Using work-stealing to parallelize symbolic algorithms and parity game solvers

Tom van Dijk (JKU Linz)

PLR, 18 July 2018

Topics

- Using work-stealing to parallelize for multi-core machines
- Reasoning on scalable datastructures
- How to make a parallel BDD package
- How to parallelize parity game solving

Thinking about parallelism

Uncontroversial opinions

- Parallel programming is about maximizing performance

Thinking about parallelism

Uncontroversial opinions

- Parallel programming is about maximizing performance
- Parallel programming is about minimizing communication

Thinking about parallelism

Uncontroversial opinions

- Parallel programming is about maximizing performance
- Parallel programming is about minimizing communication
- Shared memory does not exist
  - cachelines
  - cache hierarchy
  - limited bandwidth
  - networks of processors and memories

Thinking about parallelism

Uncontroversial opinions

- Parallel programming is about maximizing performance
- Parallel programming is about minimizing communication
- Shared memory does not exist
  - cachelines
  - cache hierarchy
  - limited bandwidth
  - networks of processors and memories
- Importance of multi-core
  - Most developers have multi-core systems
  - Multi-core systems are less restrictive than GPUs
  - Multi-core parallelism combines nicely with computer clusters

Thinking about parallelism

Uncontroversial opinions

- Parallel programming is about maximizing performance
- Parallel programming is about minimizing communication
- Shared memory does not exist
  - cachelines
  - cache hierarchy
  - limited bandwidth
  - networks of processors and memories
- Importance of multi-core
  - Most developers have multi-core systems
  - Multi-core systems are less restrictive than GPUs
  - Multi-core parallelism combines nicely with computer clusters
- In principle, consider each problem as parallelizable!

Outline

1. Introduction
2. Work-stealing
3. Parallel decision diagrams
4. Parity games

Work stealing

Why work-stealing

- Many algorithms have a natural recursive definition
- Solve the problem by solving subproblems
- Subproblems are independent
- Compute subproblems in parallel

Examples

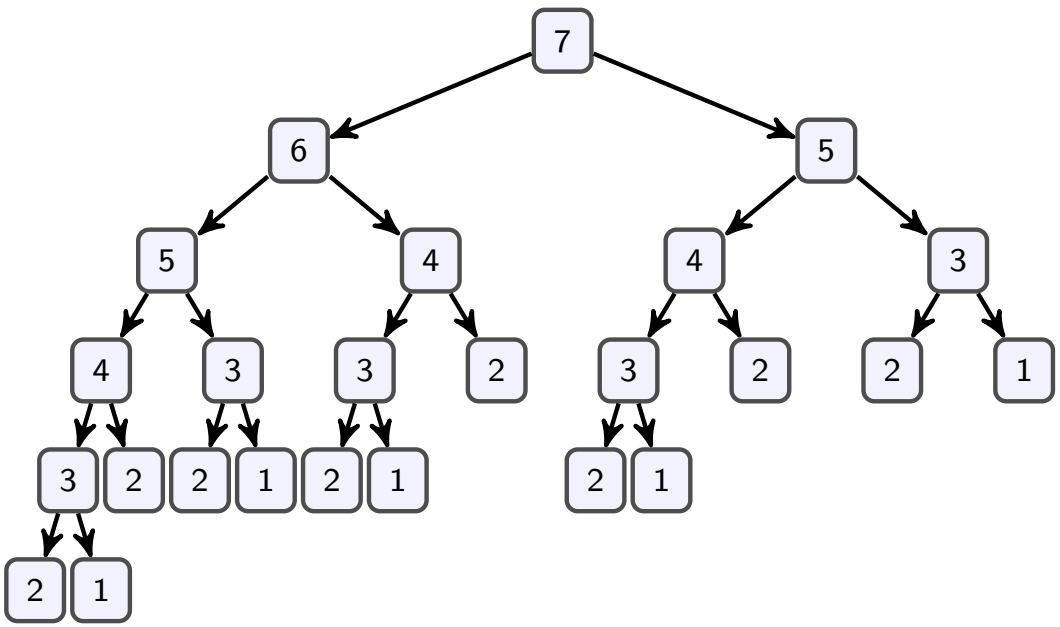
- Reachability
- Backward search
- BDD operations
- Data parallelism in e.g. arrays

Task parallelism

```
1 def fib(k):
2     if k < 2: return k
3     x ← fib(k − 1)
4     y ← fib(k − 2)
5     return x + y
```

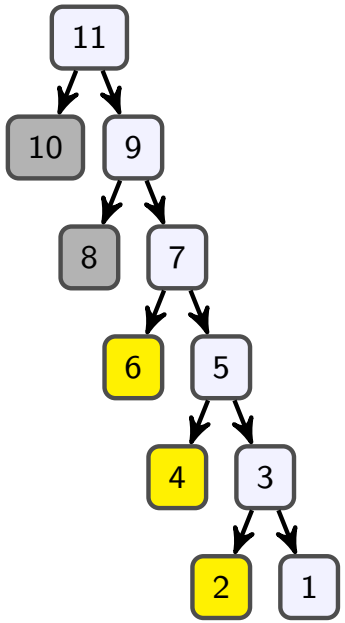
$\Rightarrow$

```
1 def fib(k):
2     if k < 2: return k
3     spawn fib(k − 1)
4     spawn fib(k − 2)
5     y ← sync
6     x ← sync
7     return x + y
```

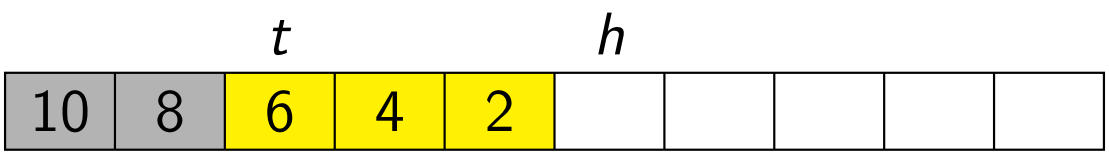


Example: calculate `fib(11)`

Task graph:



Task deque:



Work-stealing related to the deque operations

Work stealing operation	Deque operation
`spawn` a task	`push` to head
`sync` a task	`peek` and `pop` from head
`steal` a task	`steal` from tail

- Each worker has 1 **deque**
- Owner uses **push**/**pop**/**peek** on its deque
- Other workers (thieves) use **steal** on other deques
- Policy: steal from the thief

Deques for work-stealing

Implementations (blue = non-blocking)

- Fully shared deques
  - Frigo e.a. ("Cilk" 1998)
  - ABP (1998)
  - Chase and Lev (2005)
  - Hendler e.a. (2006)
- Private deques (message passing flavor)
  - Acar e.a. (2013)
- Split deques
  - Faxén ("Wool" 2008, 2010)
  - Dinan e.a. (2009)
  - Van Dijk & Van de Pol ("Lace", 2013)

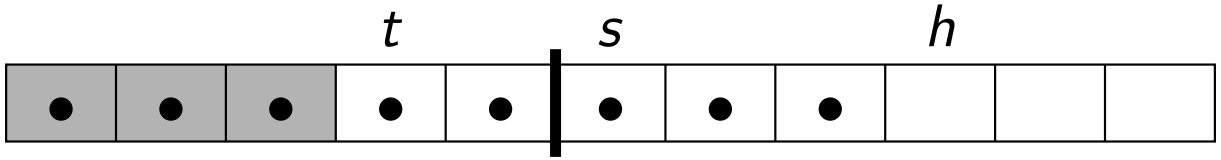
Deques for work-stealing

Challenges

- Avoid hidden and unnecessary communication
  - false sharing (variables accessed by thieves / owner)
  - unnecessary memory writes *and reads!*
- Avoid using locks/mutexes
  - (solved using lock-free operations)
- Avoid expensive memory fences like in Cilk-THE
  - (mostly solved using the split idea)
- Avoid overhead, especially since most tasks are never stolen
  - (solved using the "direct task stack")

Deques for work-stealing

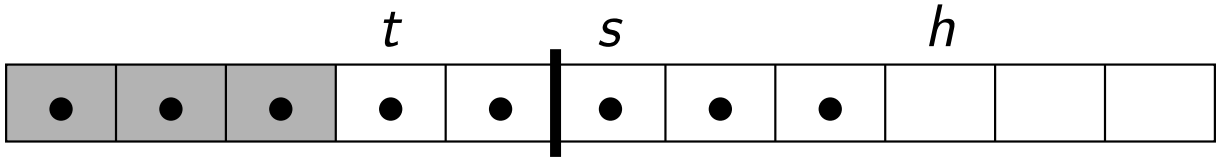
Deque is described by variables **tail** (t), **split** (s), **head** (h).



- Tasks are **shared** or **private**
- The first t tasks are **stolen**
- Thieves steal with **atomic cas** on t and s together
- Owner modifies s and h with normal memory operations
- Extra flag: *movesplit*
- Direct task stack: parameters *and results* stored in the deque!

Deques for work-stealing

Deque is described by variables **tail** (t), **split** (s), **head** (h).

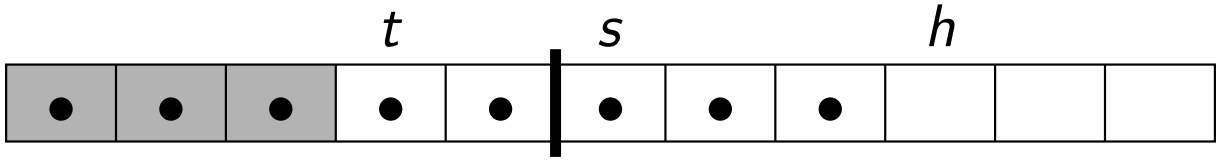


Operations

`push`	Add task to head. Maybe move split to the right.
`peek`	Maybe move split to the left. Return status of head.
`pop`	Remove task from head.
`steal`	Move tail with **atomic cas**, or set movesplit.

Deques for work-stealing

Deque is described by variables **tail** (t), **split** (s), **head** (h).

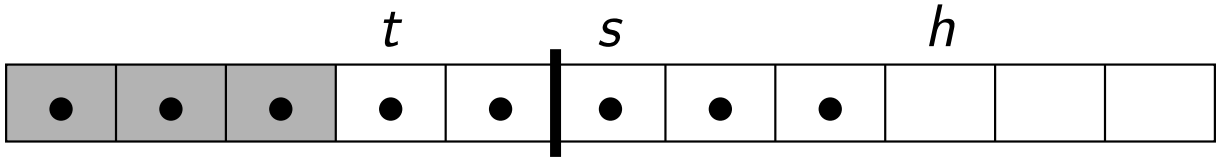


Communication is key

Cacheline	Contents	Thief access	Owner access
Shared 1	`tail`, `split`	Often	Sometimes
Shared 2	flag `movesplit`	Sometimes	Often
Private	`head`, `osplit`	–	Often

Deques for work-stealing

Deque is described by variables **tail** (t), **split** (s), **head** (h).

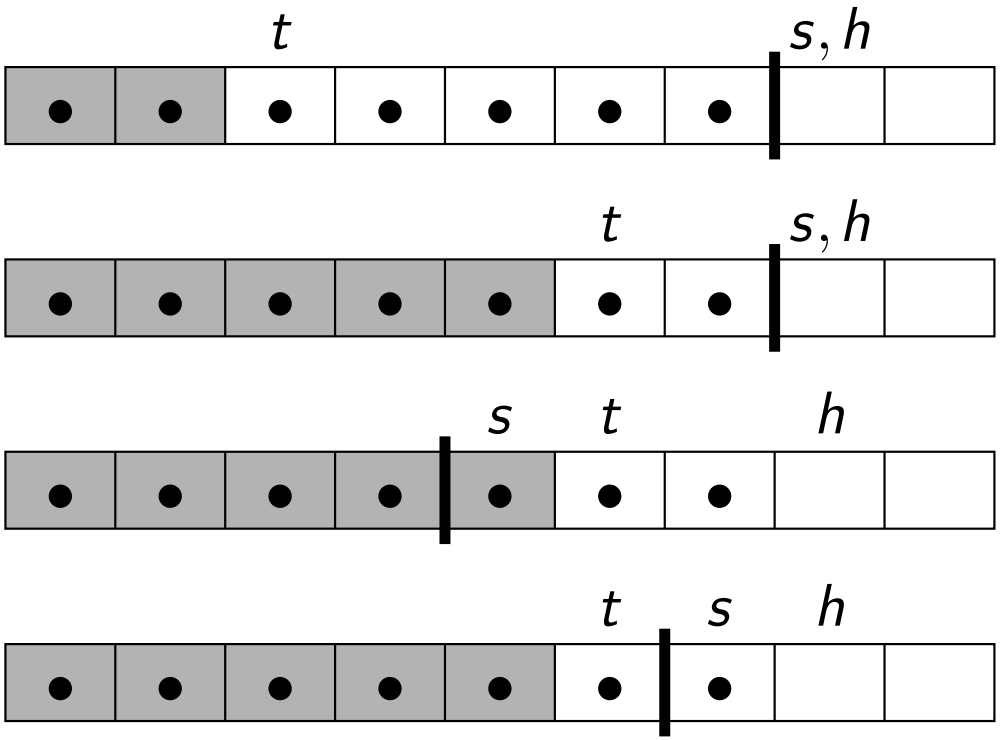


Communication is key

`push`	Add task to head. Maybe move split to the right. Private; Shared 2; (Shared 1)
`peek`	Maybe move split to the left. Return status of head. (Shared 1, race condition); Private
`pop`	Remove task from head. Private
`steal`	Move tail with **atomic cas**, or set movesplit. Shared 1, Shared 2

Deques for work-stealing

Moving the split point back



Work-stealing

Benchmarks

- `fib`(50) – 20,365,011,073 tasks
- `N-queens`(15) – 171,129,071 tasks
- `uts`(T3L) – Unbalanced Tree Search, 111,345,630 tasks
- `matmul`(4096) – 3,595,117 tasks
- No cut-off point, fine-grained, very small tasks

Measurements

- 48-core AMD machine (4 sockets, 12 cores per socket)
- Wallclock time around parallel part, 48 workers.

Work-stealing

Results	Benchmark time			Speedup	
	T_S	T_1	T_{48}	T_S/T_{48}	T_1/T_{48}
fib 50	149.2	144	4.13	34.5	34.9
uts T3L	43.11	44.2	2.23	18.7	19.9
uts T3L *	43.11	44.3	1.154	37.4	38.2
queens 15	533	602	12.63	42.2	47.7
matmul 4096	773	781	16.46	47.0	47.5

* = with extension to fix issues with leapfrogging

Summary

- Non-blocking split deque has low overhead and good speedup
- Designed to minimize communication
- No locking, just **atomic cas**
- Only require memory fence in `shrink`
- Lace can be found at `https://github.com/trolando/lace`

Outline

1 Introduction

2 Work-stealing

3 Parallel decision diagrams

4 Parity games

Context

Transition system

- Tuple $(S, \rightarrow)$ with states S and transitions $\rightarrow : S \times S$
- Maybe transition labels (Labeled transition system)
- Maybe state labels (Kripke structure)

Example model checking tasks

- Check some property, for example in CTL or LTL
- Count number of reachable states to satisfy our curiosity
- Find shortest path to a bad state
- Perform bisimulation minimisation

Context

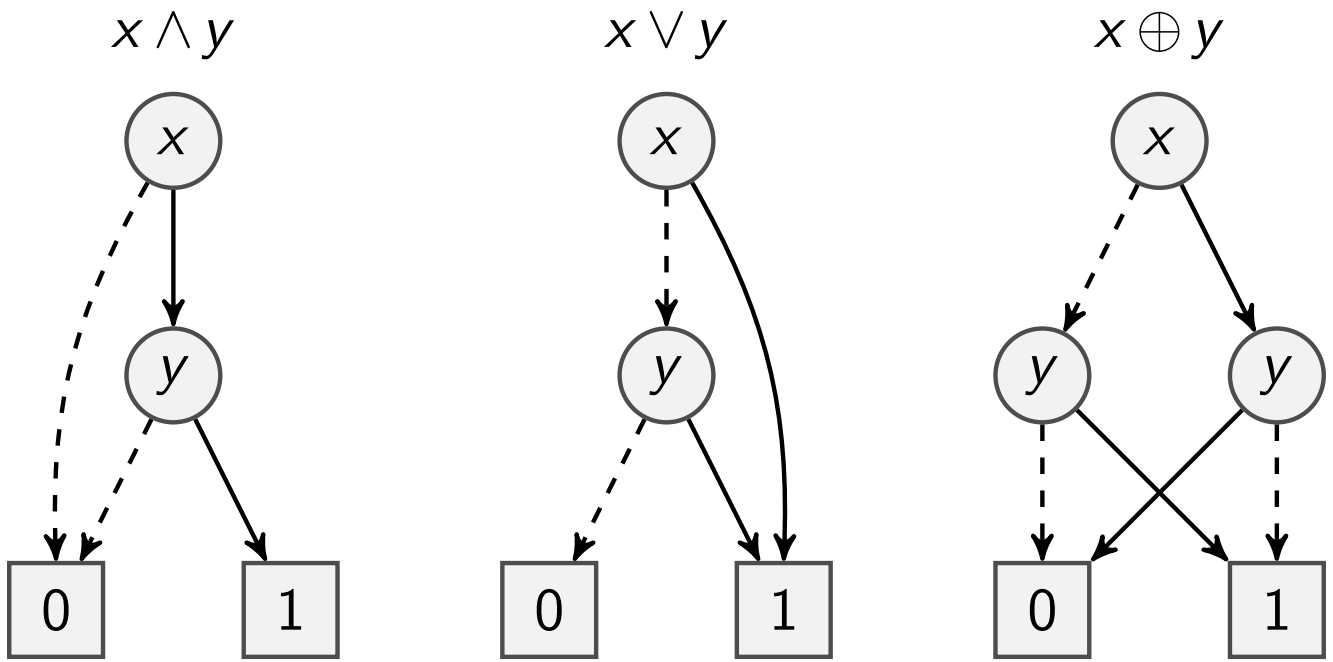
Storing sets in memory:

Explicit	Symbolic (as sets)
Set of states:	Boolean function:
$\overline{x_0}\overline{x_1}\overline{x_2}$ (0,0,0)	$\overline{x_0} \vee \overline{x_1} \vee \overline{x_2}$
$\overline{x_0}\overline{x_1}x_2$ (0,0,1)	
$\overline{x_0}x_1\overline{x_2}$ (0,1,0)	All states where:
$\overline{x_0}x_1x_2$ (0,1,1)	$(0,*,*)$ or $(*,0,*)$ or $(*,*,0)$
$x_0\overline{x_1}\overline{x_2}$ (1,0,0)	
$x_0\overline{x_1}x_2$ (1,0,1)	
$x_0x_1\overline{x_2}$ (1,1,0)	

- Use symbolic methods to deal with large state spaces
- Work with characteristic function rather than individual states

Binary Decision Diagrams

- A BDD is a directed acyclic graph encoding a $\mathbb{B}^k \to \mathbb{B}$ function
- Paths represent valuations of $\mathbb{B}^k$
- Path to $\boxed{1}$, then valuation is in the set



Typical (MT)BDD operation

```
1 def apply(x, y, F):
2     if (x,y,F) ∈ cache:  return cache[(x,y,F)]
3     if x and y are terminals:  return F(x,y)
4     v = topVar(x,y)
5     low ← apply(x_{v=0}, y_{v=0}, F)
6     high ← apply(x_{v=1}, y_{v=1}, F)
7     result ← BDDnode(v, high, low)
8     cache[(x,y,F)] ← result
9     return result
```

Typical features of a BDD operation

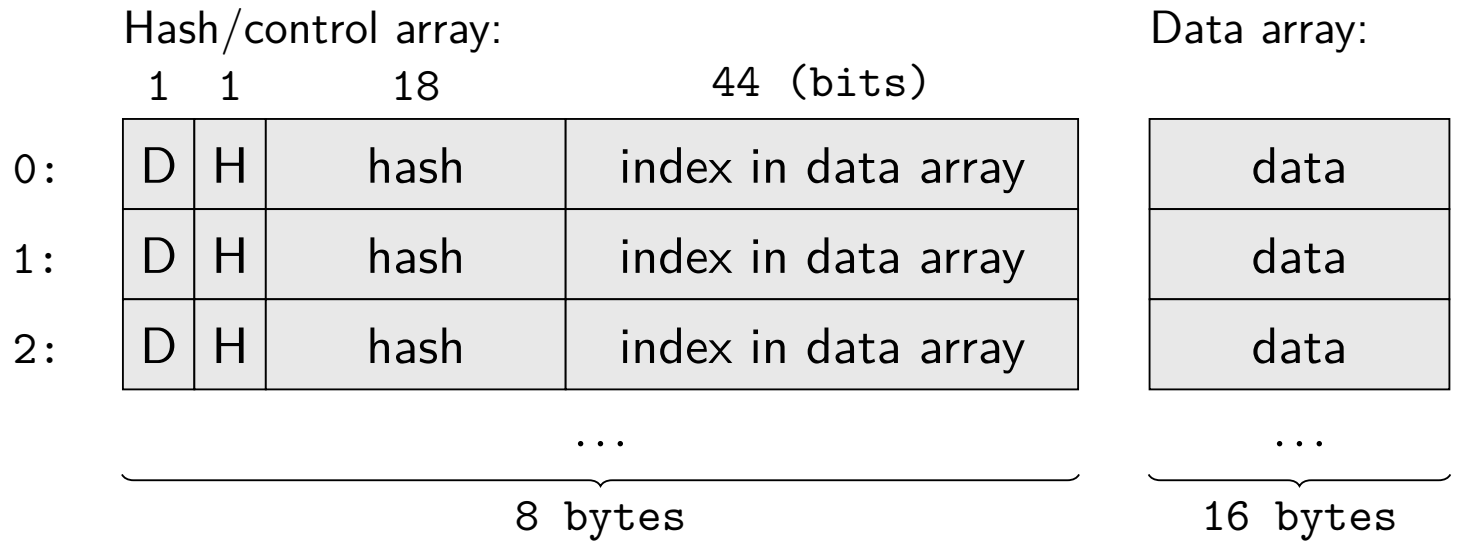
- Two recursive calls
- Accessing an operation cache
- Creating a BDD node in the nodes table

Datastructures

Safe and scalable datastructures

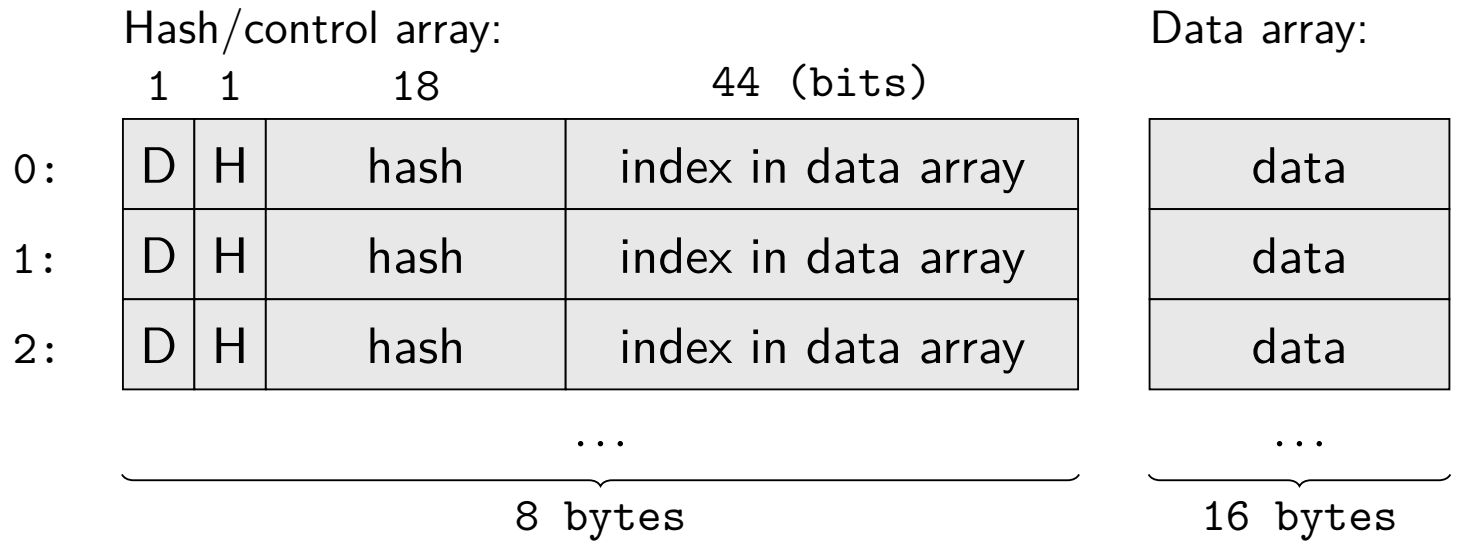
- BDD operations are "memory-heavy"
- Using locks/mutexes is bad for parallel performance
- *Lockless* design using the atomic `compare_and_swap`
- Operation cache
  - Short-lived 1-bit lock on bucket during write
  - No waiting: greedy abort during "write-lock"
  - 1 cas instruction per write
- Nodes table
  - No locks
  - 1 cas instruction per insert operation
  - 1 extra cas per 512 inserts

Nodes table



- Store the BDD nodes in a "data array"
- Hash BDD nodes in a "hash array"
- Manipulate hash array using `compare_and_swap` (64-bit)
- *Lockless* technique: shortlived mini-locks on buckets that only block other threads with the same hash
  - Use other hash bits for "hash" than for linear probing

Nodes table



- Phase 1: find empty hash bucket or existing entry
  - Using cache-line aware version of linear probing
  - On empty hash bucket: go to Phase 2
- Phase 2: find empty data bucket (linear search)
  - On empty data bucket: use cas to set "D"
- Phase 3: find empty hash bucket or existing entry
  - Continue search from Phase 1
  - On empty hash bucket: use cas to set "H", "hash" and "index"
  - On existing: undo setting "D" in phase 2 using cas

Operation Cache

- Store results of DD operations: 192 bits $\rightarrow$ 64 bits
- 32 bytes per cache entry
- Separated into data array and hash array
- Writing to the cache with cas-lock
  1. Lock hash array bucket (parameterised with hash of key)
  2. Write in data array
  3. Release hash array bucket
- Lossy (overwriting) cache + greedy abort
  - Abort if locked (regardless of key)
  - Abort if different key (when reading)
- Solve race condition with a counter in the hash array

Garbage collection

Parallel garbage collection

- Stop the world (all workers do garbage collection)
- Triggered when the unique table is “full”

Modular garbage collection

- Clear hash table (hash/control array)
- Call all marking handlers (mark for rehash)
  - Internal list of references during operations
  - Custom user reference management
- Optional: resize tables
- Rehash all marked nodes

Computing reachability

- Compute reflexive transitive closure of $\mathcal{T}$ applied to $\mathcal{S}$
- Use a frontier set (only new states)

```
1 def ClosureFS(S, T):
2     states ← ∅
3     frontier ← S
4     while frontier ≠ ∅:
5         next ← RelProd(frontier, T)
6         frontier ← next \ states
7         states ← states ∪ frontier
8     return states
```

Computing reachability

- Compute reflexive transitive closure of $\mathcal{T}$ applied to $\mathcal{S}$
- Multiple transition relations

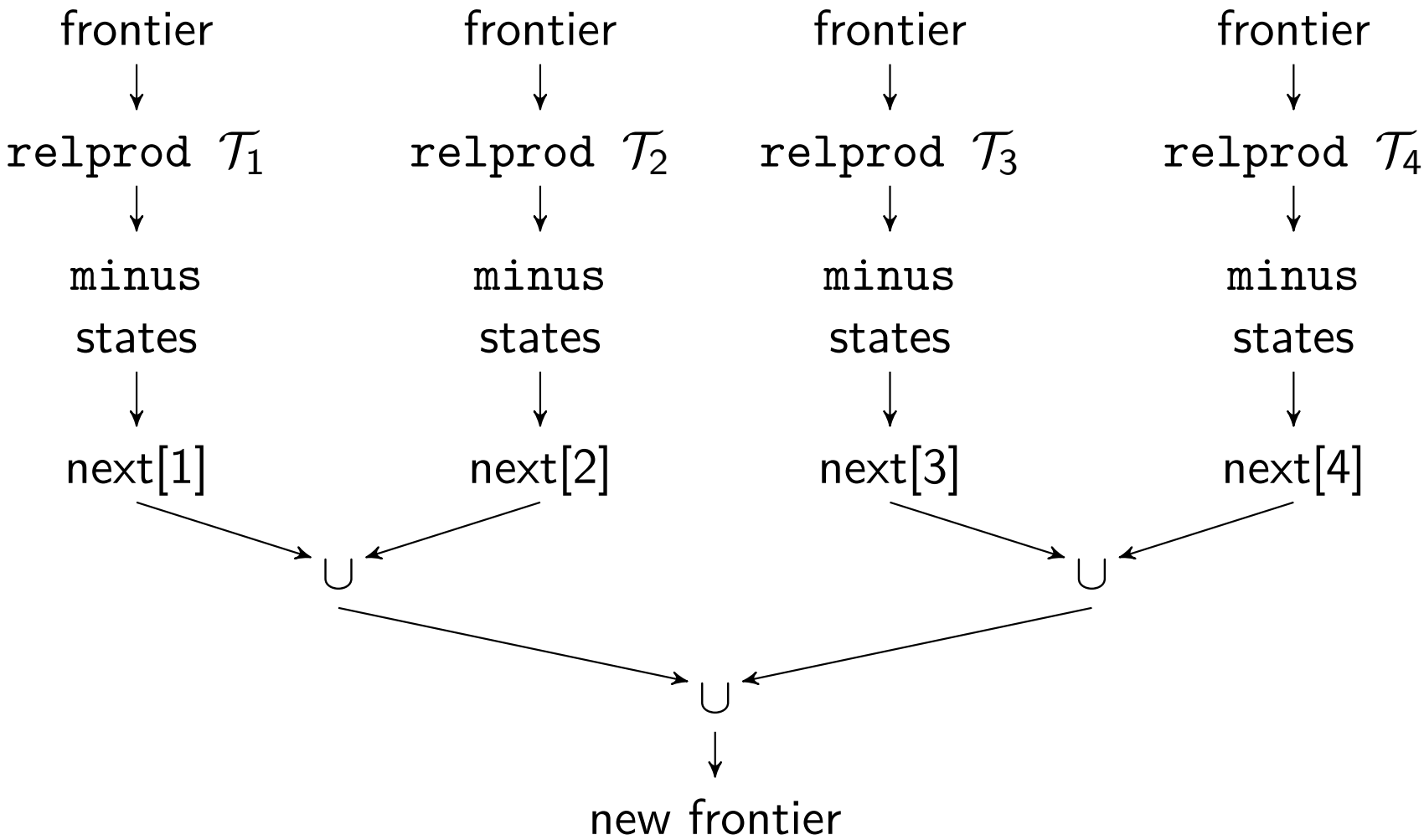
```
1  def ClosureFS(S, T_1, ..., T_N):
2      states ← ∅
3      frontier ← S
4      while frontier ≠ ∅:
5          foreach T_i do
6              next[i] ← RelProd(frontier, T_i)
7              next[i] ← next[i] \ states
8          end
9          frontier = next[1] ∪ ... ∪ next[N]
10         states ← states ∪ frontier
11     return states
```

Computing reachability

- Compute reflexive transitive closure of $\mathcal{T}$ applied to $\mathcal{S}$
- Multiple transition relations
- On-the-fly learning via (explicit) next-state interface

```
1  def ClosureFS(S, T_1, ..., T_N):
2      states ← ∅
3      frontier ← S
4      while frontier ≠ ∅:
5          foreach T_i do
6              T_i ← Learn(frontier, T_i)
7              next[i] ← RelProd(frontier, T_i)
8              next[i] ← next[i] \ states
9          end
10         frontier = next[1] ∪ ... ∪ next[N]
11         states ← states ∪ frontier
12     return states
```

Parallel reachability



Results

Benchmarks

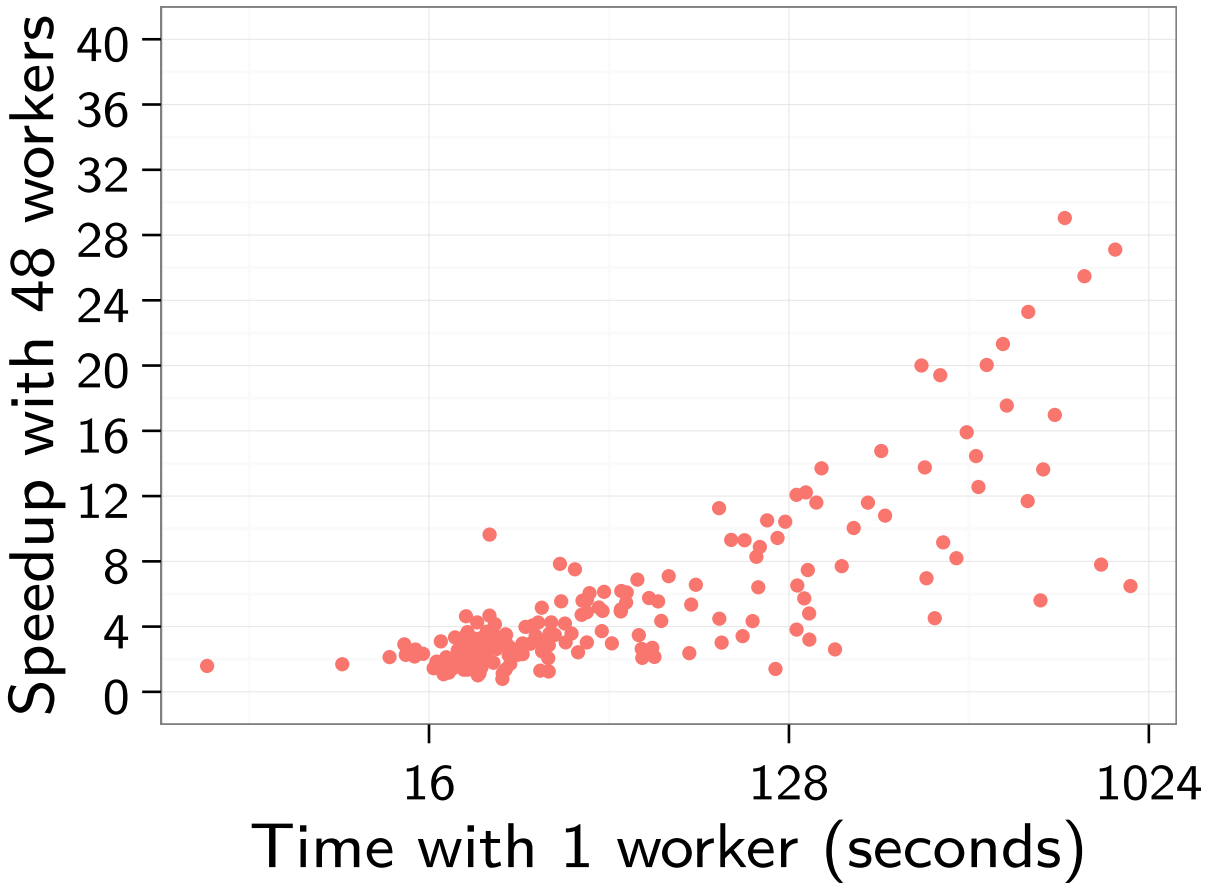
- Machine: 48-core Opteron 6168 (1.9GHz), 128GB memory.
- Experiments using entire BEEM database (269 DiVinE models)
  - Several timed out (1200 seconds)
  - Each data point repeated at least 3 times
- Variants:
  - bfs: only low-level parallel (no changes to model checker)
  - par also high-level parallel (multiple transition relations)

Results

Experiment	T_1	T_{16}	T_{24}	T_{32}	T_{48}	T_{48}/T_1
`blocks.4` (bfs)	630.04	45.88	33.24	27.01	21.69	**29.0**
`blocks.4` (par)	629.54	41.61	29.26	23.04	16.58	**38.0**
`telephony.8` (bfs)	843.06	66.28	47.91	39.17	31.10	**27.1**
`telephony.8` (par)	843.70	58.17	41.17	32.76	24.68	**34.2**
`lifts.8` (bfs)	377.36	36.61	30.06	27.68	26.11	**14.5**
`lifts.8` (par)	377.52	25.92	18.68	15.18	12.03	**31.4**
`firewire_tree.1` (bfs)	16.43	11.24	11.12	11.36	11.35	**1.4**
`firewire_tree.1` (par)	16.40	1.09	0.97	0.94	0.99	**16.5**
Sum of all bfs	20745	3902	3667	3625	3737	**5.6**
Sum of all par	20756	1851	1552	1403	1298	**16.0**

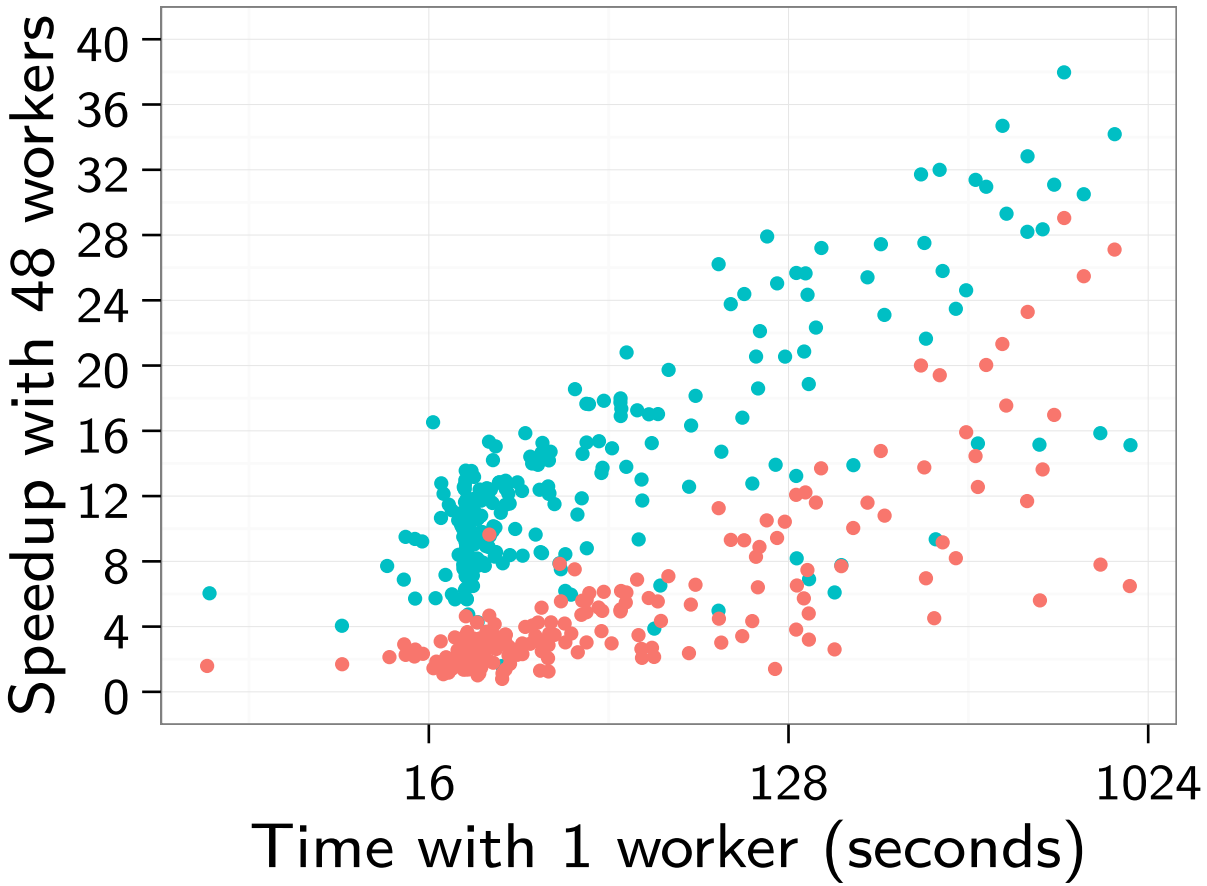
Results

Only low-level parallelism:



Results

Low-level parallelism plus parallel learning and parallel calls



Summary

- Parallel BDD package Sylvan
- Operations implemented with work-stealing
- Shared datastructures use lockless techniques
- Mix of low-level and high-level parallelism
- (Not in the slides: application to bisimulation minimisation)

Outline

1 Introduction

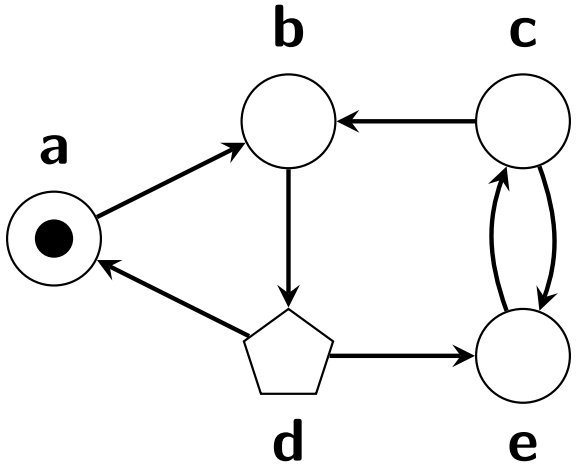
2 Work-stealing

3 Parallel decision diagrams

4 Parity games

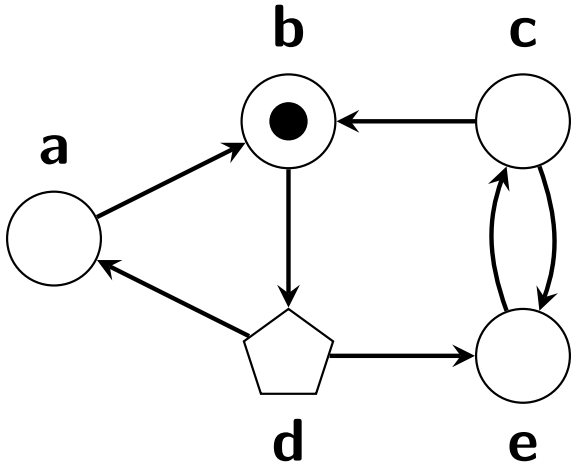
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



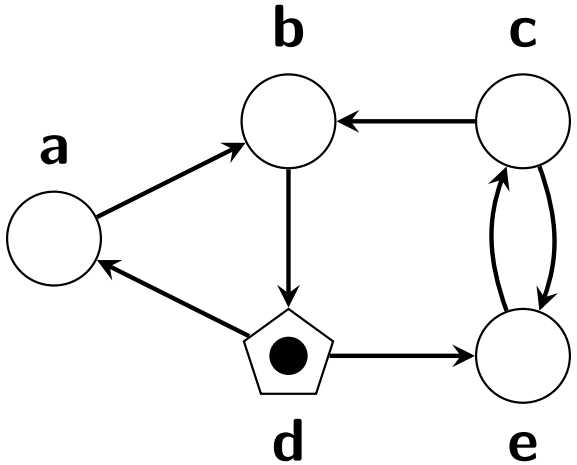
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



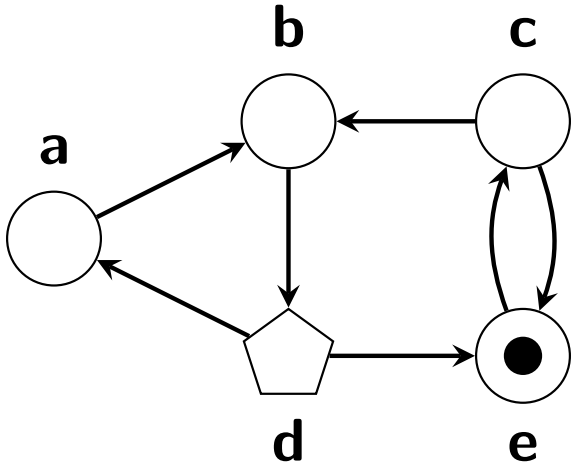
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



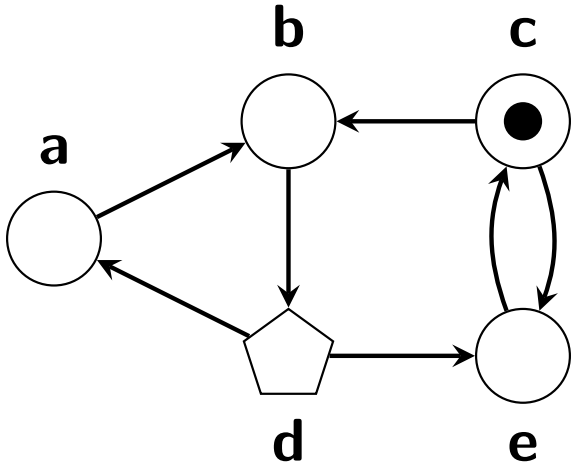
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



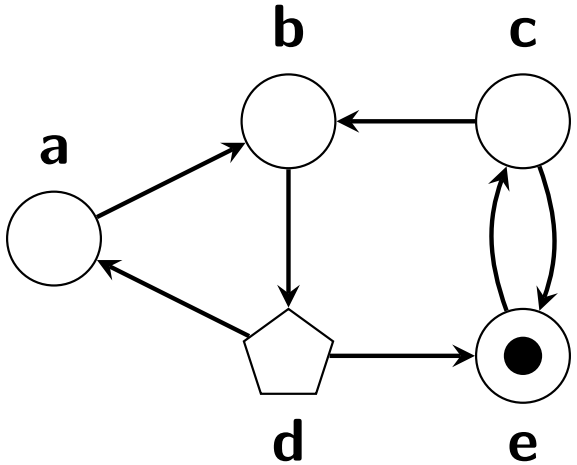
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



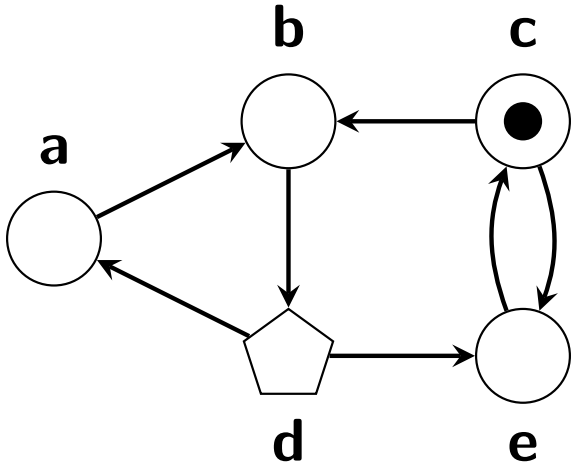
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



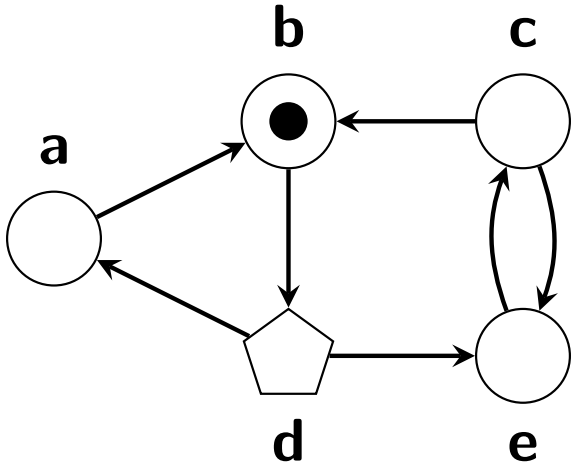
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



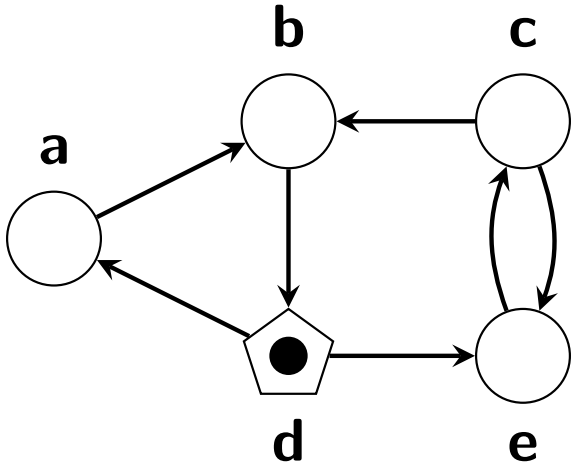
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



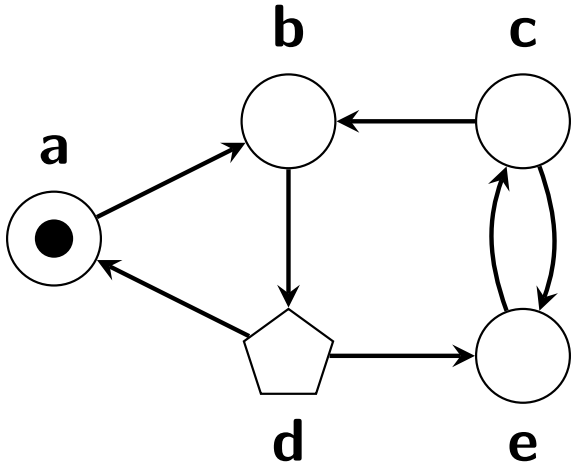
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



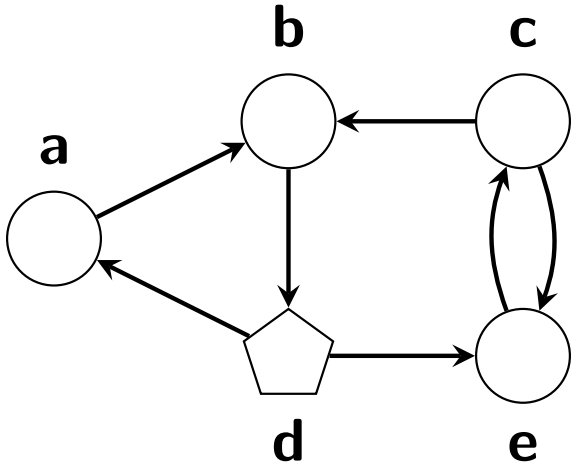
Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



Parity Games

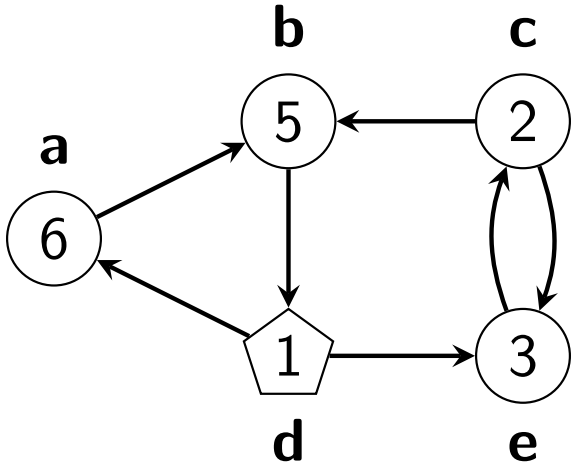
- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



How do we determine who wins a play?

Parity Games

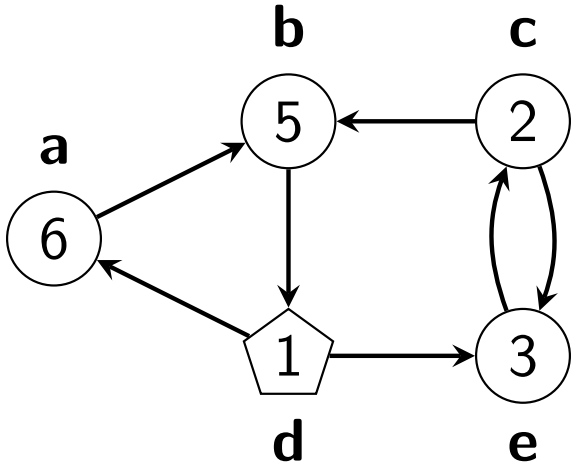
- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



- Each vertex has a priority $\{0, 1, 2, \ldots, d\}$
- Highest priority seen infinitely often determines winner
- Player Even wins if this number is even

Parity Games

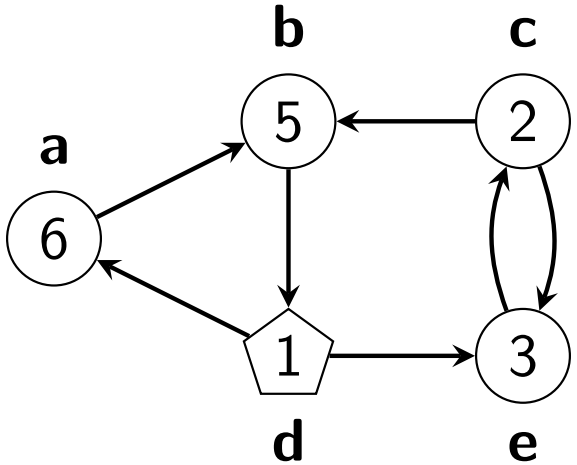
- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



How do we determine who wins a vertex?

Parity Games

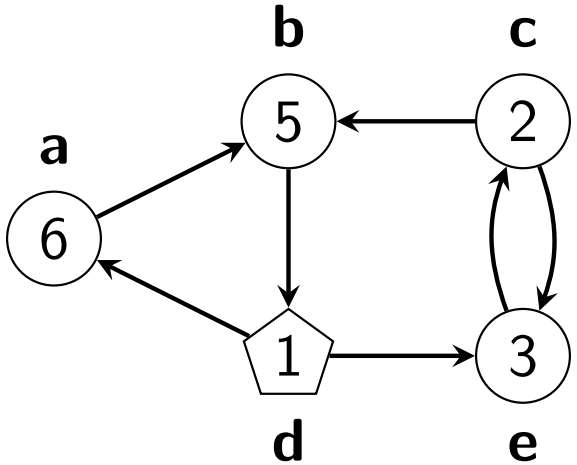
- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



A player wins a vertex if it has a strategy
to win all plays from that vertex

Parity Games

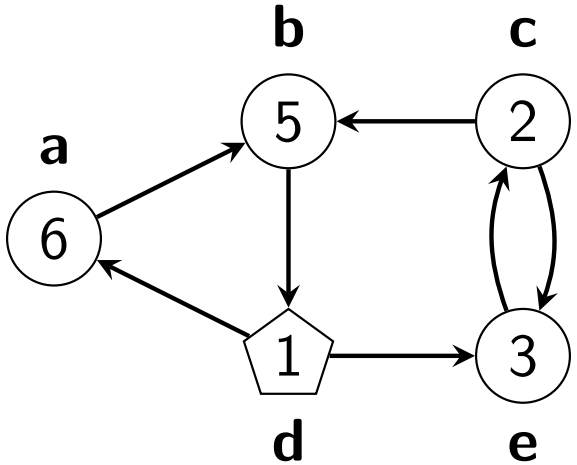
- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



Which vertices are won by which player?

Parity Games

- A parity game is played on a directed graph
- Two players: Even ○ and Odd ⬠
- The players move a token along the edges of the graph
- Each vertex is owned by one player who chooses a successor



Player Odd wins all vertices with strategy $\{\mathbf{d} \to \mathbf{e}\}$

Parity Games

Known facts of parity games

- Some vertices are won by Even, some vertices are won by Odd
- The winner has a memoryless strategy to win

Memoryless strategy

"If I always play from v to w, then I win all plays from v"

Parity Games

Known facts of parity games

- Some vertices are won by Even, some vertices are won by Odd
- The winner has a memoryless strategy to win

Memoryless strategy

"If I always play from v to w, then I win all plays from v"

Solving a parity game

- Determine the winner of each vertex
- Compute the strategy for each player

Parity Games

Why do we want to solve parity games?

- As expressive as nested least and greatest fixpoint operators
- Polynomial-time equivalent to:
  - modal μ-calculus model-checking
  - solving Boolean Equation Systems
- Backend for LTL model checking and LTL synthesis

Parity Games

Why do we want to solve parity games?

- As expressive as nested least and greatest fixpoint operators
- Polynomial-time equivalent to:
  - modal μ-calculus model-checking
  - solving Boolean Equation Systems
- Backend for LTL model checking and LTL synthesis

Open question: Is solving parity games in **P**?

- The problem is in **NP** $\cap$ **co-NP**
- The problem is in **UP** $\cap$ **co-UP**
- It is believed a polynomial solution exists

Parity Games

(Incomplete list of) published algorithms

McNaughton/Zielonka	$\mathcal{O}(e \cdot n^d)$, $\mathcal{O}(2^n)$	1998
Small Progress Measures	$\mathcal{O}(d \cdot e \cdot (n/d)^{d/2})$	1998
Strategy Improvement	$\mathcal{O}(n \cdot e \cdot 2^e)$	2000
Dominion Decomposition	$\mathcal{O}(n^{\sqrt{n}})$	2006
Big Step	$\mathcal{O}(e \cdot n^{d/3})$	2007
APT	$\mathcal{O}(n^d)$	2016
Priority Promotion	Exponential	2016
Quasi-Polynomial (multiple)	$\mathcal{O}(n^{6+\log d})$	2016 – 2018
Tangle Learning	–	2018

Parity Games

Strategy improvement

1. Set initial strategy for both players
   - Odd has a random strategy
   - Even only plays profitable continuations, otherwise halts
   - This fixes a single finite/infinite play from each vertex
2. Compute for each vertex the current value of the play
   1. Reset all values to ∞
   2. Backward search from halting Even vertices
3. Greedy improve strategy for Odd based on best successor
   - For each vertex, pick successor that gives the best play
4. Repeat 2 – 3 until stable
5. Greedy improve strategy for Even based on best successor
6. Repeat 2 – 5 until stable
7. Return result

Attractor computation

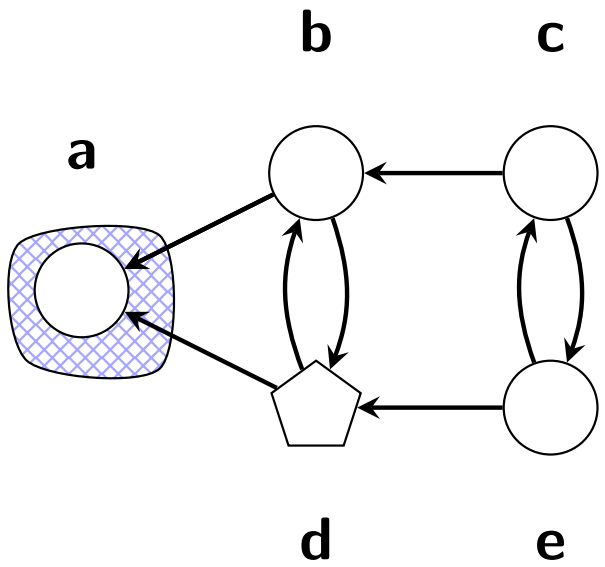
Compute all vertices from which player α can ensure arrival in a target set

Start with the target set A, then iteratively add vertices to A:

- All vertices of α with an edge to A
- All vertices of $\overline{\alpha}$ with only edges to A

Example of attractor computation

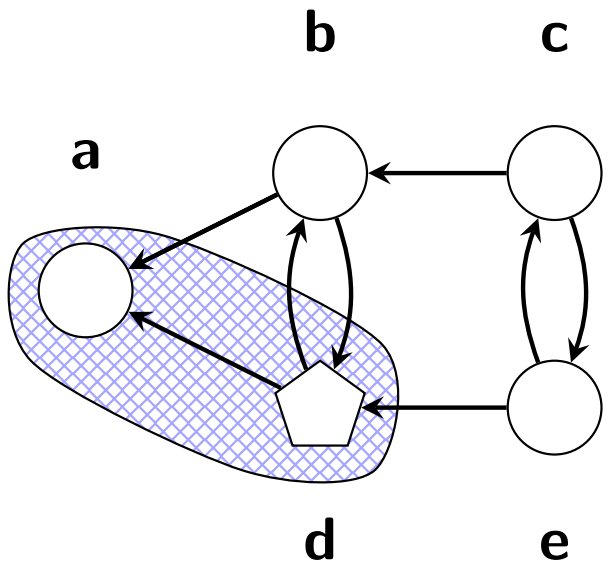
Computing the ⬠-attractor to **a**



Initial set: $\{\mathbf{a}\}$
Can attract: **d** but not **b**

Example of attractor computation

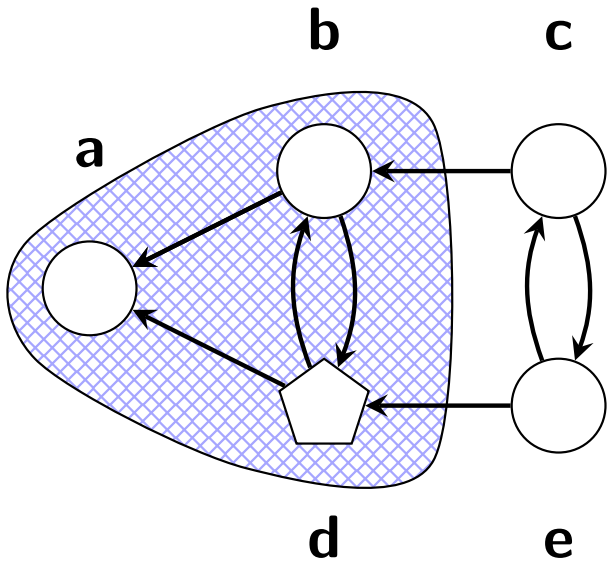
Computing the ⬠-attractor to **a**



Current set: $\{\mathbf{a}, \mathbf{d}\}$
Can attract: **b** but not **e**

Example of attractor computation

Computing the ⬠-attractor to **a**



Current set: $\{\mathbf{a}, \mathbf{b}, \mathbf{d}\}$
Can attract: neither **c** nor **e**

Parallelizing Parity Game Algorithms

Parallel McNaughton/Zielonka

- Algorithm is "many attractor computations"
- Attractor computation is a backward search
- Implement backward search with work-stealing and **atomic cas**

Parallel strategy improvement

- Initial strategy: **embarrasingly parallel**
- Reset values to ∞: **embarrasingly parallel**
- Compute backward search: **work-stealing and atomic cas**
- Greedy improve strategy: **embarrasingly parallel**

Empirical evaluation

Benchmarks

- Benchmarks by Jeroen Keiren.
- 313 model checking and 216 equivalence checking games.
- Furthermore 420 random games of up to 20000 vertices.

Solvers

- Oink *
- PGSolver (Friedmann et al)
- parallel-si (Fearnley) *
- SPGSolver (Di Stasio, Arcucci et al) *
- * = Both sequential and multi-core versions

Empirical evaluation

	Model checking		Equiv checking		Random games		Total	
zlk-1	113	0	560	0	**5**	0	**679**	0
zlk-8	111	0	781	0	26	0	918	0
pbeszlk	**67**	0	**533**	0	340	0	941	0
zlk	87	0	903	0	**5**	0	995	0
spg-seq	1262	1	2639	2	728	0	4629	3
spg-mc	474	0	4304	2	51885	39	56663	41
pgzlk	42881	32	44469	35	36859	19	124210	86

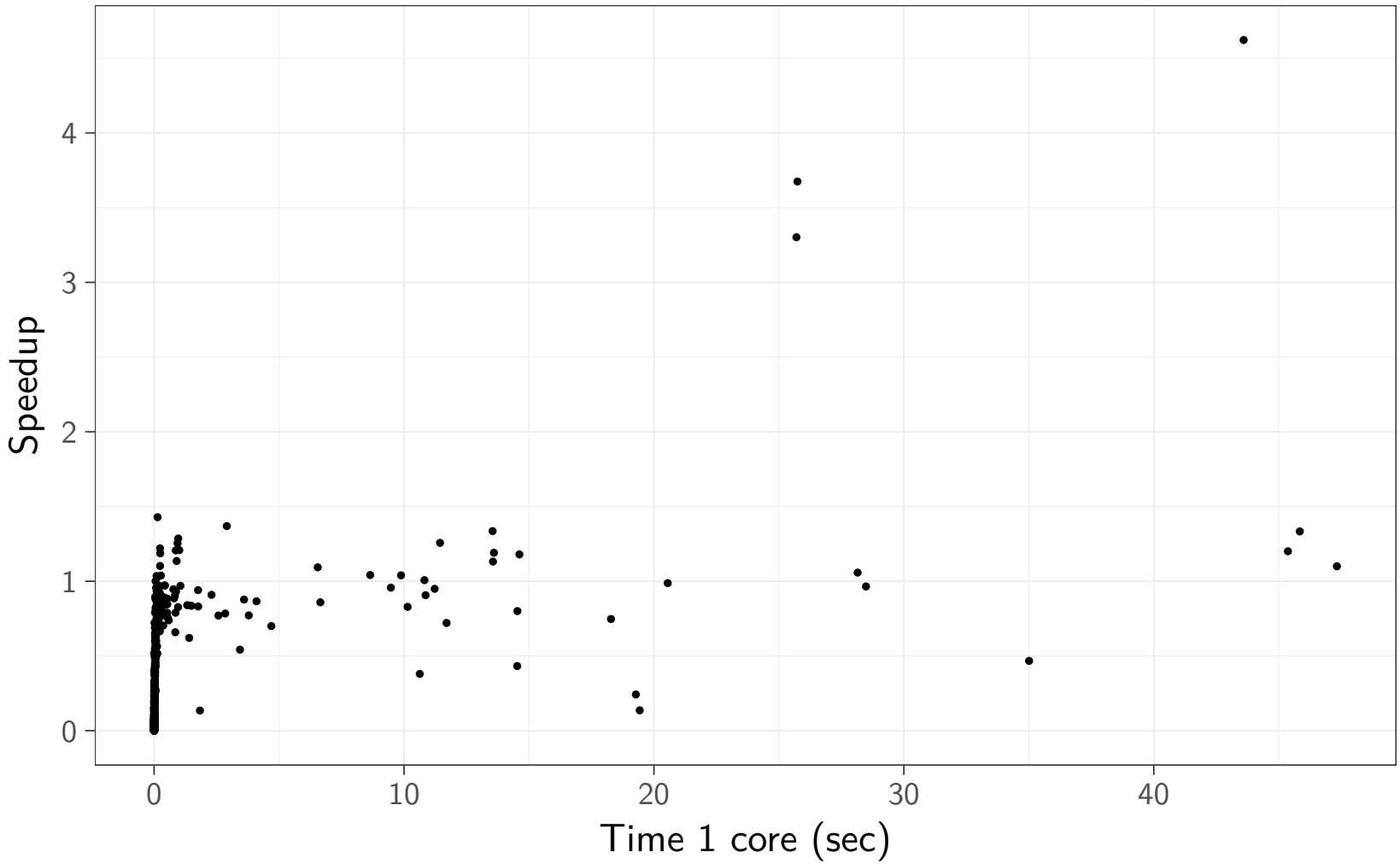
Runtimes in sec. (PAR2) and number of timeouts (10 minutes) of the four solvers PGSolver (`pgzlk`), SPGSolver (`spg`), pbespgsolve (`pbeszlk`) and Oink (`zlk`).

Empirical evaluation

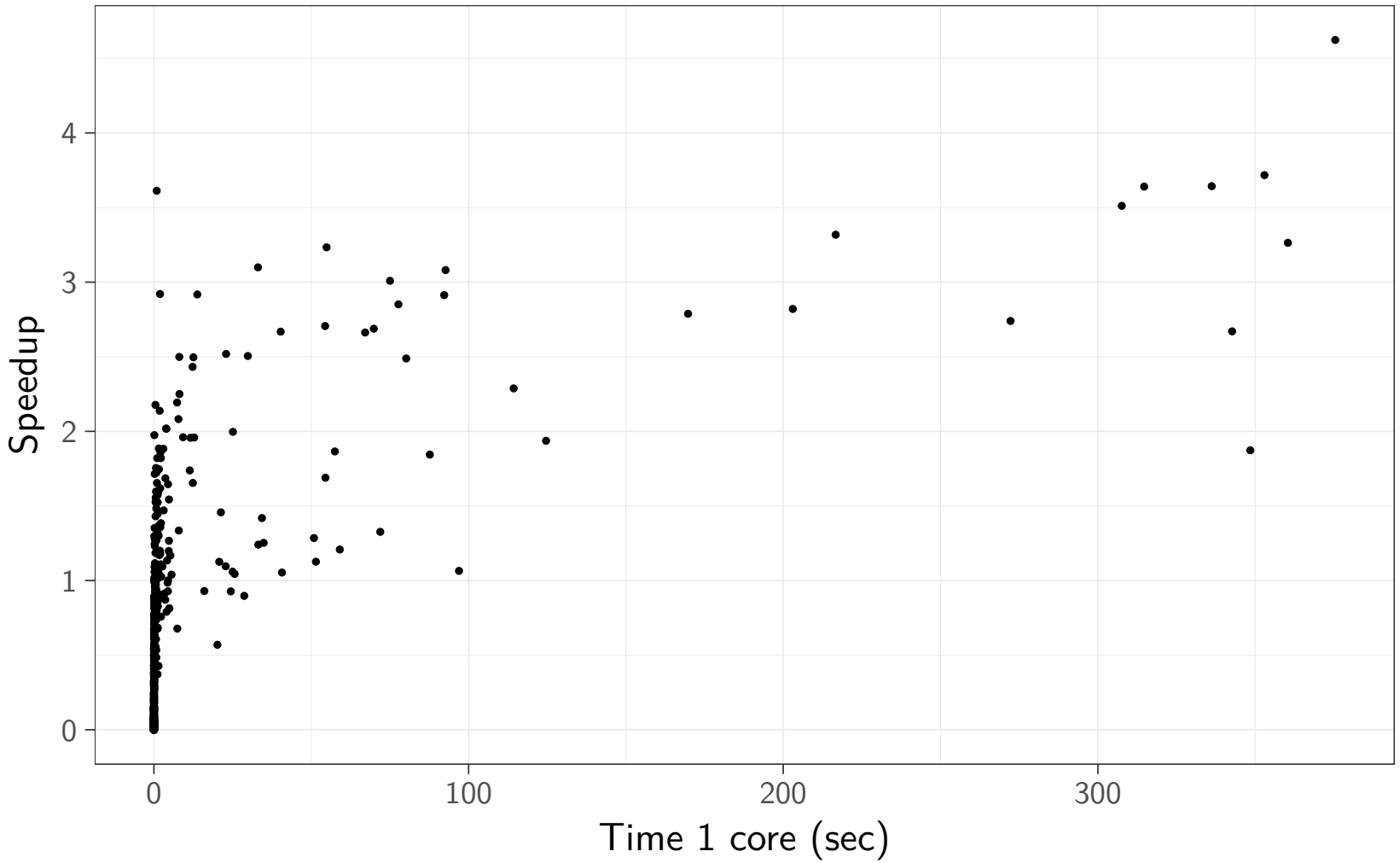
	Model checking		Equiv checking		Random games		Total	
psi-8	956	0	**1474**	0	**551**	0	**2981**	0
psi	**889**	0	4085	1	565	0	5540	1
psi-1	1771	1	6640	4	613	0	9024	5
parsi-seq	1897	1	6596	4	1697	0	10190	5
parsi-mc8	1939	1	4885	2	47217	39	54041	42
parsi-mc1	3908	2	10508	7	47633	39	62049	48
pgsi	101137	78	69492	55	39020	27	209649	160

Runtimes in sec. (PAR2) and number of timeouts (10 minutes) of the three solver PGSolver (`pgsi`), the solver by Fearnley with sequential (`parsi-seq`) and multi-core variants, and Oink with sequential (`psi`) and multi-core variants.

Empirical evaluation



Empirical evaluation



Summary

- Parity game solving algorithms can be parallelized
- Mixed results on benchmarks
- See for yourself: `https://www.github.com/trolando/oink`

Conclusions

- Work-stealing is a great way to parallelize for the multi-core
- Good results for parallel decision diagrams
- Mixed results for parallel parity game solvers
- Based on my publications in:
  - AVOCS 2012, PDMC 2012 (First results)
  - Mucocos 2013 (Lace)
  - TACAS 2015, Thesis 2016, STTT 2017 (Sylvan)
  - TACAS 2015 (symbolic model checking)
  - SETTA 2015 (probabilistic symbolic model checking)
  - TACAS 2016, STTT 2018 (symbolic bisimulation minimisation)
  - PPAM 2016, SPIN 2017 (lifted to distributed computing)
  - TACAS 2018 (parity games)