Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Division of Computer Science

Parallelizing Concuerror: A Dynamic Partial Order Reduction Testing Tool for Erlang Programs

Διπλωματική Εργασία

PANAGIOTIS FYTAS

Επιβλέπων : Konstantinos Sagonas
Associate Professor NTUA

Αθήνα, Νοέμβριος 2018

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Division of Computer Science

Parallelizing Concuerror: A Dynamic Partial Order Reduction Testing Tool for Erlang Programs

Διπλωματική Εργασία

PANAGIOTIS FYTAS

Επιβλέπων : Konstantinos Sagonas
Associate Professor NTUA

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 01η Νοεμβρίου 2018.

....	....	....
Konstantinos Sagonas Associate Professor NTUA	Nikolaos S. Papaspyrou Associate Professor NTUA	Nectarios Koziris Professor NTUA

Αθήνα, Νοέμβριος 2018

....

Panagiotis Fytas

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Panagiotis Fytas, 2018.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η επαλήθευση και ο έλεγχος ταυτόχρονων προγραμμάτων είναι μία πολύπλοκη εργασία. Λόγω του μη ντετερμινιστικού τρόπου με τον οποίο δουλεύει ο scheduler ενός συστήματος, μπορεί μόνο μερικά interleavings από processes να είναι ελαττωματικά και συνεπώς, για να ελεγχθεί η ορθότητα ενός προγράμματος, πρέπει να εξεταστούν όλα τα πιθανά schedulings. Ο αριθμός, όμως, αυτών των schedulings είναι υψηλής τάξης εκθετικός σε σχέση με το μέγεθος του προγράμματος. Η πιο αποτελεσματική μέθοδος για να λυθεί αυτό το πρόβλημα είναι το Stateless model checking με τη χρήση αναγωγής σε δυναμικές σχέσεις μερική διάταξης (Dynamic Partial Order Reduction ή DPOR). Καθώς τα παράλληλα συστήματα επεξεργασίας έχουν γίνει το κυρίαρχο πρότυπο της σύγχρονης αρχιτεκτονικής υπολογιστών, η παραλληλοποίηση των διαφόρων DPOR αλγορίθμων είναι αναγκαία για την κλιμάκωση αυτών των αλγορίθμων στους σύγχρονους υπολογιστές.

Αυτή η διπλωματική εργασία ασχολείται με τη παραλληλοποίηση του Concuerror, ένα stateless model checking εργαλείο που χρησιμοποιεί διάφορους DPOR αλγόριθμους για να ελέγξει ταυτόχρονα προγράμματα γραμμένα σε Erlang. Συγκεκριμένα, θα εστιάσουμε στο να καταστήσουμε εφικτή την παράλληλη εξερεύνηση διαφορετικών interleavings απο τον Concuerror και στο να σχεδιάσουμε παράλληλες εκδοχές για τα τους δύο πιο αποδοτικούς αλγορίθμους του Concuerror: τον source-DPOR και τον optimal-DPOR. Επίσης, θα αξιολογήσουμε την επιτάχυνση και την κλιμακωσιμότητα των υλοποιήσεων μας σε διάφορα benchmarks που χρησιμοποιούνται ευρέως για την αξιολόγηση DPOR αλγορίθμων.

Λέξεις κλειδιά

Stateless Model Checking, Systematic Concurrency Testing, Dynamic Partial Order Reduction, Parallelization

Abstract

Testing and verifying concurrent programs is quite a daunting task. Due to the non-determinism of the scheduler, errors can occur only on specific process interleaving sequences and therefore, all possible different schedulings should be examined. Currently, the most practical method of dealing with the combinatorial state space explosion of this problem is a technique called Dynamic Partial Order Reduction (DPOR). As parallel processing has become the dominant paradigm in modern computer architecture, developing parallel versions of DPOR algorithms is essential for scaling those algorithms to modern computers.

This thesis is concerned with the parallelization of Concuerror, a stateless model checking tool that uses various DPOR techniques for testing concurrent Erlang programs. Specifically, we are going to focus on modifying Concuerror, in order to be able to explore different interleavings in parallel, and on developing parallel versions for the two most efficient DPOR algorithms implemented in Concuerror: source-DPOR and optimal-DPOR. Also, we are going to evaluate the speedup and scalability of our implementation on certain benchmarks that are widely used for evaluating DPOR algorithms.

Key words

Stateless Model Checking, Systematic Concurrency Testing, Dynamic Partial Order Reduction, Parallelization

Ευχαριστίες

Panagiotis Fytas,

Αθήνα, 01η Νοεμβρίου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-16, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Νοέμβριος 2018.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Acknowledgements

,
Athens, November 01, 2018

This thesis is also available as Technical Report CSD-SW-TR-1-16, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, November 2018.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Acknowledgements 11

Περιεχόμενα 13

Κατάλογος πινάκων 15

Κατάλογος σχημάτων 17

Κώδικες 19

List of Algorithms 21

1. Introduction 23

2. Preliminaries 25
   2.1 The Erlang Language 25
      2.1.1 Concurrent Erlang 25
      2.1.2 Distributed Erlang 25
   2.2 Testing Concurrent Programs 26
   2.3 Framework 26
      2.3.1 Abstraction Model 27
      2.3.2 Event Dependencies 27
      2.3.3 Independence and Races 28
   2.4 Concuerror Overview 29
      2.4.1 Instrumenter 29
      2.4.2 Scheduler 29
      2.4.3 Logger 29

3. Dynamic Partial Order Reduction 31
   3.1 Basic DPOR Concepts 31
   3.2 Source Sets 32
   3.3 Source-DPOR 32
   3.4 Wakeup Trees 33
      3.4.1 Operations on Wakeup Trees 34
   3.5 Optimal-DPOR 34

4. Parallelizing Source-DPOR Algorithm . . . 37
4.1 Existing Work . . . 37
4.2 Parallel source-DPOR . . . 38
4.2.1 Basic Idea . . . 38
4.2.2 Algorithm . . . 39
4.2.3 Load Balancing . . . 41
4.2.4 A Simple Example . . . 41

5. Parallelizing Optimal-DPOR Algorithm . . . 45
5.1 Parallel optimal-DPOR - A First Attempt . . . 45
5.1.1 Basic Idea . . . 45
5.1.2 Algorithm . . . 45
5.1.3 Example . . . 47
5.1.4 Performance Evaluation . . . 50
5.1.5 Performance Analysis . . . 50
5.2 Scalable Parallel optimal-DPOR . . . 52
5.3 Basic Idea . . . 52

6. Implementation Details . . . 53
6.1 Dealing with Processes . . . 53
6.2 Execution Sequence Replay . . . 54
6.3 Logger Modifications . . . 55

7. Performance Evaluation . . . 57
7.1 Tests Overview . . . 57
7.2 Source DPOR . . . 58
7.2.1 Preformance Results . . . 58
7.2.2 Performance Analysis . . . 59
7.3 Optimal DPOR . . . 60
7.3.1 Preformance Results . . . 60
7.4 Final Comments . . . 60

8. Concluding Remarks . . . 61

Βιβλιογραφία . . . 63

Κατάλογος πινάκων

5.1 Parallel Optimal-DPOR performance. 50

7.1 Sequential Source-DPOR vs Optimal-DPOR for our benchmarks 57

Κατάλογος σχημάτων

4.1 Simple readers-writes example 41
4.2 Interleavings explored by the sequential source-DPOR 42
4.3 Initial interleaving explored by the parallel algorithm 42
4.4 Exploration of the assigned traces by each scheduler 43
4.5 Execution Tree if Scheduler 1 returned first 44

5.1 *Lastzero* 2 example 48
5.2 Interleavings explored by the sequential optimal-DPOR 48
5.3 Initial interleaving explored by the parallel optimal-DPOR 49
5.4 Execution time for readers 10 by optimal-DPOR. 50
5.5 Planner Queue and Frontier sizes for the execution of lastzero. 51

7.1 Performance for indexer 15 58
7.2 Performance for readers 15 58
7.3 Performance for lastzero 11 59
7.4 Speedup comparison for source-DPOR 59

Κώδικες

6.1 Environment variables 55

List of Algorithms

1 Source-DPOR 32
2 Optimal-DPOR 35

3 Controller Loop 39
4 Frontier Partitioning 39
5 Scheduler Exploration Loop 40
6 Handling Scheduler Response 40

7 Controller for Optimal-DPOR - First Attempt 46
8 Optimal Frontier Partitioning 47
9 Handling Scheduler and Planner Response 47

Κεφάλαιο 1

Introduction

Nowadays, the necessity for concurrency is undeniable. Moore's Law [Moor06] dictates that the number of transistors that can be placed inside a dense integrated circuit doubles roughly every two years. However, during the past years, energy and heat restrictions have significantly thwarted the ability to accelerate a single chip. Multithreaded architectures is the main way with which modern technology can keep up with Moore's Law and therefore, concurrent programming is essential for scaling software to modern hardware. Apart from that, through concurrency, it becomes possible for long-running tasks to not delay short running ones and consequently, concurrent programs offer an increased availability of services making them essential for numerous applications, such as web services.

Nevertheless, due to the fact that concurrent processes and threads share resources and have to communicate with each other, concurrent programming is a more challenging endeavor, compared to sequential programming. To make matters worse, processes are scheduled in a non-deterministic manner, which can lead to errors that may occur only on specific rare interleavings. Therefore, concurrency bugs can be extremely hard to detect and reproduce, making techniques such as unit testing, ineffective in detecting such errors. Testing concurrent programs requires the systematic exploration of all possible interleavings (or at least a sufficient subset of those).

Advances made in model checking have led to *Stateless Model Checking* [Gode97], which systematically explores the state-space of a given program, without storing the global states, in order to verify that each reachable state satisfies a given property. Stateless model checking is a practical approach that does not have large memory requirements and has been implemented in tools such as CHESS [Musu08] and Verisoft [Gode05]. Still, this technique suffers from combinatorial explosion. However, different interleavings that can be obtained from each other by swapping adjacent and independent execution steps, can be considered equivalent. Partial Order Reduction (POR) [Gode96, Clar99, Valm91] algorithms utilize this observation to successfully diminish the size of the explored state-space. Dynamic Partial Order Reduction (DPOR) [Flan05, Abdu14] algorithms manage to achieve an even increased reduction, by detecting dependencies more accurately. DPOR techniques have been successfully implemented in tools like Concuerror [Chri13, Goto11], Nidhugg [Abdu15], Inspect [Yang08] and Eta [Sims11].

Parallelizing DPOR algorithms is essential in order to make them scalable to modern computer architectures, but also to potentially achieve significant speedups that will alleviate the effect of the exponential state-space explosion. The parallelization of DPOR algorithms that use persistent sets [Flan05, Lei06, Valm91] has already been been examined and parallel versions have been implement for Inspect [Yang07] and Eta [Sims12].

On this thesis, we are going to focus on the parallelization of Concuerror, a stateless model checker used for testing concurrent Erlang programs. Specifically, we are going to examine the parallelization of source-DPOR [Abdu14] and optimal-DPOR [Abdu14], which are the main algorithms implemented in Concuerror.

Κεφάλαιο 2

Preliminaries

2.1 The Erlang Language

Erlang is a declarative programming language with built-in support for concurrency, distribution, fault-tolerance, on-the-fly code reloading and automatic memory management. Erlang was initially developed by Ericsson in 1986 with the purpose of programming industrial telecommunications systems. However, it was later realized that it was also suitable for soft real-time applications [Vird96]. In 1998, Erlang/OTP (Open Telecom Platform) was released as open source and since then has been used commercially by various companies, including Ericsson, for a wide variety of large-scale applications.

2.1.1 Concurrent Erlang

The main strength of Erlang stems from its built-in concurrency support. In the core of this are the lightweight Erlang processes, each having its own program counter, process dictionary and call stack. Additionally, Erlang implements its processes through the runtime system of BEAM (the VM of Erlang) and therefore they are not mapped to OS threads. These processes use minimal memory, can be task switched extremely fast, can run in parallel and thousands of them can exist in a single machine. A process is identified globally by its Pid (process identifier).

Erlang concurrency is mainly based on message passing, since the state of Erlang processes is ("mostly") not shared. As such, the operator ''!" can be used to asynchronously send messages between processes, which can be of any datatype. The message is placed on the "mailbox" (a message queue) of the receiving process, until it is extracted by a *receive* expression. The *receive* expression uses pattern matching to scan the mailbox in a FIFO order for a message that matches that pattern. If no such message is found, the receiving process is blocked at the *receive*, waiting for a new message to be sent or for a timeout to occur, in case the *receive* expression had an *after* part.

Starting a process can be done efficiently through the BIF (built-in function) (*spawn/1*) (and its variants). By calling this function a new concurrent erlang process is created in order to evaluate the function specified at the arguments of *spawn*. The Pid of this new process is the return value of the *spawn* function.

It is often claimed that Erlang has no shared memory between different processes [Arms07] and inter-process communication is solely based on message passing. However, that is not entirely true since it is possible for different processes to access the same memory through the ETS (Erlang Term Storage) module.

2.1.2 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 [Arms07]. A node is assigned to a name of the form "name@host". Erlang nodes can connect with each other using cookies and they can communicate over the network. Pids continue to be unique over different nodes(globally). However, inside two different nodes, two different processes can have the same local Pid.

Distributed Erlang Programs can run on different nodes. An Erlang process can be spawned on any node, local or remote. All primitives (‘‘!”, *receive*, etc.) operate over the network similarly as they would on the same node.

2.2 Testing Concurrent Programs

Testing a concurrent program is significantly more challenging than testing a sequential one. The main reason behind this increased difficulty, is the non-deterministic way in which processes and threads are scheduled by computers. On a given input, a concurrent program may lead to different results depending on how its processes were scheduled. Specifically, errors can exist only on particular interleavings, that may have a small probability of occurring, making normal testing methods ineffective in detecting the existence of such bugs. Therefore, in order to test and verify concurrent programs it is essential that all possible interleavings are explored. *Model Checking* does this by exploring the complete state space of a program. In realistic scenarios, this approach is extremely inefficient, since storing the state of each process can have extreme memory requirements [Gode97]. *Stateless Model Checking* solves this issue by using a run-time scheduler to navigate the complete (reachable) state space without storing the actual state of the processes. Nevertheless, the number of possible interleavings increases exponentially with the length of the program and therefore, this method suffers from combinatorial explosion.

However, the complete set of possible interleavings does not need to be explored, since different interleavings can be equivalent, as long as they can be obtained from each other by reordering adjacent, independent execution steps. Such interleavings belong to the same *Mazurkiewicz trace* [Mazu87]. As a result, only a single interleaving from each different Mazurkiewicz trace needs to be examined in order to sufficiently test a program. This observation is utilized by various *Partial Order Reduction* (POR) [Gode96, Clar99, Valm91] techniques, which try to examine at most one interleaving from different Mazurkiewicz traces in order to alleviate the state space explosion that occurs when testing larger programs.

POR algorithms try to avoid exploring redundant interleavings by maintaining information regarding potential races between different processes. Such information can either be gained by *statically* analyzing the code of the program [Kurs98], or by *dynamically* detecting dependencies during the runtime of the program [Flan05]. *Dynamic Partial Order Reduction* (DPOR) utilizes the latter technique and generally outperforms static POR, since static information can be imprecise, which can lead to ineffective reduction of the state space.

Still, most DPOR techniques, fail to always guarantee that an optimal amount of interleavings gets fully explored (only one interleaving from each Mazurkiewicz trace), even when coupled with other reduction techniques, such as *sleep sets* [Gode96]. Redundant exploration can be reduced by using variations, such as source-DPOR [Abdu14], or even altogether avoided, through optimal-DPOR [Abdu14].

2.3 Framework

Here we present the abstractions that we use to model concurrent systems as well as the relations that occur in such a system [Abdu14].

2.3.1 Abstraction Model

Our abstraction system assumes concurrent programs have a finite amount of processes and that the code executes deterministically. Additionally, the state space of the program does not contain cycles and all execution sequences are considered to be finite (this does not exclude execution sequences which are blocked due to a deadlock).

The complete execution of a process p (also refered to as an actor process) splits into different execution steps, which are to be executed atomically. Each step combines a singular global statement along with the local statements (that do not have any explicit affect on the state of other processes) that take place before the next global statement. This acts as an optimization by reducing the total execution steps that can be interleaved and subsequently, the amount of interleavings that are to be examined [Gode97].

We use Σ to denote the set of global states ($s \in \Sigma$) and s_0 to denote the unique initial state. If a process cannot continue, the execution of the processes is considered to *block* in a state s. An execution sequence E of a system is a finite sequence of execution steps of its processes that is performed from s_0. Since each execution step is deterministic, 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, which is denoted as $s_{[E]}$. For a state s, let $enabled(s)$ denote the set of processes p that are enabled in s (i.e., for which execute $p(s)$ is defined). We use . to denote concatenation of sequences of processes. Thus, if p is not blocked after E, then $E.p$ is an execution sequence.

An *event* of E is a particular occurrence of a process in E. We use $\langle p, i\rangle$ to denote the i_{th} event of process p in the execution sequence E. In other words, the event $\langle p, i\rangle$ is the i_{th} execution step of process p in the execution sequence E. We use $dom(E)$ to denote the set of events $\langle p, i\rangle$ which are in E, i.e., $\langle p, i\rangle \in dom(E)$ iff E contains at least i steps of p. We will use $e, e', \dots$, to range over events. We use $proc(e)$ to denote the process p of an event $e = \langle p, i\rangle$. If $E.w$ is an execution sequence, obtained by concatenating E and w, then $dom_{[E]}(w)$ denotes $dom(E.w)$ $dom(E)$, i.e., the events in $E.w$ which are in w. As a special case, we use $next_{[E]}(p)$ to denote $dom_{[E]}(p)$. The notation $<_E$ is used to denote the total order between events in E, i.e. $e <_E e'$ denotes that e occurs before e' in E. We use $E' \leq E$ to denote that the sequence E' is a prefix of the sequence E.

2.3.2 Event Dependencies

Here we denote the happens-before relation between two events in an execution sequence E, a vital concept in DPOR algorithms, by using the notation $\rightarrow_E$. Specifically, the relation $e \rightarrow_E e'$, where events e, e' are in $dom(E)$, means that e "happens-before" e' in the execution sequence E. The DPOR algorithms presented here use the $happens\text{-}before\ assignment$, which is a function that assigns such a "happens-before" relation to events in any execution sequence. Usually, such a function is implemented using *vector clocks* [Matt88] to create relations concerning accesses to the same variables or sending and receiving the same message.

Definition 2.1. (Happens-Before Assignment)
A happens-before assignment, which assigns a unique happens-before relation $\rightarrow_E$ to any execution sequence E, is valid if it satisfies the following properties for all execution sequences E.

1. $\rightarrow_E$ is a partial order on $dom(E)$, which is included in $<_E$.
2. The execution steps of each process are totally ordered, i.e., $\langle p, i\rangle \rightarrow_E \langle p, i+1\rangle$ whenever $\langle p, i+1\rangle \in dom(E)$.

3. If E' is a prefix of E then $\rightarrow_E$ and $\rightarrow_{E'}$ are the same on $dom(E')$.

4. Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence which has exactly the same "happens-before" relation $\rightarrow_{E'}$ as $\rightarrow_E$. This means that the relation $\rightarrow_E$ induces a set of equivalent execution sequences, all with the same "happens-before" relation. We use $E \simeq E'$ to denote that E and E' are linearizations of the same "happens-before" relation, and $[E]_{\simeq}$ to denote the equivalence class of E.

5. If $E \simeq E'$ then $s_{[E]} = s_{[E']}$ (i.e., two equivalent traces will lead to the same state).

6. For any sequences E, E' and w, such that $E.w$ is an execution sequence, we have $E \simeq E'$ if and only if $E.w \simeq' E'.w$.

7. If p, q and r are different processes, then if $next_{[E]}(P) \rightarrow_{E.p.r} next_{[E.p]}(r)$ and $next_{[E]}(p) \not\rightarrow_{E..p.q} next_{[E.p]}(q)$, then $next_{[E]}(p) \rightarrow_{E.p.q.r} next_{[E.p.q]}(r)$.

For the happens-before relations that concern us the first six properties are fairly obvious. As far as the seventh property is concerned, if the next step of p happens before the next step of r after the sequence E, then the step of p still happens before the step of r even when some step of another process, which is not dependent with p, is inserted between p and r. This property is true in most practical computation models, such as the message passing and shared memory systems that concern us. As a special case, properties 4 and 5 together imply that if we have two consecutive events e and e' in E, such as that $e \not\rightarrow_E e'$, then they can be swapped without affecting the global state after the two events have occured.

2.3.3 Independence and Races

Here we define the concept of independence between events of a computation. If $E.p$ and $E.w$ are two execution sequences, then $E \models p\Diamond w$ denotes that $E.p.w$ is also an execution sequence such that $next_{[E]}(p) \not\rightarrow_{E.p.w} e$ for any $e \in dom_{[E.p]}(w)$. To elaborate, $E \models p\Diamond w$ means that the next event of p would not "happen before" any event in w in the execution sequence $E.p.w$. Simply, this notation states that p is independent with w after E. In the special case when w contains only one process q, then $E \models p\Diamond q$ denotes that the next steps of p and q are independent after E. We use $E \not\models p\Diamond w$ to denote that $E \models p\Diamond w$ does not hold.

We use the notation $w\backslash p$, where w is a sequence and $p \in w$, to denote the sequence w with its first occurrence of p removed, and $w \upharpoonright p$ to denote the prefix of w up to but not including the first occurrence of p. Considering an execution sequence E and an event $e \in dom(E)$, we use $pre(E, e)$ to denote the prefix of E up to, but not including, the event e. We also use the notation $notdep(e, E)$ to refer to the sub-sequence of E consisting of the events that occur after e but do not "happen after" e (i.e., the events e' that occur after e such that $e \not\rightarrow_E e'$).

Intuitively, two events, e and e' in an execution sequence E, where e occurs before e' in E, are in a race if

- e happens-before e' in E, and
- e and e' are "concurrent", i.e., there is an equivalent execution sequence $E' \simeq E$ in which e and e' are adjacent.

We use the notation $e \lessdot_E e'$ to denote that events e and e' are in a race, or more formally, that $proc(e) \neq proc(e')$, that $e \rightarrow_E e'$, and that there is no event $e'' \in dom(E)$, different from e' and e, such that $e \rightarrow_E e'' \rightarrow_E e'$.

Let $e \lesssim_E e'$ denote that $e \lessdot_E e'$, and that the race can be reversed. Formally, if $E' \lesssim E$ and e occurs immediately before e' in E', then $proc(e')$ was not blocked before the occurrence of e. This concept is useful, because whenever a DPOR algorithm detects a race, it will check whether the events in the race can be executed in the reverse order. Since the events are related by the happens-before relation, this may lead to a different global state and therefore the algorithm must try to explore a corresponding execution sequence.

2.4 Concuerror Overview

Concuerror [Chri13, Goto11] 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. Specifically, Concuerror navigates the state space of a program, under a given test suite with a specified input, to check whether certain errors occur in specific interleavings or verify the absence of any errors. Such errors include abnormal process exits, uncaught exceptions, assertion violations and deadlocks. Concuerror's functionality can be mainly described through its main components: the Instrumenter, the Scheduler and the Logger.

2.4.1 Instrumenter

Concuerror instruments the code of a program without having to make modification to the Erlang VM. Instead, it utilizes a source to source translation that adds preemptions points to various points in the code of a program. When the execution of a program reaches a preemption point, the process will yield its execution by blocking on a receive statement, until a continuation message is sent from the Scheduler.

This makes it possible to control how the processes of a program are scheduled and therefore, recreate a specific interleaving. Moreover, this allows for the modification of specific BIFs that interact with the global state of a program, by inserting a preemption point before such function calls and controlling their execution.

2.4.2 Scheduler

In order to explore the complete state space of a concurrent program, it is vital that we are able to "force" specific schedulings (interleavings) of its processes. The Scheduler is responsible for controlling the execution of the processes to produce the required interleavings and at the same time check for and handle possible errors that may occur.

The scheduler is also responsible for determining which interleavings are to be checked. This is done by implementing various DPOR algorithms (persistent-DPOR, source-DPOR, optimal-DPOR). The default algorithm currently used by Concuerror is optimal-DPOR. However, the user can specify the technique that Concuerror will use to search the state space, through the --*dpor* option.

The functionality of the Scheduler can be divided into two main parts: *the exploration phase* and the *planning phase* (in accordance to most DPOR algorithms, as described in Chapter 3). The planning phase is responsible for determining which interleavings need to be explored and the exploration phase is responsible for producing those interleavings.

2.4.3 Logger

Testing programs is useless without providing the user with information on how an error was produced. This is the responsibility of the Logger. During its execution, the Scheduler logs information regarding the explored interleavings. The Logger is responsible for compiling that information in order to print the trace of a scheduling that led to a potential error. At the same time, when used in developer mode, the Logger is essential for providing debugging information to Concuerror developers.

Κεφάλαιο 3

Dynamic Partial Order Reduction

3.1 Basic DPOR Concepts

Generally, DPOR algorithms use a depth-first backtrack search to explore the state-space of a concurrent system. This exploration is driven by two basic concepts: *persistent sets* and *sleep sets*, which make sure to explore a sufficient portion (at least one interleaving from different Mazurkiewicz traces) of the state-space, while trying to minimize any unnecessary exploration.

Intuitively, a persistent set at a state s is (specific) subset of $enabled(s)$ whose exploration guarantees that all non-equivalent interleavings (from different Mazurkiewicz traces) will be explored. This is vital in proving the correctness of Classic DPOR algorithms [Flan05], on the assumption (that is taken into account by our abstraction) that our state space is acyclic and finite. The way that such sets are constructed differs from paper to paper [Flan05, Lei06, Valm91], and those variations can lead to different degrees of state-space reduction.

The sleep set technique, being complimentary to persistent sets (it does not contribute to the soundness of algorithm), aims to further reduce the number of the explored interleavings. A sleep set at an execution sequence E contains processes, whose exploration would be redundant, preventing equivalent interleavings from being explored.

Specifically, after $E.p$ has been explored, process p is added to the sleep set at E. From this point on, p will exist in any sleep set of an execution sequence of the form $E.w$, provided that $E.w$ is also an execution sequence and $E \models p \Diamond w$. The processes in the sleep set are not going to be executed from this point on, unless a dependency gets detected. For instance, p will be removed from the sleep set at $E.w.q$ if a dependency gets detected between the next steps of q and p.

It can be proved [Gode96] that sleep sets will eventually block all the redundant interleavings and thus the only interleavings that are going to be explored fully will belong to different Mazurkiewicz traces. However, this does not mean sleep sets avoid all redundant explorations. To elaborate, sleep sets make it possible for the exploration of an interleaving, which belongs to the same Mazurkiewicz trace as another interleaving that has already been explored, to eventually block, by having all of its enabled processes appear in the sleep set. This is called *sleep-set blocking* and it means that all possible traces, from this point on, are redundant and therefore, need not be explored further. Sleep sets do not guarantee optimality for a DPOR algorithm, since redundant traces get explored, albeit not completely.

Source-DPOR [Abdu14] replaces persistent sets with *source sets* in order to achieve a significantly better reduction in the amount of the explored interleavings. However, source-DPOR still suffers from sleep-set blocking. Optimal-DPOR [Abdu14] combines the concept of source sets with *wakeup trees* to fully avoid sleep-set blocking and lead to the exploration of an optimal subset of interleavings.

3.2 Source Sets

Before defining source sets formally, we need to define the concepts of possible initial steps in an execution sequence [Abdu14]:

Definition 3.1. (Initials after an execution sequence $E.w$, $I_{[E]}(w)$)
For an execution sequence $E.w$, let $I_{[E]}(w)$ denote the set of processes that perform events e in $dom_{[E]}(w)$ that have no "happens-before" predecessors in $dom_{[E]}(w)$. More formally, $p \in I_{[E]}(w)$ if $p \in w$ and there is no other event $e \in dom_{[E]}(w)$ with $e \rightarrow_{E.w} next_{[E]}(p)$.

By relaxing this definition, we can get the definition of Weak Initials, WI:

Definition 3.2. (Weak Initials after an execution sequence $E.w$, $WI_{[E]}(w)$)
For an execution sequence $E.w$, let $WI_{[E]}(w)$ denote the union of $I_{[E]}(w)$ and the set of processes that perform events p such that $p \in enabled(s_{[E]})$.

To clarify these notations, for an execution sequence $E.w$:

- $p \in I_{[E]}(w)$ iff there is a sequence w' such that $E.w \simeq E.p.w'$, and
- $p \in WI_{[E]}(w)$ iff there are sequences w' and v such that $E.w.v \simeq E.p.w'$.

Definition 3.3. (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 $WI_{[E]}(w) \cap P = \emptyset$.

A direct consequence of this definition is that every set of processes that can cover the complete state space after an execution sequence E can be considered a source set of E.

3.3 Source-DPOR

Here we present the source-DPOR algorithm [Abdu14].

Algorithm 1: Source-DPOR

```
Function Explore(E,Sleep)
    if $\exists p \in (enabled(s_{[E]}) \backslash Sleep)$ then
        $backtrack(E) := p$;
        while $\exists p \in (backtrack(E) \backslash Sleep)$ do
            foreach $e \in dom(E)$ such that $e \precsim_{E.p} next_{[E]}(p)$ do
                let $E' = pre(E, e)$;
                let $u = notdep(e, E).p$;
                if $I_{[E']}(u) \cap backtrack(E') = \emptyset$ then
                    add some $q' \in I_{[E']}(u)$ to $backtrack(E')$;
            let $Sleep' := \{q \in Sleep \mid E \models p \diamondsuit q\}$;
            $Explore(E.p, Sleep')$;
            add $p$ to $Sleep$;
```

An execution step $Explore(E, Sleep)$ is responsible for the explorations of all Mazurkiewicz traces that begin with the prefix E. These explorations start by initializing $backtrack(E)$ with an arbitrary enabled process that is not in the sleep set ($Sleep$). From this point forward, for each process p that exists in $backtrack(E)$ source-DPOR will perform two main phases.

During the first phase (race detection), we find all events e that occur in E (i.e., $e \in dom(E)$) which are racing with the next event of p, and that race can be reversed ($e \lesssim_{E.p} next_{[E]}(p)$). For each such event e, we are trying to reverse that race by ensuring that the next event of p gets performed before e, or using the symbolism of the algorithm, that a sequence equivalent to $E.notdep(e, E).p.proc(e).z$ (z is any continuation of the execution sequence) is explored. Such a trace could be explored by taking the next available step from any process in $I_{[E']}(notdep(e, E).p)$ (where $E' = pre(E, e)$) at E'. Therefore, a process from $I_{[E']}(notdep(e, E).p)$ is added to the backtrack set at E', provided that it is not already there.

During the last phase (exploration), we recursively explore $E.p$. The sleep set at $E.p$ is initialized appropriately by taking the sleep set at E and removing all processes whose next step is dependent with the next step of p. This ensures that at $E.p$ we will not consider the races of processes that have already been considered, unless they race with the new scheduled process p. After $E.p$ finished its exploration, p is added to the sleep set at E, because we want to refrain from executing an equivalent trace.

Practically, on Concuerror the algorithm is structured differently. The main difference is that the algorithm goes into the race detection phase when an interleaving has reached its end. At this point all the races that occurred in the interleaving are detected and backtrack points are inserted in the appropriate prefixes of the complete execution sequence. Then the exploration continues by exploring the backtrack set of the longest prefix first. This does not affect the soundness of the algorithm since the only thing that changes is the order in which new interleavings are explored.

We should note here that $Explore(E, Sleep)$ does not need any additional information from the various prefixes of E that has not already been established. However, $Explore(E, Sleep)$ can add backtrack points to the various prefixes of E. This is vital and must be taken into consideration while trying to parallelize source-DPOR.

3.4 Wakeup Trees

In order to achieve optimality, we must completely avoid having sleep-set blocked interleavings. This is achieved by combining a mechanism called wakeup trees [Abdu14] with source sets.

Notice that in source-DPOR a sequence of the form $E'.notdep(e, E).p.proc(e).z$ needs to be explored, but only a single process from the Initials set of $notdep(e, E).p$ is potentially added to the backtrack set. Therefore, a piece of information on how to reverse the race gets lost. This may lead to sleep-set blocking, since an alternative sequence could be explored instead. Intuitively, wakeup trees hold in the form of a tree the fragments that need to be explored in order to explore the necessary interleavings, while avoid sleep-set blocking.

In order to define wakeup trees, we first present the generalizations of the concepts of Initials and Weak Initials so they can contain sequences of processes instead of just processes:

- $v \sqsubseteq_{[E]} w$ denotes that exists a sequence v' such that $E.v.v'$ and $E.w$ are execution sequences with the relation $E.v.v' \simeq E.w$. What this means is that after E, v is a possible way to start an execution that is equivalent to w. To connect this to the concept of Initials we have $p \in I_{[E]}(w)$ iff $p \sqsubseteq_{[E]} w$.

- $v \sim_{[E]} w$ denotes that exist sequences v' and w' such that $E.v.v'$ and $E.w.w'$ are execution sequences with the relation $E.v.v' \simeq E.w.w'$. What this means is that after

E, v is a possible way to start an execution that is equivalent to $E.w.w'$. To connect this to the concept of Weak Initials we have $p \in WI_{[E]}(w)$ iff $p \sim_{[E]} w$.

Definition 3.4. (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 3.5. (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.

3.4.1 Operations on Wakeup Trees

An important operation used by the optimal-DPOR algorithm is the insertion of new initial fragments of interleavings, which need to be explored, into the wakeup tree.

Considering a wakeup tree $\langle B, \prec \rangle$ after $\langle E, P \rangle$ and some sequence w with $E.w$ being an execution sequence such that $WI_{[E]}(w) \cap P = \emptyset$, the following properties are used to define the $insert_{[E]}(w, \langle B, \prec \rangle)$:

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

Let v be the smallest (according to the $\prec$ order) in B with $v \sim_{[E]} w$. The operation $insert_{[E]}(w, \langle B, \prec \rangle)$ can either be taken as $\langle B, \prec \rangle$, provided that v is a leaf, or by adding $v.w'$ as a leaf and ordering it after all existing nodes in B of form $v.w''$, where w' is the shortest sequence with $w \sqsubseteq_{[E]} v.w'$.

Let us also describe the $subtree(\langle B, \prec \rangle, p)$ operation. For a wakeup tree $\langle B, \prec \rangle$ and a process $p \in B$, $subtree(\langle B, \prec \rangle), p)$ is used to denote the subtree of $\langle B, \prec \rangle$ rooted at p. More formally, $subtree(\langle B, \prec \rangle), p) = \langle B', \prec' \rangle$ where $B' = \{w \mid p.w \in B\}$ and $\prec'$ is the extension of $\prec$ to B'

3.5 Optimal-DPOR

Here the optimal-DPOR algorithm is presented as described in its original paper [Abdu14].

Similarly, with the other algorithms, optimal-DPOR has two different phases: race detection and state exploration. However, the algorithm is structured differently. In the same way that source-DPOR is structured at Concuerror, optimal-DPOR only detects the races when a maximal execution sequence has been reached (i.e., there exist no enabled processes). This is necessary because the condition for inserting new wakeup trees is only valid when the fragment that is going to be inserted contains all the events in the complete executions that do not happen after e and those that occur after e.

Algorithm 2: 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
                $insert_{[E']}(v, wut(E'))$;
    else
        if $WuT \neq \langle\{\langle\rangle\}, \emptyset\rangle$ then
            $wut(E) := WuT$;
        else
            choose $p \in enabled(s_{[E]})$;
            $wut(E) := \langle\{p\}, \emptyset\rangle$;
        $sleep(E) := Sleep$;
        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 $wut(E)$;
```

The race detection phase works mostly similarly with source-DPOR. The main differences have to do with the fact that we require the knowledge of the sleep set for every prefix E' and that the concepts of Weak Initials is used instead of the Initials to determine whether a fragment is going to be inserted at the wakeup tree, which is rooted at the prefix E'.

In the exploration phase of a non-maximal execution sequence, the wakeup tree of that sequence is initialized to the given WuT. If WuT is empty, then an arbitrary enabled process is chosen, in the same way that it would for the non-optimal algorithms. Afterwards, for every process that exists in WuT the explore function is going to be called recursively, with the appropriate subtree of $wut(E)$. This guarantees that the complete fragment gets explored. After the recursive call finishes, the sequences that were explored are removed from the wakeup tree. Sleep sets are handled in a similar way with previous algorithms.

As the name suggests, optimal-DPOR is optimal in the sense that it never explores two maximal execution sequences that belong to the same Mazurkiewicz trace, since it can be proven that no interleaving is sleep-set blocked [Abdu14].

Κεφάλαιο 4

Parallelizing Source-DPOR Algorithm

Concuerror utilizes primarily DPOR algorithms to systematically test concurrent Erlang programs. Therefore, parallelizing Concuerror entails designing parallel versions for its DPOR algorithms. Since Concuerror is written in Erlang, which is a functional language that is based on message passing to share data between processes, we are going to follow a message passing approach, while developing our algorithms.

In this chapter, we are going to present the parallel version of the source-DPOR algorithm. Let us first discuss some existing work in parallelizing persistent-set based DPOR algorithms.

4.1 Existing Work

When parallelizing DPOR algorithms, ideally we would like to develop parallel algorithms that explore exactly the same number of interleavings as their sequential versions. In other words, we want to retain the soundness of our algorithms, while simultaneously not exploring more interleavings than necessary

At a first glance, parallelizing DPOR algorithms may seem straightforward. Since the state space of a program contains no cycles, we should simply distribute that state space into multiple workers-schedulers. For example, in the case of Algorithm 1, we would assign a prefix of the form $E.p$ to a scheduler. That scheduler would be responsible for the execution of $explore(E.p, Sleep)$. However this approach leads to two main issues [Yang07].

Firstly, DPOR algorithms detect races and update the exploration frontier in a non-local manner. While calls to $explore(E.p, Sleep)$ may guarantee that for all maximal execution sequences of form $E.w$, the algorithm has explored some execution sequence E' which is in $[E.w]_{\simeq}$, backtrack points may also be inserted in the prefixes of $E.p$. For instance, lets assume that we assign to one scheduler the exploration of $E.p$ and to another scheduler the exploration of $E.q$. Since we are using message passing as our programming model, our schedulers would use different copies of the prefix E. Both of those explorations may lead to adding the same process r to the backtrack at E. This would mean that both those schedulers would end up calling $explore(E.r, Sleep)$. Therefore, different schedulers may end up fully exploring identical interleavings. In order to combat those redundant explorations, Yang et al. [Yang07] suggest a heuristic that simply modifies the lazy addition of backtrack entries to the exploration frontier [Flan05] to become more eager. Adding backtrack entries more eagerly, i.e., earlier in the exploration phase, reduces the chances of two different workers exploring identical interleavings. In the above scenario, this strategy could have led to r being added to the backtrack at E before assigning the exploration of $E.p$ and $E.q$ to different schedulers. Therefore, both of our schedulers would have know that r exists in the backtrack of E and none of them would have added it, avoiding the duplicate exploration of $E.r$. However, this is simply a heuristic, which means that depending on the tested program, a significant amount of redundant computations may still occur. Particularly, this heuristic fails to prevent redundant explorations, when branches in the code lead to different races.

Secondly, the size of different chunks of the state space cannot be known a priori. This means that some form of load balancing is essential to achieve linear speedup. Yang et al. [Yang07] suggest using a centralized load balancer to unload work from a scheduler. Specifically, a scheduler calls the load balancer when the total number of backtrack entries in the execution sequence of the scheduler exceed a threshold. However, for different programs and different number of workers, different threshold values should be used [Sims12]. Still, Yang et al. provide no insight into the problem of selecting an appropriate threshold.

Simsa et al. [Sims12] provide a more appropriate way to solve these issues. By using a centralized $Controller$, which keeps track of the current $execution\ tree$ (a tree whose branches correspond to the current execution sequences E of the schedulers), they assure that no redundant explorations occur. They also suggests the use of $time\ slicing$ to achieve load balancing.

4.2 Parallel source-DPOR

Here we are going to present how to efficiently parallelize the source-DPOR algorithm, by modifying the parallel algorithm presented by Simsa et al. [Sims12].

4.2.1 Basic Idea

Normally, DPOR algorithms perform a depth-first search of the state space to check for erroneous interleavings. Instead, we are going to use multiple depth-first searches (by partitioning the frontier of our search) to explore our state space.

We are going to use a centralized $Controller$ who will be responsible for distributing the exploration frontier to different worker-schedulers. The Controller is also going to oversee the parallel exploration, so we avoid exploring more interleavings than the sequential version.

In order to avoid redundant explorations, we are going to use the concept of $ownership$ [Sims12] of a node of a state space. A scheduler exclusively owns a node of the state space if it is either contained as a backtrack entry within the part of the frontier that was assigned to that specific scheduler, or if it is a descendant of a node that the scheduler owns. All other nodes, are considered to have a $disputed$ ownership.

The scheduler, when conducting its depth-first search, is going to be allowed to explore only nodes that it owns. When encountering disputed nodes, the scheduler will report back to the Controller, which will keep track of the complete active frontier. If that disputed node does not exist within the complete frontier, then no other scheduler has explored that node and therefore, the scheduler can $claim\ ownership$ over the disputed node and continue with exploring it. Otherwise, the ownership of the disputed node has been claimed by some other scheduler and therefore, that node can be discarded from the frontier of our scheduler.

For example, if a scheduler is responsible for exploring an execution sequence E that has a single backtrack entry of p at E, then that scheduler owns every node that is a descendant of $E.p$ i.e., it owns every execution sequence that begins with the prefix $E.p$ (or else, the complete subtree that is rooted at $E.p$). Now lets assume that during the exploration of the subtree that is rooted at $E.p$, r has been added at the backtrack at E. Since this node is disputed, the scheduler will not explore this backtrack entry. Instead, it will report back to the Controller, in order to determine whether some other entry has already claimed ownership over $E.r$.

4.2.2 Algorithm

Algorithm 3: Controller Loop

```
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 $Frontier$;
            $spawn(S, explore\_loop(E_c, Budget))$;
        $Frontier, T \leftarrow wait\_scheduler\_response(Frontier, T)$;
```

The logic of the Controller is shown in Algorithm 3. The Controller maintains a $Frontier$, which is a set of execution sequences E, and an execution tree T, which contains as branches the execution sequences of the $Frontier$. For as long as there exists an execution sequence at the $Frontier$ ($Frontier \neq \emptyset$), the Controller will partition its $Frontier$ to at most N execution sequences. Then, the Controller will try to assign all of its unassigned execution sequences to any idle scheduler, by spawning $explore\_loop(E_c, Budget)$ functions. Finally, it will block until it receives a response from a scheduler.

Algorithm 4: Frontier Partitioning

```
Function $partition(Frontier, N)$
    for all $E \in Frontier$ 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 $Frontier$;
    return $Frontier$;
```

During the *partitioning* phase (Algorithm 4), we inspect the current $Frontier$ to determine whether we should create additional execution sequences. Every execution sequence that contains more than one backtrack entry is split into multiple sequences until either the $Frontier$ contains N sequences or all sequences have exactly one backtrack entry. It is vital to modify sleep sets appropriately because, if we were to simply remove backtrack entries, our algorithm would have an increased amount of sleep-set blocked interleavings. In addition, we would also end up potentially re-adding the same backtrack entries, which would lead to exploring duplicate interleavings.

Algorithm 5 details how the schedulers explore their assigned state space. A call to $explore\_loop(E_0, Budget)$ guarantees that for all maximal execution sequences of form $E_0.w$, the algorithm has explored some execution sequence E'_0 which is in $[E_0.w]_{\simeq}$ We

Algorithm 5: Scheduler Exploration Loop

```
Function explore_loop($E_0$, Budget)
    $StartTime \leftarrow get\_time()$;
    $E \leftarrow E_0$;
    repeat
        $E' \leftarrow explore(E)$;
        $plan\_more\_interleavings(E')$;
        $E'' \leftarrow get\_next\_execution\_sequence(E')$;
        $CurrentTime \leftarrow get\_time()$;
    until $CurrentTime - StartTime > Budget$ or $size(E'') \leq size(E_0)$;
    send $E''$ to Controller ;
```

use $explore(E)$ and $plan\_more\_interleavings(E')$ as a high level way to describe the main phases (state exploration and race detection) of the sequential source-DPOR. The $plan\_more\_interleavings(E')$ function could add backtrack points in prefixes of E_0. This could lead to different schedulers exploring identical interleavings. We avoid this by having the function $get\_next\_execution\_sequence(E')$ return the largest prefix of E' that has a non-empty backtrack set. This leads to a depth-first exploration of the assigned state space before considering interleavings outside of the state space. The exploration continues until we encounter a prefix of E_0 ($size(E'') \leq size(E_0)$). This is necessary to assert that the specific scheduler will not explore interleavings outside of its state space. When the exploration terminates, the backtrack points added to the prefixes of E_0 will be reported back to the Controller.

Algorithm 6: Handling Scheduler Response

```
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 $Frontier$;
    return $Frontier, T$;
```

When the Controller receives a response (an execution sequence E) from a scheduler (Algorithm 6), it will try and report any new backtrack entries in E to the execution tree T. This is done through the $update\_execution\_tree(E, T)$ function. This function iterates over the execution sequence and the execution tree simultaneously and those backtrack entries of E that are not found in the execution tree, are added to it and they are not removed from the execution sequence. This means that this execution sequence is the first to *claim ownership* over those entries and the state space that exists under them. Any backtrack entries that already exist in T are removed from the execution sequence E (and added to the sleep sets at the appropriate prefixes of E), because some other execution sequence has already claimed their ownership. This updated execution sequence is then added to the $Frontier$ of the Controller. Through this we make sure that two schedulers cannot explore identical interleavings.

When updating the execution tree, we also use the initial execution sequence that was assigned to a scheduler (the one denoted as E_0 at Algorithm 5) to figure out which parts of the execution tree have already been explored. Those parts are deleted from the execution

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

Σχήμα 4.1: Simple readers-writes example

tree. This is mandatory in order to keep the size of the execution tree proportionate to the size of our current $Frontier$.

4.2.3 Load Balancing

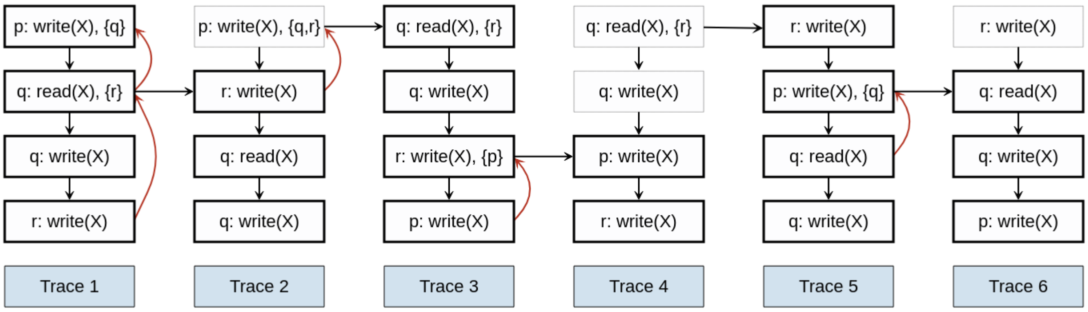
In order to achieve decent speedups and scalability it is necessary to have load-balancing [Sims12]. This done through time-slicing the exploration of execution sequences. This is the reason behind the use of $Budget$ in Algorithm 5. By having schedulers return after a certain time-slice, we can make sure that even if their assigned state space was larger compared to that of other schedulers, they will eventually exit and have their execution sequence and subsequently, their state space, partitioned. How effective is this method is determined by two variables, the upper limit N to the number of execution sequences in our $Frontier$ and the $Budget$ of a scheduler.

Higher upper limit N means that a larger work pool is available to the workers and they do not always have to wait for the $update\_execution\_tree(E, T)$ function to terminate. Therefore, the utilization of the schedulers increases. However, a larger N means increased memory requirements, since more execution sequences are active at each time. More importantly, it also means that the state space splits into smaller fragments. This increases how frequently a scheduler discovers disputed nodes, which leads to an increase in the communication between the schedulers and the controller. Setting this limit to double the amount of schedulers, produces decent results for most test cases[Sims12].

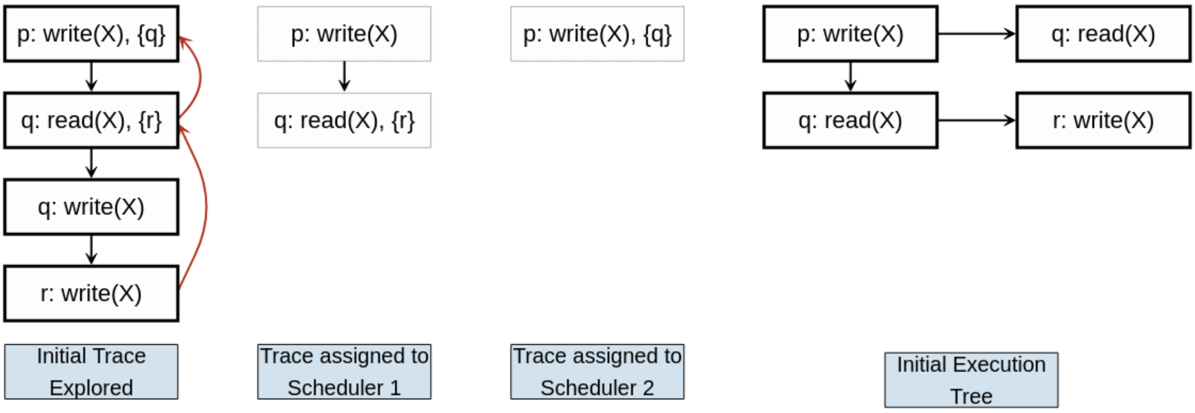
Smaller $Budget$ values lead to more balanced workload, since the work is distributed more frequently. However, extremely low values may lead to an increased communication overhead between the Controller and the schedulers. This can also cause the Controller to become a significant bottleneck. The best way to deal with this, is to pick an initial value $Budget$ of around 1-10 seconds. When a scheduler starts a new exploration, the value of its budget will be dynamically assigned by the Controller depending on the amount of idle schedulers. For instance, the first execution should have a budget of $\frac{Budget}{n}$ were n is the total amount of schedulers, which are all idle. When half the schedulers are idle, this value should be $\frac{Budget}{2}$, etc. This makes it possible to have reduced communication (higher budget) during periods with many busy schedulers and a better balancing (lower budget) during periods with many idle schedulers.

4.2.4 A Simple Example

Let us consider the example in Figure 4.1. In this case we have 3 processes that write and read a shared variable x. Figure 4.2 represents the traces explored during the sequential source-DPOR. We use a bold rectangle to represent a new event and a faint rectangle to denote a replayed event. The red edges represent the races that are detected and planned. The source set at a state is represented inside the brackets.

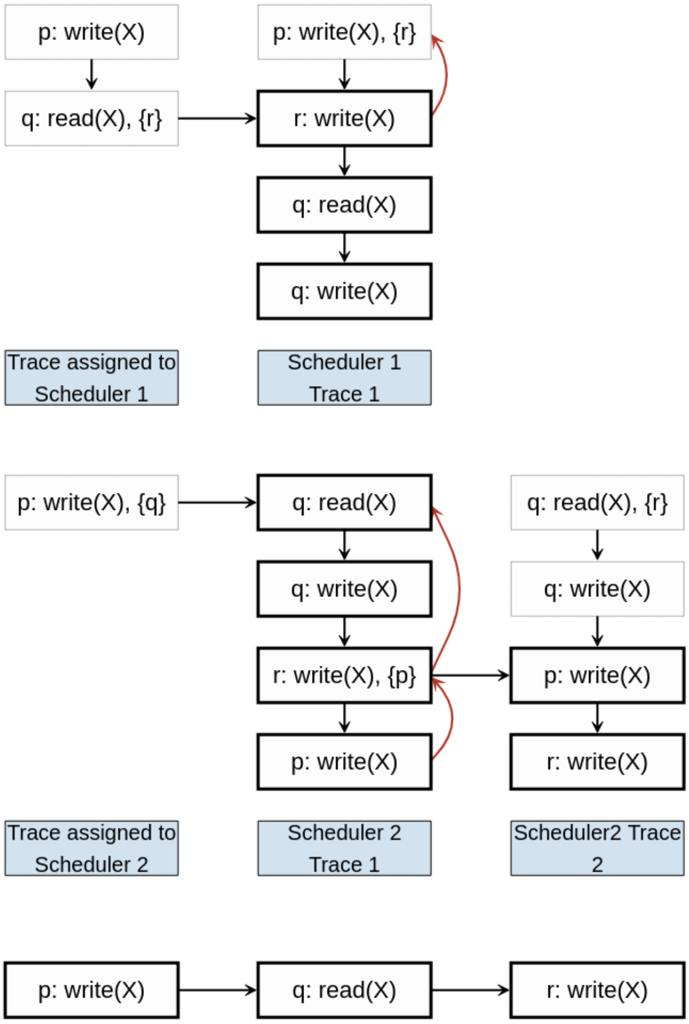

p: write(X), {q}
q: read(X), {r}
q: write(X)
r: write(X)
Trace 1
p: write(X), {q,r}
r: write(X)
q: read(X)
q: write(X)
Trace 2
q: read(X), {r}
q: write(X)
r: write(X), {p}
p: write(X)
Trace 3
q: read(X), {r}
q: write(X)
p: write(X)
r: write(X)
Trace 4
r: write(X)
p: write(X), {q}
q: read(X)
q: write(X)
Trace 5
r: write(X)
q: read(X)
q: write(X)
p: write(X)
Trace 6

Σχήμα 4.2: Interleavings explored by the sequential source-DPOR


p: write(X), {q}
q: read(X), {r}
q: write(X)
r: write(X)
Initial Trace Explored
p: write(X)
q: read(X), {r}
Trace assigned to Scheduler 1
p: write(X), {q}
Trace assigned to Scheduler 2
p: write(X)
q: read(X)
q: read(X)
r: write(X)
Initial Execution Tree

Σχήμα 4.3: Initial interleaving explored by the parallel algorithm

Figure 4.3 depicts the initial step of the parallel source-DPOR. This initial execution sequence, along with detected races is partitioned into fragments which get assigned to different schedulers. This image also contains the initial execution tree which represents the state space that exists in our exploration frontier at this point.



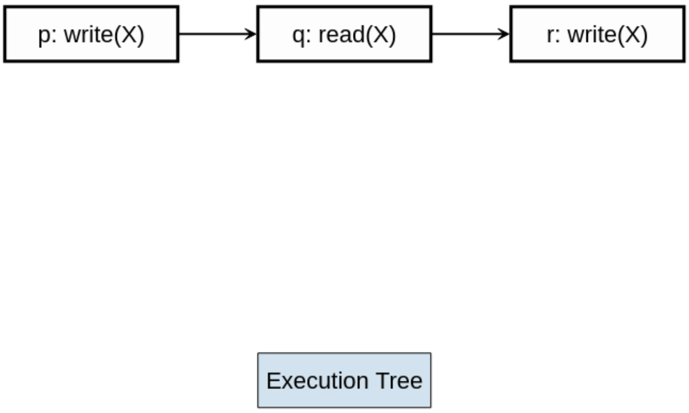
p: write(X)
q: read(X), {r}
p: write(X), {r}
r: write(X)
q: read(X)
q: write(X)
Trace assigned to Scheduler 1
Scheduler 1 Trace 1
p: write(X), {q}
q: read(X)
q: write(X)
r: write(X), {p}
p: write(X)
q: read(X), {r}
q: write(X)
p: write(X)
r: write(X)
Trace assigned to Scheduler 2
Scheduler 2 Trace 1
Scheduler2 Trace 2
p: write(X)
q: read(X)
r: write(X)

Σχήμα 4.4: Exploration of the assigned traces by each scheduler

In Figure 4.4 we illustrate how each scheduler explores its assigned execution sequence. The first scheduler explores its first trace: $p.r.q.q$ (equivalent to the second trace at Figure 4.2). After the race detection takes place, there are no backtrack entries added below its assigned trace. The only execution sequence planned is the sequence r. However, this sequence is not explored since it does not belong to the state space of the scheduler. The Controller assigned to the second scheduler the execution sequence q. After exploring its first trace, two more races are detected. It is important to notice here that the first trace of the second scheduler is equivalent to the third trace of the sequential algorithm. However, the sequential algorithm detects only one race. This happens because on the sequential algorithm the race between $q : read(x)$ and $r : write(x)$ had already been detected at the second interleaving and so its planning gets skipped. On the contrary, on the parallel algorithm this interleaving is explored by another scheduler and therefore, there is no knowledge of this race been already detected. This leads to both our schedulers having detected the same race. Nevertheless, in both schedulers this new backtrack entry is outside of their state space.

This means that they will have to report their results back to the Controller. The schedulers that reports first its results, will be the one to update the execution tree by inserting the new entry found. This scheduler will add its execution sequence to the frontier, which will be again partitioned (no need for a partition here since the unexplored frontier will only have one race). Then this execution sequence will be assigned to an idle scheduler. The scheduler that reports second to the Controller, will not be able to insert its backtrack entry into the execution tree, because that entry will already be there, and the backtrack entry gets removed from its execution sequence. This execution sequence will be left with no more backtrack entries and as such it will not be inserted into the frontier. This guarantees that we do not explore identical interleavings more than once.

Lets assume that Scheduler 1 was the one that managed to report first to the Controller. Then the execution tree will be the one depicted in Figure 4.5. Notice here that the states explored by the Scheduler 1 were deleted from the execution tree. This keeps the size of the execution tree proportional to the size of the current exploration frontier. At this point Scheduler 1 will add its execution sequence to the frontier. Then the Controller will assign this sequence to Scheduler 1 (since Scheduler 2 has not yet returned), and Scheduler 1 will explore the last 2 traces (trace 5 and 6 from the sequential example). After both Scheduler 1 and 2 have returned the execution will have finished since there will no more traces left to explore.



Σχήμα 4.5: Execution Tree if Scheduler 1 returned first

Κεφάλαιο 5

Parallelizing Optimal-DPOR Algorithm

In this chapter, we are going to present a first attempt in parallelizing the optimal-DPOR algorithm, provide an analysis on why this attempt fails to achieve any speedup and then present an improved parallel optimal-DPOR algorithm.

5.1 Parallel optimal-DPOR - A First Attempt

5.1.1 Basic Idea

Parallelizing the optimal-DPOR algorithm is significantly more complicated than source-DPOR. We do know that whenever a call to $Explore(E, Sleep, WuT)$ returns during Algorithm 2, then for all maximal execution sequences of form $E.w$, the algorithm has explored some execution sequence E' which is in $[E.w]_{\simeq}$ [Abdu14]. However, the complete WuT at some execution sequence E cannot be known until we have completed exploring all execution sequences which are ordered before E, according to the total order of our state space (Definition 3.4). This happens because the $insert_{[E']}(v, wut(E'))$ function can add entries to any wakeup tree of an execution sequence that is ordered after the current execution sequence.

Therefore, when assigning an incomplete wakeup tree to a scheduler, there is no guarantee that the scheduler will explore the complete assigned state space. This means that if a scheduler inserts a fragment into a wakeup tree owned by a different scheduler, we cannot know if that fragment (or a different but equivalent fragment) was indeed explored. As a result, the concept of the ownership of a backtrack entry, as defined in Chapter 4, cannot remain the same for the optimal-DPOR algorithm.

Another main issue with optimal-DPOR, is the fact that the $insert_{[E']}(v, wut(E'))$ function may end up rearranging the sequence of the events within v. This means that two execution sequences, while different, may be equivalent. This can potentially lead to different schedulers inserting in their wakeup trees execution sequences that while different, may produce equivalent interleavings and therefore, the optimality of the algorithm could be lost.

However, we do know that any leaf of $\langle B, \prec \rangle$ remains a leaf of $insert_{[E]}(w, \langle B, \prec \rangle)$ [Abdu14]. This means that, during the sequential algorithm, any fragment that is inserted into a wakeup tree is a fragment that must be explored, unless it is removed during the exploration phase of the algorithm. Therefore, when we insert a fragment into a wakeup tree, we can explore it out of order, as long as we are careful to remove execution sequences that would have been removed on the sequential algorithm. We can take advantage of this to create an algorithm that can explore many interleavings in parallel but race detects each explored interleaving sequentially.

5.1.2 Algorithm

Due to the fact that we need to have parallel exploration of interleavings but sequential planning, we need to decouple the normal exploration loop of a scheduler into two different parts: *state exploration* and *race detection−planning*. Our workers (the schedulers) will be responsible for the first part. For the second part, we are going to use a centralized Planner. However, in order to be able to better distribute the available work to the schedulers when the Planner is busy, we are also going to use a Controller.

Algorithm 7: Controller for Optimal-DPOR - First Attempt

```
Function controller_loop(Schedulers)
    $E_0 \leftarrow$ an arbitrary initial execution sequence;
    $Frontier \leftarrow [E_0]$;
    $T \leftarrow$ an execution tree rooted at $E_0$;
    $PlannerQueue \leftarrow empty$;
    while $size(Frontier) > 0$ and $size(PlannerQueue) > 0$ do
        $partition(Frontier)$;
        while exists an idle scheduler $S$ and an unassigned execution sequence $E$ in $Frontier$ do
            $E_c \leftarrow$ a copy of $E$;
            $spawn(S, explore(E_c))$;
        while the Planner is idle and PlannerQueue $\neq$ empty do
            $E \leftarrow PlannerQueue.pop()$;
            $update\_trace(E, T)$;
            $spawn(Planner, plan(E))$;
        $wait\_response(Frontier, T, PlannerQueue)$;
```

Algorithm 7 describes the functionality of the Controller. Similarly to the source-DPOR parallel version, the Controller is responsible for maintaining the current Frontier (as well as partitioning it) and the current Execution Tree and for assigning execution sequences to schedulers, for as long we have idle schedulers and available work.

Apart from that, the Controller also maintains a queue of fully explored execution sequences that need to be race detected. When the Planner is idle and the queue is not empty, the execution sequence is updated (through $update\_trace(E, T)$) and then is sent to the Planner so its races can be detected. When updating the execution sequence from the execution tree, the subtrees of the execution tree which are ordered after the execution sequence (according to the ordering of our state space Definition 3.4) are inserted into the execution tree as *not_owned* wakeup trees. This guarantees that no redundant fragments are going to be inserted for future explorations and therefore, the algorithm remains optimal.

The $plan(E)$ function race detects the fully explored execution sequence E according to the logic of optimal-DPOR (Algorithm 2). When the planning of the sequence is finished the results are reported back to the Controller. The $explore(E)$ function explores the execution sequence E until a maximal execution sequence has been reached and reports back that execution sequence to the Controller.

Partitioning the exploration frontier (Algorithm 8) has two main differences, compared to the parallel source-DPOR. Firstly, the frontier gets partitioned completely, so we can maximize the parallelization of the exploration phase. Secondly, the entries that are distributed from one execution sequence, are not simply removed from the backtrack and added to the sleep set. It is vital here to maintain the correct ordering between the interleavings (Definition 3.4), because during the exploration phase of the optimal-DPOR algorithm, branches can be removed from the wakeup tree. Maintaining the ordering will keep this behavior intact in the parallel version. Therefore, the given entry is marked as *not_owned* at the distributed sequence. The function $split\_wut\_at(v, wut(E'_c))$ splits the copy of the

Algorithm 8: Optimal Frontier Partitioning

```
Function partition(Frontier)
    for all $E \in Frontier$ do
        while $wakeup\_tree\_leaves(E) > 1$ do
            $E' \leftarrow$ a prefix of $E$ with $wut(E') \neq \emptyset$ ;
            $v \leftarrow$ a leaf $\in wut(E')$;
            $E'_c \leftarrow$ a copy of $E'$;
            mark $v$ as $not\_owned$ at $wut(E')$;
            $\{Prefix, v, Suffix\} \leftarrow split\_wut\_at(v, wut(E'_c))$;
            add the first processes of $Prefix$ to $sleep(E'_c)$;
            mark $Suffix$ as $not\_owned$ at $wut(E'_c)$;
            add $E'_c$ to $Frontier$;
```

wakeup tree to three parts: the $Prefix$ (the wakeup tree entries ordered before the branch v), the leaf v and the $Suffix$ (the wakeup entries ordered after v). The first processes processes of the entries of the $Prefix$ are added to the sleep set at the new execution sequence E'_c (e.g. if $p.q.r$ is a leaf in $Prefix$, then p is added to $sleep(E'_c)$). The $Suffix$ entries are marked as not_owned at E'_c.

Algorithm 9: Handling Scheduler and Planner Response

```
Function wait_response(Frontier, T, PlannerQueue)
    receive a message M;
    if M is sent from a Scheduler then
        $E \leftarrow M$;
        $PlannerQueue.push(E)$;
    else if M is sent from the Planner then
        $E \leftarrow M$;
        $update\_execution\_tree(E, T)$;
        add $E$ to $Frontier$;
```

After assigning the available work to the available schedulers and the Planner, the Controller will wait for a response either from a scheduler or the Planner (Algorithm 9). When a response is received from a scheduler, the fully explored received execution sequence will be added to the queue of the Planner and the Controller will continue with its loop (Algorithm 7). If a response is received from the Planner, the Controller will update the execution tree T by adding the new wakeup trees that were inserted by the Planner and by deleting the suffix of the execution sequence that was just explored and has no wakeup trees. We delete this part in order to have the execution tree only contain the part of the state space that is either currently getting explored or is planned to be explored. If we were to not delete those suffixes, the size of the execution tree would eventually be the size of our complete state space.

5.1.3 Example

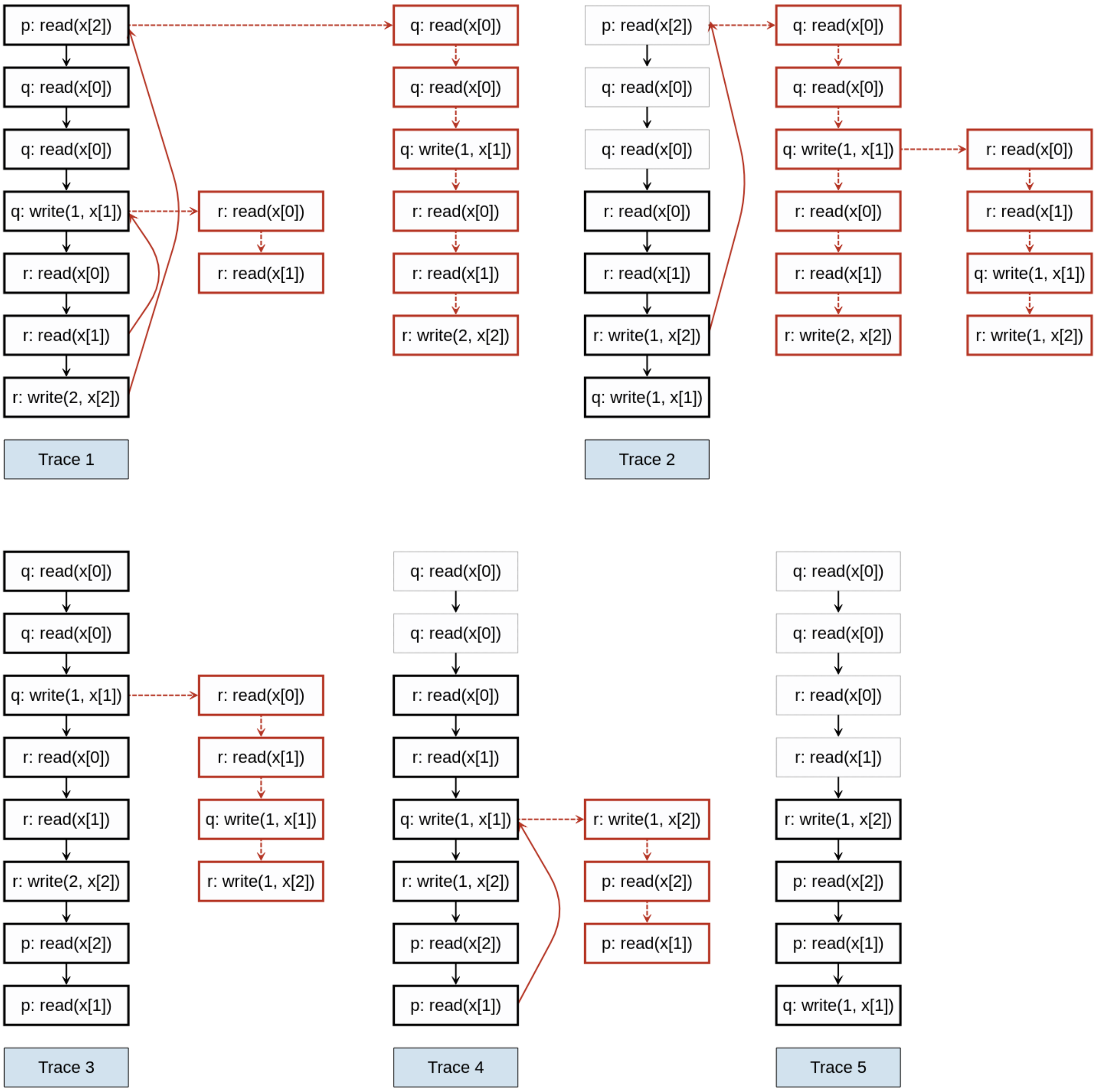
In Figure 5.1 we can see the pseudocode of $lastzero$ 2, were we have an array of 3 elements (initially all elements have a zero value) and three processes. The first process (p) searches the array for the zero element with the highest index. The other two processes increase their assigned element by a value of 1.

```
p:
i = N
while x[i] != 0 and i > 0:
    i = i – 1
```

```
q:
R1 = x[0]
R2 = x[0]
assert(R1 == R2)
x[1] = R2 + 1
```

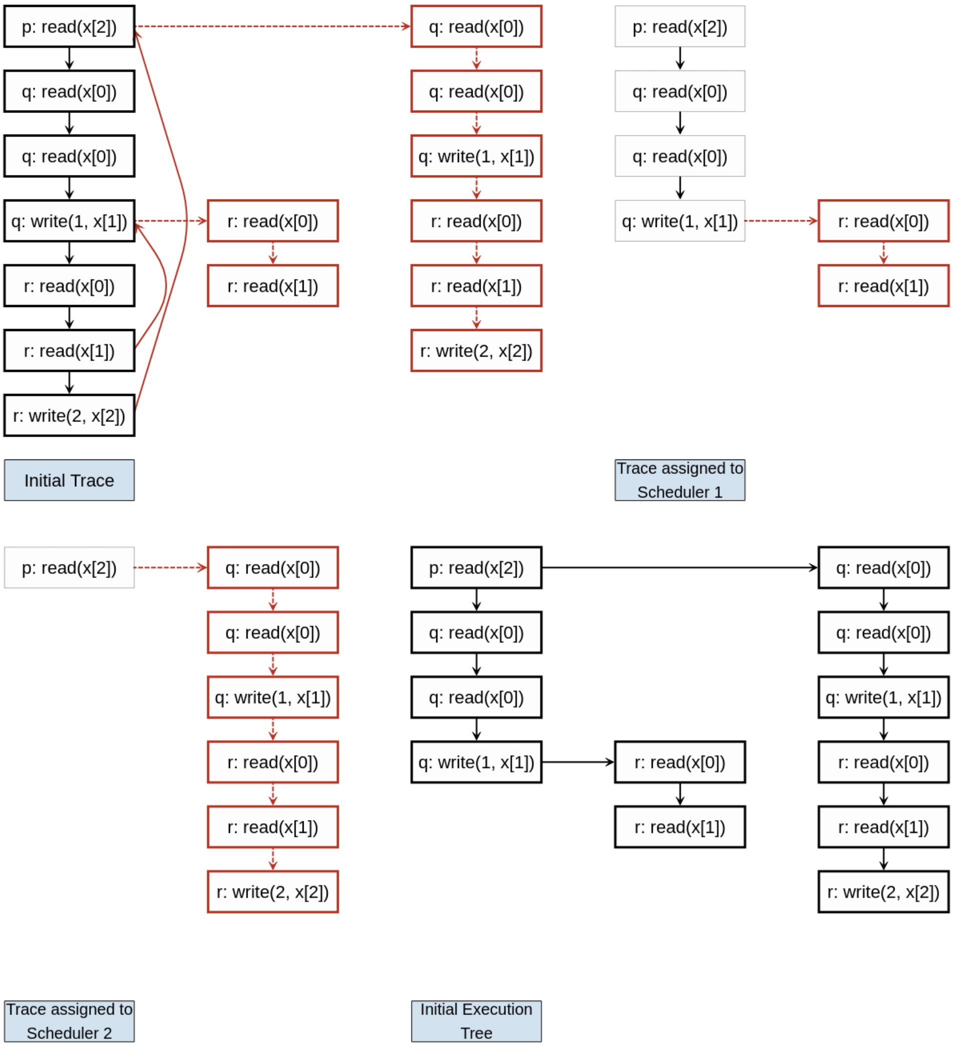
```
r:
R3 = x[0]
R4 = x[1]
assert(R3 == R4)
x[2] = R4 + 1
```

Σχήμα 5.1: *Lastzero* 2 example


p: read(x[2])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
Trace 1
r: read(x[0])
r: read(x[1])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
p: read(x[2])
q: read(x[0])
q: read(x[0])
r: read(x[0])
r: read(x[1])
r: write(1, x[2])
q: write(1, x[1])
Trace 2
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
r: read(x[0])
r: read(x[1])
q: write(1, x[1])
r: write(1, x[2])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
p: read(x[2])
p: read(x[1])
Trace 3
r: read(x[0])
r: read(x[1])
q: write(1, x[1])
r: write(1, x[2])
q: read(x[0])
q: read(x[0])
r: read(x[0])
r: read(x[1])
q: write(1, x[1])
r: write(1, x[2])
p: read(x[2])
p: read(x[1])
Trace 4
r: write(1, x[2])
p: read(x[2])
p: read(x[1])
q: read(x[0])
q: read(x[0])
r: read(x[0])
r: read(x[1])
r: write(1, x[2])
p: read(x[2])
p: read(x[1])
q: write(1, x[1])
Trace 5

Σχήμα 5.2: Interleavings explored by the sequential optimal-DPOR

Figure 5.2 represents the traces explored during the sequential optimal-DPOR. We use a black bold rectangle to represent a new event and a faint rectangle to denote a replayed event. The continuous red edges represent the races that are detected and planned. The red nodes represent the wakeup tree entries at each trace.


p: read(x[2])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
Initial Trace
r: read(x[0])
r: read(x[1])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
p: read(x[2])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
Trace assigned to Scheduler 1
p: read(x[2])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
Trace assigned to Scheduler 2
p: read(x[2])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
q: read(x[0])
q: read(x[0])
q: write(1, x[1])
r: read(x[0])
r: read(x[1])
r: write(2, x[2])
Initial Execution Tree

Σχήμα 5.3: Initial interleaving explored by the parallel optimal-DPOR

Figure 5.3 depicts the initial step of the parallel optimal-DPOR. An arbitrary execution sequence is explored initially and then its races are detected and planned in the form of wakeup trees. The wakeup trees are distributed into different fragments and all unassigned fragments are assigned into the idle schedulers. Also the execution tree is initialized with the current exploration frontier.

In this example, lets assume that Scheduler 2 finishes first the exploration of its assigned trace. The controller will then receive the new explored trace and will add this trace to the queue of the Planner. Since the Planner is idle, this trace will be sent to the Planner to be race detected. While race detecting this trace, no more interleavings will be planned. This trace is equivalent to the 3rd trace of the sequential execution (Figue 5.2). Notice here that in the sequential algorithm this trace had an additional wakeup tree. This wakeup tree was planned by the 2nd trace of the sequential algorithm, which has yet to be race detected in our example. Therefore, traces 4 and 5 of the sequential algorithm cannot be planned from the 3rd trace but only from the 2nd. This makes apparent the main issue of the parallelization of the optimal-DPOR: the complete wakeup tree at some execution sequence E cannot be known until we have completed exploring all execution sequences which are ordered before E, according to the total order of our state space (Definition 3.4)

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%	23m32.843s	50m98,312s	49m21,219s
readers 10	59.1%	40.9%	43.267s	59.699s	54.592s

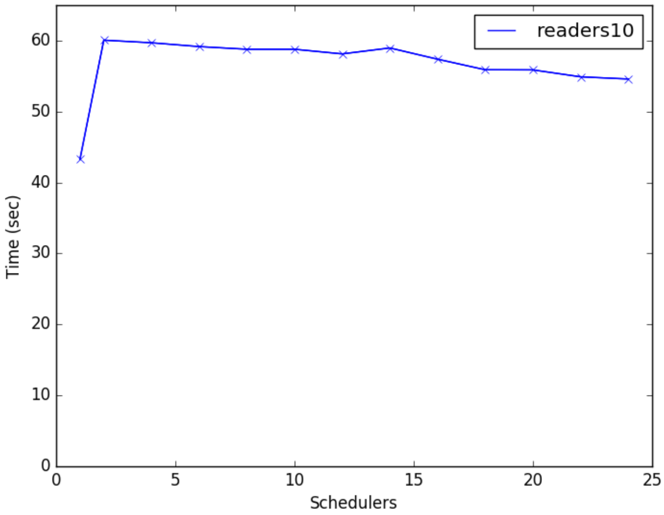
Πίνακας 5.1: Parallel Optimal-DPOR performance.

5.1.4 Performance Evaluation

We are going to use the following two synthetic benchmarks to evaluate our first attempt in parallelizing the optimal-DPOR algorithm:

- $readers\ N$: 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.

Our parallel optimal-DPOR fails to achieve any type of speedup, as depicted in Figure 5.4. Table 5.1 holds information about how our implementation runs on various test cases. The exploration and planning time rows show what percentage of the time of the sequential algorithm is spend on exploring and planning interleavings respectively. Those measurements will help us explain the reason for this lack of speedup.



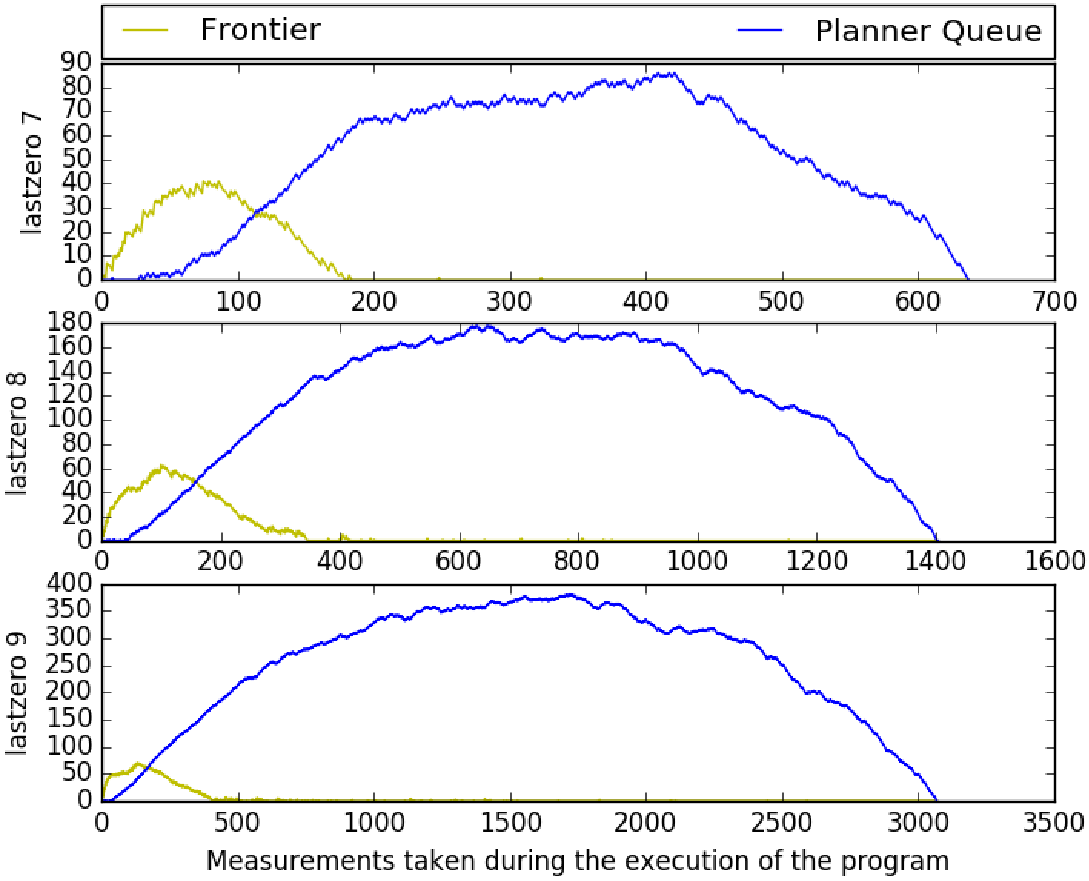
Σχήμα 5.4: Execution time for readers 10 by optimal-DPOR.

5.1.5 Performance Analysis

This lack of performance is justified by the following reasons.

Firstly, lets look at the exploration and planning percentages of the sequential algorithm, presented at Table 5.1. While the percentage of exploration time in the case of a small test case like readers10 is around 40%, we have observed that for larger test cases this figure varies between 10%-30%. This means that even on an ideal setup, with zero overhead and infinite schedulers, our speedup could never exceed a factor of 1.429 (on a test case with 70% planning time, if we consider that the exploration phase happens instantly the speedup would be $1.429 = \frac{100}{70}$, since the planning phase would still have to take place sequentially). This reason, by itself, makes it impossible for our algorithm to achieve good performance and scalability.

What is more, the rate with which new interleavings are planned for exploration leads to schedulers not being sufficiently utilized throughout the execution of the algorithm. Specifically, we have noticed that the execution of the DPOR algorithm can be split into three consecutive phases. In the first phase, race detecting a single interleaving generates a large amount of interleavings that need to be explored and therefore the exploration Frontier is relatively large. This means that the schedulers have enough work and are being kept utilized. During the second phase, race detecting an interleaving generates a relatively small amount of new interleavings that need to be explored. Generally, throughout this phase, not enough work is being generated and the schedulers tend to stay idle. In the third phase, barely any new interleavings are planned, until the complete state space has been explored, at which point the algorithm terminates.



Σχήμα 5.5: Planner Queue and Frontier sizes for the execution of lastzero.

This pattern becomes visible from Figure 5.5. We have run our algorithm with four schedulers for the lastzero 7,8 and 9 benchmarks. To create this graph we have taken measurements regarding the size of queue of the Planner and the exploration Frontier of the schedulers. The measurements are taken after either the exploration or the planning of an interleaving has finished. In this graph, the three aforementioned phases can be clearly observed. After the first phase ends, we notice that the relative large exploration Frontier that has been created, gets “consumed” in an extremely fast rate, due to the fact that we have four scheduler consuming from it, while only one Planner producing in it. Also, like we mentioned before, the exploration of a single trace is significantly faster than its race detection. Consequently, during the second phase there are very few traces within the exploration Frontier and therefore, the schedulers stay underutilized. We can also notice from the graph that this effect intensifies while our state space increases. The main reason behind this is the fact that as the state space of a program increases, the planning of an interleaving becomes even more slow, compared to the exploration of an interleaving.

To make matters worse, our algorithm also has a significant overhead. As mentioned before, communication between the controller and the workers can be a substantial bottleneck. The parallel source-DPOR deals with it by assigning a state space to the workers, minimizing the need for communication. However, in this attempt the use of the centralized planner leads to the schedulers having to report a trace back to the controller every single time an exploration reaches its end.

5.2 Scalable Parallel optimal-DPOR

5.3 Basic Idea

Like we mentioned in the previous section, trying to modify the parallel source-DPOR algorithm to develop a parallel-DPOR algorithm has two main issues.

Firstly, in source-DPOR the backtrack entries are

Κεφάλαιο 6

Implementation Details

In Chapter 4 we described how the Source-DPOR and Optimal-DPOR algorithms can be parallelized, by using multiple schedulers to explore different interleavings concurrently. In order to accomplish this, some modification are to be made. Here we summarize those necessary modifications.

We should briefly describe how the scheduler works in the sequential implementation. The scheduler starts by exploring completely an arbitrary interleaving (with a maximal sequence E), through the function *explore/1*. It continues by calling *plan_more_interleaving/1*, in order to detect the races of the explored interleaving and plan the exploration of future interleavings according to the logic of the used algorithm. Let's assume that we must explore a sequence $E'.p$, where E' is a prefix of E and p a process. The next invocation of *explore/1* will reset all actor processes, to force them back to their initial state, and then it will replay the prefix E' to recreate the global state at E', without completely recreating the events in the prefix. After the replay is done, the processes in the backtrack (or in the wakeup tree) will be scheduled and finally the remaining events will be scheduled arbitrarily so as no more processes are enabled. The scheduler will then try and plan more interleavings. We are finished when there are no more interleavings left to explore (or until an error is found if the option *keep_going* is set to *false*).

Let us, also, describe the main datatypes used in the scheduler:

- $event()$: corresponds to the event e of a process p, according to our notation. It contains the Erlang Pid (process identifier) of the actor process p, as well as information about the code (e.g. the BIF) of this specific event.
- $\#event\_tree\{\}$: refers to either the backtrack or the wakeup tree at a specific point.
- $\#trace\_state\{\}$: holds information about an execution step of an execution sequence E, such as the backtrack (or the wakeup tree) and the sleep set at this point.
- $\#scheduler\_state\{\}$ a record that contains information regarding the state of the scheduler such as the algorithm used and, most importantly, the current trace, which is a list of $\#trace\_state\{\}$ records. This list roughly corresponds to the execution sequence E as defined in our framework.

6.1 Dealing with Processes

In the sequential Concuerror, for each process of a tested program, the scheduler needs to spawn only one process. The scheduler will then control the execution of the processes of a program to produce different interleavings. In order to concurrently explore different interleavings, for every process in the tested program, each parallel scheduler must spawn its own process. This technically means that we should have different Erlang processes that correspond to the same process of the tested program. Erlang processes are characterized by their Pid, which is globally unique. The Pid of a process is also used in Concuerror to identify a process and, therefore, characterize a trace. When transferring traces between schedulers any Pid found anywhere in the trace should change to reflect the Pids of the different schedulers.

This means that a mapping should be created between the Pids of the different schedulers. This mapping can be established through the *symbolic names* that Concuerror assigns to the tested process with the following logic:

$$Symbol(p) = \begin{cases} ``P" & \text{if } p \text{ is the initial process} \\ Symbol(q).i & \text{if } p \text{ is the } i_{th} \text{ child of } q \end{cases}$$

However, creating such mappings is not enough to guarantee that a trace can be transferred between different schedulers. It is important that the same execution sequence leads to the same global state regardless of the scheduler that explores it. Specifically, Erlang gives the ability to compare Pids. For instance, the ordering of two Pids could change the outcome of a branch in a program. This could result in the same trace leading to different global states on different schedulers. What is more, through the use of the BIF *pid_to_list/1*, a Pid could exist in the form of a string in some trace and as a result we would have to try and parse every string in a trace to check whether is refers to a Pid.

We solve these issues by having each scheduler run on its own Erlang node. It is possible for two processes, located on different nodes, to have the same local Pid. While trying to implement such a mechanism, we have encountered two main issues:

- Erlang does not give the option to request specific Pids. Nevertheless, the Erlang VM of a node assigns Pids in a sequential ordering. For example, after spawning a process with $< 0.110.0 >$ as a local Pid, then next process spawned in that node would have a Pid of $< 0.111.0 >$. We can use this to preemptively spawn processes on different nodes with the same Pid, by creating a *process_spawner*. We require that nothing besides our schedulers runs on our nodes and therefore, there will be no interference with sequence of the spawned Pids on the node. Firstly, we must reach a consensus between the different schedulers as to the initial local Pid. This consensus can be achieved by having each scheduler send to the *process_spawner* the first available local Pid in their node (we can get this by spawning a dummy process). The *process_spawner* chooses the maximum local Pid and sends it to all the schedulers. The schedulers can then spawn a process with this maximum Pid by spawning and killing dummy processes until they reach the requested Pid. Then they can spawn a specified (by the user) amount or processes preemptively. The i_{th} processes spawned this way on different nodes, will all have the same local Pid. Thanks to that, we can have different processes on different nodes with the same local Pid that corresponds to the same symbolic process.

- Simply sending a trace between schedulers on different nodes will result in the Erlang VM changing every Pid on the trace to their global values. Those values, however, are unique. We can avoid this by transforming every Pid on that trace to a string (by using the *pid_to_list/1* BIF) before sending the trace. When we send the transformed trace the VM will not interfere with the transformed Pids. The local Pids can then be recovered from the receiving scheduler by using the *list_to_pid/1* BIF. These local Pids will refer to processes with the same symbolic name on different nodes.

6.2 Execution Sequence Replay

Even after fixing the Pid issue, replaying traces on different schedulers is not going to work. There are two basic reasons behind this.

Firstly, during the execution of certain events (such as events related to ETS tables or spawning) Concuerror uses various ETS tables to keep track of specific information. When Concuerror explores a trace in replay mode, it makes sure that such information exists and drives the tested processes to the appropriate state. Therefore, when Concuerror explores a trace it creates some side-effects on the its global state. Those side-effects will not exist on a different Erlang node, since ETS tables are not shared between nodes.

```erlang
Pid = spawn(fun() ->
                receive
                    exit ->
                        ok
                end
            end),
Lambda =
    fun() ->
         Pid ! exit
    end.
```

Listing 6.1: Environment variables

Secondly, user defined lambda functions can have some environment variables. The value of those variables is immutable once the lambda function is defined. In Listing 6.1, if a trace contains an event that applies this function, this event will not be able to be properly replayed by a scheduler other than the one that created it, since the Pid environment variable cannot be changed. The only reasonable way to solve this is to change how replaying works.

Specifically, we need to have two different replay modes: *pseudo* and *actual*. Pseudo replay is used when replaying traces that were created by the same scheduler and works exactly like the replay of the sequential Concuerror. Actual replay recreates the events and the side-effects of a trace and is used for replaying interleavings received from other schedulers. On built-in events, we achieve this by setting an *actual_replay* flag to *true* and by making changes on how the *concuerror_callback* module handles those replays. On other events, we set their *event_info* value to *undefined* forcing those events to be recreated.

6.3 Logger Modifications

The Logger is necessary for reporting erroneous interleavings. If we were to keep the logger working as is, the reports from the different schedulers would interleave with each other leading unreadable output. There are two ways solve this.

We can implement a *logger_wrapper* function that is going to be responsible for organizing

Κεφάλαιο 7

Performance Evaluation

In this chapter, we are going to present the performance results of our parallel source-DPOR and optimal-DPOR algorithms, implemented in Concuerror. We are going to evaluate our results on some "standard" and synthetic benchmarks that are normally used to test DPOR algorithms. Finally, we will try to explain the behavior of the parallel program as reflected in those charts, drawing related conclusions whenever possible.

7.1 Tests Overview

First we are going to give a brief overview of the programs tested:

- *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.

- *writers* N: This is a modification of the readers test. Here we have a reader process that reads a variable and N writer processes that write that variable. The fact that we have many writers creates more races and therefore more work for our program.

- *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.

Table 7.1 contains information about the traces explored and duration of those explorations for the sequential versions of Source-DPOR and Optimal-DPOR.

The benchmarks were performanced 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

Benchmark	Traces for Source-DPOR	Traces for Optimal-DPOR	Time for Source-DPOR	Time for Optimal-DPOR
lastzero 11	60073	7168	50m39.201s	23m32.843s
indexer 15	4096	4096	18m40.386s	12m45.39
readers 15	32768	32768	39m17.533s	52m43.585s

Πίνακας 7.1: Sequential Source-DPOR vs Optimal-DPOR for our benchmarks

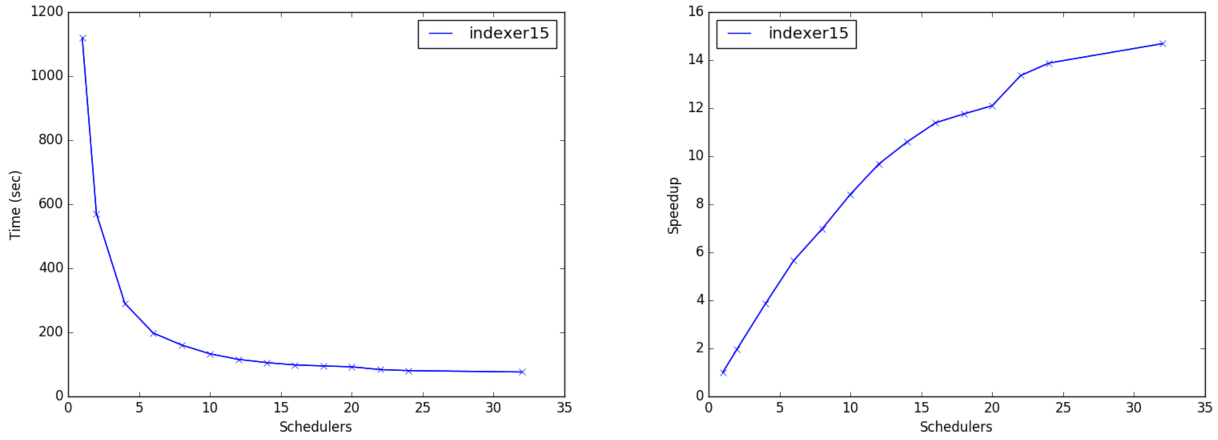
version (Erlang/OTP 21.1). While running our tests, we are using the –*keep_going* flag to continue exploring our state space, even after an error is found. We do this so we can evaluate how fast the complete state space gets explored.

7.2 Source DPOR

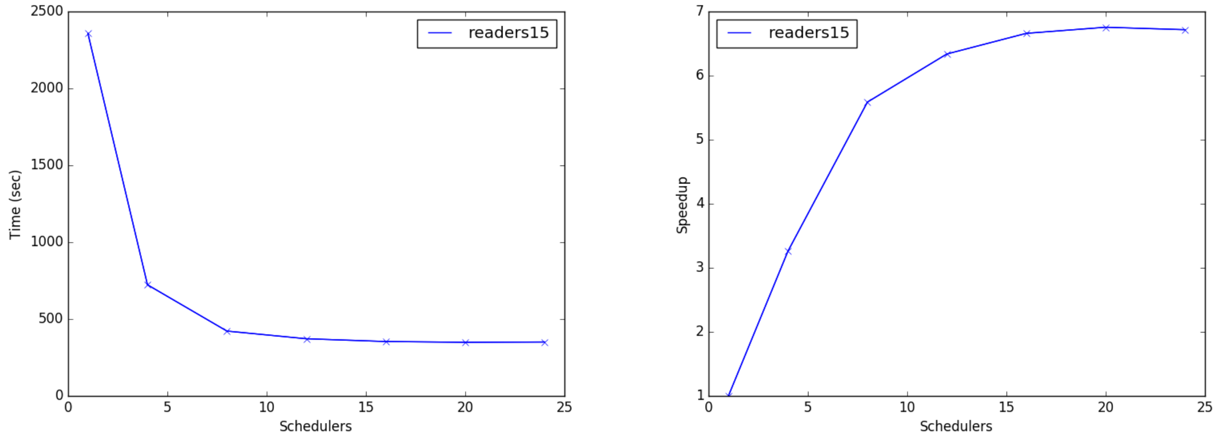
7.2.1 Preformance Results

Here we are going to present graphs depicting the execution time and the speedup ($\frac{T_{serial}}{T_{parallel}}$) of the source DPOR algorithm for different numbers of schedulers and for various test cases. Also we are going combine the speedups for those test cases in a comparative graph (Figure 7.4).

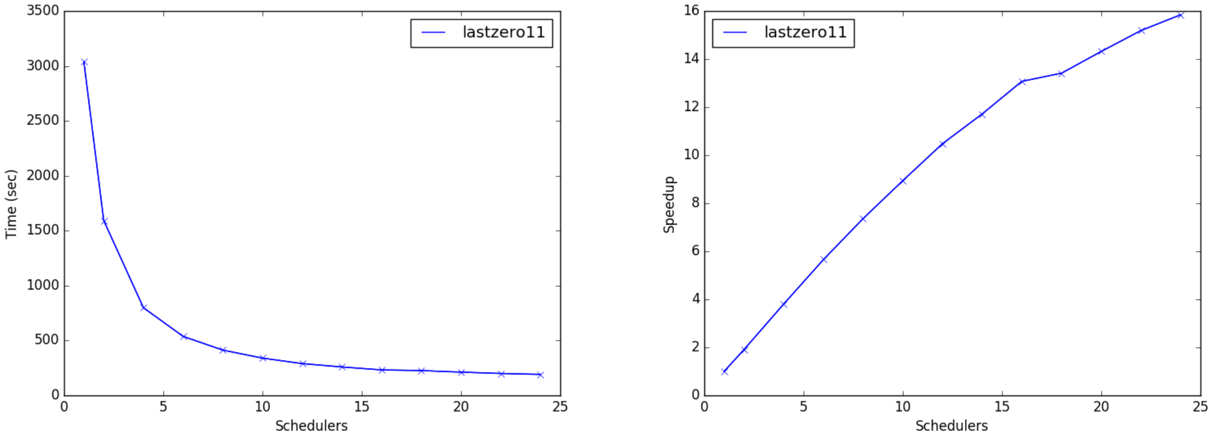
(Note: more benchmarks will be added)



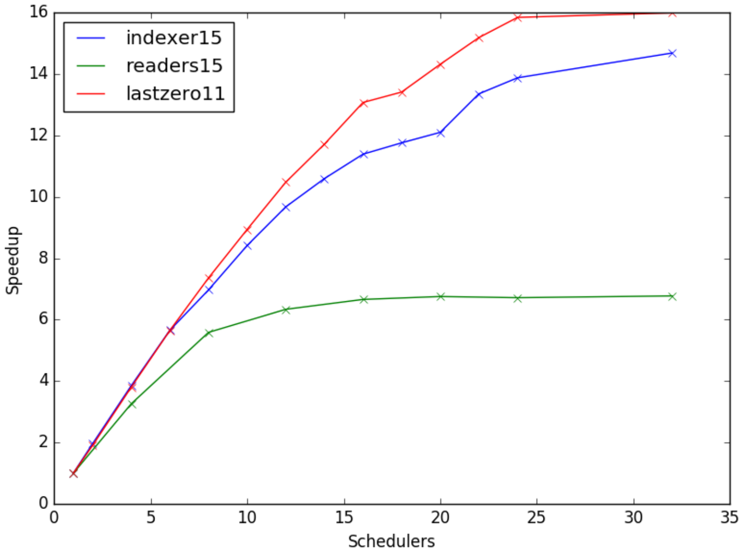
Σχήμα 7.1: Performance for indexer 15



Σχήμα 7.2: Performance for readers 15



Σχήμα 7.3: Performance for lastzero 11



Σχήμα 7.4: Speedup comparison for source-DPOR

7.2.2 Performance Analysis

As we can see in our charts, our parallel implementation of source-DPOR in Concuerror significantly reduces the testing time of our test cases. Namely, in the cases of both lastzero and indexer we have a speedup by a factor of 1.9 for 2 schedulers and of 3.9 for 4 schedulers. This makes parallel source DPOR highly usable in personal computers.

Additionally, we are getting decent scalability in test cases like lastzero11 and indexer15. Particularly, with 32 schedulers we manage as speedup by a factor of 16 in the case of lastzero. However, through the scalability charts we can notice that with more schedulers the scalability of those test cases starts to decline. This happens because we are using a centralize controller which becomes a bottleneck, since the more the schedulers the higher the chance the controller will be busy and therefore, unable to distribute work. The most important reason behind this drop in scalability though, is the fact that with more available schedulers the assignment of the state space of the program becomes more fine grained. A more fine grained assignment leads to the schedulers exploring their assigned state space faster and having more races found outside their state space (since the state space is smaller). Consequently, the communication with the controller is more frequent. This situation is problematic for two reasons: the communication between a scheduler and the controller has a non-negligible overhead and when the amount of communications between the schedulers and the controller increases, the controller becomes an even bigger bottleneck. As such, we come to the conclusion that a test case cannot scale beyond a certain point based on the number of its interleavings.

From Figure 7.4 we can notice the scalability of readers15 breaks significantly faster compared to that of the other test cases, despite the fact that readers15 has more explored traces than indexer15 (Table 7.1). This means that number of the traces explored is not the only scalability factor and different test cases with equivalent sizes can have varied results.

7.3 Optimal DPOR

7.3.1 Preformance Results

7.4 Final Comments

The fact that optimal-DPOR lacks speedup does not mean that we have not benefited from our parallelization. The good resutls produced by the parallel source-DPOR help the source-DPOR algorithm outperform the sequential optimal-DPOR.

When there are no sleep-set blocked interleavings the sequential source-DPOR can run faster than the optimal-DPOR (Table 7.1). In the case of readers15, for instance, the sequential source-DPOR is already faster by the optimal algorithm by around 13 minutes. If we combine this with speedup of the parallel version, the benefit of our parallelization becomes clear.

Still, even when we have a significant amount of sleep-set blocked interleavings, like in the case of lastzero11, our parallel source-DPOR can fairly easily catch up with the optimal-DPOR. Specifically, the optimal algorithm has an execution time of 23m32.843s and explores 7168 interleavings, while the source-Algorithm has an execution time of 50m39.201s and explores 60073 interleavings. By using two schedulers, the runtime of source-Algorithm drops to 26m33.051s and with four schedulers to 13m19,016s. As we add schedulers source-DPOR will outperform the optimal algorithm even more.

Κεφάλαιο 8

Concluding Remarks

Βιβλιογραφία

[Abdu14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, "Optimal Dynamic Partial Order Reduction", in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pp. 373–384, New York, NY, USA, 2014, ACM.

[Abdu15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson and Konstantinos Sagonas, "Stateless Model Checking for TSO and PSO", in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pp. 353–367, Berlin, Heidelberg, 2015, Springer-Verlag.

[Arms07] Joe Armstrong, *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf, 2007.

[Aron18] Stavros Aronis, *Effective Techniques for Stateless Model Checking*, Ph.D. thesis, 2018.

[Chri13] M. Christakis, A. Gotovos and K. Sagonas, "Systematic Testing for Detecting Concurrency Errors in Erlang Programs", in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 154–163, March 2013.

[Clar99] E.M. Clarke, O. Grumberg, M. Minea and D. Peled, "State space reduction using partial order techniques", *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, Nov 1999.

[Flan05] Cormac Flanagan and Patrice Godefroid, "Dynamic Partial-order Reduction for Model Checking Software", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pp. 110–121, New York, NY, USA, 2005, ACM.

[Gode96] Patrice Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Gode97] Patrice Godefroid, "Model Checking for Programming Languages Using VeriSoft", in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pp. 174–186, New York, NY, USA, 1997, ACM.

[Gode05] Patrice Godefroid, "Software Model Checking: The VeriSoft Approach", *Form. Methods Syst. Des.*, vol. 26, no. 2, pp. 77–101, March 2005.

[Goto11] Alkis Gotovos, Maria Christakis and Konstantinos Sagonas, "Test-driven Development of Concurrent Programs Using Concuerror", in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pp. 51–61, New York, NY, USA, 2011, ACM.

[Kurs98] R. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigün, “Static partial order reduction”, in Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 345–357, Berlin, Heidelberg, 1998, Springer Berlin Heidelberg.

[Lei06] Yu Lei and Richard H. Carver, “Reachability Testing of Concurrent Programs”, *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 382–403, June 2006.

[Matt88] Friedemann Mattern, “Virtual Time and Global States of Distributed Systems”, in *PARALLEL AND DISTRIBUTED ALGORITHMS*, pp. 215–226, North-Holland, 1988.

[Mazu87] Antoni Mazurkiewicz, “Trace theory”, in W. Brauer, W. Reisig and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pp. 278–324, Berlin, Heidelberg, 1987, Springer Berlin Heidelberg.

[Moor06] G. E. Moore, “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.”, *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sept 2006.

[Musu08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar and Iulian Neamtiu, “Finding and Reproducing Heisenbugs in Concurrent Programs”, in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI’08, pp. 267–280, Berkeley, CA, USA, 2008, USENIX Association.

[Sims11] J. Simsa, R. Bryant, G. Gibson and J. Hickey, “Efficient Exploratory Testing of Concurrent Systems”, *CMU-PDL Technical Report*, vol. 113, November 2011.

[Sims12] Jirí Simsa, Randy Bryant, Garth A. Gibson and Jason Hickey, “Scalable Dynamic Partial Order Reduction”, in *RV*, 2012.

[Valm91] Antti Valmari, “Stubborn sets for reduced state space generation”, in Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pp. 491–515, Berlin, Heidelberg, 1991, Springer Berlin Heidelberg.

[Vird96] Robert Virding, Claes Wikström and Mike Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[Wiki18] Wikipedia contributors, “Preemption (computing) — Wikipedia, The Free Encyclopedia”, `https://en.wikipedia.org/w/index.php?title=Preemption_(computing)&oldid=838871409`, 2018. [Online; accessed 18-October-2018].

[Yang07] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan and Robert M. Kirby, “Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software”, in *Proceedings of the 14th International SPIN Conference on Model Checking Software*, pp. 58–75, Berlin, Heidelberg, 2007, Springer-Verlag.

[Yang08] Y. Yang, X. Chen and G. Gopalakrishnan, *Inspect: A Runtime Model Checker for Multithreaded C Programs*, University of Utah Tech, 2008.