Parallelizing Dynamic Partial Order Reduction Algorithms in Concuerror

Παναγιώτης Φυτάς

ΣΗΜΜΥ - ΕΜΠ

03112113

Summary

Section 1

Background

Concurrent Computing

Concurrency is difficult to get right:

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks
- Race conditions

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
  - Interleaving non-determinism

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
  - Interleaving non-determinism
  - Timing non-determinism

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
  - Interleaving non-determinism
  - Timing non-determinism

Concurrent Computing

Concurrency is difficult to get right:

- Deadlocks
- Race conditions
- Resource starvation
- Scheduling non-determinism
    - Interleaving non-determinism
    - Timing non-determinism

Errors can occur only on specific rare interleavings. Detecting and reproducing bugs becomes extremely hard.

Modeling our Problem

- An interleaving (or a trace) represents a scheduling of the concurrent program.

Modeling our Problem

- An interleaving (or a trace) represents a scheduling of the concurrent program.
- The state space is the set of all possible interleavings.

Modeling our Problem

- An interleaving (or a trace) represents a scheduling of the concurrent program.
- The state space is the set of all possible interleavings.
- In order to verify a program, the complete state space must be explored.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.
- Combinatorial state space explosion.

Stateless Model Checking

- Stateless Model Checking systematically explores all possible interleavings.
- Combinatorial state space explosion.
- Different interleavings can be equivalent (Mazurkiewicz trace).

Stateless Model Checking

```
p:                q:
read(x)           read(y)
write(x)          read(x)
```

Stateless Model Checking

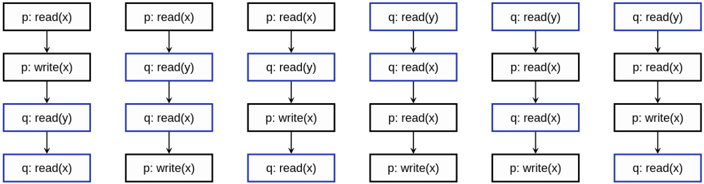


Figure: State space of a program.

Partial Order Reduction

Partial Order Reduction tries to avoid exploring equivalent interleavings through race detection.

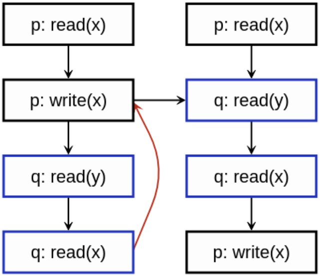


Figure: Interleavings explored by POR

Partial Order Reduction

- Static Partial Order Reduction: Dependencies are tracked before execution, by statically analyzing the code.

Partial Order Reduction

- Static Partial Order Reduction: Dependencies are tracked before execution, by statically analyzing the code.
- Dynamic Partial Order Reduction (DPOR): Actual dependencies are observed during runtime.

Terminology

- The complete execution of a process p splits into different execution steps, which are to be executed atomically. Those steps are referred to as $events$. Each event must be deterministic.

Terminology

- The complete execution of a process p splits into different execution steps, which are to be executed atomically. Those steps are referred to as $events$. Each event must be deterministic.
- An execution sequence E of a system is a finite sequence of execution steps of its processes that is performed from a unique initial state.

Terminology

- The complete execution of a process p splits into different execution steps, which are to be executed atomically. Those steps are referred to as $events$. Each event must be deterministic.
- An execution sequence E of a system is a finite sequence of execution steps of its processes that is performed from a unique initial state.
- An execution sequence E is uniquely characterized by the sequence of processes that perform steps in E. For instance, $p.p.q$ denotes the execution sequence where first p performs two steps, followed by a step of q.

Terminology

- The complete execution of a process p splits into different execution steps, which are to be executed atomically. Those steps are referred to as $events$. Each event must be deterministic.
- An execution sequence E of a system is a finite sequence of execution steps of its processes that is performed from a unique initial state.
- An execution sequence E is uniquely characterized by the sequence of processes that perform steps in E. For instance, $p.p.q$ denotes the execution sequence where first p performs two steps, followed by a step of q.
- The sequence of processes that perform steps in E also uniquely determines the global state of the system after E.

Erlang

Erlang is a programming language that has build-in support for:

- Concurrency

Erlang

Erlang is a programming language that has build-in support for:

- Concurrency
- Distribution

Concurrent Erlang

In the core of concurrent Erlang are its lightweight processes which:

- are implemented by the Erlang VM's (BEAM) runtime system, not by operating system threads.

Concurrent Erlang

In the core of concurrent Erlang are its lightweight processes which:

- are implemented by the Erlang VM's (BEAM) runtime system, not by operating system threads.
- are uniquely identified by their Pid (Process Identifier).

Concurrent Erlang

In the core of concurrent Erlang are its lightweight processes which:

- are implemented by the Erlang VM's (BEAM) runtime system, not by operating system threads.
- are uniquely identified by their Pid (Process Identifier).
- communicate with each other mainly though message passing (actor model).

Concurrent Erlang

In the core of concurrent Erlang are its lightweight processes which:

- are implemented by the Erlang VM's (BEAM) runtime system, not by operating system threads.
- are uniquely identified by their Pid (Process Identifier).
- communicate with each other mainly though message passing (actor model).

Concurrent Erlang

In the core of concurrent Erlang are its lightweight processes which:

- are implemented by the Erlang VM's (BEAM) runtime system, not by operating system threads.
- are uniquely identified by their Pid (Process Identifier).
- communicate with each other mainly though message passing (actor model).

Erlang also has support for shared memory through the build-in ETS (Erlang Term Storage) module.

Distributed Erlang

- An Erlang node is an Erlang runtime system containing a complete virtual machine which contains its own address space and set of processes.

Distributed Erlang

- An Erlang node is an Erlang runtime system containing a complete virtual machine which contains its own address space and set of processes.
- Erlang nodes can communicate over the network, remotely or through the localhost.

Distributed Erlang

- An Erlang node is an Erlang runtime system containing a complete virtual machine which contains its own address space and set of processes.
- Erlang nodes can communicate over the network, remotely or through the localhost.
- Pids continue to be unique over different nodes (globally).

Distributed Erlang

- An Erlang node is an Erlang runtime system containing a complete virtual machine which contains its own address space and set of processes.
- Erlang nodes can communicate over the network, remotely or through the localhost.
- Pids continue to be unique over different nodes (globally).
- Inside two different nodes, two different processes can have the same local Pid.

Distributed Erlang

- An Erlang node is an Erlang runtime system containing a complete virtual machine which contains its own address space and set of processes.
- Erlang nodes can communicate over the network, remotely or through the localhost.
- Pids continue to be unique over different nodes (globally).
- Inside two different nodes, two different processes can have the same local Pid.
- All primitives operate over the network similarly as they would on the same node.

Concuerror

Concuerror is a tool that uses various stateless model checking techniques in order to systematically test an Erlang program, with the aim of detecting and reporting concurrency-related runtime errors. Its main components are:

- Instrumenter:

Concuerror

Concuerror is a tool that uses various stateless model checking techniques in order to systematically test an Erlang program, with the aim of detecting and reporting concurrency-related runtime errors. Its main components are:

- Instrumenter:
  - adds preemption points to various points in the code of a tested program.

Concuerror

Concuerror is a tool that uses various stateless model checking techniques in order to systematically test an Erlang program, with the aim of detecting and reporting concurrency-related runtime errors. Its main components are:

- Instrumenter:
  - adds preemption points to various points in the code of a tested program.
  - makes it possible to produce specific interleaving.

Concuerror

Concuerror is a tool that uses various stateless model checking techniques in order to systematically test an Erlang program, with the aim of detecting and reporting concurrency-related runtime errors. Its main components are:

- Instrumenter:
  - adds preemption points to various points in the code of a tested program.
  - makes it possible to produce specific interleaving.
- Scheduler:

Concuerror

Concuerror is a tool that uses various stateless model checking techniques in order to systematically test an Erlang program, with the aim of detecting and reporting concurrency-related runtime errors. Its main components are:

- Instrumenter:
  - adds preemption points to various points in the code of a tested program.
  - makes it possible to produce specific interleaving.
- Scheduler:
  - uses mainly source-DPOR or optimal-DPOR, to determine which interleavings need to be checked.

Concuerror

Concuerror is a tool that uses various stateless model checking techniques in order to systematically test an Erlang program, with the aim of detecting and reporting concurrency-related runtime errors. Its main components are:

- Instrumenter:
  - adds preemption points to various points in the code of a tested program.
  - makes it possible to produce specific interleaving.
- Scheduler:
  - uses mainly source-DPOR or optimal-DPOR, to determine which interleavings need to be checked.
  - controls the execution of the processes to produce those interleavings.

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.
- Develop parallel version for optimal-DPOR algorithm.

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.
- Develop parallel version for optimal-DPOR algorithm.
- Implement those parallel algorithms in Concuerror.

Aim of Thesis

- Develop parallel version for source-DPOR algorithm.
- Develop parallel version for optimal-DPOR algorithm.
- Implement those parallel algorithms in Concuerror.
- Evaluate the performance of our implementation.

Section 2

Sequential DPOR Algorithms

General DPOR Concepts

DPOR: performs a DFS using a backtrack set. Two main techniques:

- Persistent sets: only a provably sufficient subset of the enabled processes gets explored.

General DPOR Concepts

DPOR: performs a DFS using a backtrack set. Two main techniques:

- Persistent sets: only a provably sufficient subset of the enabled processes gets explored.
- Sleep sets: prevents redundant exploration.

Optimality in DPOR

- A DPOR algorithm is optimal if, for every maximal execution sequence E, it explores exactly one interleaving from $[E]_{\simeq}$.

Optimality in DPOR

- A DPOR algorithm is optimal if, for every maximal execution sequence E, it explores exactly one interleaving from $[E]_{\simeq}$.
- Persistent-set based DPOR is not optimal.

Source Sets

Definition 1 (Weak Initials after an execution sequence $E.w$, $WI_{[E]}(w)$)

$p \in WI_{[E]}(w)$ if and only if there are sequences w' and v such that $E.w.v \simeq E.p.w'$.

Definition 2 (Source Sets)

Let E be an execution sequence, and let W be a set of sequences, such that $E.w$ is an execution sequence for each $w \in W$. A set T of processes is a source set for W after E if for each $w \in W$ we have $WI_{[E]}(w) \cap T \neq \emptyset$.

A source set of a sequence w at an execution sequence E, contains the process that can perform the "first steps" after E that can reproduce sequences equivalent to w.

Source-DPOR

```
Function Explore(E,Sleep)
  if ∃p ∈ (enabled(s_[E])\Sleep) then
    backtrack(E) := p;
    while ∃p ∈ (backtrack(E)\Sleep) do
      foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
        let E′ = pre(E, e);
        let u = notdep(e, E).p;
        if I_[E′](u) ∩ backtrack(E′) = ∅ then
          add some q′ ∈ I_[E′](u) to backtrack(E′);
        end
      end
      let Sleep′ := {q ∈ Sleep | E ⊨ p♢q};
      Explore(E.p, Sleep′);
      add p to Sleep;
    end
  end
```

Algorithm 1: Source-DPOR

Source-DPOR

Source-DPOR in action:

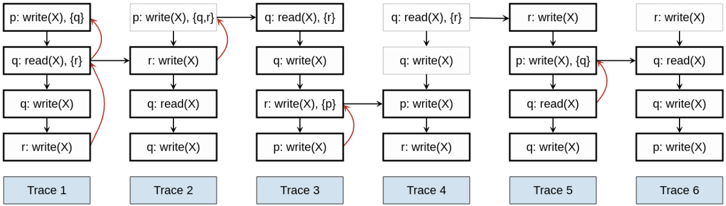


Figure: Interleavings explored by the source-DPOR.

Motivation for Wakeup Trees

```
Function Explore(E,Sleep)
  if ∃p ∈ (enabled(s_[E]) \ Sleep) then
    backtrack(E) := p;
    while ∃p ∈ (backtrack(E) \ Sleep) do
      foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p)
       do
        let E' = pre(E, e);
        let u = notdep(e, E).p;
        if I_[E'](u) ∩ backtrack(E') = ∅ then
          add some q' ∈ I_[E'](u) to backtrack(E');
        end
      end
      let Sleep' := {q ∈ Sleep | E ⊨ p◇q};
      Explore(E.p, Sleep');
      add p to Sleep;
    end
  end
```

- $E'.u$ needs to be explored.
- But only a single process gets added to the backtrack.
- This may lead to sleep-set blocking.

Wakeup Trees

Definition 3

(Ordered Tree)

An *ordered tree* is a pair $\langle B, \prec \rangle$, where B (the set of nodes) is a finite prefix-closed set of sequences of processes with the empty sequence $\langle\rangle$ being the root. The children of a node w, of form $w.p$ for some set of processes p, are ordered by $\prec$. In $\langle B, \prec \rangle$, such an ordering between children has been extended to the total order $\prec$ on B by letting $\prec$ be the induced post-order relation between the nodes in B. This means that if the children $w.p_1$ and $w.p_2$ are ordered as $w.p_1 \prec w.p_2$, then $w.p_1 \prec w.p_2 \prec w$ in the induced post-order.

Definition 4

(Wakeup Tree)

Let E be an execution sequence and P a set of processes. a *wakeup tree* after $\langle E, P \rangle$ is an ordered tree $\langle B, \prec \rangle$, for which the following properties hold:

- $WI_{[E]}(w) \cap P = \emptyset$ for every leaf w of B.
- For every node in B of the form $u.p$ and $u.w$ such that $u.p \prec u.w$ and $u.w$ is a leaf the $p \notin WI_{[E.u]}(w)$ property must hold true.

Wakeup Trees

Intuitively, wakeup trees hold sequences of processes that need to be explored. They can visualized in the following way:

Wakeup Trees

Intuitively, wakeup trees hold sequences of processes that need to be explored. They can visualized in the following way:

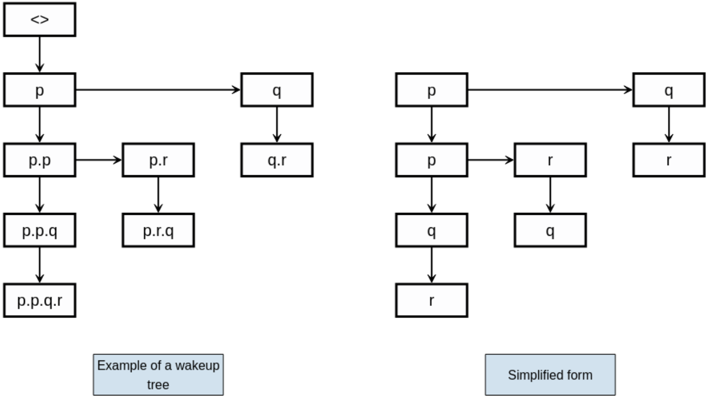


Figure: Visualizing wakeup trees.

Wakeup Trees

Inserting new sequences ($insert_{[E]}(w, \langle B, \prec \rangle)$) in the wakeup tree has the following properties:

- $insert_{[E]}(w, \langle B, \prec \rangle)$ is also a wakeup tree after $\langle E, P \rangle$.

Wakeup Trees

Inserting new sequences ($insert_{[E]}(w, \langle B, \prec \rangle)$) in the wakeup tree has the following properties:

- $insert_{[E]}(w, \langle B, \prec \rangle)$ is also a wakeup tree after $\langle E, P \rangle$.
- Any leaf of $\langle B, \prec \rangle$ remains a leaf of $insert_{[E]}(w, \langle B, \prec \rangle)$.

Wakeup Trees

Inserting new sequences ($insert_{[E]}(w, \langle B, \prec\rangle)$) in the wakeup tree has the following properties:

- $insert_{[E]}(w, \langle B, \prec\rangle)$ is also a wakeup tree after $\langle E, P\rangle$.
- Any leaf of $\langle B, \prec\rangle$ remains a leaf of $insert_{[E]}(w, \langle B, \prec\rangle)$.
- $insert_{[E]}(w, \langle B, \prec\rangle)$, while it may not contain w as a leaf, it contains a leaf u with $u \sim_{[E]} w$.

Optimal-DPOR

Function *Explore*(E,$Sleep$,WuT)

- **if** $enabled(s_{[E]}) = \emptyset$ **then**
  - **foreach** $e, e' \in dom(E)$ *such that* ($e \lesssim_E e'$) **do**
    - **let** $E' = pre(E, e)$;
    - **let** $v = notdep(e, E).proc(e')$;
    - **if** $sleep(E') \cap WI_{[E']}(v) = \emptyset$ **then**
      - $insert_{[E']}(v, wut(E'))$;
    - **end**
  - **end**
- **else**
  - **if** $WuT \neq \langle\{\langle\rangle\}, \emptyset\rangle$ **then**
    - $wut(E) := WuT$;
  - **else**
    - choose $p \in enabled(s_{[E]})$;
    - $wut(E) := \langle\{p\}, \emptyset\rangle$;
  - **end**
  - $sleep(E) := Sleep$;
  - **while** $\exists p \in wut(E)$ **do**
    - **let** $p = min_{\prec}\{p \in wut(E)\}$;
    - **let** $Sleep' := \{q \in sleep(E) \mid E \models p\Diamond q\}$;
    - **let** $WuT' = subtree(wut(E), p)$;
    - $Explore(E.p, Sleep', WuT')$;
    - add p to $sleep(E)$;
    - remove all sequences of form $p.w$ from $wut(E)$;
  - **end**
- **end**

Algorithm 2: Optimal-DPOR

Optimal-DPOR example

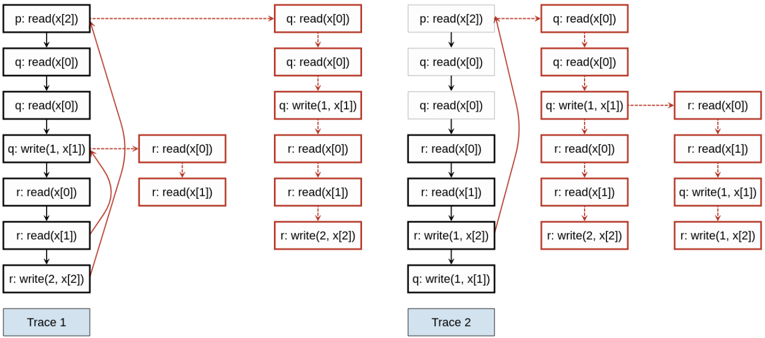


Figure: Optimal-DPOR exlporation.

Optimal-DPOR example

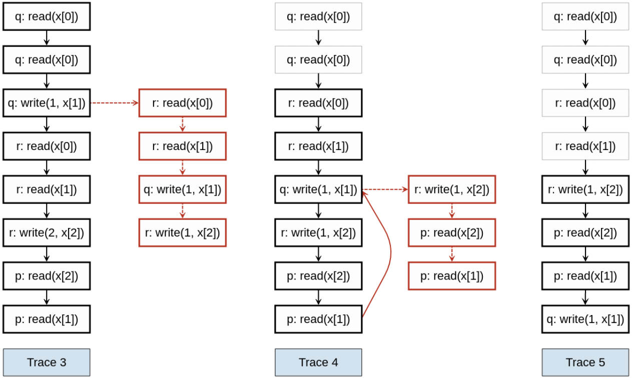


Figure: Optimal-DPOR exlporation.

Section 3

Parallelizing source-DPOR and optimal-DPOR

Parallel source-DPOR

Lets assume that we have an execution sequence E and that p and q are processes in $backtrack(E)$. We could:

- Assign the exploration of $E.p$ and $E.q$ to different workers-schedulers.

Parallel source-DPOR

Lets assume that we have an execution sequence E and that p and q are processes in $backtrack(E)$. We could:

- Assign the exploration of $E.p$ and $E.q$ to different workers-schedulers.
- The exploration frontier gets updated in a non-local manner.

Parallel source-DPOR

Lets assume that we have an execution sequence E and that p and q are processes in $backtrack(E)$. We could:

- Assign the exploration of $E.p$ and $E.q$ to different workers-schedulers.
- The exploration frontier gets updated in a non-local manner.
- Who would be responsible for exploring $E.r$?

Parallel source-DPOR

Lets assume that we have an execution sequence E and that p and q are processes in $backtrack(E)$. We could:

- Assign the exploration of $E.p$ and $E.q$ to different workers-schedulers.
- The exploration frontier gets updated in a non-local manner.
- Who would be responsible for exploring $E.r$?
- How do we now that the subtrees are balanced?

Basic Idea

We are going to use a centralized Controller who will be responsible for:

- assigning work to different schedulers, by partitioning Execution sequences.

Basic Idea

We are going to use a centralized Controller who will be responsible for:

- assigning work to different schedulers, by partitioning Execution sequences.
- resolving conflicts.

Basic Idea

The Controller will keep track of:

- the Frontier of our search: the set of the execution sequences assigned to different schedulers.

Basic Idea

The Controller will keep track of:

- the Frontier of our search: the set of the execution sequences assigned to different schedulers.
- the Execution Tree: the subtree of the state space that is currently being explored i.e., the Frontier combined in a tree form. A path from the root of tree to a leaf uniquely determines an execution sequence.

Basic Idea

Also, lets introduce the concept of Execution Tree node ownership:

- A scheduler exclusively **owns** a node of the state space if:

Basic Idea

Also, lets introduce the concept of Execution Tree node ownership:

- A scheduler exclusively **owns** a node of the state space if:
    - it is an assigned backtrack entry

Basic Idea

Also, lets introduce the concept of Execution Tree node ownership:

- A scheduler exclusively **owns** a node of the state space if:
  - it is an assigned backtrack entry
  - it is a descendant of a node that the scheduler owns.

Basic Idea

Also, lets introduce the concept of Execution Tree node ownership:

- A scheduler exclusively **owns** a node of the state space if:
  - it is an assigned backtrack entry
  - it is a descendant of a node that the scheduler owns.
- All other nodes, are considered to have a **disputed** ownership.

Controller Logic

Function *controller_loop*(N, $Budget$, $Schedulers$)
  $E_0 \leftarrow$ an arbitrary initial execution sequence;
  $Frontier \leftarrow [E_0]$;
  $T \leftarrow$ an execution tree rooted at E_0;
  while $Frontier \neq \emptyset$ **do**
    $Frontier \leftarrow partition(Frontier, N)$;
    while *exists an idle scheduler* S *and an unassigned execution sequence* E *in* $Frontier$ **do**
      $E_c \leftarrow$ a copy of E;
      mark E as assigned in $Frontier$;
      $spawn(S, explore\_loop(E_c, Budget))$;
    end
    $Frontier, T \leftarrow wait\_scheduler\_response(Frontier, T)$;
  end

Frontier Partitioning

Function *partition*($Frontier$, N)
  for *all E ∈ Frontier* **do**
    while $total\_backtrack\_entries(E) > 1$ ***and*** $size(Frontier) < N$ **do**
      $E' \leftarrow$ the smallest prefix of E that has a backtrack entry ;
      $p \leftarrow$ a process $\in backtrack(E')$;
      $E'_c \leftarrow$ a copy of E';
      remove p from $backtrack(E')$;
      add p to $sleep(E')$;
      add $backtrack(E')$ to $sleep(E'_c)$;
      add E'_c to $Frontier$;
    end
  end
  return $Frontier$;

Scheduler Exploration Loop

Function *explore_loop*(E_0, $Budget$)

$StartTime \leftarrow get\_time()$;

$E \leftarrow E_0$;

repeat

$E' \leftarrow explore(E)$;

$E' \leftarrow plan\_more\_interleavings(E')$;

$E \leftarrow get\_next\_execution\_sequence(E')$;

$CurrentTime \leftarrow get\_time()$;

until $CurrentTime - StartTime > Budget$ ***or*** $size(E) \leq size(E_0)$;

send E to Controller ;

Parallel source-DPOR

Function *wait_scheduler_response(Frontier, T)*
- **receive** E from a scheduler;
- remove E from $Frontier$;
- $E', T \leftarrow update\_execution\_tree(E, T)$;
- **if** E' *has at least one backtrack entry* **then**
  - add E' to $Frontier$;
- **end**
- **return** $Frontier, T$;

Algorithm 3: Handling Scheduler Response

Resolving conflicts

$update\_execution\_tree(E, T)$:

- iterates over the execution sequence and the execution tree simultaneously .
- backtrack entries of E that are not found in the execution tree, are added to it.
- ownership is claimed over them.

Load Balancing

- Load balancing through time-slicing.

Load Balancing

- Load balancing through time-slicing.
- Schedulers report back to Controller after their running time exceeds their budget.

Load Balancing

- Load balancing through time-slicing.
- Schedulers report back to Controller after their running time exceeds their budget.
- Smaller budget leads to better load balance, but also a larger communication overhead.

Load Balancing

- Load balancing through time-slicing.
- Schedulers report back to Controller after their running time exceeds their budget.
- Smaller budget leads to better load balance, but also a larger communication overhead.
- We chane the value of budget depending on the number of idle schedulers.

A simple example

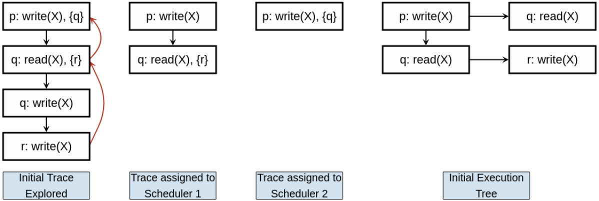


Figure: Initial execution sequences.

A simple example

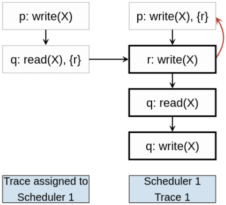


Figure: Scheduler 1 exploration.

A simple example

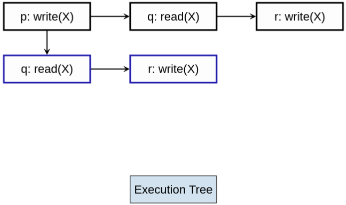


Figure: Execution Tree after Scheduler 1 reports to Controller.

A simple example

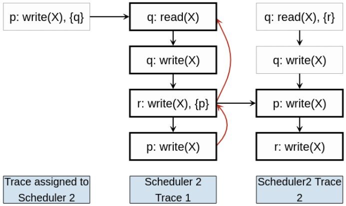


Figure: Scheduler 2 exploration.

Parallel optimal-DPOR - A first attempt

Any leaf of $\langle B, \prec \rangle$ remains a leaf of $insert_{[E]}(w, \langle B, \prec \rangle)$.
We use this to develop a parallel algorithm that uses:

- a single planner that is responsible for race detecting interleavings sequentially.
- multiple schedulers that simply explore in parallel the interleavings that generated by the planner
- a Controller who is responsible for managing the queue of the planner and for assigning interleavings to schedulers for exploration

Parallel optimal-DPOR - A first attempt

This attempt fails to provide any speedup:

Benchmark	Planning Time (%)	Exploration Time(%)	Sequential Time	Time for 4 Schedulers	Time for 24 Schedulers
readers 15	71.7%	28.3%	52m43.585s	98m28.251s	97m13.762s
lastzero 15	80.5%	19.5%	13m32.843s	24m98,312s	24m21,219s
readers 10	59.1%	40.9%	43.267s	59.699s	54.592s

Table: Parallel optimal-DPOR performance.

Parallel optimal-DPOR - A first attempt

Our attempt fails because:

- We have parallelized the exploration phase of our algorithm, but have kept the most time consuming phase sequential.

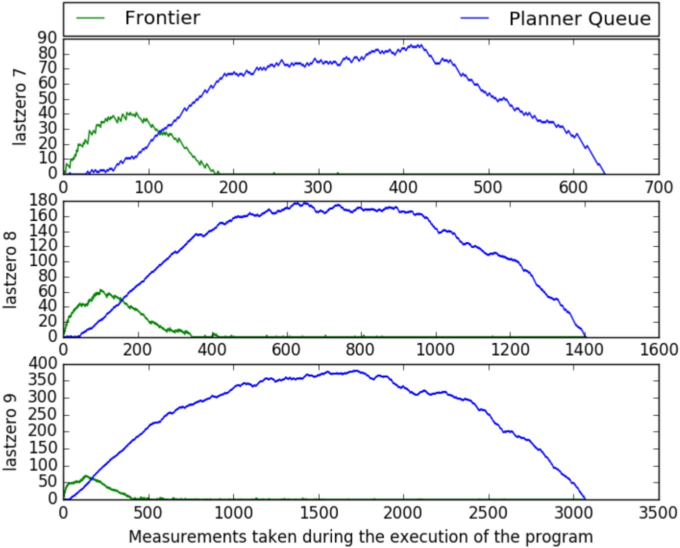
Parallel optimal-DPOR - A first attempt

Our attempt fails because:

- We have parallelized the exploration phase of our algorithm, but have kept the most time consuming phase sequential.
- We have noticed that the planner, during most of the execution of our program, does not generate enough interleavings to keep the schedulers busy.

Parallel optimal-DPOR - A first attempt

This behavior can be observed from the following graphs:



Scalable parallel optimal-DPOR

Lets assume that we have an execution sequence E and that w, v are leaf sequences of $wut(E)$:

- Assign the exploration of $E.w$ and $E.v$ to different workers-schedulers.

Scalable parallel optimal-DPOR

Lets assume that we have an execution sequence E and that w, v are leaf sequences of $wut(E)$:

- Assign the exploration of $E.w$ and $E.v$ to different workers-schedulers.
- Those schedulers will explore the subtrees rooted at the assigned sequences.

Changing the concept of ownership

We need to modify the concept of ownership as follows:

- A leaf sequence assigned to a scheduler is **owned** by that scheduler.

Changing the concept of ownership

We need to modify the concept of ownership as follows:

- A leaf sequence assigned to a scheduler is **owned** by that scheduler.
- This scheduler also owns every node in the subtree rooted at that leaf. If v is an owned leaf sequence in $wut(E)$ a scheduler owns every execution sequence that has a prefix of $E.v$.

Changing the concept of ownership

We need to modify the concept of ownership as follows:

- A leaf sequence assigned to a scheduler is **owned** by that scheduler.
- This scheduler also owns every node in the subtree rooted at that leaf. If v is an owned leaf sequence in $wut(E)$ a scheduler owns every execution sequence that has a prefix of $E.v$.
- Leaf sequence assigned to different schedulers are marked as **not owned**

Changing the concept of ownership

We need to modify the concept of ownership as follows:

- A leaf sequence assigned to a scheduler is **owned** by that scheduler.
- This scheduler also owns every node in the subtree rooted at that leaf. If v is an owned leaf sequence in $wut(E)$ a scheduler owns every execution sequence that has a prefix of $E.v$.
- Leaf sequence assigned to different schedulers are marked as **not owned**
- All other nodes are considered **disputed**.

Resolving conflicts

Conflicts are harder to resolve:

- Conflicts occur between sequences, instead of processes.

Resolving conflicts

Conflicts are harder to resolve:

- Conflicts occur between sequences, instead of processes.
- $insert_{[E]}(v, wut(E))$ may end up inserting at $wut(E)$ an execution sequence different than v.

Resolving conflicts

Conflicts are harder to resolve:

- Conflicts occur between sequences, instead of processes.
- $insert_{[E]}(v, wut(E))$ may end up inserting at $wut(E)$ an execution sequence different than v.
- different sequences may be equivalent.

Resolving conflicts

To resolve them:

- We modify $insert_{[E]}(v, wut(E))$ to insert sequences in the Execution Trees.

Resolving conflicts

To resolve them:

- We modify $insert_{[E]}(v, wut(E))$ to insert sequences in the Execution Trees.
- If a disputed sequence can be inserted into the execution tree, ownership is claimed it.

Resolving conflicts

To resolve them:

- We modify $insert_{[E]}(v, wut(E))$ to insert sequences in the Execution Trees.
- If a disputed sequence can be inserted into the execution tree, ownership is claimed it.
- Otherwise, it is marked as not owned.

Scalable parallel optimal-DPOR

This approach is similar to the parallel source-DPOR except that:

- $update\_execution\_tree(E, T)$ $inserts$ sequences in the Execution Tree, instead of processes.

Scalable parallel optimal-DPOR

This approach is similar to the parallel source-DPOR except that:

- $update\_execution\_tree(E, T)$ $inserts$ sequences in the Execution Tree, instead of processes.
- Schedulers are assigned the state space under a sequence, instead of a single process.

Comparing the two algorithms

Therefore:

- the Controller has a higher computational complexity.

Comparing the two algorithms

Therefore:

- the Controller has a higher computational complexity.
- the schedulers communicate with the Controller more frequently.

Modifying Concuerror

- Sequential Concuerror has one process for every process of tested program.

Modifying Concuerror

- Sequential Concuerror has one process for every process of tested program.
- Different schedulers must be able to explore different interleavings concurrently.

Modifying Concuerror

- Sequential Concuerror has one process for every process of tested program.
- Different schedulers must be able to explore different interleavings concurrently.
- Each scheduler must have its own set of processes.

Modifying Concuerror

- Sequential Concuerror has one process for every process of tested program.
- Different schedulers must be able to explore different interleavings concurrently.
- Each scheduler must have its own set of processes.
- Create mappings between the different copies of a process.

The issue with Pids

- Different Pids per process.

The issue with Pids

- Different Pids per process.
- Erlang allows comparison between Pids.

The issue with Pids

- Different Pids per process.
- Erlang allows comparison between Pids.
- The result of those comparisons must remain the same on different schedulers.

The issue with Pids

- Different Pids per process.
- Erlang allows comparison between Pids.
- The result of those comparisons must remain the same on different schedulers.
- pid_to_list BIF makes things worse.

The issue with Pids

- We want different process with the same Pid.

The issue with Pids

- We want different process with the same Pid.
- Inside an Erlang node, process from that node are reference by their local Pid.

The issue with Pids

- We want different process with the same Pid.
- Inside an Erlang node, process from that node are reference by their local Pid.
- Processes from different nodes can have the same local Pid.

The issue with Pids

- We want different process with the same Pid.
- Inside an Erlang node, process from that node are reference by their local Pid.
- Processes from different nodes can have the same local Pid.
- Pids are generated sequentially within a node.

Different processes with the same Pid

- Each scheduler runs in its own node.

Different processes with the same Pid

- Each scheduler runs in its own node.
- Reach a consensus between the different schedulers as to the initial local Pid.

Different processes with the same Pid

- Each scheduler runs in its own node.
- Reach a consensus between the different schedulers as to the initial local Pid.
- Spawn processes after the consensus been reached.

Different processes with the same Pid

- Each scheduler runs in its own node.
- Reach a consensus between the different schedulers as to the initial local Pid.
- Spawn processes after the consensus been reached.
- The i_{th} processes spawned this way on different nodes, will all have the same local Pid.

Modifying Concuerror

Concuerror now runs distributed.

Section 4

Performance Evaluation

Benchmarks

- The benchmarks were performed on a multiprocessor with 64 AMD Opteron 6276(2.3 GHz) cores, 126 GB of memory, running Linux 4.9.0-8amd64 and running the later Erlang version (Erlang/OTP 21.1).
- While running our tests, we are using the $-keep\_going$ flag to continue exploring our state space, even after an error is found. We do this so we can evaluate how fast the complete state space gets explored, regardless of whether errors exist.

Benchmarks

Lets give a brief overview of our benchmarks:

- $indexer\ N$: This test uses a Compare and Swap (CAS) primitive instruction to check if a specific element of a matrix is set to 0 and if so, set it to a new value. This is implemented in Erlang by using ETS tables and specifically the $insert\_new/2$ function. This function returns false if the key of the inserted tuple exists (the entry is set to 0) or it inserts the tuple if the key is not found. N refers to the number of threads that are performing this function.
- $readers\ N$: This benchmark uses a writer process that writes a variable and N reader processes that read that variable.
- $lastzero\ N$: In this test we have $N + 1$ processes that read and write on an array of $N + 1$ size, which has all its values initialized with zero. The first process reads the array in order to find the zero element with the highest index. The other N processes read an array element and update the next one.
- $rush\ hour$: a program that uses processes and ETS tables to solve the Rush Hour puzzle in parallel, using A*search. Rush hour is a complex but self-contained (917 lines of code) program.

Results for sequential algorithms

Lets present here number of interleavings for our benchmarks, as well as their performance for sequential algorithm and the parallel algorithm with one scheduler:

Benchmark	Traces for source-DPOR	Traces for optimal-DPOR	Time for source-DPOR	Time for parallel source-DPOR with 1 scheduler	Time for optimal-DPOR	Time for parallel optimal-DPOR with 1 scheduler
lastzero 11	60073	7168	49m8.510s	53m59.169s	14m8.266s	17m50.494s
indexer 17	262144	262144	186m8.136sec	205m24.872sec	193m54.320sec	252m21.033sec
readers 15	32768	32768	37m68.865s	46m28.711s	51m40.792s	67m50.643s
rush hour	46656	46656	52m36.889s	56m3.521s	51m11.184s	58m32.962s

Table: Sequential performance of source-DPOR and optimal-DPOR on four benchmarks.

We can notice that the overhead for our parallel optimal-DPOR appears to be larger than the overhead of the parallel source-DPOR. This is to be expected since updating the execution tree in the Controller should be more expensive for the optimal algorithm.

Evaluation on readers 15

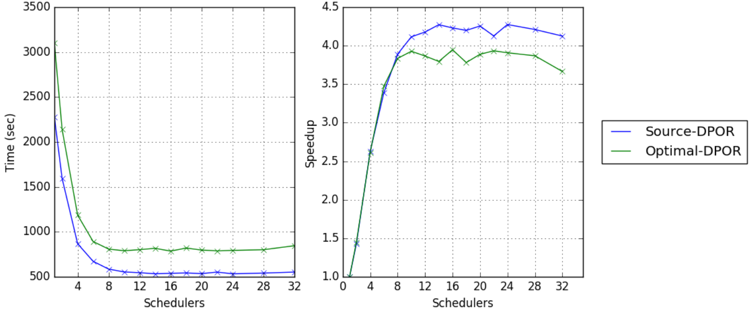


Figure: Performance of readers 15 with Budget of 10000.

Evaluation on readers 15

Trying to figure out why our algorithm fails scale for readers 15:

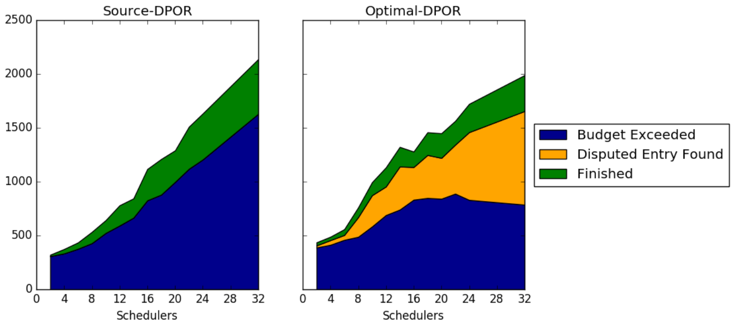


Figure: Number of times schedulers stopped their execution with a Budget of 10000.

Evaluation on readers 15

Increase budget to 30000ms:

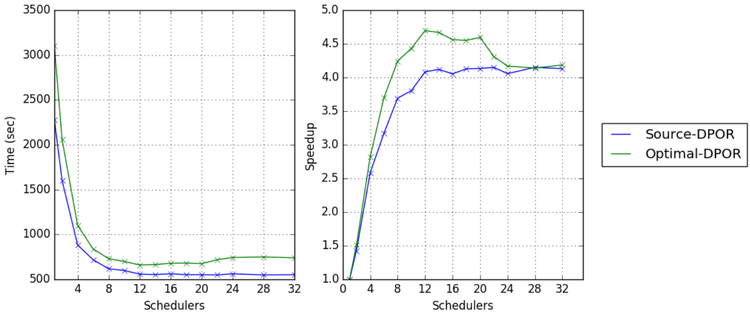


Figure: Performance of readers 15 with budget of 30000.

Evaluation on readers 15

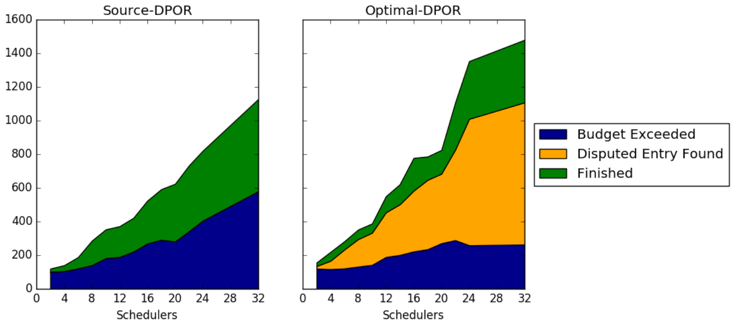


Figure: Number of times schedulers stopped their execution with a Budget of 30000.

Evaluation on readers 15

Increasing the Budget:

- Reduces the performance of source-DPOR, since it causes load imbalance.

Evaluation on readers 15

Increasing the Budget:

- Reduces the performance of source-DPOR, since it causes load imbalance.
- Increases the performance of optimal-DPOR, since finding disputed entries also leads to load balancing and therefore, optimal-DPOR does not need as frequent load balance.

Evaluation on lastzero 11

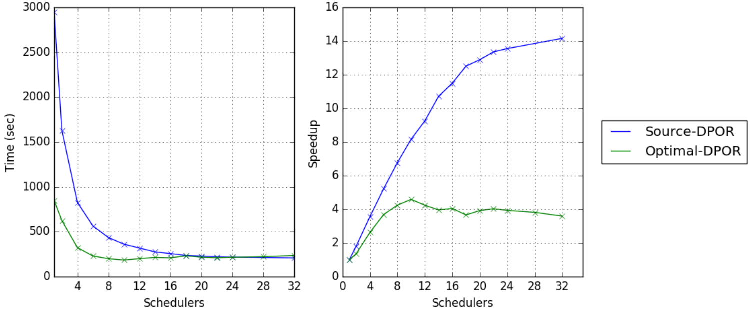


Figure: Performance of lastzero 11 with Budget of 10000 for source and 30000 for optimal.

Evaluation on rush hour

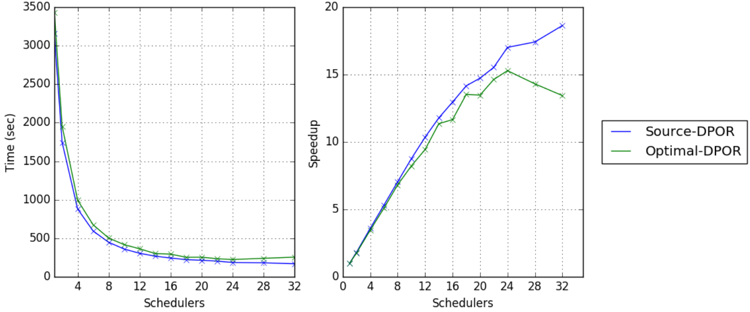


Figure: Performance of rush hour with Budget of 10000 for source and 30000 for optimal.

Evaluation on indexer 17

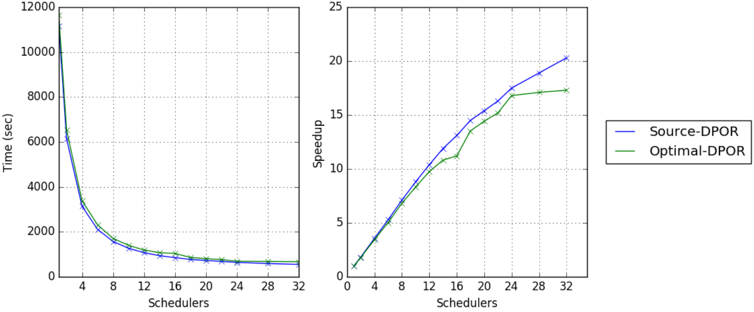


Figure: Performance of indexer 17 with Budget of 10000 for source and 30000 for optimal.

Evaluation on rush hour

Both of our algorithms provide high speed and decent scalability on this test case. However, after 24 schedulers we notice that the scalability of optimal-DPOR starts to break. This is because:

- the communication between the Controller and the schedulers in the case of optimal-DPOR is substantially more frequent.

Evaluation on rush hour

Both of our algorithms provide high speed and decent scalability on this test case. However, after 24 schedulers we notice that the scalability of optimal-DPOR starts to break. This is because:

- the communication between the Controller and the schedulers in the case of optimal-DPOR is substantially more frequent.
- the $update\_execution\_tree$ function in optimal-DPOR is more “expensive”.

Recap

Depending on the benchmark, Source-DPOR:

- Can provide a speedup by a factor of 20.

Recap

Depending on the benchmark, Source-DPOR:

- Can provide a speedup by a factor of 20.
- Retain scalability for 32 schedulers.

Recap

Depending on the benchmark, Source-DPOR:

- Can provide a speedup by a factor of 20.
- Retain scalability for 32 schedulers.

Recap

Depending on the benchmark, Source-DPOR:

- Can provide a speedup by a factor of 20.
- Retain scalability for 32 schedulers.

Optimal-DPOR:

- Can provide a speedup by a factor of 17.

Recap

Depending on the benchmark, Source-DPOR:

- Can provide a speedup by a factor of 20.
- Retain scalability for 32 schedulers.

Optimal-DPOR:

- Can provide a speedup by a factor of 17.
- Retain scalability for up to 24 schedulers.

Conclusion

During this thesis we:

- Parallelized source-DPOR and optimal-DPOR.

Conclusion

During this thesis we:

- Parallelized source-DPOR and optimal-DPOR.
- Implemented our algorithms in Concuerror.

Conclusion

During this thesis we:

- Parallelized source-DPOR and optimal-DPOR.
- Implemented our algorithms in Concuerror.
- Achieved significant speedups.

Conclusion

During this thesis we:

- Parallelized source-DPOR and optimal-DPOR.
- Implemented our algorithms in Concuerror.
- Achieved significant speedups.
- Achieved scalability up to 32 schedulers on specific benchmarks.

Thank you for your attention!