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

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 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, . . .

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)

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)

Required: parallelisation of smart algorithms!

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)

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)

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

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

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)

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)

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

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 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

- 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

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

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)

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)

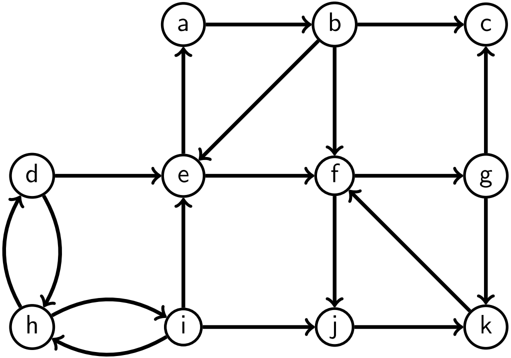
Vincent Bloemen: Parallel Strongly Connected Components (2016-2019)

- 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

Strongly Connected Component (SCC)

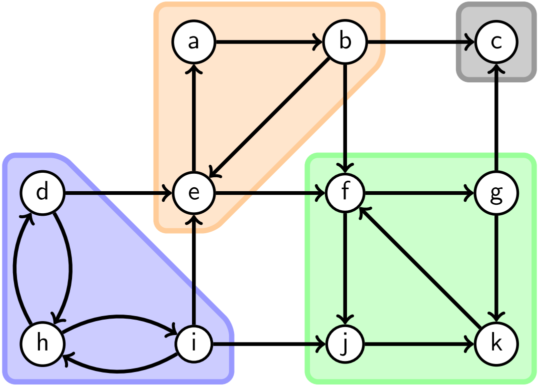
Setting: finite graph with directed edges
SCCs: maximal components of $\twoheadrightarrow \cap \twoheadleftarrow$



Strongly Connected Component (SCC)

Setting: finite graph with directed edges

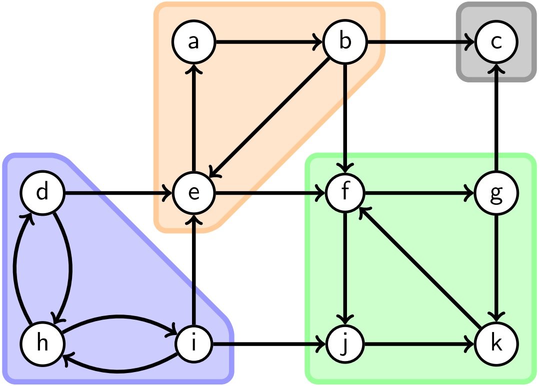
SCCs: maximal components of $\twoheadrightarrow \cap \twoheadleftarrow$



Strongly Connected Component (SCC)

Setting: finite graph with directed edges

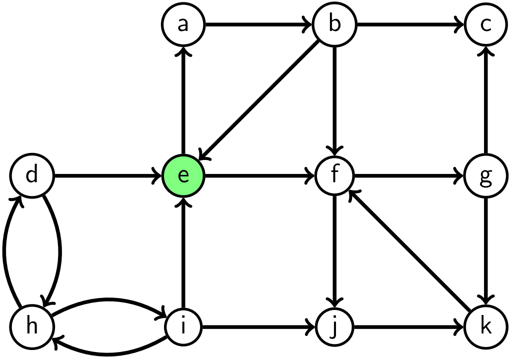
SCCs: maximal components of $\twoheadrightarrow \cap \twoheadleftarrow$



Applications: LTL model checking, fairness, evaluation of Markov Chains

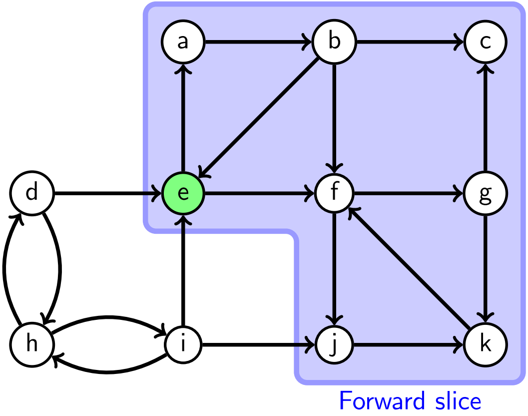
Forward-Backward (FB) parallel SCC Algorithm

1. Select a pivot node



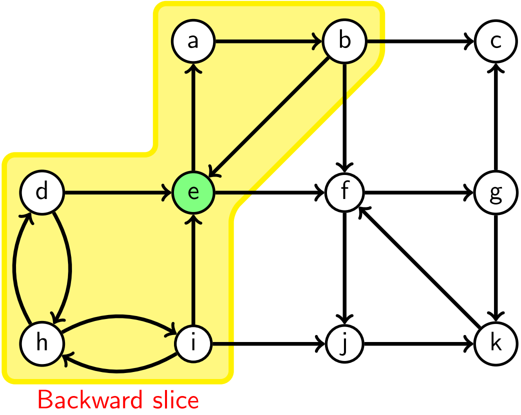
Forward-Backward (FB) parallel SCC Algorithm

2. Compute its forward reachable set (F)

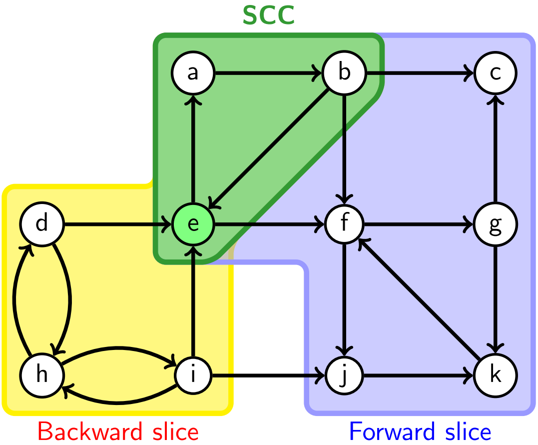


Forward-Backward (FB) parallel SCC Algorithm

3. Compute its backward reachable set (B)

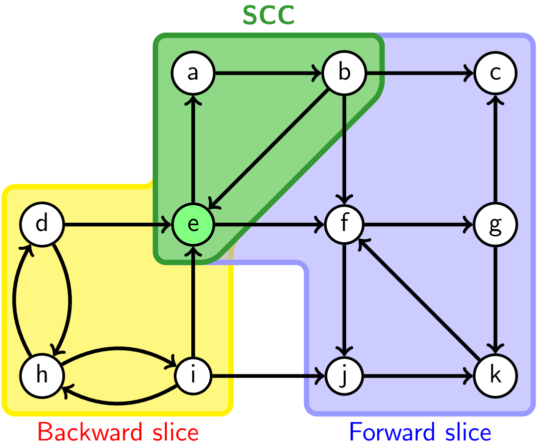


4. The intersection F ∩ B is the SCC of the pivot



Forward-Backward (FB) parallel SCC Algorithm

4. The intersection F ∩ B is the SCC of the pivot

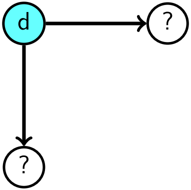


Remaining slices can be processed independently in parallel

Finding SCCs on-the-fly (path-based algorithm)

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

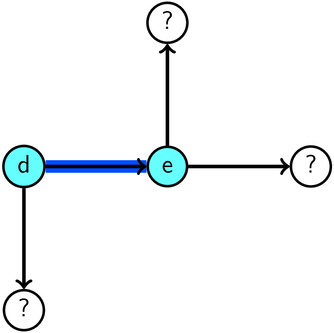
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function



Finding SCCs on-the-fly (path-based algorithm)

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

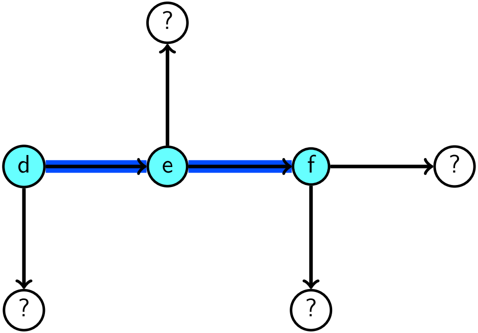
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function



Finding SCCs on-the-fly (path-based algorithm)

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

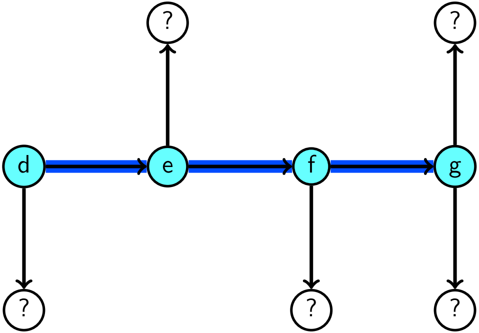
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function



Finding SCCs on-the-fly (path-based algorithm)

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

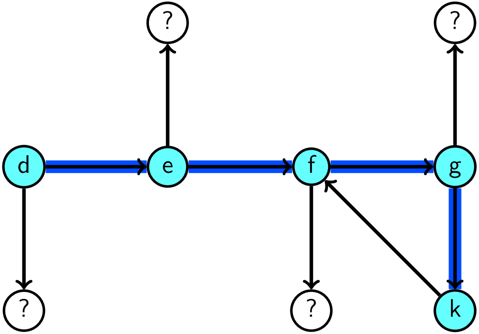
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function



Finding SCCs on-the-fly (path-based algorithm)

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

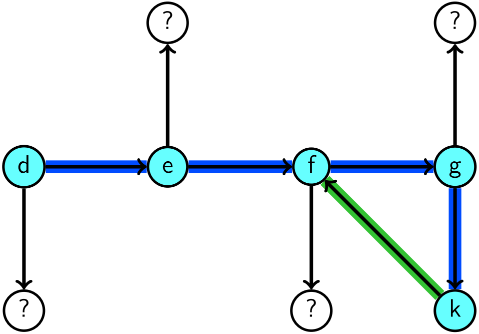
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function



Finding SCCs on-the-fly (path-based algorithm)

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

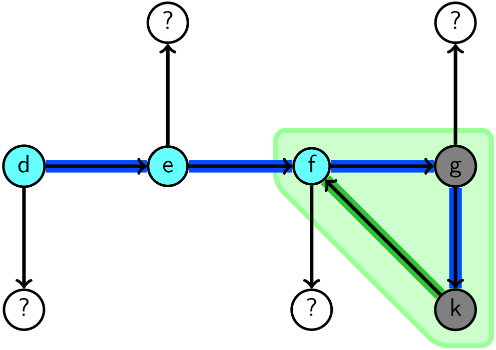
- bug finding: early termination when a bug in the model is detected
- portability: we restrict model access to a next-state function



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

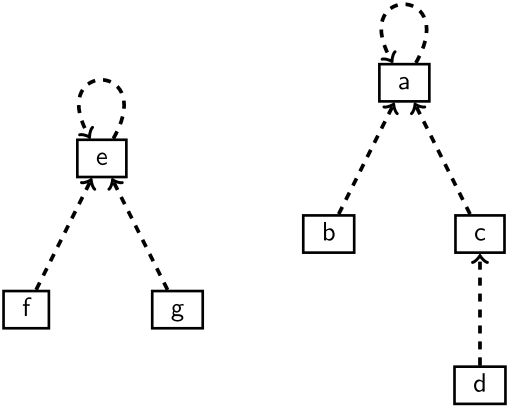


NB: this is not yet necessarily a maximal SCC,
since its successors are not completely explored

Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure *[Tarjan, van Leeuwen, JACM 1984]*:

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

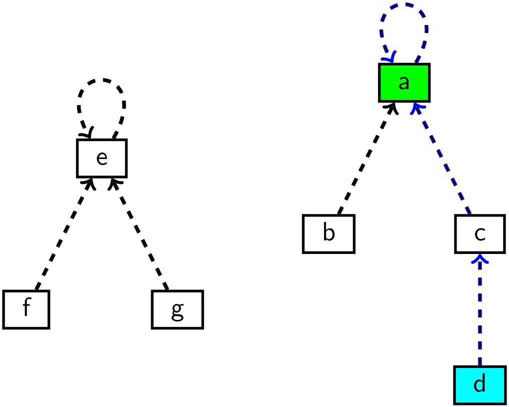


Reversed forest, nodes direct towards their representative root

Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure *[Tarjan, van Leeuwen, JACM 1984]*:

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

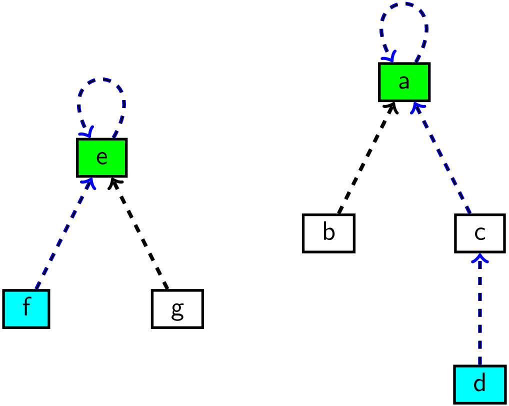


`Find(d)`: recursively searches the parent edges to find the root

Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure *[Tarjan, van Leeuwen, JACM 1984]*:

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

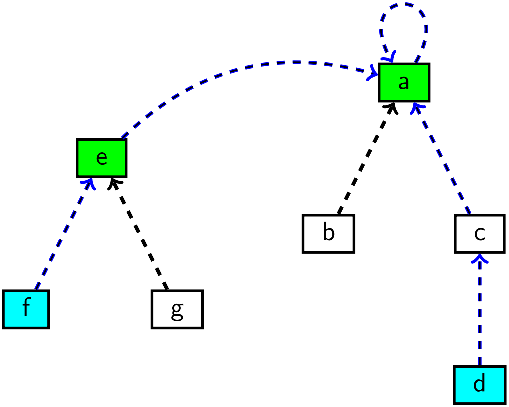


`Unite(f,d)`: Find the roots of f and d,

Maintain (partial) SCCs in a Union-Find data structure

Union-Find structure *[Tarjan, van Leeuwen, JACM 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 $\varnothing$

$\dots$
$\dots$
w
$\dots$
$\dots$
v

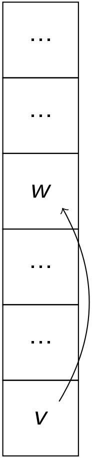
R

```
procedure SCC(v)
  Visited := Visited ∪ {v}
  R.push(v)
  for each w ∈ next_state(v)

```

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 $\emptyset$

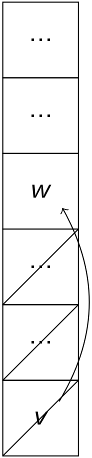


R

```
procedure SCC(v)
  Visited := Visited ∪ {v}
  R.push(v)
  for each w ∈ next_state(v)
    if w ∈ Explored                 // complete SCC
      then continue
    else if w ∉ Visited             // unseen state
      then SCC(w)
    else                            // cycle found

```

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

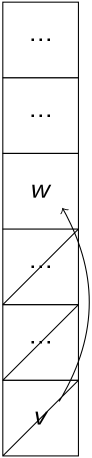


R

```
procedure SCC(v)
  Visited := Visited ∪ {v}
  R.push(v)
  for each w ∈ next_state(v)
    if w ∈ Explored              // complete SCC
      then continue
    else if w ∉ Visited          // unseen state
      then SCC(w)
    else                         // cycle found
      while S.find(v) ≠ S.find(w) do
        S.union(R.pop(), R.top())
```

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 $\varnothing$



R

```
procedure SCC(v)
· Visited := Visited ∪ {v}
· R.push(v)
· for each w ∈ next_state(v)
· · if w ∈ Explored · · · · · · · · // complete SCC
· · · then continue
· · else if w ∉ Visited · · · · // unseen state
· · · then SCC(w)
· · else · · · · · · · · · · · · // cycle found
· · · while S.find(v) ≠ S.find(w) do
· · · · S.union(R.pop(), R.top())
· if v = R.top() then · · · · ·  // completed SCC
· · report SCC S.enum(v)
· · Explored := Explored ∪ S.enum(v)
· · R.pop()
```

Comparing SCC algorithms

Tarjan's SCC algorithm [1972]

- Worst case $O(m + n)$
- On-the-fly
- Inherently DFS

Forward-Backward algorithm

- Worst case $O(n(m + n))$
- Requires predecessors
- BFS is sufficient

Comparing SCC algorithms

Tarjan's SCC algorithm [1972]

- Worst case $O(m+n)$
- On-the-fly
- Inherently DFS

Variants (DFS-based):

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

Forward-Backward algorithm

- Worst case $O(n(m+n))$
- Requires predecessors
- BFS is sufficient

Variants (BFS-based):

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

Comparing SCC algorithms

Tarjan's SCC algorithm [1972]

- Worst case $O(m+n)$
- On-the-fly
- Inherently DFS

Variants (DFS-based):

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

Forward-Backward algorithm

- Worst case $O(n(m+n))$
- Requires predecessors
- BFS is sufficient

Variants (BFS-based):

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

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

Comparing SCC algorithms

Tarjan's SCC algorithm [1972]

- Worst case $O(m+n)$
- On-the-fly
- Inherently DFS

Forward-Backward algorithm

- Worst case $O(n(m+n))$
- Requires predecessors
- BFS is sufficient

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]

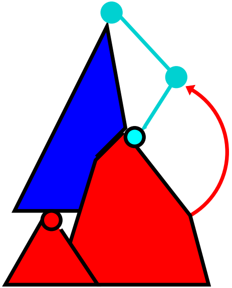
FB is easier to parallellize, but harder to use and less efficient!

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

Intermezzo: Parallel Random Nested-DFS (ATVA 2011, 2012)

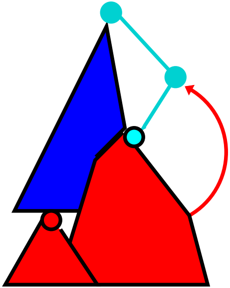
Inspiration: Swarmed Verification (Holzmann, Spin) for bug finding

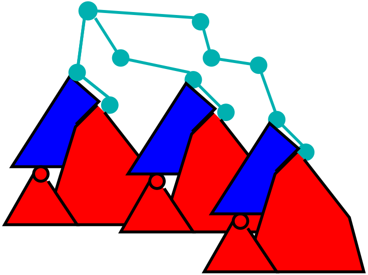


Nested Depth-First Search for LTL model checking

Intermezzo: Parallel Random Nested-DFS (ATVA 2011, 2012)

Inspiration: Swarmed Verification (Holzmann, Spin) for bug finding



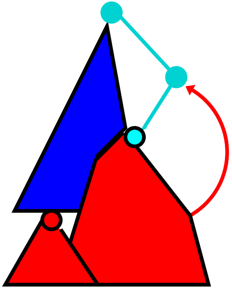


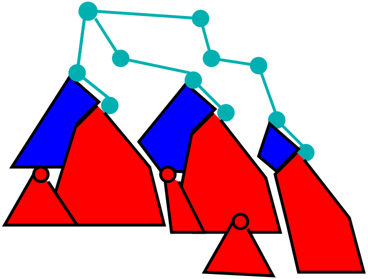
Nested Depth-First Search for LTL model checking

- Every worker performs its own NDFS in a randomized direction
- Parallel search speeds up finding bugs only: duplicate work

Intermezzo: Parallel Random Nested-DFS (ATVA 2011, 2012)

Inspiration: Swarmed Verification (Holzmann, Spin) for bug finding



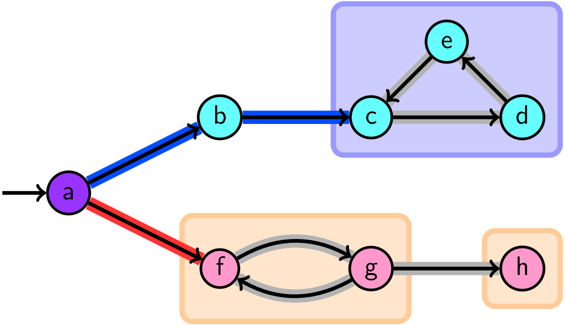


Nested Depth-First Search for LTL model checking

- 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

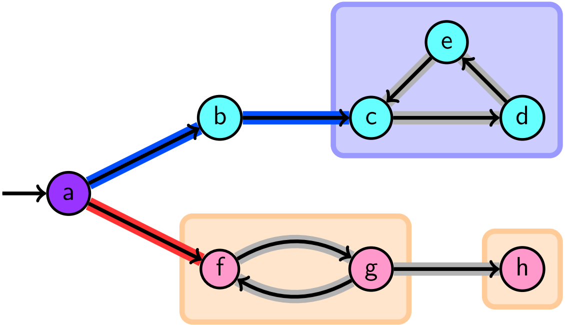
Parallel DFS + random successor order + sharing information on SCCs



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

Parallel Random DFS for SCCs

Parallel DFS + random successor order + sharing information on SCCs

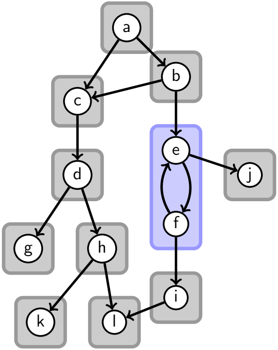


What happens if two workers start working on the same SCC?

- 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

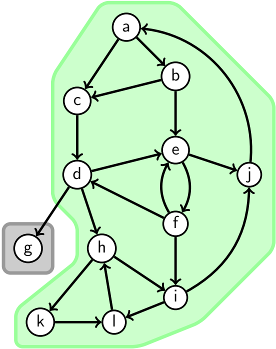
Handling Small and Large SCCs Sequentially

Small SCCs



Parallelizes well

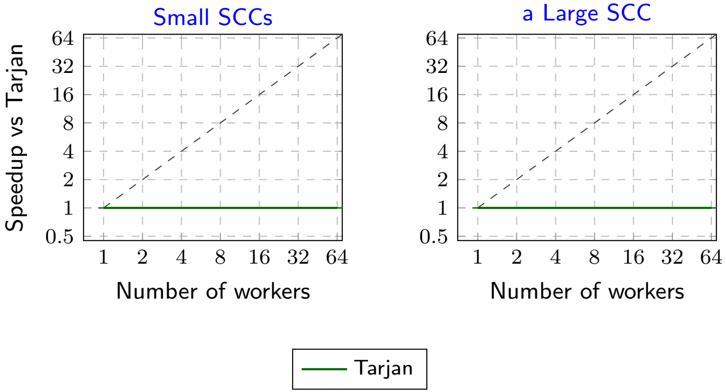
Large SCCs



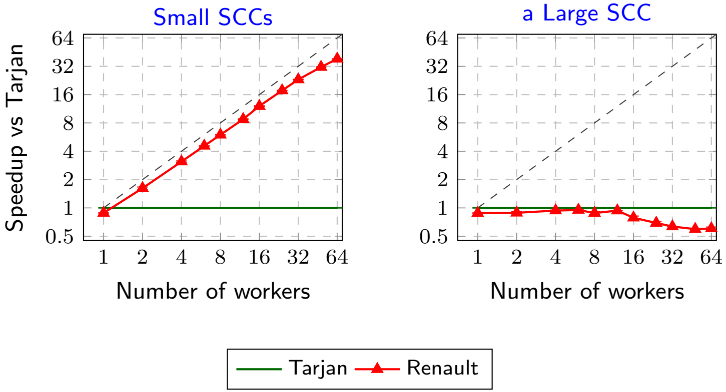
No performance gain

Bottom line: we cannot afford to handle single SCCs sequentially

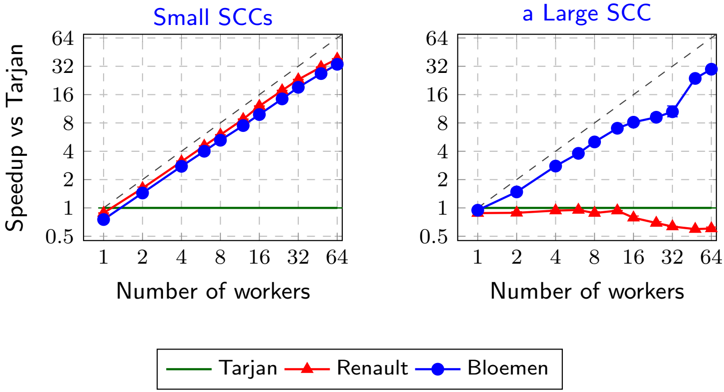
Speedup in practice



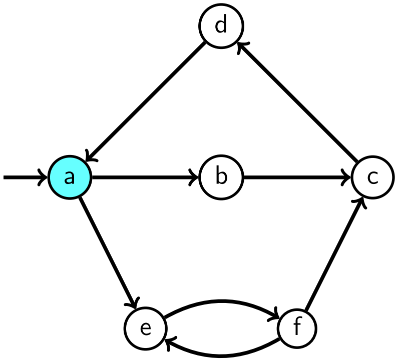
Speedup in practice



Speedup in practice

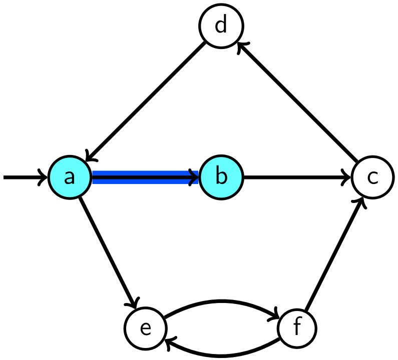


UF-SCC: Communicate partially found SCCs [Bloemen]

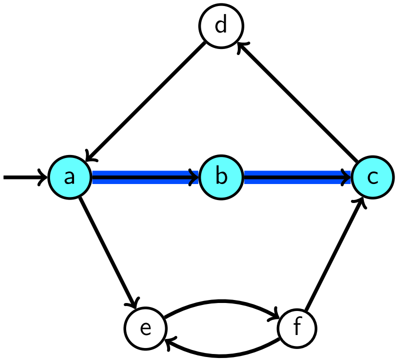


Blue worker happens to visit $a \to b \to c \to d$

UF-SCC: Communicate partially found SCCs [Bloemen]

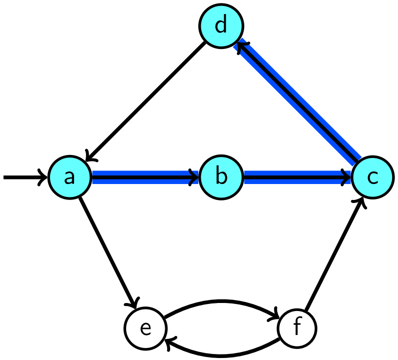


Blue worker happens to visit $a \rightarrow b \rightarrow c \rightarrow d$



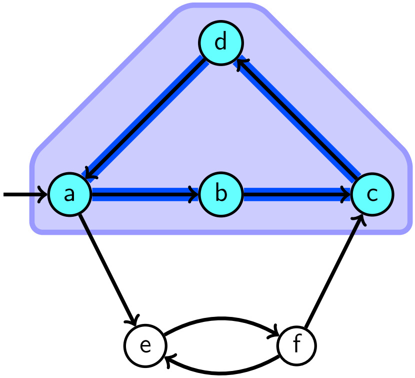
Blue worker happens to visit $a \to b \to c \to d$

UF-SCC: Communicate partially found SCCs [Bloemen]



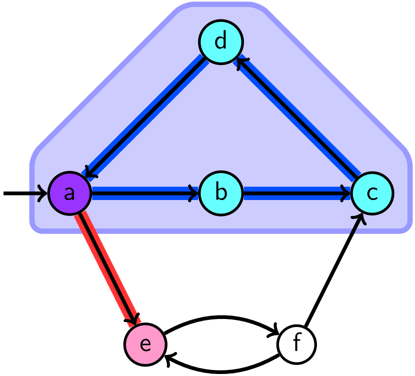
Blue worker happens to visit $a \to b \to c \to d$

UF-SCC: Communicate partially found SCCs [Bloemen]



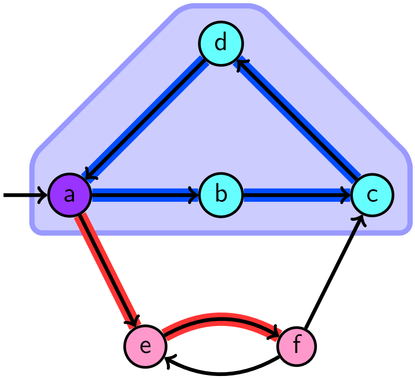
Blue worker detects and unites partial SCC $\{a, b, c, d\}$

UF-SCC: Communicate partially found SCCs [Bloemen]



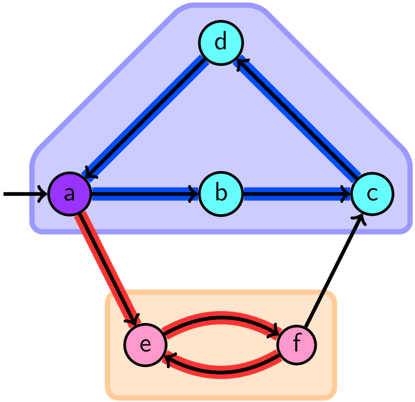
Red worker happens to visit $a \rightarrow e \rightarrow f$

UF-SCC: Communicate partially found SCCs [Bloemen]



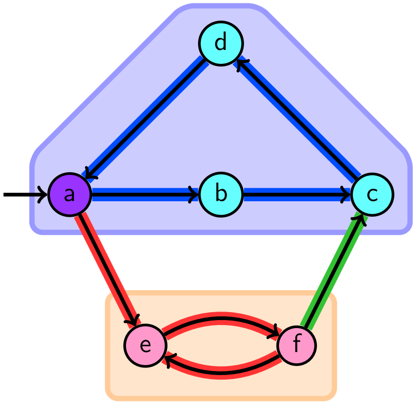
Red worker happens to visit $a \to e \to f$

UF-SCC: Communicate partially found SCCs [Bloemen]



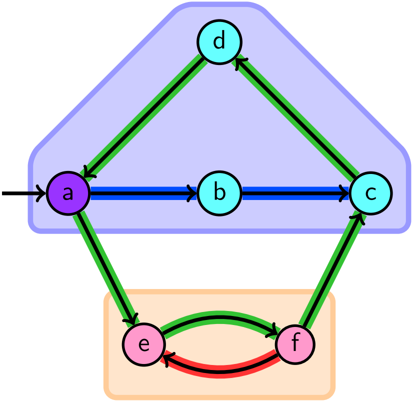
Red worker detects and unites partial SCC $\{e, f\}$

UF-SCC: Communicate partially found SCCs [Bloemen]



Red worker continues exploration $f \to c$

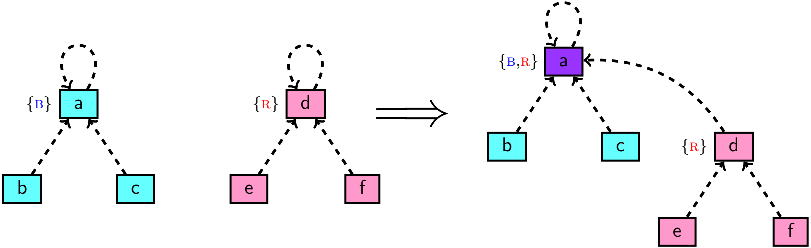
UF-SCC: Communicate partially found SCCs [Bloemen]



But how does Red worker know that it visited a state “equivalent” to c?

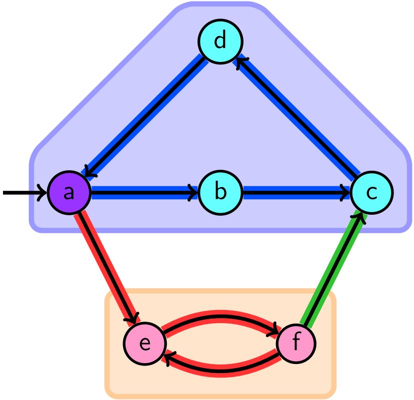
Union-Find with a worker set

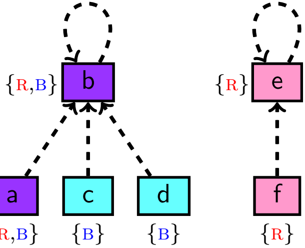
Store a bit-set of worker IDs in the union-find roots



Union-Find with a worker set

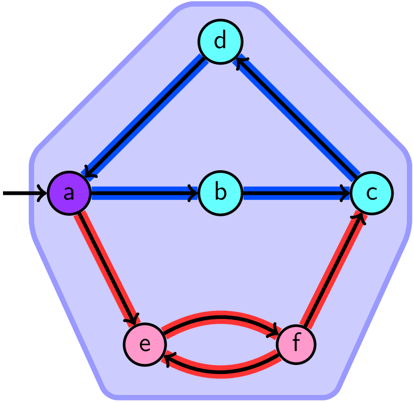
Check if the partial SCC of the successor has been visited before

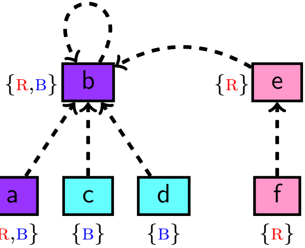




Union-Find with a worker set

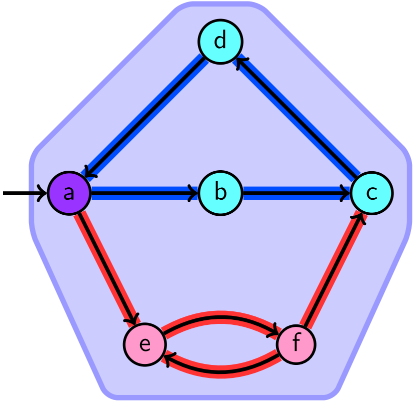
Check if the partial SCC of the successor has been visited before

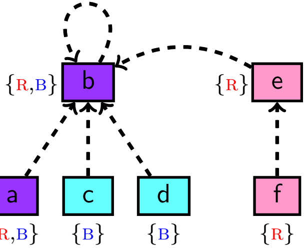




Union-Find with a worker set

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

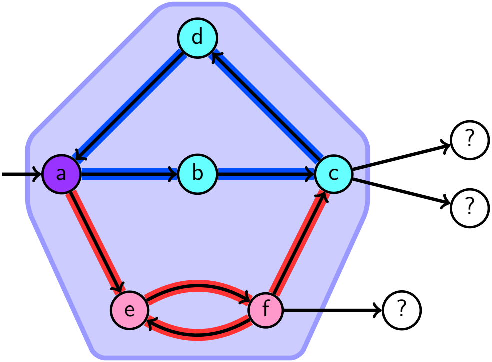




But how do we know when the SCC is complete?

Distinguish fully explored states

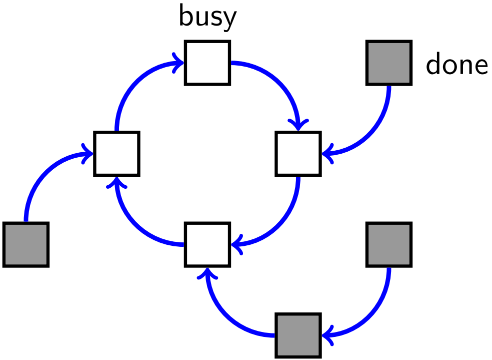
- Track which states of the SCC still have to be explored
  - An SCC is complete if all its states have been fully explored
- Evenly distribute the remaining work
  - Otherwise one worker may end up doing all the work

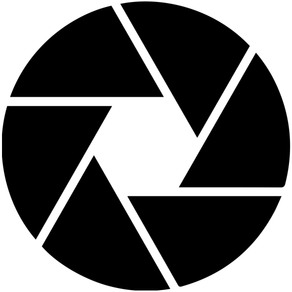


Cyclic list of BUSY states

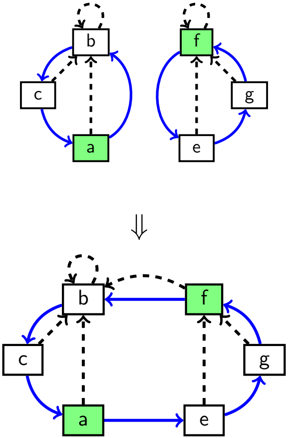
Keep track of the TODO list of BUSY states

- 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



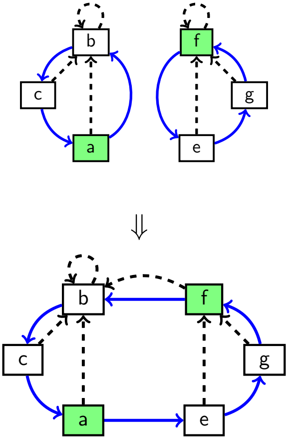


List operations

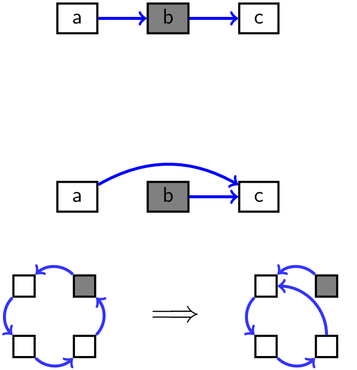
Merge lists (Union)	Remove element


List operations

Merge lists (Union)



Remove element



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 ∈ randomize(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 ∈ randomize(next_state(v'))
· · · if w ∈ DEAD                         // ignore completed SCC
· · · · then continue
· · · else if p ∉ S.find(w)              // state yet unseen by p
· · · · then UFSCC_p(w)
· · · else
· · · · while ¬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 $O((m+n).\alpha(n).p)$: linear-time, but no speed-up

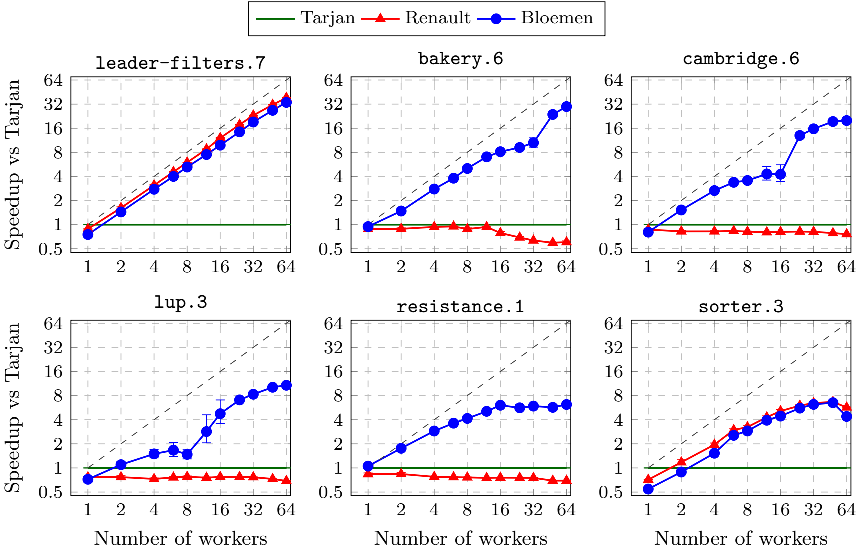
Model checking graphs are "broad", so workers spread out evenly.
Observed wall clock: $O((m+n).\alpha(n)/p)$: linear-time and linear speed-up

Can we guarantee even more? Maybe!

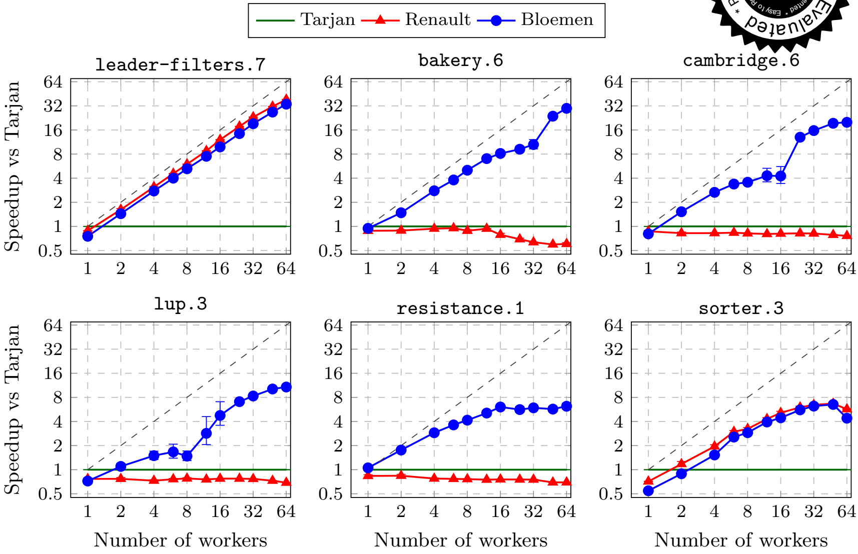
- S.V. Jayanti, R.E. Tarjan, E. Boix-Adserà [PODC'19]
  Randomized Concurrent Set Union and Generalized Wake-Up

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



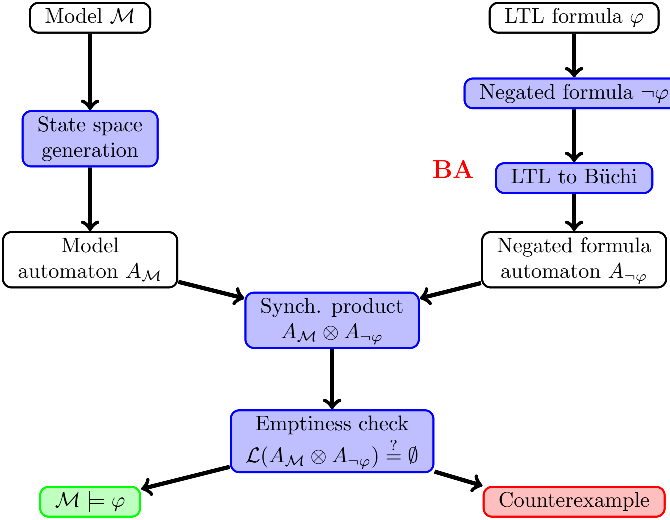
Speedup graphs of selected BEEM models



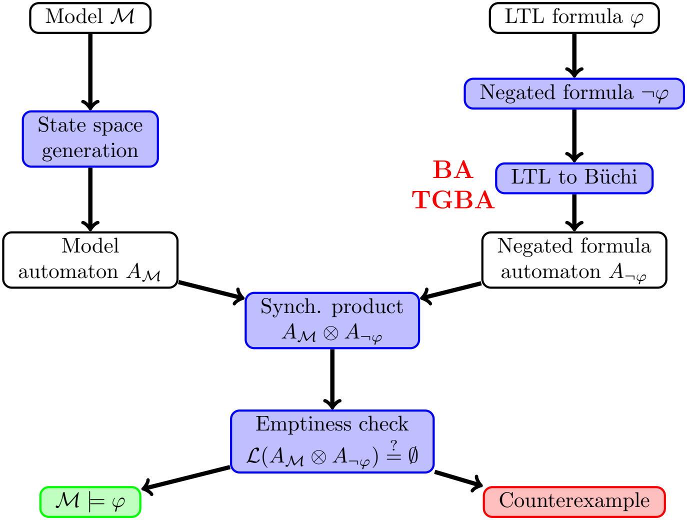
Overview

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

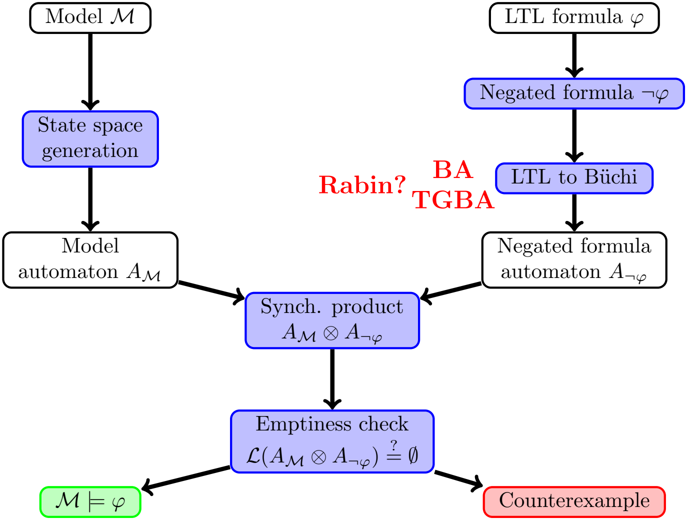
Recall: Automata-based LTL model checking



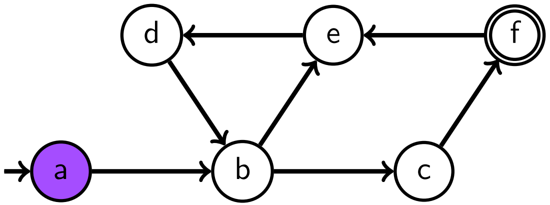
Recall: Automata-based LTL model checking



Recall: Automata-based LTL model checking



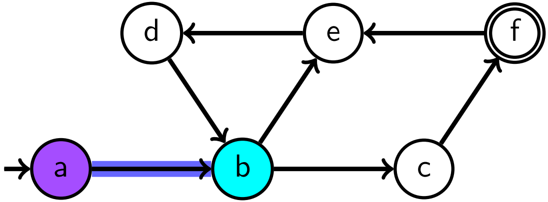
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Blue worker explores $a \rightarrow b \rightarrow e \rightarrow d$

UFSCC for LTL Model Checking with Büchi Automata

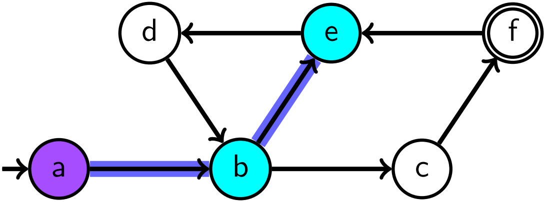
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Blue worker explores $a \to b \to e \to d$

UFSCC for LTL Model Checking with Büchi Automata

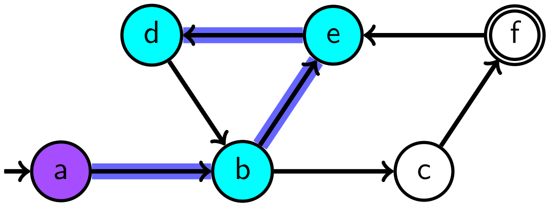
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Blue worker explores $a \to b \to e \to d$

UFSCC for LTL Model Checking with Büchi Automata

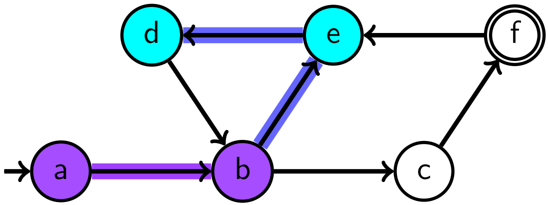
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Blue worker explores $a \to b \to e \to d$

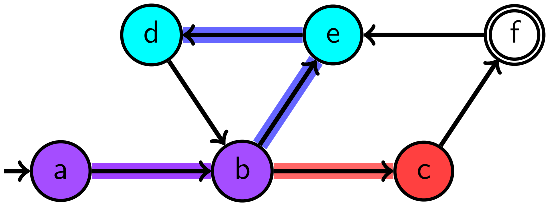
UFSCC for LTL Model Checking with Büchi Automata

LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



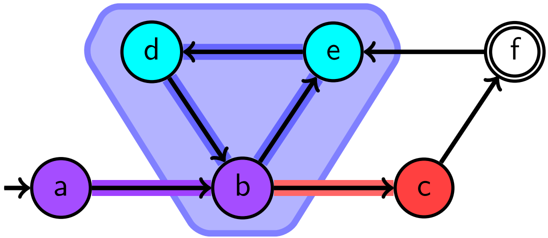
Red worker explores $a \to b \to c$

LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Red worker explores $a \to b \to c$

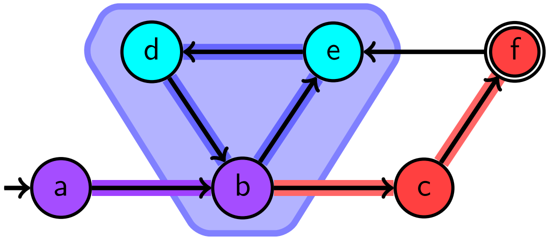
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Blue worker detects and shares partial SCC $\{b, d, e\}$

UFSCC for LTL Model Checking with Büchi Automata

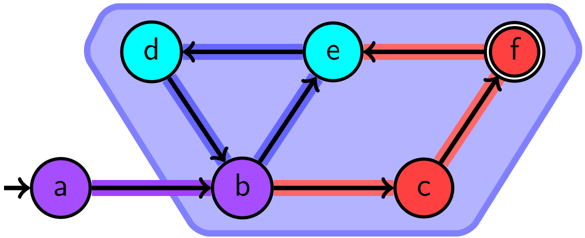
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Red worker detects complete, accepting SCC $\{b, c, d, e, f\}$

UFSCC for LTL Model Checking with Büchi Automata

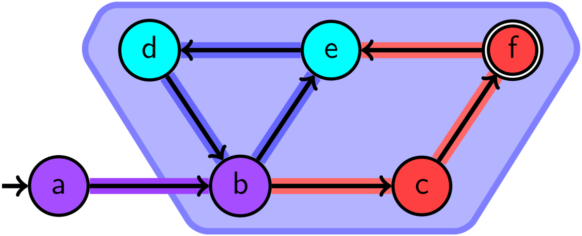
LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton



Red worker detects complete, accepting SCC $\{b, c, d, e, f\}$

UFSCC for LTL Model Checking with Büchi Automata

LTL model checking reduces to the following graph problem:
Find a reachable accepting SCC in a Büchi-automaton

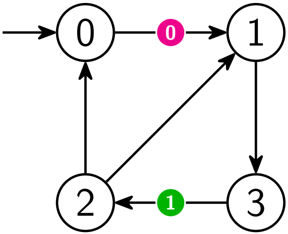


Accepting cycle has been found, while no single worker traversed it!

Accepting Cycle for TGBA

Transition-based Generalized Büchi Automata

$\mathrm{Inf}(⓿) \wedge \mathrm{Inf}(❶)$

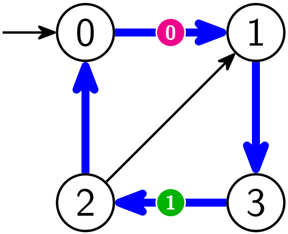


Advantage: TGBA are more concise and natural for LTL

Accepting Cycle for TGBA

Transition-based Generalized Büchi Automata

$$\mathrm{Inf}(⓿) \wedge \mathrm{Inf}(❶)$$

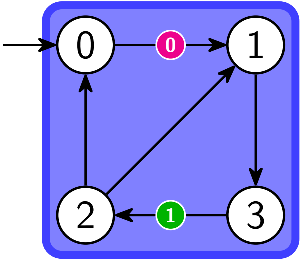


Advantage: TGBA are more concise and natural for LTL

Accepting Cycle for TGBA

Transition-based Generalized Büchi Automata

$\mathsf{Inf}(0) \wedge \mathsf{Inf}(1)$

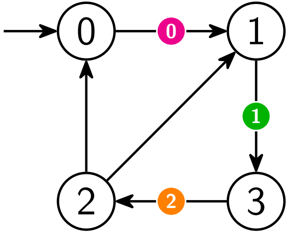


Advantage: TGBA are more concise and natural for LTL

Store all encountered accepting marks at the UF-root

Accepting Cycle for Generalized Rabin Automata

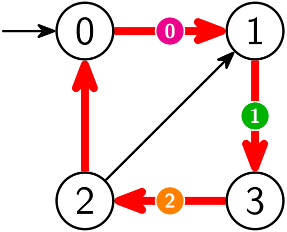
$$\mathsf{Fin}(⓿) \wedge \mathsf{Inf}(➊) \wedge \mathsf{Inf}(➋)$$



[Bloemen, Duret-Lutz, vdPol, SPIN 2017]
Can handle all Rabin conditions sequentially or in parallel

Accepting Cycle for Generalized Rabin Automata

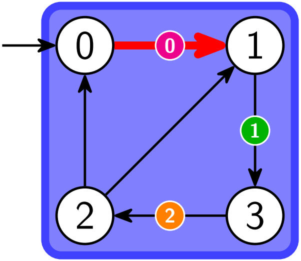
$$\mathrm{Fin}(⓿) \wedge \mathrm{Inf}(❶) \wedge \mathrm{Inf}(❷)$$



[Bloemen, Duret-Lutz, vdPol, SPIN 2017]
Can handle all Rabin conditions sequentially or in parallel

Accepting Cycle for Generalized Rabin Automata

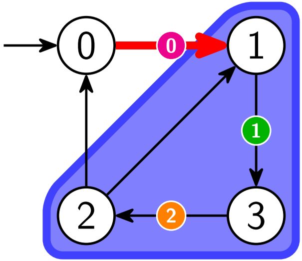
$$\mathsf{Fin}(0) \land \mathsf{Inf}(1) \land \mathsf{Inf}(2)$$



[Bloemen, Duret-Lutz, vdPol, SPIN 2017]
Can handle all Rabin conditions sequentially or in parallel

Accepting Cycle for Generalized Rabin Automata

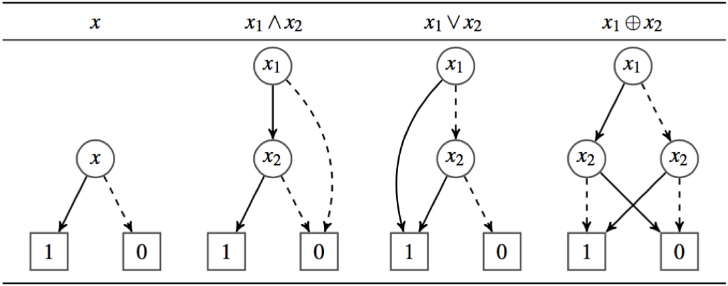
$\mathrm{Fin}(⓿) \wedge \mathrm{Inf}(❶) \wedge \mathrm{Inf}(❷)$



[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 [Tom van Dijk]

Apply($\otimes, leaf_1, leaf_2$) $= leaf_1 \otimes leaf_2$

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

let $z = min(topvar(B_1), topvar(B_2))$

$L =$ Apply($\otimes, B_1|_{z=0}, B_2|_{z=0}$)

$H =$ Apply($\otimes, B_1|_{z=1}, B_2|_{z=1}$)

$R =$ MakeUniqueNode(z, L, H)

in R

- Two recursive calls

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($\otimes, B_1, B_2$) $=$
  let $z = min(topvar(B_1), topvar(B_2))$
    $L =$ Apply($\otimes, B_1|_{z=0}, B_2|_{z=0}$)
    $H =$ Apply($\otimes, B_1|_{z=1}, B_2|_{z=1}$)
    $R =$ MakeUniqueNode(z, L, H)

  in R

- Two recursive calls
- MakeUniqueNode uses concurrent shared 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 $z = min(topvar(B_1), topvar(B_2))$

$L =$ Apply($\otimes, B_1|_{z=0}, B_2|_{z=0})$

$H =$ Apply($\otimes, B_1|_{z=1}, B_2|_{z=1})$

$R =$ MakeUniqueNode(z, L, 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

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 $(\otimes, B_1, B_2) \to R$ in cache, return R
Apply($\otimes, B_1, B_2$) =
    let $z = min(topvar(B_1), topvar(B_2))$
        L = spawn Apply($\otimes, B_1|_{z=0}, B_2|_{z=0}$)
        H = spawn Apply($\otimes, B_1|_{z=1}, B_2|_{z=1}$)
        R = MakeUniqueNode(z,sync L, sync H)
        store $(\otimes, B_1, B_2) \to R$ 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 [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

Sylvan Framework for Multi-core Decision Diagrams

Features of 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

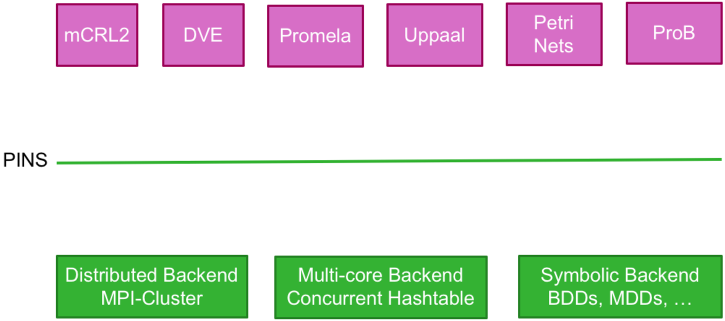
Applications

- 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

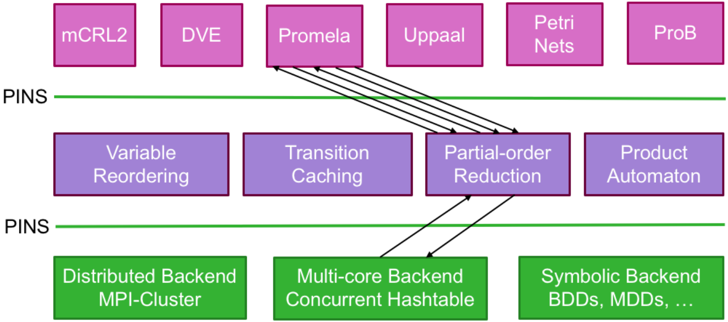
https://github.com/utwente-fmt/ltsmin



LTSmin: high-performance model checker

LTSmin and its language-independent interface PINS

`https://github.com/utwente-fmt/ltsmin`

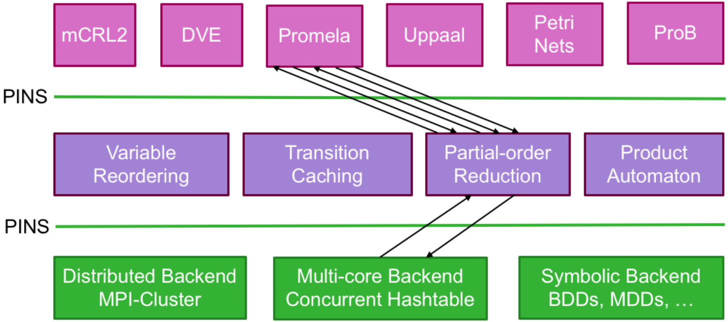


- Parallel LTL-X model checking with partial-order reduction
- Symbolic reachability with saturation and bisimulation reduction
- Distributed reachability and bisimulation reduction

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

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

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 [CAV'10]
  LTSmin: Distributed and Symbolic Reachability
- A. Laarman, J. van de Pol, M. Weber [NFM'11]
  Multi-Core LTSmin: Marrying Modularity and Scalability
- G. Kant, A. Laarman, J. Meijer, J. vd Pol, S. Blom, T. v Dijk [TACAS'15]
  LTSmin: High-Performance Language-Independent Model Checking

Literature on parallel DFS-based SCC detection

- R.E. Tarjan [SIAM 1972]
  Depth-First Search and Linear Graph Algorithms
- E.W. Dijkstra [Prentice Hall 1976]
  A Discipline of Programming
- G. Lowe [TACAS'14]
  Concurrent Depth-First Search Algorithms
- E. Renault, A. Duret-Lutz, F. Kordon, D. Poitrenaud [TACAS'15]
  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
- A. Laarman, L. Langerak, J. vd Pol, M. Weber, A. Wijs [ATVA'11]
  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. Bloemen, 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 μ-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