2-4-4	11,182	98	3.63	0.36	10.11×	0.18	0.02	7.71×
brp-3-4-4	40,592	328	13.78	0.98	14.08×	0.18	0.02	7.71×
brp-4-4-4	109,422	858	39.71	2.16	18.38×	4.52	0.23	19.64×
dkr-3	11,455	2	4.46	0.33	13.39×	0.63	0.05	11.79×
dkr-4	909,593	2	349.24	15.31	22.81×	25.73	1.37	18.82×
franklin-3-2	11,805	2	3.62	0.29	12.58×	0.28	0.04	6.64×
franklin-3-3	41,401	2	11.94	0.66	17.96×	0.95	0.07	13.55×
franklin-4-2	272,241	2	50.97	2.40	21.28×	2.19	0.18	12.28×
franklin-4-3	5,269,441	2	807.72	32.69	24.71×	31.94	1.56	20.52×
hesselink-2	540,736	72	7.64	0.79	9.71×	0.19	0.03	6.33×
hesselink-3	13,834,800	189	37.10	2.76	13.46×	0.94	0.13	7.36×
hesselink-4	142,081,536	384	114.37	7.98	14.33×	2.79	0.38	7.44×
hesselink-5	883,738,000	675	351.69	23.93	14.70×	8.33	1.11	7.49×
swp-2-4	2,589,056	511	116.16	5.08	22.88×	2.32	0.13	18.09×
swp-3-2	52,380	121	4.41	0.31	14.07×	0.11	0.01	11.00×
swp-3-3	1,652,724	1,093	135.99	6.21	21.88×	2.34	0.12	19.51×
swp-4-2	140,352	341	8.13	0.46	17.85×	0.28	0.03	11.13×
swp-4-3	7,429,632	5,461	420.09	17.13	24.52×	10.59	0.49	21.68×
WMS	155,034,776	1	0.36	0.22	1.66×	0.10	0.11	0.93×

PARITY GAMES

GIJS KANT



Recall:

- mCRL2 process → LPS (linear specification)
- LTS x μ-calculus = BES (Boolean Equation System, nested fixpoints)
- LPS x μ-calculus = PBES (Groote, Mateescu)
- BES ~ PG (Parity Game: LTS with players, priorities)

1. Compute symbolic parity game (SPG) from PBES using PINS
   - Basically, this is symbolic state space generation
2. Solve SPG by a BDD-based version of Zielonka's algorithm
   - Basically, repeated "attractor computations" ~ backward reachability

LINK WITH MCRL2

- mcrl22lps example.mcrl2 example.lps
- lps2pbes –f formula.mu example.lps example.pbes
- pbes2lts-sym –pg-solve –vset=mddmc example.pbes

OVERVIEW – 10 YEARS OF LTSMIN

HIGH-PERFORMANCE SYMBOLIC MODEL CHECKING

1. Introduction: High-performance Model Checking – 10 years LTSmin
   - Motivation of a research line
   - LTSmin overview

2. Multi-core Distributed BDDs
   - Sylvan: multi-core BDDs, MDDs
   - Distributed Hardware Architecture

3. Applications in Symbolic Verification
   - State Space Generation
   - Symbolic Bisimulation Reduction
   - Solving Parity Games Symbolically