Concurrent Algorithms and Data Structures for Model Checking

Jaco van de Pol

Alfons Laarman, Tom van Dijk, Vincent Bloemen

Aarhus University, Denmark
University of Twente, The Netherlands

CONCUR + FMICS, Aug 30, 2019
Model checking requires the exploration of very large, implicit graphs. These graphs are generated from specifications (models, programs).
Model checking requires the exploration of very large, implicit graphs. These graphs are generated from specifications (models, programs).

**Smart Algorithms: exponential gains in time/memory**

- Partial Order Reduction: only representative interleavings
- Binary Decision Diagrams: concise representation with logic
- Symmetry Reduction, Abstraction, ...
Smart Parallel Algorithms for Model Checking

Model checking requires the exploration of very large, implicit graphs. These graphs are generated from specifications (models, programs).

**Smart Algorithms: exponential gains in time/memory**
- Partial Order Reduction: only representative interleavings
- Binary Decision Diagrams: concise representation with logic
- Symmetry Reduction, Abstraction, ...

**Parallel Algorithms: at most linear speedup in \# processors**
- Clusters of computers (distributed memory)
- Multi-core processors (parallel algorithms, NUMA)
- GPU (many-core, not considered here)
Model checking requires the exploration of very large, implicit graphs. These graphs are generated from specifications (models, programs).

**Smart Algorithms: exponential gains in time/memory**
- Partial Order Reduction: only representative interleavings
- Binary Decision Diagrams: concise representation with logic
- Symmetry Reduction, Abstraction, ...

**Parallel Algorithms: at most linear speedup in \# processors**
- Clusters of computers (distributed memory)
- Multi-core processors (parallel algorithms, NUMA)
- GPU (many-core, not considered here)

**Required: parallelisation of smart algorithms!**
Model checking requires the exploration of very large, implicit graphs.
These graphs are generated from specifications (models, programs).

**Smart Algorithms:** exponential gains in time/memory
- Partial Order Reduction: only representative interleavings
- Binary Decision Diagrams: concise representation with logic
- Symmetry Reduction, Abstraction, …

**Parallel Algorithms:** at most linear speedup in \# processors
- Clusters of computers (distributed memory)
- Multi-core processors (parallel algorithms, NUMA)
- GPU (many-core, not considered here)

**Required:** parallelisation of smart algorithms!
**Challenge:** efficiency = time-optimal + linear speedup
Opportunities and obstacles in parallel model checking

Distributed Model Checking

- More memory is available (NoW = Network of Workstations)
- Price: communication costs
- Main limitation: latency and throughput of the network
- Redesign algorithms (load balancing, latency hiding, speculation)

Multi-core Model Checking

- State space is available in shared memory: efficient communication
- Main limitation: memory bus contention, cache coherence, locking
- Graphs: irregular memory access (hash tables, BDDs)
- Computer architecture: from SMP to NUMA
- Efficiency: lock-free (CAS, memory barriers), be cache-line aware

In both cases, thorough experimental evaluation is important
Opportunities and obstacles in parallel model checking

Distributed Model Checking
- More memory is available \((\text{NoW} = \text{Network of Workstations})\)
- Price: communication costs
- Main limitation: latency and throughput of the network
- Redesign algorithms (load balancing, latency hiding, speculation)

Multi-core Model Checking
- State space is available in shared memory: efficient communication
- Main limitation: memory bus contention, cache coherence, locking
- Graphs: irregular memory access (hash tables, BDDs)
- Computer architecture: from SMP to NUMA
- Efficiency: lock-free (CAS, memory barriers), be cache-line aware
Opportunities and obstacles in parallel model checking

Distributed Model Checking

- More memory is available \((\text{NoW} = \text{Network of Workstations})\)
- Price: communication costs
- Main limitation: latency and throughput of the network
- Redesign algorithms (load balancing, latency hiding, speculation)

Multi-core Model Checking

- State space is available in shared memory: efficient communication
- Main limitation: memory bus contention, cache coherence, locking
- Graphs: irregular memory access (hash tables, BDDs)
- Computer architecture: from SMP to NUMA
- Efficiency: lock-free (CAS, memory barriers), be cache-line aware

In both cases, thorough experimental evaluation is important
History: successful PDMC workshop series (2002-2012)

- 1990 Kimura & Clarke, parallel algorithm for constructing BDDs (SMP)
- 1997 Stern & Dill, parallelizing the Mur$\phi$ verifier (NoW, MPI)
History: successful PDMC workshop series (2002-2012)

- 1990 Kimura & Clarke, parallel algorithm for constructing BDDs (SMP)
- 1997 Stern & Dill, parallelizing the Mur$\phi$ verifier (NoW, MPI)
- 2000 Heyman, Grumberg & Schuster, distributed symbolic model checking
- 2001 Barnat, Brim & Cerna, distributed LTL model checking (DiVinE)
- 2002 Behrman etal, distributed timed model checking (Uppaal)
- 2003 Blom, Lisser, Orzan, vdPol, Weber, distributed bisimulation ($\mu$CRL)
- 2004 Chung & Ciardo, saturation NoW (symbolic reachability for Petri nets)
History: successful PDMC workshop series (2002-2012)

- 1990 Kimura & Clarke, parallel algorithm for constructing BDDs (SMP)
- 1997 Stern & Dill, parallelizing the Mur\(\phi\) verifier (NoW, MPI)
- 2000 Heyman, Grumberg & Schuster, distributed symbolic model checking
- 2001 Barnat, Brim & Cerna, distributed LTL model checking (DiVinE)
- 2002 Behrman etal, distributed timed model checking (Uppaal)
- 2003 Blom, Lisser, Orzan, vdPol, Weber, distributed bisimulation (\(\mu\)CRL)
- 2004 Chung & Ciardo, saturation NoW (symbolic reachability for Petri nets)
- 2007 Barnat, Brim & Rockai, scalable multi-core LTL model checking
- 2007 Holzman & Bosnacki, multi-core model checking with SPIN
- 2008 Holzman, Joshi & Groce, swarmed verification with SPIN
- 2009 Ciardo, parallel symbolic reachability is difficult
History: successful PDMC workshop series (2002-2012)

- 1990 Kimura & Clarke, parallel algorithm for constructing BDDs (SMP)
- 1997 Stern & Dill, parallelizing the Murϕ verifier (NoW, MPI)
- 2000 Heyman, Grumberg & Schuster, distributed symbolic model checking
- 2001 Barnat, Brim & Cerna, distributed LTL model checking (DiVinE)
- 2002 Behrman etal, distributed timed model checking (Uppaal)
- 2003 Blom, Lisser, Orzan, vdPol, Weber, distributed bisimulation (µCRL)
- 2004 Chung & Ciardo, saturation NoW (symbolic reachability for Petri nets)
- 2007 Barnat, Brim & Rockai, scalable multi-core LTL model checking
- 2007 Holzman & Bosnacki, multi-core model checking with SPIN
- 2008 Holzman, Joshi & Groce, swarmed verification with SPIN
- 2009 Ciardo, parallel symbolic reachability is difficult

Parallel model checking compromises on worst case performance

**Challenge:** scalable & efficient multi-core model checking
History: successful PDMC workshop series (2002-2012)

- 1990 Kimura & Clarke, parallel algorithm for constructing BDDs (SMP)
- 1997 Stern & Dill, parallelizing the Murϕ verifier (NoW, MPI)
- 2000 Heyman, Grumberg & Schuster, distributed symbolic model checking
- 2001 Barnat, Brim & Cerna, distributed LTL model checking (DiVinE)
- 2002 Behrmann etal, distributed timed model checking (Uppaal)
- 2003 Blom, Lisser, Orzan, vdPol, Weber, distributed bisimulation (µCRL)
- 2004 Chung & Ciardo, saturation NoW (symbolic reachability for Petri nets)
- 2007 Barnat, Brim & Rockai, scalable multi-core LTL model checking
- 2007 Holzman & Bosnacki, multi-core model checking with SPIN
- 2008 Holzman, Joshi & Groce, swarmed verification with SPIN
- 2009 Ciardo, parallel symbolic reachability is difficult

Parallel model checking compromises on worst case performance

**Challenge: scalable & efficient multi-core model checking**

- 2011 Laarman & vdPol, Multi-core Nested DFS
- 2013 Van Dijk & vdPol, Scalable multi-core BDD algorithms
- 2016 Bloemen & vdPol, Multi-core DFS SCC algorithm
3 PhD theses from University of Twente

- Alfons Laarman: Parallel Nested Depth-First Search (2010-2014)
  - lock-free hashtable, state compression (make-over: Freark vd Berg)
  - parallel NDFS (now formally verified by Wytse Oortwijn)
  - compatible with partial-order reduction: LTL-X model checking

- Tom van Dijk: Concurrent Decision Diagrams (2012-2016)
  - concurrent garbage collection, lossy cache, task scheduler
  - parallel symbolic reachability, bisimulation minimisation, saturation
  - heterogeneous distributed + multi-core version (Wytse Oortwijn)

- Vincent Bloemen: Parallel Strongly Connected Components (2016-2019)
  - based on DFS and sharing info on partial SCCs
  - concurrent Union-Find structure + iterable cyclic list
  - LTL model checking with Büchi automata, Rabin automata, etc.
3 PhD theses from University of Twente

**Alfons Laarman: Parallel Nested Depth-First Search (2010-2014)**
- lock-free hashtable, state compression (make-over: Freark vd Berg)
- parallel NDFS (now formally verified by Wytse Oortwijn)
- compatible with partial-order reduction: LTL-X model checking
### 3 PhD theses from University of Twente

<table>
<thead>
<tr>
<th>Alfons Laarman: Parallel Nested Depth-First Search (2010-2014)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- lock-free hashtable, state compression (make-over: Freark vd Berg)</td>
</tr>
<tr>
<td>- parallel NDFS (now formally verified by Wytse Oortwijn)</td>
</tr>
<tr>
<td>- compatible with partial-order reduction: LTL-X model checking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tom van Dijk: Concurrent Decision Diagrams (2012-2016)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- concurrent garbage collection, lossy cache, task scheduler</td>
</tr>
<tr>
<td>- parallel symbolic reachability, bisimulation minimisation, saturation</td>
</tr>
<tr>
<td>- heterogeneous distributed + multi-core version (Wytse Oortwijn)</td>
</tr>
</tbody>
</table>
3 PhD theses from University of Twente

**Alfons Laarman: Parallel Nested Depth-First Search (2010-2014)**
- lock-free hashtable, state compression (make-over: Freark vd Berg)
- parallel NDFS (now formally verified by Wytse Oortwijn)
- compatible with partial-order reduction: LTL-X model checking

**Tom van Dijk: Concurrent Decision Diagrams (2012-2016)**
- concurrent garbage collection, lossy cache, task scheduler
- parallel symbolic reachability, bisimulation minimisation, saturation
- heterogeneous distributed + multi-core version (Wytse Oortwijn)

**Vincent Bloemen: Parallel Strongly Connected Components (2016-2019)**
- based on DFS and sharing info on partial SCCs
- concurrent Union-Find structure + iterable cyclic list
- LTL model checking with Büchi automata, Rabin automata, etc.
Overview

1. Introduction

2. Strongly Connected Components
   - A simple parallel SCC algorithm
   - Dijkstra’s sequential SCC algorithm
   - A parallel DFS algorithm for SCCs

3. Multicore Model Checking
   - Explicit-state LTL model checking
   - Symbolic model checking
   - LTSmin: high-performance model checker

4. Conclusion
Strongly Connected Component (SCC)

Setting: finite graph with directed edges
SCCs: maximal components of $\rightarrow \cap \leftarrow$
Strongly Connected Component (SCC)

Setting: finite graph with directed edges
SCCs: maximal components of $\rightarrow \cap \leftarrow$
Strongly Connected Component (SCC)

Setting: finite graph with directed edges
SCCs: maximal components of $\rightarrow \cap \leftarrow$

Applications: LTL model checking, fairness, evaluation of Markov Chains
Forward-Backward (FB) parallel SCC Algorithm

1. Select a pivot node
Forward-Backward (FB) parallel SCC Algorithm

2. Compute its forward reachable set (F)
3. Compute its backward reachable set (B)
4. The intersection $F \cap B$ is the SCC of the pivot.
Forward-Backward (FB) parallel SCC Algorithm

4. The intersection $F \cap B$ is the SCC of the pivot

Remaining slices can be processed independently in parallel
Finding SCCs on-the-fly (path-based algorithm)

For model checking, an on-the-fly SCC algorithm is preferable:
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function
Finding SCCs on-the-fly (path-based algorithm)

For model checking, an on-the-fly SCC algorithm is preferable:
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function

NB: this is not yet necessarily a maximal SCC, since its successors are not completely explored
Finding SCCs on-the-fly (path-based algorithm)

For model checking, an on-the-fly SCC algorithm is preferable:
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function

NB: this is not yet necessarily a maximal SCC, since its successors are not completely explored
For model checking, an on-the-fly SCC algorithm is preferable:

- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function

NB: this is not yet necessarily a maximal SCC, since its successors are not completely explored
Finding SCCs on-the-fly (path-based algorithm)

For model checking, an on-the-fly SCC algorithm is preferable:
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function

NB: this is not yet necessarily a maximal SCC, since its successors are not completely explored
Finding SCCs on-the-fly (path-based algorithm)

For model checking, an on-the-fly SCC algorithm is preferable:

- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function

NB: this is not yet necessarily a maximal SCC, since its successors are not completely explored
Finding SCCs on-the-fly (path-based algorithm)

For model checking, an on-the-fly SCC algorithm is preferable:

- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function

NB: this is not yet necessarily a maximal SCC, since its successors are not completely explored
Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure [Tarjan, van Leeuwen, J ACM 1984]:

- supports disjoint subsets, which can be merged
- basic functions: Union and Find (unique representative)

Reversed forest, nodes direct towards their representative root
Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure [Tarjan, van Leeuwen, J ACM 1984]:
- supports disjoint subsets, which can be merged
- basic functions: Union and Find (unique representative)

### Find(d)
- recursively searches the parent edges to find the root
Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure [Tarjan, van Leeuwen, J ACM 1984]:
- supports disjoint subsets, which can be merged
- basic functions: Union and Find (unique representative)

Unite(f,d): Find the roots of f and d,
Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure [Tarjan, van Leeuwen, J ACM 1984]:
- supports disjoint subsets, which can be merged
- basic functions: Union and Find (unique representative)

Unite(f,d): Find the roots of f and d, and update one of them
Dijkstra’s SCC Algorithm [1976]

Uses stack $R$ (push, pop, top) and disjoint-set $S$ (union, find, enum).
Also maintains sets $Visited$ and $Explored$, initially $\emptyset$

```plaintext

1  procedure SCC(v)
2    Visited := Visited $\cup \{v\}$
3    R.push(v)
4    for each $w \in$ next_state(v)
5
```

Jaco van de Pol, Aarhus+Twente
Dijkstra’s SCC Algorithm [1976]

Uses stack $R$ (push, pop, top) and disjoint-set $S$ (union, find, enum).
Also maintains sets $Visited$ and $Explored$, initially $\emptyset$.

```
1 procedure SCC(v)
2 · Visited := Visited $\cup \{v\}$
3 · $R.push(v)$
4 · for each $w \in$ next_state($v$)
5 · · if $w \in$ Explored // complete SCC
6 · · then continue
7 · · else if $w \notin$ Visited // unseen state
8 · · then SCC($w$)
9 · · else // cycle found
10```

Jaco van de Pol, Aarhus+Twente
Concurrency for Model Checking 11 / 40
Dijkstra’s SCC Algorithm [1976]

Uses stack $R$ (push, pop, top) and disjoint-set $S$ (union, find, enum).
Also maintains sets $Visited$ and $Explored$, initially $\emptyset$.

```
procedure SCC($v$)
    Visited := Visited $\cup \{v\}$
    $R$.push($v$)
    for each $w \in$ next_state($v$)
        if $w \in$ Explored
            then continue
        else if $w \notin$ Visited
            then SCC($w$)
        else
            while $S$.find($v$) $\neq$ $S$.find($w$) do
                $S$.union($R$.pop(), $R$.top())
    $R$.pop()
```

Dijkstra’s SCC Algorithm [1976]

Uses stack \( R \) (push, pop, top) and disjoint-set \( S \) (union, find, enum).
Also maintains sets \( Visited \) and \( Explored \), initially \( \emptyset \).

1. procedure \( SCC(v) \)
2. \( \cdot \) \( Visited := Visited \cup \{v\} \)
3. \( \cdot \) \( R.push(v) \)
4. \( \cdot \) \( \text{for each } w \in \text{next}_{\text{state}}(v) \)
5. \( \cdot \) \( \cdot \) \( \text{if } w \in \text{Explored} \quad \text{// complete SCC} \)
6. \( \cdot \) \( \cdot \) \( \text{then continue} \)
7. \( \cdot \) \( \cdot \) \( \text{else if } w \not\in \text{Visited} \quad \text{// unseen state} \)
8. \( \cdot \) \( \cdot \) \( \text{then } SCC(w) \)
9. \( \cdot \) \( \cdot \) \( \text{else} \quad \text{// cycle found} \)
10. \( \cdot \) \( \cdot \) \( \text{while } S.find(v) \neq S.find(w) \text{ do} \)
11. \( \cdot \) \( \cdot \) \( S.union(R.pop(), R.top()) \)
12. \( \cdot \) \( \cdot \) \( \text{if } v = R.top() \text{ then} \quad \text{// completed SCC} \)
13. \( \cdot \) \( \cdot \) \( \text{report } SCC \ S.enum(v) \)
14. \( \cdot \) \( \cdot \) \( \text{Explored := Explored } \cup S.enum(v) \)
15. \( \cdot \) \( \cdot \) \( R.pop() \)
### Comparing SCC algorithms

<table>
<thead>
<tr>
<th>Tarjan’s SCC algorithm [1972]</th>
<th>Forward-Backward algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Worst case $O(m + n)$</td>
<td>- Worst case $O(n(m + n))$</td>
</tr>
<tr>
<td>- On-the-fly</td>
<td>- Requires predecessors</td>
</tr>
<tr>
<td>- Inherently DFS</td>
<td>- BFS is sufficient</td>
</tr>
</tbody>
</table>
### Comparing SCC algorithms

<table>
<thead>
<tr>
<th>Tarjan’s SCC algorithm [1972]</th>
<th>Forward-Backward algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Worst case $O(m + n)$</td>
<td>- Worst case $O(n(m + n))$</td>
</tr>
<tr>
<td>- On-the-fly</td>
<td>- Requires predecessors</td>
</tr>
<tr>
<td>- Inherently DFS</td>
<td>- BFS is sufficient</td>
</tr>
</tbody>
</table>

**Variants (DFS-based):**
- double DFS (transposed graph)
  - Kosaraju‘78, Sharir‘81
- path-based SCC algorithms
  - Purdom‘70, Munro‘71, Dijkstra‘76

**Variants (BFS-based):**
- original FB algorithm
  - Fleischer, Hendrickson, Pinar [‘00]
- R-OBF: trims trivial SCCs
  - Barnat, Chaloupka, vdPol [‘09]
Comparing SCC algorithms

<table>
<thead>
<tr>
<th>Tarjan’s SCC algorithm [1972]</th>
<th>Forward-Backward algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Worst case $O(m + n)$</td>
<td>• Worst case $O(n(m + n))$</td>
</tr>
<tr>
<td>• On-the-fly</td>
<td>• Requires predecessors</td>
</tr>
<tr>
<td>• Inherently DFS</td>
<td>• BFS is sufficient</td>
</tr>
</tbody>
</table>

Variants (DFS-based):
- double DFS (transposed graph)
  - Kosaraju‘78, Sharir‘81
- path-based SCC algorithms
  - Purdom‘70, Munro‘71, Dijkstra‘76

Variants (BFS-based):
- original FB algorithm
  - Fleischer, Hendrickson, Pinar ['00]
- R-OBF: trims trivial SCCs
  - Barnat, Chaloupka, vdPol ['09]

FB is easier to parallelize, but harder to use and less efficient!
### Comparing SCC algorithms

<table>
<thead>
<tr>
<th>Tarjan’s SCC algorithm [1972]</th>
<th>Forward-Backward algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Worst case $O(m + n)$</td>
<td>- Worst case $O(n(m + n))$</td>
</tr>
<tr>
<td>- On-the-fly</td>
<td>- Requires predecessors</td>
</tr>
<tr>
<td>- Inherently DFS</td>
<td>- BFS is sufficient</td>
</tr>
</tbody>
</table>

**Variants (DFS-based):**
- double DFS (transposed graph)  
  Kosaraju‘78, Sharir‘81
- path-based SCC algorithms  
  Purdom‘70, Munro‘71, Dijkstra‘76

**Variants (BFS-based):**
- original FB algorithm  
  Fleischer, Hendrickson, Pinar [‘00]
- R-OBF: trims trivial SCCs  
  Barnat, Chaloupka, vdPol [‘09]

**FB is easier to parallelize, but harder to use and less efficient!**

**Complexity theory of parallel graph algorithms:**
- Reif (1985): *Depth-First Search is inherently sequential* (P-complete)
- Amato (1993): SSSP in $O(\log^2(n))$ time on $O(n^{2.376})$ processors
Inspiration: Swarmed Verification (Holzmann, Spin) for bug finding

Nested Depth-First Search for LTL model checking
Inspiration: Swarmed Verification (Holzmann, Spin) for bug finding

Nested Depth-First Search for LTL model checking

- Every worker performs its own NDFS in a randomized direction
- Parallel search speeds up finding bugs only: duplicate work
Inspiration: Swarmed Verification (Holzmann, Spin) for bug finding

Nested Depth-First Search for LTL model checking

- Every worker performs its own NDFS in a randomized direction
- Parallel search speeds up finding bugs only: duplicate work
- EP 2011: Share much, repair violations of DFS order: sequential work
- LvdP 2011: Share less, avoid violations of DFS order: some locking
Parallel Random DFS for SCCs

Parallel DFS + random successor order + sharing information on SCCs

What happens if two workers start working on the same SCC?

G. Lowe (TACAS'14): suspend and sequential repair procedure
E. Renault et al. (TACAS'15): share complete SCCs only
V. Bloemen et al. (PPoPP'16): share partial SCCs as well

Jaco van de Pol, Aarhus+Twente
Concurrency for Model Checking 14 / 40
Parallel Random DFS for SCCs

Parallel DFS + random successor order + sharing information on SCCs

What happens if two workers start working on the same SCC?

- G. Lowe (TACAS’14): suspend and sequential repair procedure
- E. Renault et al. (TACAS’15): share complete SCCs only
- V. Bloemen et al. (PPoPP’16): share partial SCCs as well
Handling Small and Large SCCs Sequentially

Small SCCs

- Parallelizes well

Large SCCs

- No performance gain

**Bottom line:** we cannot afford to handle single SCCs sequentially
Speedup in practice

- **Small SCCs**
- **a Large SCC**

Graphs showing speedup vs. the number of workers for both small and large strongly connected components (SCCs) compared to Tarjan's algorithm.
Speedup in practice

**Small SCCs**

- Graph showing speedup vs number of workers for Small SCCs.
- The graph includes data points for both Tarjan and Renault algorithms.

**a Large SCC**

- Graph showing speedup vs number of workers for a Large SCC.
- The graph includes data points for both Tarjan and Renault algorithms.

- Legend: Tarjan (green line) and Renault (red triangles).

---

Jaco van de Pol, Aarhus+Twente
Concurrency for Model Checking 16 / 40
Speedup in practice

**Small SCCs**

**a Large SCC**

![Graph showing speedup vs workers for small SCCs and a large SCC.](image)

- **X-axis:** Number of workers
- **Y-axis:** Speedup vs Tarjan

Legend:
- **Tarjan** (green line)
- **Renault** (red triangles)
- **Bloemen** (blue circles)
Blue worker happens to visit $a \rightarrow b \rightarrow c \rightarrow d$
Blue worker happens to visit $a \rightarrow b \rightarrow c \rightarrow d$
Blue worker happens to visit $a \rightarrow b \rightarrow c \rightarrow d$
Blue worker happens to visit $a \rightarrow b \rightarrow c \rightarrow d$
Blue worker detects and unites partial SCC \( \{a, b, c, d\} \)
Red worker happens to visit $a \rightarrow e \rightarrow f$
Red worker happens to visit $a \rightarrow e \rightarrow f$
UF-SCC: Communicate partially found SCCs [Bloemen]

Red worker detects and unites partial SCC \{e, f\}
Red worker continues exploration $f \rightarrow c$
But how does Red worker know that it visited a state “equivalent” to c?
Union-Find with a worker set

Store a bit-set of worker IDs in the union-find roots
Check if the partial SCC of the successor has been *visited* before

But how do we know when the SCC is complete?

Jaco van de Pol, Aarhus+Twente
Union-Find with a worker set

Check if the partial SCC of the successor has been *visited* before
Union-Find with a worker set

Check if the partial SCC of the successor has been visited before

But how do we know when the SCC is complete?
Distinguish fully explored states

- **Track** which states of the SCC still have to be explored
  - An SCC is complete if all its states have been fully explored
- **Evenly distribute** the remaining work
  - Otherwise one worker may end up doing all the work
Cyclic list of BUSY states

- **BUSY**: There may be some *unexplored successors* from this state
- **DONE**: This state has been *fully explored* by some worker
- Workers can *concurrently* pick states from the cyclic list
List operations

<table>
<thead>
<tr>
<th>Merge lists (Union)</th>
<th>Remove element</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td></td>
</tr>
</tbody>
</table>

Jaco van de Pol, Aarhus+Twente
List operations

<table>
<thead>
<tr>
<th>Merge lists (Union)</th>
<th>Remove element</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Jaco van de Pol, Aarhus+Twente
Concurrency for Model Checking 22 / 40
Algorithm: code for worker $p$

Uses local stacks $R_p$ (push, pop, top) and shared disjoint-set $S$ (union, find, claim, equal, cyclic list)

```
procedure UFSCC_p(v)
  · S.claim(v,p) // Add $p$ to workers, $v$ to cyclic list
  · $R_p$.push(v)
  · while $v' := S.PickFromList(v)$
    · · for each $w \in \text{randomize}(\text{next\_state}(v'))$
    · · ·
    · · ·
    · · ·
    · · ·
    · · ·
  · S.RemoveFromList(v')
  · if $v = R_p.top()$ then report $R_p.pop()$ // report the SCC
```
Algorithm: code for worker $p$

Uses local stacks $R_p$ (push, pop, top) and shared disjoint-set $S$ (union, find, claim, equal, cyclic list)

```
procedure UFSCC$_p$(v)
  · $S$.claim(v,p)  // Add $p$ to workers, $v$ to cyclic list
  · $R_p$.push(v)
  · while $v' := S$.PickFromList(v)
    · · for each $w \in \text{randomize(next\_state(v'))}$
    · · · if $w \in \text{DEAD}$  // ignore completed SCC
    · · · · then continue
    · · · else if $p \notin S$.find(w)  // state yet unseen by $p$
    · · · · then UFSCC$_p$(w)
    · · · else
    · · · · while $\neg S$.equal(v,w)  // merge states on cycle
    · · · · · $S$.union($R_p$.pop(), $R_p$.top())
    · · · · $S$.RemoveFromList(v')
    · · · if $v = R_p$.top() then report $R_p$.pop()  // report the SCC
```
Time Complexity and Speed-Up

- $n$: number of states (nodes), $m$: number of transitions (edges)
- $\alpha(n)$: inverse of Ackermann function (amortized complexity of UF)
- $p$: number of workers
Time Complexity and Speed-Up

- $n$: number of states (nodes), $m$: number of transitions (edges)
- $\alpha(n)$: inverse of Ackermann function (amortized complexity of UF)
- $p$: number of workers

In the worst case, all workers visit the whole graph in lockstep, so total amount of work is $O((m + n).\alpha(n).p)$: linear-time, but no speed-up

Model checking graphs are “broad”, so workers spread out evenly. Observed wall clock: $O((m + n).\alpha(n)/p)$: linear-time and linear speed-up
Time Complexity and Speed-Up

- $n$: number of states (nodes), $m$: number of transitions (edges)
- $\alpha(n)$: inverse of Ackermann function (amortized complexity of UF)
- $p$: number of workers

In the worst case, all workers visit the whole graph in lockstep, so total amount of work is $O((m + n)\cdot \alpha(n)\cdot p)$: linear-time, but no speed-up

Model checking graphs are “broad”, so workers spread out evenly. Observed wall clock: $O((m + n)\cdot \alpha(n)/p)$: linear-time and linear speed-up

Can we guarantee even more? Maybe!

- S.V. Jayanti, R.E. Tarjan, E. Boix-Adserà [PODC’19]
  
  Randomized Concurrent Set Union and Generalized Wake-Up reports the first concurrent union-find algorithm with a total work complexity that grows sublinear in $p$, the number of processes.
Speedup graphs of selected BEEM models

Consistent * Complete* WellDocumented* EasytoReuse*

Evaluated PoP Artifact * AEC

Number of workers

Speedup vs Tarjan

leader-filters.7

bakery.6

cambridge.6

lup.3

resistance.1

sorter.3

Number of workers

Number of workers

Number of workers
Speedup graphs of selected BEEM models

- **leader-filters.7**
- **bakery.6**
- **cambridge.6**
- **lup.3**
- **resistance.1**
- **sorter.3**

Each graph shows the speedup vs Tarjan for different models with varying numbers of workers. The models include leader-filters, bakery, cambridge, lup, resistance, and sorter.
Overview

1. Introduction

2. Strongly Connected Components
   - A simple parallel SCC algorithm
   - Dijkstra’s sequential SCC algorithm
   - A parallel DFS algorithm for SCCs

3. Multicore Model Checking
   - Explicit-state LTL model checking
   - Symbolic model checking
   - LTSmin: high-performance model checker

4. Conclusion
Recall: Automata-based LTL model checking

Model $\mathcal{M}$

State space generation

Model automaton $A_\mathcal{M}$

LTL formula $\varphi$

Negated formula $\neg \varphi$

LTL to Büchi

Negated formula automaton $A_{\neg \varphi}$

Synch. product $A_\mathcal{M} \otimes A_{\neg \varphi}$

Emptiness check $\mathcal{L}(A_\mathcal{M} \otimes A_{\neg \varphi}) \not\models \emptyset$

$\mathcal{M} \models \varphi$

Counterexample
Recall: Automata-based LTL model checking

Model $\mathcal{M}$

State space generation

Model automaton $A_\mathcal{M}$

LTL formula $\varphi$

Negated formula $\neg \varphi$

LTL to Büchi

Negated formula automaton $A_{\neg \varphi}$

Synch. product $A_\mathcal{M} \otimes A_{\neg \varphi}$

Emptiness check $\mathcal{L}(A_\mathcal{M} \otimes A_{\neg \varphi}) \models \emptyset$

$\mathcal{M} \models \varphi$

Counterexample

BA

TGBA
Recall: Automata-based LTL model checking

- Model $\mathcal{M}$
- State space generation
- Model automaton $A_M$
- LTL formula $\varphi$
- LTL to Büchi
- Negated formula $\neg \varphi$
- Negated formula automaton $A_{\neg \varphi}$
- Synch. product $A_M \otimes A_{\neg \varphi}$
- Emptiness check $\mathcal{L}(A_M \otimes A_{\neg \varphi}) \models \emptyset$
- $\mathcal{M} \models \varphi$ Counterexample
- $\text{BA}$
- $\text{TGBA}$
- Rabin?
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Blue worker explores $a \rightarrow b \rightarrow e \rightarrow d$
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Blue worker explores $a \rightarrow b \rightarrow e \rightarrow d$
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Blue worker explores $a \rightarrow b \rightarrow e \rightarrow d$
LTL model checking reduces to the following graph problem: Find a reachable accepting SCC in a Büchi-automaton.

Blue worker explores $a \rightarrow b \rightarrow e \rightarrow d$.
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Red worker explores $a \rightarrow b \rightarrow c$
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Red worker explores $a \rightarrow b \rightarrow c$
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Blue worker detects and shares partial SCC \{b, d, e\}
LTL model checking reduces to the following graph problem:

Find a reachable accepting SCC in a Büchi-automaton

Red worker detects complete, accepting SCC \{b, c, d, e, f\}
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Red worker detects complete, accepting SCC \( \{b, c, d, e, f\} \)
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

Accepting cycle has been found, while no single worker traversed it!
Accepting Cycle for TGBA

Transition-based Generalized Büchi Automata

\[ \text{Inf}(0) \land \text{Inf}(1) \]

Advantage: TGBA are more concise and natural for LTL
Accepting Cycle for TGBA

Transition-based Generalized Büchi Automata

$$\text{Inf}(0) \land \text{Inf}(1)$$

Advantage: TGBA are more concise and natural for LTL
Accepting Cycle for TGBA

Transition-based Generalized Büchi Automata

\[ \text{Inf}(0) \land \text{Inf}(1) \]

Advantage: TGBA are more concise and natural for LTL

Store all encountered accepting marks at the UF-root
Accepting Cycle for Generalized Rabin Automata

\[ \text{Fin}(0) \land \text{Inf}(1) \land \text{Inf}(2) \]

[Bloemen, Duret-Lutz, vdPol, SPIN 2017]
Can handle all Rabin conditions sequentially or in parallel
Accepting Cycle for Generalized Rabin Automata

\[ \text{Fin}(0) \land \text{Inf}(1) \land \text{Inf}(2) \]

Can handle all Rabin conditions sequentially or in parallel

[Bloemen, Duret-Lutz, vdPol, SPIN 2017]
Accepting Cycle for Generalized Rabin Automata

\[ \text{Fin}(0) \land \text{Inf}(1) \land \text{Inf}(2) \]

[Bloemen, Duret-Lutz, vdPol, SPIN 2017]
Can handle all Rabin conditions sequentially or in parallel
Accepting Cycle for Generalized Rabin Automata

\[ \text{Fin}(0) \land \text{Inf}(1) \land \text{Inf}(2) \]

[Bloemen, Duret-Lutz, vdPol, SPIN 2017]
Can handle all Rabin conditions sequentially or in parallel
Adapt the UF-SCC procedure by postponing “fin”–labels
Binary Decision Diagrams

- Concise, canonical, representation for Boolean functions
- Used in Symbolic Model Checking to represent sets of states
Apply Operator on Binary Decision Diagrams

Towards Multi-Core BDD

Apply(⊗, leaf₁, leaf₂) = leaf₁ ⊗ leaf₂

Apply(⊗, B₁, B₂) =

let z = min(topvar(B₁), topvar(B₂))
L = Apply(⊗, B₁|z=0, B₂|z=0)
H = Apply(⊗, B₁|z=1, B₂|z=1)
R = MakeUniqueNode(z, L, H)
in R

- Two recursive calls
Apply Operator on Binary Decision Diagrams

Towards Multi-Core BDD

Apply\((\otimes, \text{leaf}_1, \text{leaf}_2) = \text{leaf}_1 \otimes \text{leaf}_2\)

Apply\((\otimes, B_1, B_2 ) = \)

\[\text{let } z = \min(\text{topvar}(B_1), \text{topvar}(B_2))\]

\[L = \text{Apply}(\otimes, B_1|z=0, B_2|z=0)\]

\[H = \text{Apply}(\otimes, B_1|z=1, B_2|z=1)\]

\[R = \text{MakeUniqueNode}(z, L, H)\]

in \(R\)

- Two recursive calls
- MakeUniqueNode uses \textit{concurrent} shared hashtable
Apply Operator on Binary Decision Diagrams

Towards Multi-Core BDD

Apply(⊗, leaf₁, leaf₂) = leaf₁ ⊗ leaf₂

Apply(⊗, B₁, B₂) = if (⊗, B₁, B₂) \rightarrow R in cache, return R

Apply(⊗, B₁, B₂) =

let z = min(topvar(B₁), topvar(B₂))

L = Apply(⊗, B₁|z=0, B₂|z=0)

H = Apply(⊗, B₁|z=1, B₂|z=1)

R = MakeUniqueNode(z, L, H)

store (⊗, B₁, B₂) \rightarrow R in cache

in R

- Two recursive calls
- MakeUniqueNode uses concurrent shared hashtable
- Caching uses concurrent lossy hashtable
Apply Operator on Binary Decision Diagrams

Towards Multi-Core BDD

[Tom van Dijk]

Apply( \( \otimes \), leaf\(_1\), leaf\(_2\) ) = leaf\(_1\) \( \otimes \) leaf\(_2\)

Apply( \( \otimes \), B\(_1\), B\(_2\) ) = if \( \otimes, B_1, B_2 \) \( \rightarrow \) R in cache, return R

Apply( \( \otimes \), B\(_1\), B\(_2\) ) =

let \( z = \min(\text{topvar}(B_1), \text{topvar}(B_2)) \)

\( L = \text{spawn} \) Apply( \( \otimes \), B\(_1\)\( |z=0\), B\(_2\)\( |z=0\) )

\( H = \text{spawn} \) Apply( \( \otimes \), B\(_1\)\( |z=1\), B\(_2\)\( |z=1\) )

\( R = \text{MakeUniqueNode}(z, \text{sync} \ L, \text{sync} \ H) \)

store \( \otimes, B_1, B_2 \) \( \rightarrow \) R in cache

in R

- Two recursive calls
- MakeUniqueNode uses concurrent shared hashtable
- Caching uses concurrent lossy hashtable
- Spawn/Sync requires a fine-grained task scheduler (deque)
# Sylvan Framework for Multi-core Decision Diagrams

## Features of Sylvan

- **Support**: BDD, Multiway/Multiterminal DDs, ZDDs, ...
- **Programmable interface**: (C, C++, Python)
- **Ported to RDMA**: Multicore/Distributed  

[Wytse Oortwijn, SPIN17]

Missing: dynamic variable reordering
## Features of Sylvan

[https://github.com/utwente-fmt/sylvan](https://github.com/utwente-fmt/sylvan)

- Support: BDD, Multiway/Multiterminal DDs, ZDDs, ...
- Programmable interface (C, C++, Python)
- Ported to RDMA: Multicore/Distributed  
  [Wytse Oortwijn, SPIN17]

Missing: dynamic variable reordering

## Applications

- Symbolic Reachability with BFS strategy and Saturation
- Symbolic Bisimulation Reduction / CTMC lumping
- Symbolic Parity Game Solving (Zielonka’s algorithm)
LTSmin: high-performance model checker

LTSmin and its language-independent interface PINS
https://github.com/utwente-fmt/ltsmin

Parallel LTL-X model checking with partial-order reduction
Symbolic reachability with saturation and bisimulation reduction
Distributed reachability and bisimulation reduction

Competition Awards: RERS 2012, 2013, 2016; MCC 2016 gold in LTL

Jaco van de Pol, Aarhus+Twente
LTSM in: high-performance model checker

LTSM in and its language-independent interface PINS
https://github.com/utwente-fmt/ltsm in

- Parallel LTL-X model checking with partial-order reduction
- Symbolic reachability with saturation and bisimulation reduction
- Distributed reachability and bisimulation reduction
LTSmin: high-performance model checker

LTSmin and its language-independent interface PINS
https://github.com/utwente-fmt/ltsmin

- Parallel LTL-X model checking with partial-order reduction
- Symbolic reachability with saturation and bisimulation reduction
- Distributed reachability and bisimulation reduction
- Competition Awards: RERS 2012, 2013, 2016; MCC 2016 gold in LTL
Conclusion

Concurrent Datastructures
- hash-tables, lossy cache, union-find, deque
- mostly lock-less, use CAS, NUMA-aware programming

Total amount of work: try to avoid duplicate work

Speedup bottlenecks: try to avoid sequential repair

Careful reconsider necessary invariants

Recent directions of interest
- GPU algorithms and implementations
- Parallel SAT/QBF solving
- Parallel parameter synthesis (probability, time)
- Parallel strategy synthesis for games

Jaco van de Pol, Aarhus+Twente
## Conclusion

### Concurrent Datastructures
- hash-tables, lossy cache, union-find, deque
- mostly lock-less, use CAS, NUMA-aware programming

### Parallel Algorithms, in particular parallel DFS-based
- Total amount of work: try to avoid duplicate work
- Speedup bottlenecks: try to avoid sequential repair
- Careful reconsider necessary invariants

---

Jaco van de Pol, Aarhus+Twente

Concurrency for Model Checking
Conclusion

Concurrent Datastructures
- hash-tables, lossy cache, union-find, deque
- mostly lock-less, use CAS, NUMA-aware programming

Parallel Algorithms, in particular parallel DFS-based
- Total amount of work: try to avoid duplicate work
- Speedup bottlenecks: try to avoid sequential repair
- Careful reconsider necessary invariants

Recent directions of interest
- GPU algorithms and implementations
- Parallel SAT/QBF solving
- Parallel parameter synthesis (probability, time)
- Parallel strategy synthesis for games
Literature: Overview and LTSmin tool

- J. Barnat, V. Bloemen, A. Duret-Lutz, Laarman, Petrucci, vd Pol, Renault
  Parallel Model Checking Algorithms for Linear-Time Temporal Logic
  In: Handbook of Parallel Constraint Reasoning 2018: 457-507

- T. van Dijk, J. van de Pol
  Multi-core Decision Diagrams
  In: Handbook of Parallel Constraint Reasoning, 2018: pp. 509-545

- S. Blom, J. van de Pol, M. Weber
  LTSmin: Distributed and Symbolic Reachability [CAV’10]

- A. Laarman, J. van de Pol, M. Weber
  Multi-Core LTSmin: Marrying Modularity and Scalability [NFM’11]

- G. Kant, A. Laarman, J. Meijer, J. vd Pol, S. Blom, T. v Dijk
  LTSmin: High-Performance Language-Independent Model Checking [TACAS’15]
Literature on parallel DFS-based SCC detection

- R.E. Tarjan [SIAM 1972]
  Depth-First Search and Linear Graph Algorithms
  A Discipline of Programming
- G. Lowe [TACAS’14]
  Concurrent Depth-First Search Algorithms
  Parallel Explicit Model Checking for Generalized Büchi Automata
- V. Bloemen, A. Laarman, J. van de Pol [PPoPP’16]
  Multi-Core On-The-Fly SCC Decomposition
- S.V. Jayanti, R.E. Tarjan, E. Boix-Adserà [PODC’19]
  Randomized Concurrent Set Union and Generalized Wake-Up.
Literature on parallel LTL model checking

- Jiri Barnat, Lubos Brim, Jakub Chaloupka [ASE’03]
  Parallel Breadth-First Search LTL Model-Checking
- Jiri Barnat, Lubos Brim, Petr Rockai [SPIN’07]
  Scalable Multi-core LTL Model-Checking
  Multi-core Nested Depth-First Search
- S. Evangelista, L. Petrucci [ATVA’11]
  Parallel Nested Depth-First Searches for LTL Model Checking
- A. Laarman, S. Evangelista, L. Petrucci, J. van de Pol [ATVA’12]
  Improved Multi-Core Nested Depth-First Search
- A. Laarman, M. Olesen, A. Dalsgaard, K. Larsen, J. vd Pol [CAV’13]
  Multi-core Emptiness Checking of Timed Büchi Automata
- V. Bloomen, A. Duret-Lutz, J. van de Pol [SPIN’17]
  Explicit state model checking with Büchi and Rabin automata
Literature on parallel BDDs and symbolic model checking

- S. Kimura, E.M. Clarke [ICCD’90]
  A parallel algorithm for constructing Binary Decision Diagrams
- O. Grumberg, T. Heyman, A. Schuster [CAV’01]
  Distributed Symbolic Model Checking for $\mu$-Calculus
- T. van Dijk, J. van de Pol [TACAS’15]
  Sylvan: Multi-Core Decision Diagrams
- Tom van Dijk, Jaco van de Pol [TACAS’16]
  Multi-core Symbolic Bisimulation Minimisation
- W. Oortwijn, T. van Dijk, J. van de Pol [SPIN’17]
  Distributed binary decision diagrams for symbolic reachability
- Tom van Dijk, Jeroen Meijer, Jaco van de Pol [TACAS’19]
  Multi-core On-The-Fly Saturation
Parallel SCC with UF and Cyclic List