Software and Systems Modeling (2023) 22:521–545
https://doi.org/10.1007/s10270-022-01060-7

SPECIAL SECTION PAPER

Linear parallel algorithms to compute strong and branching bisimilarity

Jan Martens[1] · Jan Friso Groote[1] · Lars B. van den Haak[1] · Pieter Hijma[1,2] · Anton Wijs[1]

Received: 6 April 2022 / Revised: 5 August 2022 / Accepted: 30 September 2022 / Published online: 6 December 2022
© The Author(s) 2022

Abstract
We present the first parallel algorithms that decide strong and branching bisimilarity in linear time. More precisely, if a transition system has n states, m transitions and $|Act|$ action labels, we introduce an algorithm that decides strong bisimilarity in $\mathcal{O}(n + |Act|)$ time on $\max(n, m)$ processors and an algorithm that decides branching bisimilarity in $\mathcal{O}(n + |Act|)$ time using up to $\max(n^2, m, |Act|n)$ processors.

Keywords Strong bisimulation · Branching bisimulation · RCPP · Parallel algorithms · PRAM

1 Introduction

The notion of *bisimilarity* for Kripke structures and labelled transition systems (LTSs) is commonly used to define behavioural equivalence. Deciding this equivalence is essential for modelling and verifying discrete event systems [1,2]. Kanellakis and Smolka proposed a partition refinement algorithm for obtaining the bisimilarity relation for Kripke structures [3]. The proposed algorithm has a run time complexity of $\mathcal{O}(nm)$ where n is the number of states and m is the number of transitions of the input. Later, a more sophisticated refinement algorithm running in $\mathcal{O}(m\ log\ n)$ steps was proposed by Paige and Tarjan [4].

In recent years, the increase in the speed of sequential chip designs has stagnated due to a multitude of factors such as energy consumption and heat generation. In contrast, parallel devices, such as graphics processing units (GPUs), keep increasing rapidly in computational power. In order to profit from the acceleration of these devices, we require algorithms with massive parallelism. The article ‘There’s plenty of room at the Top: What will drive computer performance after Moore’s law’ by Leierson et al. [5] indicates that the advance in computational performance will come from software and algorithms that can employ hardware structures with a massive number of simple, parallel processors, such as GPUs. In this article, we propose two such algorithms to decide strong and branching bisimilarity.

For strong bisimilarity, we improve on the best-known theoretical time complexity for parallel bisimulation algorithms using a higher degree of parallelism. The proposed algorithm improves the run time complexity to $\mathcal{O}(n)$ on $\max(n, m)$ processors, and we base it on the sequential algorithm of Kanellakis and Smolka [3]. This time complexity matches the theoretical lower bound for parallel partition refinement algorithms given in [6]. The larger number of processors used in this algorithm favours the increasingly parallel design of contemporary and future hardware. In addition, the algorithm is *optimal* w.r.t. the sequential Kanellakis–Smolka algorithm, meaning that overall, it does not perform more work than its sequential counterpart.

We first present our algorithm on Kripke structures where transitions are unlabelled. However, as labelled transition systems (LTSs) are commonly used, and labels are not

Communicated by Gwen Salaun.

✉ Jan Martens
j.j.m.martens@tue.nl

Jan Friso Groote
j.f.groote@tue.nl

Lars B. van den Haak
l.b.v.d.haak@tue.nl

Pieter Hijma
pieter@cs.vu.nl

Anton Wijs
a.j.wijs@tue.nl

[1] Eindhoven University of Technology, De Groene Loper 5, Eindhoven 5612 AE, The Netherlands

[2] Vrije Universiteit, De Boelelaan 1105, Amsterdam 1081 HV, The Netherlands

straightforward to incorporate in an efficient way (cf. for instance [7]), we discuss how to extend our algorithm to take action labels into account. This leads to an algorithm with a run time complexity of $\mathcal{O}(n + |Act|)$ on $\max(n, m)$ processors, with Act the set of action labels.

Our algorithm has been designed for and can be analysed with the Concurrent Read Concurrent Write (CRCW) PRAM model, following notations from [8]. This model extends the normal RAM model, allowing multiple processors to work with shared memory. In the CRCW PRAM model, parallel algorithms can be described in a straightforward and elegant way. In reality, no device exists that completely adheres to this PRAM model. Still, with recent advancements, hardware gets better and better at approximating the model since the number of parallel threads keeps growing. We demonstrate this by translating the PRAM algorithm to GPU code. We straightforwardly implemented our algorithm that decides strong bisimilarity in CUDA and experimented with an NVIDIA Titan RTX, a state-of-the-art GPU, showing that our algorithm performs mostly in line with what our PRAM algorithm predicts.

The present article is an extended version of our conference paper [9]. We have extended that work with a new algorithm that decides branching bisimilarity. Algorithms deciding branching bisimilarity typically propagate information over τ-paths sequentially, which can take $\mathcal{O}(n)$ time. We opt to use up to n^2 processors to propagate this information in constant time. In this way, we manage to keep a linear time bound on the algorithm. As far as we know, this is the first algorithm deciding branching bisimilarity that uses extra parallelism to propagate this information. We explain this new algorithm and provide correctness proofs.

The article is structured as follows: In Sect. 2, we recall the necessary preliminaries on the CRCW PRAM model and state the partition refinement problems this article focuses on. In Sect. 3, we propose a parallel algorithm to compute strong bisimulation for Kripke structures, which is also called the relational coarsest partition problem (RCPP). In this section, we also prove the correctness of the algorithm and provide a complexity analysis. In Sect. 4, we discuss the details for an adjustment to the algorithm that deals with multiple action labels, thereby supporting LTSs, which forms the bisimulation coarsest refinement problem (BCRP). In this section, we also provide a complexness and correctness argument, and the results of experiments on a GPU implementation. In Sect. 5, we discuss the modification of the algorithm to compute branching bisimilarity (branching-BCRP). We discuss the details of the modifications, prove them correct and give the complete algorithm. In Sect. 6, we discuss related work. Lastly, in Sect. 7 we draw conclusions and discuss future work.

2 Preliminaries

2.1 The PRAM model

A *parallel random access machine* (PRAM) is a natural extension of the normal random access machine (RAM), where an arbitrary number of parallel processors can access the memory. Following the definitions of [8], we use a version of a PRAM that is able to concurrently read and concurrently write (CRCW PRAM). It differs from the model introduced in [10] in which the PRAM model was only allowed to concurrently read from the same memory address, but concurrent writes (to the same address) could not happen.

A CRCW PRAM consists of a sequence of numbered processors $P_0, P_1, \ldots$. These processors have all the natural instructions of a normal RAM, such as addition, subtraction and conditional branching based on the equality and less-than operators. There is an infinite amount of common memory the processors have access to. The processors have instructions to read from and write to the common memory. In addition, a processor P_i has an instruction to obtain its unique index i.

All the processors have the same program and run in a synchronised way in a single-instruction, multiple-data (SIMD) fashion. In other words, all processors execute the program in lock step. Parallelism can be achieved by distributing the data elements over the processors and having the processors apply the program instructions to 'their' data elements.

We assume that one arbitrary write will succeed whenever a concurrent write happens to the same memory cell. This is called the *arbitrary* CRCW PRAM.

A parallel program for a PRAM is called *optimal* w.r.t. a sequential algorithm if the total work done by the program does not exceed the work done by the sequential algorithm. More precisely, if T is the parallel run time and P the number of processors used, then the algorithm is optimal w.r.t. a sequential algorithm running in S steps if $P \cdot T \in \mathcal{O}(S)$.

2.2 Strong bisimulation

To formalise concurrent system behaviour, we use LTSs.

Definition 2.1 (*Labelled Transition System*) A labelled transition system (LTS) is a three-tuple $A = (S, Act, \rightarrow)$ where S is a finite set of states, Act a finite set of action labels and $\rightarrow \subseteq S \times Act \times S$ the transition relation.

Let $A = (S, Act, \rightarrow)$ be an LTS. Then, for any two states $s, t \in S$ and $a \in Act$, we write $s \xrightarrow{a} t$ iff $(s, a, t) \in \rightarrow$.

Kripke structures differ from LTSs in the fact that the states are labelled as opposed to the transitions. In the current article, for convenience, instead of using Kripke structures where appropriate, we reason about LTSs with a single action label, i.e. $|Act| = 1$. Computing the coarsest partition of such an LTS can be done in the same way as for Kripke structures, apart from the fact that in the latter case, a different initial partition is computed that is based on the state labels (see, for instance, [11]).

Definition 2.2 (*Strong bisimulation*) On an LTS $A = (S, Act, \rightarrow)$ a relation $R \subseteq S \times S$ is called a strong bisimulation relation if and only if it is symmetric and for all $s, t \in S$ with sRt and for all $a \in Act$ with $s \xrightarrow{a} s'$, we have:

$$\exists t' \in S. t \xrightarrow{a} t' \wedge s' R t'$$

Whenever we refer to bisimulation, we mean strong bisimulation. Two states $s, t \in S$ in an LTS A are called *bisimilar*, denoted by $s \leftrightarroweq t$, iff there is a bisimulation relation R for A that relates s and t.

A *partition* π of a finite set of states S is a set of subsets that are pairwise disjoint and whose union is equal to S, i.e. $\bigcup_{B \in \pi} B = S$. Every element $B \in \pi$ of this partition π is called a *block*.

We call partition π' a *refinement* of π iff for every block $B' \in \pi'$ there is a block $B \in \pi$ such that $B' \subseteq B$. We say a partition π of a finite set S induces the relation $R_\pi = \{(s, t) \mid \exists B \in \pi.\ s \in B \text{ and } t \in B\}$. This is an equivalence relation of which the blocks of π are the equivalence classes.

Given an LTS $A = (S, Act, \rightarrow)$ and two states $s, t \in S$, we say that s *reaches* t with action $a \in Act$ iff $s \xrightarrow{a} t$.

A state s *reaches* a set $U \subseteq S$ with an action a iff there is a state $t \in U$ such that s reaches t with action a.

A set of states $V \subseteq S$ is called *stable* under a set of states $U \subseteq S$ iff for all actions a either all states in V reach U with a, or no state in V reaches U with a. A partition π is stable under a set of states U iff each block $B \in \pi$ is stable under U. The partition π is called stable iff it is stable under all its own blocks $B \in \pi$.

Fact 2.3 [4] Stability is inherited under refinement, i.e. given a partition π of S and a refinement π' of π. If π is stable under $U \subseteq S$, then π' is also stable under U.

2.3 Problems

The main problem we focus on in this work is called the bisimulation coarsest refinement problem (**BCRP**). It is defined as follows:

Input: An LTS $M = (S, Act, \rightarrow)$.

Output: The partition π of S which is the coarsest partition, i.e. has the smallest number of blocks, that forms a bisimulation relation.

In a Kripke structure, the transition relation forms a single binary relation, since the transitions are unlabelled. This is also the case when an LTS has a single action label. In that case, the problem is called the relational coarsest partition problem (**RCPP**) [3,4,12]. This problem is defined as follows:

Input: A set S, a binary relation $\rightarrow: S \times S$ and an initial partition π_0

Output: The partition π which is the coarsest refinement of π_0 and which forms a bisimulation relation.

It is known that BCRP is not significantly harder than RCPP as there are intuitive translations from LTSs to Kripke structures [13, Dfn. 4.1]. However, some non-trivial modifications can speed up the algorithm in some cases; hence, we discuss both problems separately.

In Sect. 3, we discuss the basic parallel algorithm for RCPP, and in Sect. 4, we discuss the modifications required to efficiently solve the BCRP problem for LTSs with multiple action labels.

3 Relational coarsest partition problem

In this section, we discuss a sequential algorithm based on the one of Kanellakis and Smolka [3] for RCPP (Sect. 3.1). This is the basic algorithm that we adapt to the parallel PRAM algorithm (Sect. 3.2). The algorithm starts with an input partition π_0 and refines all blocks until a stable partition is reached. This stable partition will be the coarsest refinement that defines a bisimulation relation.

3.1 The sequential algorithm

The sequential algorithm, Algorithm 1, works as follows. Given are a set S, a transition relation $\rightarrow \subseteq S \times S$, and an initial partition π_0 of S. Initially, we mark the partition as not necessarily stable under all blocks by putting these blocks in a set *Waiting*. In any iteration of the algorithm, if a block B of the current partition is not in *Waiting*, then the current partition is stable under B. If *Waiting* is empty, the partition is stable under all its blocks, and the partition represents the required bisimulation.

As long as some blocks are in *Waiting* (line 3), a single block $B \in \pi$ is taken from this set (line 4), and we split the current partition such that it becomes stable under B. Therefore, we refer to this block as the *splitter*. The set $S' = \{s \in S \mid \exists t \in B. s \rightarrow t\}$ is the reverse image of B (line 6). This set consists of all states that can reach B, and we use S' to define our new blocks. All blocks B' that have a non-empty intersection with S', i.e. $B' \cap S' \neq \emptyset$, and are not a subset of S', i.e. $B' \cap S' \neq B'$ (line 7), are split into the subset of states in S' and the subset of states that are not in S' (lines 9–10). These two new blocks are also added to the set of *Waiting* blocks (line 11–12). The number of states is finite, and blocks can be split only a finite number of times. Hence, blocks are only finitely often put in *Waiting*, so the algorithm is guaranteed to terminate.

Algorithm 1 Sequential algorithm based on Kanellakis–Smolka

```
1: $\pi := \pi_0$
2: $Waiting := \pi$
3: while $Waiting \neq \emptyset$ do
4:   for all $B \in Waiting$ do
5:     $Waiting := Waiting \setminus \{B\}$
6:     $S' := \{s \in S \mid \exists t \in B . s \rightarrow t\}$
7:     for all $B' \in \pi$ with $\emptyset \subset B' \cap S' \subset B'$ do
8:       // Split $B'$ into $B' \cap S'$ and $B' \setminus S'$
9:       $\pi := \pi \setminus \{B'\}$
10:      $\pi := \pi \cup \{B' \cap S', B' \setminus S'\}$
11:      $Waiting := Waiting \setminus \{B'\}$
12:      $Waiting := Waiting \cup \{B' \cap S', B' \setminus S'\}$
13:    end for
14:  end for
15: end while
```

3.2 The PRAM algorithm

Next, we describe a PRAM algorithm to solve RCPP that is based on the sequential algorithm given in Algorithm 1.

3.2.1 Block representation

Given an LTS $A = (S, Act, \rightarrow)$ with $|Act| = 1$ and $|S| = n$ states, we assume that the states are labelled with unique indices $0, \ldots, n-1$. A partition π in the PRAM algorithm is represented by assigning a block label from a set of block labels L_B to every state. The number of blocks can never be larger than the number of states; hence, we use the indices of the states as block labels: $L_B = S$. We exploit this in the PRAM algorithm to efficiently select a new block label whenever a new block is created. We select the block label of a new block by electing one of its states to be the *leader* of that block and using the index of that state as the block label. By doing so, we maintain an invariant that the leader of a block is also a member of the block.

In a partition π, whenever a block $B \in \pi$ is split into two blocks B' and B'', the leader s of B, which is part of B' becomes the leader of B', and for B'', a new state $t \in B''$ is elected to be the leader of this new block. Since the new leader is not part of any other block, the label of t is fresh with respect to the block labels that are used for the other blocks. This method of using state leaders to represent subsets was first proposed in [14,15].

3.2.2 Data structures

The common memory contains the following information:

1. $n : \mathbb{N}$, the number of states of the input.
2. $m : \mathbb{N}$, the number of transitions of the input relation.
3. The input, a fixed numbered list of transitions. For every index $0 \leq i < m$ of a transition, a source $source_i \in S$ and target $target_i \in S$ are given, representing the transition $source_i \rightarrow target_i$.
4. $C : L_B \cup \{\bot\}$, the label of the current block that is used as a splitter; $\bot$ indicates that no splitter has been selected.
5. The following is stored in lists of size n, for each state with index i:
   (a) $mark_i : \mathbb{B}$, a mark indicating whether state i is able to reach the splitter.
   (b) $block_i : L_B$, the block of which state i is a member.
6. The following is stored in lists of size n, for each potential block with block label i:
   (a) $new\_leader_i : L_B$ the leader of the new block when a split is performed.
   (b) $waiting_i : \mathbb{B}$ indicating whether π might not be stable w.r.t. the block and should be checked as splitter.

As input, we assume that each state with index i has an input variable $I_i \in L_B$ that is the initial block label. In other words, the values of the I_i variables together encode π_0. Using this input, the initial values of the block label $block_i$ variables are calculated to conform to our block representation with leaders. Furthermore, in the initialisation, executed on n processors, $waiting_i$ = false for all i that are not used as block label, and true otherwise.

3.2.3 The algorithm

We provide our first PRAM algorithm in Algorithm 2. The PRAM is started with $\max(n, m)$ processors. These processors are used for transitions, states and blocks.

The algorithm performs initialisation (lines 1–6) using n processors, where each block selects a new leader (lines 3–4). In line 3, we exploit the feature of a CRCW PRAM that only one write will succeed when multiple processors try to write to the same memory location. In line 4, the write that has succeeded is the same for each block, ensuring that the leader is one of its own states. Next, the algorithm sets the initial block to waiting. Subsequently, the algorithm enters a single loop that we explain in three separate parts.

Splitter selection (lines 8–14), using n processors. Every variable $mark_i$ is set to false. After this, every processor with index i will check $waiting_i$.[1] If block i is marked waiting, the processor tries to write i in the variable C. If multiple write accesses to C happen concurrently in this iteration, then according to the arbitrary PRAM model

[1] Here processor i is used to both directly go over state i and block i. Checking $waiting_{block_i}$ would also be correct. However, we can directly access the blocks using n processors since there are a maximum of n blocks.

(see Sect. 2), only one process j will succeed in writing, setting $C := j$ as the splitter in this iteration.

Mark states (lines 15–17), using m processors. Every processor i is responsible for the transition $s_i \rightarrow t_i$ and checks if t_i ($target_i$) is in the current block C (line 15). If this is the case the processor writes true to $mark_{source_i}$ where $source_i$ is s_i. This mark now indicates that s_i reaches block C.

Splitting blocks (lines 18–26), using n processors. Every processor i compares the mark of state i, i.e. $mark_i$, with the mark of the leader of the block in which state i resides, i.e. $mark_{block_i}$ (line 20). If the marking is different, state i has to be split from $block_i$ into a new block. At line 21, a new leader is elected among the states that form the newly created block. The index of this leader is stored in $new\_leader_{block_i}$. The block $block_i$ is set to be *waiting* (line 22). After that, all involved processors update the block index for their state (line 23) and indicate that the new blocks should be checked as splitter by adding the blocks to the *waiting* set (line 24).

The steps of the program are illustrated in Fig. 1. The notation B_{s_i} refers to a block containing all states that have state s_i as their block leader. In the figure on the left, we have two blocks B_{s_1} and B_{s_4}, of which at least B_{s_4} is marked as waiting. Block B_{s_4} is selected to be splitter, i.e. $C = B_{s_4}$ at line 12 of Algorithm 2. In the figure in the middle, $mark_i$ is set to true for each state i that can reach B_{s_4} (line 16). Finally, block B_{s_4} is removed from waiting (line 19), all states compare their mark with the leader's mark, and the processor working on state s_3 discovers that the mark of s_3 is different from the mark of s_1, so s_3 is elected as leader of the new block B_{s_3} at line 21 of Algorithm 2. Both B_{s_1} and B_{s_3} are set to waiting (lines 22 and 24).

The algorithm repeats execution of the **while**-loop until no blocks are marked waiting.

Algorithm 2 The algorithm for RCPP for each processor P_i in the PRAM

```
1: if $i < n$ then
2:     $waiting_i$ := false
3:     $new\_leader_{I_i} := i$                ▷ Leader election
4:     $block_i := new\_leader_{I_i}$
5:     $waiting_{block_i}$ := true
6: end if
7: do
8:     $C := \bot$
9:     if $i < n$ then
10:        $mark_i$ := false
11:        if $waiting_i$ then
12:            $C := i$
13:        end if
14:    end if
15:    if $i < m$ and $block_{target_i} = C$ then
16:        $mark_{source_i}$ := true
17:    end if
18:    if $i < n$ and $C \neq \bot$ then
19:        $waiting_C$ := false
20:        if $mark_i \neq mark_{block_i}$ then
21:            $new\_leader_{block_i} := i$
22:            $waiting_{block_i}$ := true
23:            $block_i := new\_leader_{block_i}$
24:            $waiting_{block_i}$ := true
25:        end if
26:    end if
27: while $C \neq \bot$
```

A partition π induces a relation in which the blocks are the equivalence classes. For an input partition π_0 we call the relation induced by the coarsest refinement of π_0 that is a bisimulation relation $\leftrightarrows_{\pi_0}$.

We now prove that Algorithm 2 indeed solves RCPP. We first introduce Lemma 3.1 which is invariant throughout the execution and expresses that states which are related by $\leftrightarrows_{\pi_0}$ are never split into different blocks. This lemma implies that if a refinement forms a bisimulation relation, it is the coarsest.

3.3 Correctness

The $block_i$ list in the common memory at the start of iteration k defines a partition π_k where states $s \in S$ with equal block labels $block_i$ form the blocks:

$$\pi_k = \{\{s \in S \mid block_s = s'\} \mid s' \in S\} \setminus \{\emptyset\}$$

A run of the program produces a sequence $\pi_0, \pi_1, \ldots$ of partitions. Partition π_k is a refinement of every partition $\pi_0, \pi_1, \ldots, \pi_{k-1}$, since blocks are only split and never merged.

Lemma 3.1 *Let S be the input set of states, $\rightarrow : S \times S$ the input relation and π_0 the input partition. Let $\pi_1, \pi_2, \ldots$ be the sequence of partitions produced by Algorithm 2, then for all initial blocks $B \in \pi_0$, states $s, t \in B$ and iterations $k \in \mathbb{N}$:*

$$s \leftrightarrows_{\pi_0} t \implies \exists B' \in \pi_k.\, s, t \in B'$$

Proof This is proved by induction on k. In the base case, π_0, this is true by default. Now assume for a particular $k \in \mathbb{N}$ that the property holds. We know that the partition π_{k+1} is obtained by splitting with respect to a block $C \in \pi_k$. For two states $s, t \in S$ with $s \leftrightarrows_{\pi_0} t$ we know that s and t are in the same block in π_k. In the case that both s and t do not reach C, then $mark_s = mark_t =$ false. Since they were in the same block, they will be in the same block in π_{k+1}.

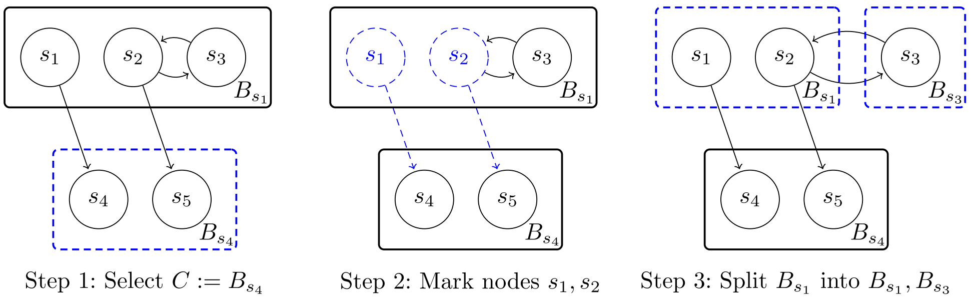

s_1
s_2
s_3
B_{s_1}
s_4
s_5
B_{s_4}
Step 1: Select $C := B_{s_4}$
Step 2: Mark nodes s_1, s_2
B_{s_3}
Step 3: Split B_{s_1} into B_{s_1}, B_{s_3}

Fig. 1 One iteration of Algorithm 2

Now consider the case that at least one of the states is able to reach C. Without loss of generality say that s is able to reach C. Then there is a transition $s \to s'$ with $s' \in C$. By Definition 2.2, there exists $t' \in S$ such that $t \to t'$ and $s' \leftrightarrows_{\pi_0} t'$. By the induction hypothesis we know that since $s' \leftrightarrows_{\pi_0} t'$, s' and t' must be in the same block in π_k, i.e. t' is in C. This witnesses that t is also able to reach C and we must have $mark_s = mark_t =$ true. Since the states s and t are both marked and are in the same block in π_k, they will remain in the same block in π_{k+1}. □

Lemma 3.2 *Let S be the input set of states with $\to: S \times S$, $L_B = S$ the block labels, and π_n the partition stored in the memory after the termination of Algorithm 2. Then the relation induced by π_n is a bisimulation relation.*

Proof Since the program finished, we know that for all block indices $i \in L_B$ we have $waiting_i =$ false. For a block index $i \in L_B$, $waiting_i$ is set to false if the partition π_k, after iteration k, is stable under the block with index i and set to true if it is split. So, by Fact 2.3, we know π_n is stable under every block B, hence stable. Next, we prove that a stable partition is a bisimulation relation.

We show that the relation R induced by π_n is a bisimulation relation. Assume states $s, t \in S$ with sRt are in block $B \in \pi_n$. Consider a transition $s \to s'$ with $s' \in S$. State s' is in some block $B' \in \pi_n$, and since the partition is stable under block B', and s is able to reach B', by the definition of stability, we know that t is also able to reach B'. Therefore, there must be a state $t' \in B'$ such that $t \to t'$ and $s'Rt'$. Finally, by the fact that R is an equivalence relation we know that R is also symmetric; therefore, it is a bisimulation relation. □

Theorem 3.3 *The partition resulting from executing Algorithm 2 forms the coarsest relational partition for a set of states S and a transition relation $\to: S \times S$, solving RCPP.*

Proof By Lemma 3.2, the resulting partition is a bisimulation relation. Lemma 3.1 implies that it is the coarsest refinement which is a bisimulation. □

3.4 Complexity analysis

Every step in the body of the while-loop can be executed in constant time. So the asymptotic complexity of this algorithm is given by the number of iterations.

Theorem 3.4 *RCPP on an input with m transitions and n states is solved by Algorithm 2 in $\mathcal{O}(n)$ time using $\max(n, m)$ CRCW PRAM processors.*

Proof In iteration $k \in \mathbb{N}$ of the algorithm, let us call the total number of blocks $N_k \in \mathbb{N}$ and the number of blocks marked as waiting $U_k \in \mathbb{N}$. Initially, $N_0 = U_0 = |\pi_0|$. In every iteration k, a number of blocks $l_k \in \mathbb{N}$ is split, resulting in l_k new blocks, so the new total number of blocks at the end of iteration k is $N_{k+1} = N_k + l_k$.

First the current block C in iteration k which was marked as waiting is no longer waiting which causes the number of waiting blocks to decrease by one. In this iteration k, the set of l_k blocks $B_1, \ldots, B_{l_k}$ are split, resulting in l_k newly created blocks. These l_k blocks are all marked as waiting. A number of blocks $l'_k \leq l_k$ of the blocks $B_1, \ldots B_{l_k}$ were not yet marked as waiting and are now set to waiting. The current block C is possibly one of these l'_k blocks which were not marked as waiting and are now waiting again. The total number of blocks marked as waiting at the end of iteration k is $U_{k+1} = U_k + l_k + l'_k - 1$.

For all $k \in \mathbb{N}$, in iteration k we calculate the total number of blocks $N_k = |\pi_0| + \sum_{i=0}^{k-1}(l_i)$ and waiting blocks $U_k = |\pi_0| - k + \sum_{i=0}^{k-1}(l_i + l'_i)$. The number of iterations is given by $k = \sum_{i=0}^{k-1}(l_i + l'_i) - U_k + |\pi_0|$. By definition, $l'_i \leq l_i$, and the total number of newly created blocks is $\sum_{i=0}^{k-1}(l_i) = N_k - |\pi_0|$; hence, $\sum_{i=0}^{k-1}(l_i + l'_i) \leq 2\sum_{i=0}^{k-1}(l_i) \leq 2N_k - 2|\pi_0|$. The number of waiting blocks is always positive, i.e. $U_k \geq 0$, and the total number of blocks can never be larger than the number of states, i.e. $N_k \leq n$, so the total number of iterations z is bounded by $z \leq 2N_z - |\pi_0| \leq 2n - |\pi_0|$. □

4 Bisimulation coarsest refinement problem

In this section, we extend our algorithm to the bisimulation coarsest refinement problem (BCRP), i.e. to LTSs with multiple action labels.

Solving BCRP can in principle be done by translating an LTS to a Kripke structure, for instance by using the method described in [16]. This translation introduces a new state for every transition, resulting in a Kripke structure with $n + m$ states. If the number of transitions is significantly larger than the number of states, then the number of iterations of our algorithm increases undesirably.

4.1 The PRAM algorithm

Instead of introducing more states, we introduce multiple marks per state, but in total we have no more than m marks. For each state s, we use a mark variable for each different outgoing action label relevant for s, i.e. for each a for which there is a transition $s \xrightarrow{a} t$ to some state t. Each state may have a different set of outgoing action labels and thus a different set of marks. Therefore, we first perform a preprocessing procedure in which we group together states as blocks in the initial partition that have the same set of outgoing action labels. This is valid, since two bisimilar states must have the same outgoing actions. Since in the sequence of produced partitions each partition is a refinement of the previous one, the algorithm has the invariant that two states of the same block have the same set of action labels. For the extended algorithm, we need to maintain extra information in addition to the information needed for Algorithm 2. For an input LTS $A = (S, Act, \rightarrow)$ with n states and m transitions, the extra information is:

1. Each action label has an index $a \in \{0, \ldots, |Act| - 1\}$.
2. The following is stored in lists of size m, for each transition $s \xrightarrow{a} t$ with transition index $i \in \{0, \ldots, m - 1\}$:
   (a) $a_i := a$, the action label of transition i,
   (b) $order_i : \mathbb{N}$, the order of this action label, with respect to the source state s. For example, if a state s has the list $[1, 3, 6]$ of outgoing action labels, and transition i has label 3, then $order_i$ is 1 (we start counting from 0).
3. $mark : [\mathbb{B}]$, a list of up to m marks, in which there is a mark for every state, action pair for which it holds that the state has at least one outgoing transition labelled with that action. This list can be interpreted as the concatenation of sublists, where each sublist contains all the marks for one state. For each state $s \in S$, we have:
   (a) $\mathit{off}(s) : \mathbb{N}$, the offset to access the beginning of the sublist of the marks of the state s in $mark$. The positions $mark_{\mathit{off}(s)}$ up to $mark_{\mathit{off}(s+1)}$ contain the sublist of marks for state s. For example, if state s has outgoing transitions with 3 distinct action labels, we know that $\mathit{off}(s+1) \equiv \mathit{off}(s)+3$, and we have 3 marks for state s. We write $mark_{\mathit{off}(s)+order_i}$ to access the mark for transition i which has source state s.
4. $mark\_length : \mathbb{N}$, the length of the mark list. This allows us to reset all marks in constant time using $mark\_length$ processors. This number is not larger than the number of transitions ($mark\_length \leq m$).
5. In a list of size n, we store for each state $s \in S$ a variable $split_s : \mathbb{B}$. This indicates if the state will be split off from its block.

With this extra information, we can modify Algorithm 2 to work with labels. The new version is given in Algorithm 3. The changes involve the following:

1. Lines 7–9: Reset the *mark* list.
2. Line 11: Reset the *split* list.
3. Line 17: When marking the transitions, we do this for the correct action label, using $order_i$. Note the indexing into *mark*. It involves the offset for the state $source_i$ and $order_i$.
4. Lines 19–21: We tag a state to be split when it differs for any action from the block leader.
5. Line 24: If a state was tagged to be split in the previous step, it should split from its leader.
6. Line 29: If any block was split, the partition may not be stable w.r.t. the splitter.

To use Algorithm 3, two preprocessing steps are required. First, we need to partition the states w.r.t. their set of outgoing action labels. This can be done with a modified version of Algorithm 2, which performs one iteration for each action label. For the second preprocessing step, we need to gather the extra information that is needed in Algorithm 3. This is done via sorting the action labels and subsequently performing some parallel segmented (prefix) sums [17]. In total, the preprocessing takes $\mathcal{O}(|Act| + \log m)$ time. For details how this is implemented, see Appendix A.

4.2 Complexity and correctness

The correctness of this algorithm follows directly from the arguments in Sect. 3.3.

Algorithm 3 The algorithm for BCRP, the lines highlighted differ from Algorithm 2.

```
1: if $i < n$ then
2:   $waiting_i$ := false
3:   $waiting_{block_i}$ := true
4: end if
5: do
6:   $C := \bot$
7:   if $i < mark\_length$ then
8:     $mark_i$ := false
9:   end if
10:   if $i < n$ then
11:     $split_i$ := false
12:     if $waiting_i$ then
13:       $C := i$
14:     end if
15:   end if
16:   if $i < m$ and $block_{target_i} = C$ then
17:     $mark_{off(source_i)+order_i}$ := true
18:   end if
19:   if $i < m$ and $mark_{off(source_i)+order_i} \neq mark_{off(block_{source_i})+order_i}$ then
20:     $split_{source_i}$ := true
21:   end if
22:   if $i < n$ & $C \neq \bot$ then
23:     $waiting_C$ := false
24:     if $split_i$ then
25:       $new\_leader_{block_i} := i$
26:       $waiting_{block_i}$ := true
27:       $block_i := new\_leader_{block_i}$
28:       $waiting_{block_i}$ := true
29:       $waiting_C$ := true
30:     end if
31:   end if
32: while $C \neq \bot$
```

Theorem 4.1 *Given an LTS $A = (S, \rightarrow, Act)$, the partition π resulting from executing Algorithm 3 is the coarsest partition that induces a bisimulation relation, solving BCRP.*

Proof The proof goes in a completely analogous way to the proof of Theorem 3.3.

We can modify the proof of Lemma 3.1, for it to still hold in the setting of labels. The *mark* for each action label is the same for two bisimilar states. This means bisimilar states have the same value for *split*. Therefore, they will always be split into the same block together. Thus, bisimilar states will always be in the same block.

The proof of Lemma 3.2 still works in the setting with labels, proving that the partition π is stable with respects to all blocks $B \in \pi$, and thereby that π induces a bisimulation relation.

Thus, these two facts imply that π is the coarsest partition inducing a bisimulation relation. □

For Algorithm 3, we need to prove why it takes a linear number of steps to construct the final partition. This is subtle, as an iteration of the algorithm does not necessarily remove a block from the waiting set.

Theorem 4.2 *BCRP on an input with m transitions and n states is solved by Algorithm 3 in $\mathcal{O}(n + |Act|)$ time using $\max(n, m)$ CRCW PRAM processors.*

Proof The total preprocessing takes $\mathcal{O}(|Act| + \log m)$ steps, after which the **while**-loop will be executed on a partitioning π_0 which was the result of the preprocessing on the partition $\{S\}$. Every iteration of the **while**-loop is executed in constant time. Using the structure of the proof of Theorem 3.4, we derive a bound on the number of iterations.

At the start of iteration $k \in \mathbb{N}$ the total number of blocks and waiting blocks are $N_k, U_k \in \mathbb{N}$, initially $U_0 = N_0 = |\pi_0|$. In iteration k, a number l_k of blocks is split into two blocks, resulting in l_k new blocks, meaning that $N_{k+1} = N_k + l_k$. All new l_k blocks are marked as waiting and a number $l'_k \leq l_k$ of the old blocks that are split were not waiting at the start of iteration k and are now marked as waiting. If $l_k = l'_k = 0$ there are no blocks split and the current block C is no longer marked as waiting. We indicate this with a variable c_k: $c_k = 1$ if $l_k = 0$, and $c_k = 0$, otherwise. The total number of iterations up to iteration k in which no block is split is given by $\sum_{i=0}^{k-1} c_i$. The number of iterations in which at least one block is split is given by $k - \sum_{i=0}^{k-1} c_i$.

If in an iteration k at least one block is split, the total number of blocks at the end of iteration k is strictly higher than at the beginning; hence, for all $k \in \mathbb{N}$, $N_k \geq k - \sum_{i=0}^{k-1} c_i$. Hence, $N_k + \sum_{i=0}^{k-1} c_i$ is an upper bound for k.

We derive an upper bound for the number of iterations in which no blocks are split using the total number of waiting blocks. In iteration k there are $U_k = \sum_{i=0}^{k-1}(l_i + l'_i) - \sum_{i=0}^{k-1} c_i + |\pi_0|$ waiting blocks. Since the sum of newly created blocks $\sum_{i=0}^{k-1}(l_i) = N_k - |\pi_0|$ and $l'_i \leq l_i$, the number of blocks marked as waiting U_k is bounded by $2N_k - \sum_{i=0}^{k-1} c_i - |\pi_0|$. Since $U_k \geq 0$ we have the bound $\sum_{i=0}^{k-1} c_i \leq 2N_k - |\pi_0|$. This gives the bound on the total number of iterations $z \leq 3N_z - |\pi_0| \leq 3n - |\pi_0|$.

With the time for preprocessing this makes the run time complexity $\mathcal{O}(n + |Act| + \log m)$. Since the number of transitions m is bounded by $|Act| \times n^2$, this simplifies to $\mathcal{O}(n + |Act|)$. □

4.3 Experimental results

In this subsection, we discuss the results of our implementation of Algorithm 3. Note that this implementation is not optimised for the specific hardware it runs on, since the goal of this article is to provide a generic parallel algorithm. This implementation is purely a proof of concept, to show that our algorithm can be mapped to contemporary hardware and to understand how the algorithm scales with the size of the input.

The implementation targets GPUs since a GPU closely resembles a PRAM and supports a large amount of parallelism. The algorithm is implemented in CUDA C++ version 11.1 with use of the Thrust library.[2] As input, we chose all benchmarks of the VLTS benchmark suite[3] for which the implementation produces a result within 10 minutes. The VLTS benchmarks are LTSs that have been obtained from various case studies derived from real concurrent system models.

The experiments were run on an NVIDIA Titan RTX with 24 GB memory and 72 Streaming Multiprocessors, each supporting up to 1024 threads in flight. Although this GPU supports 73,728 threads in flight, it is very common to launch a GPU program with one or even several orders of magnitude more threads, in particular to achieve load balancing between the Streaming Multiprocessors and to hide memory latencies. In fact, the performance of a GPU program usually relies on that many threads being launched.

Table 1 shows the results of the experiments we conducted. The $|Act|$ column corresponds to the number of different action labels. The $|Blocks|$ column indicates the number of different blocks at the end of the algorithm, where each block contains only bisimilar states. With $\#It$, we refer to the number of **while**-loop iterations that were executed (see Algorithm 3). The number of states and transitions can be derived from the benchmark name. In the benchmark 'X_N_M,' $N * 1000$ is the number of states and $M * 1000$ is the number of transitions. The T_{pre} give the preprocessing times in milliseconds, which includes the memory transfers to the GPU, sorting the transitions and initial partitioning. The T_{alg} give the times of the core algorithm, in milliseconds. The T_{BCRP} is the sum of the preprocessing and the algorithm, in milliseconds. We have not included the loading times for the files and the first CUDA API call that initialises the device. We ran each benchmark 10 times and took the averages. The standard deviation of the total times varied between 0% and 3% of the average; thus, 10 runs are sufficient. All the times are rounded with respect to the standard error of the mean.

We see that the bound as proved in Sect. 4.2 ($k \leq 3n$) is indeed respected, $\#It/n$ is at most 2.20, and in most cases below that. The number of iterations is tightly related to the number of blocks that the final partition has, the $\#It/|Blocks|$ column varies between 1.00 and 2.53. This can be understood by the fact that each iteration either splits one or more blocks or marks a block as non-waiting, and all blocks are waiting at least once. This also means that for certain LTSs the algorithm scales better than linearly in n. The preprocessing often takes the same amount of time (about a few milliseconds). Exceptions are those cases with a large number of actions and/or transitions, for instance Vasy_25_25 or Vasy_574_13561.

Concerning the run times, it is not true that each iteration takes the same amount of time. A GPU is not a perfect PRAM machine. There are two key differences. Firstly, we suspect that the algorithm is memory bound since it is performing a limited amount of computations. The memory accesses are irregular, i.e. random, which caches can partially compensate, but for sufficiently large n and m, the caches cannot contain all the data. This means that as the LTSs become larger, memory accesses become relatively slower. Secondly, at a certain moment, the maximum number of threads that a GPU can run in parallel is achieved, and adding more threads will mean more run time. These two effects can best be seen in the $T_{alg}/\#It$ column, which corresponds to the time per iteration. The values are around 0.02 up to 300, 000 transitions, but for a larger number of states and transitions, the amount of time per iteration increases.

4.4 Experimental comparison

We compared our implementation (BCRP) with an implementation of the algorithm by Lee and Rajasekaran (LR) [12] on GPUs, and the optimised GPU implementation by Wijs based on *signature-based* bisimilarity checking [18], with *multi-way splitting* (Wms) and with *single-way splitting* (Wss) [14]. Multi-way splitting indicates that a block is split into multiple blocks at once, which is achieved by computing a signature for each state in every partition refinement iteration, and splitting each block off into sets of states, each containing all the states with the same signature. The signature of a state is derived from the labels of the blocks that this state can reach in the current partition. Note that we are not including comparisons with CPU bisimulation checking tools; the fact that those tools run on completely different hardware makes a comparison problematic, and such a comparison does not serve the purpose of evaluating the feasibility of implementing Algorithm 3. Optimising our implementation to make it competitive with CPU tools is planned for future work.

The running times of the different algorithms can be found in Table 2. Similarly to our previous benchmarks, the algorithms were run 10 times on the same machine using the same VLTS benchmark suite with a time-out of 10 minutes. In some cases, the nondeterministic behaviour of the algorithms Wms and Wss led to high variations in the runs. In cases where the standard error of the mean was more than 5% of the mean value, we have added the standard error in Table 2 in parentheses. Furthermore, all the results are rounded with respect to the standard error of the mean. As a preprocessing step for the LR, Wms and Wss algorithms the input LTSs need to be sorted. We did not include this in the

[2] The source code can be found at https://github.com/sakehl/gpu-bisimulation.

[3] https://cadp.inria.fr/resources/vlts/.

Table 1 Benchmark results for BCRP (Algorithm 3) on a GPU, times (T) are in ms

Benchmark name	$\|Act\|$	$\|Blocks\|$	$\#It$	T_{pre}	T_{alg}	$\#It/n$	$\#It/\ \|Blocks\|$	T_{BCRP}/n	$T_{alg}/\#It$	T_{BCRP}
Vasy_0_1	2	9	16	0.50	0.37	0.06	1.78	0.003	0.023	0.87
Cwi_1_2	26	1132	2786	0.63	56.5	1.43	2.46	0.029	0.020	57.1
Vasy_1_4	6	28	45	0.56	1.01	0.04	1.61	0.001	0.022	1.58
Cwi_3_14	2	62	122	0.63	2.68	0.03	1.97	0.001	0.022	3.30
Vasy_5_9	31	145	193	0.84	4.22	0.04	1.33	0.001	0.022	5.06
Vasy_8_24	11	416	664	0.70	13.9	0.07	1.59	0.002	0.021	15
Vasy_8_38	81	219	319	1.12	6.64	0.04	1.46	0.001	0.021	7.76
Vasy_10_56	12	2112	3970	0.73	82.0	0.37	1.88	0.008	0.021	82.7
Vasy_18_73	17	4087	6882	1.01	142	0.37	1.68	0.008	0.021	143
Vasy_25_25	25,216	25,217	25,218	159	519	1.00	1.00	0.027	0.021	678
Vasy_40_60	3	40,006	87,823	0.87	1810	2.20	2.20	0.045	0.021	1811
Vasy_52_318	17	8142	15,985	2.52	338	0.31	1.96	0.007	0.021	340
Vasy_65_2621	72	65,536	98,730	12.2	10,050	1.51	1.51	0.154	0.102	10,060
Vasy_66_1302	81	66,929	91,120	6.70	5745	1.36	1.36	0.086	0.063	5752
Vasy_69_520	135	69,754	113,246	4.13	3780	1.62	1.62	0.054	0.033	3780
Vasy_83_325	211	83,436	148,012	4.41	3093	1.77	1.77	0.037	0.021	3097
Vasy_116_368	21	116,456	210,537	2.50	5900	1.81	1.81	0.051	0.028	5900
Cwi_142_925	7	3410	5118	4.85	238	0.04	1.50	0.002	0.047	243
Vasy_157_297	235	4289	9682	4.58	201	0.06	2.26	0.001	0.021	206
Vasy_164_1619	37	1136	1630	8.34	125	0.01	1.43	0.001	0.077	134
Vasy_166_651	211	83,436	145,029	6.13	5710	0.87	1.74	0.034	0.039	5720
Cwi_214_684	5	77,292	149,198	3.58	6948	0.70	1.93	0.032	0.047	6952
Cwi_371_641	61	33,994	85,858	4.72	4050	0.23	2.53	0.011	0.047	4050
Vasy_386_1171	73	113	199	7.38	14.0	0.00	1.76	0.000	0.070	21
Cwi_566_3984	11	15,518	23,774	16.0	3707	0.04	1.53	0.007	0.156	3723
Vasy_574_13561	141	3577	5860	71.5	3770	0.01	1.64	0.007	0.643	3841
Vasy_720_390	49	3292	3782	3.97	143	0.01	1.15	0.0002	0.038	147
Vasy_1112_5290	23	265	365	24.0	99.3	0.0003	1.38	0.0001	0.272	123
Cwi_2165_8723	26	31,906	66,132	37.0	23,660	0.03	2.07	0.011	0.358	23,700
Cwi_2416_17605	15	95,610	152,099	64.1	96,400	0.06	1.59	0.040	0.634	96,500
Vasy_6020_19353	511	7168	12,262	221	11,690	0.002	1.71	0.002	0.954	11,910
Vasy_6120_11031	125	5199	10,014	74.0	6763	0.002	1.93	0.001	0.675	6837
Vasy_8082_42933	211	408	660	281	1149	0.0001	1.62	0.0002	1.739	1429

times, nor the reading of files and the first CUDA API call (which initialises the GPU).

This comparison confirms the expectation that our algorithm in all cases (except the small LTS Cwi_1_2) outperforms LR and that LR is not suitable for massive parallel devices such as GPUs.

Furthermore, the comparison demonstrates that in most cases our algorithm (BCRP) outperforms Wss. In some benchmarks (Cwi_1_2, Cwi_214_684, Cwi_2165_8723 and Cwi_2416_17605) Wss is more than twice as fast, but in 16 other cases our algorithm is more than twice as fast. The last comparison shows that our algorithm does not outperform Wms. Wms employs multi-way splitting which is known to be very effective in practice. Furthermore, contrary to our implementation, Wms is optimised for GPUs while the focus of the current work is to improve the theoretical bounds and describe a general algorithm.

In order to understand the difference in performance between Wms and our algorithm better, we analysed the complexity of Wms [14]. In general this algorithm is quadratic in time, and the linearity claim in [14] depends on the assumption that the fan-out of ‘practical’ transition systems is bounded, i.e. every state has no more than c outgoing transitions for c a (low) constant. We designed the

Table 2 Comparing the total time of our BCRP algorithm with other algorithms, times (T) are in ms

Benchmark name	T_{BCRP}	T_{LR}	T_{Wss}	T_{Wms}
Vasy_0_1	0.87	2.29	0.49	0.45
Cwi_1_2	57.1	17	18.8	21.8
Vasy_1_4	1.58	4.78	1.68	0.62
Cwi_3_14	3.30	60	3.80	3.72
Vasy_5_9	5.06	134	35.3	3.45
Vasy_8_24	15	277	31.5	3.03
Vasy_8_38	7.76	127	35.1	5.94
Vasy_10_56	82.7	860	40.9	4.6(0.2)
Vasy_18_73	143	1354	211	21.7
Vasy_25_25	678	21,960	t.o.	416
Vasy_40_60	1811	17,710	1290	1230
Vasy_52_318	340	11,855	368	152(20)
Vasy_65_2621	10,060	t.o.	27,000	1230
Vasy_66_1302	5752	480,600	20,450	240(20)
Vasy_69_520	3780	94,800	16,090	35.4
Vasy_83_325	3097	57,190	21,500	5880
Vasy_116_368	5900	80,900	6360	2930
Cwi_142_925	243	3363	220(30)	140(20)
Vasy_157_297	206	1058	1240	579
Vasy_164_1619	134	8173	470(30)	46.8
Vasy_166_651	5720	80,210	29,660	9560
Cwi_214_684	6952	19,250	440(30)	450(50)
Cwi_371_641	4050	26,940	6970	1548
Vasy_386_1171	21	334	30.6	34.8
Cwi_566_3984	3723	98,200	6700	2200 (200)
Vasy_574_13561	3841	144,810	11,700	1853
Vasy_720_390	147	2454	1633	183
Vasy_1112_5290	123	4570	293	36.8
Cwi_2165_8723	23,700	140,170	9700	1965
Cwi_2416_17605	96,500	257,200	16,300 (1,100)	15,300
Vasy_6020_19353	11,910	107,900	34,000 (2000)	19,230
Vasy_6120_11031	6837	55,750	7010	1280
Vasy_8082_42933	1429	17,272	5530	2030

transition systems Fan_out_n for $n \in \mathbb{N}^+$ to illustrate the difference. The LTS $Fan\_out_n = (S, \{a, b\}, \rightarrow)$ has n states: $S = \{0, \ldots, n-1\}$. The transition function contains $i \xrightarrow{a} i+1$ for all states $1 < i < n-1$. Additionally, from state 0 and 1 there are transitions to every state: $0 \xrightarrow{b} i$, $1 \xrightarrow{b} i$ for all $i \in S$. This LTS has n states, $3n-3$ transitions and a maximum out degree of n transitions.

Fig. 2 shows the results of calculating the bisimulation equivalence classes for Fan_out_n, with Wms and BCRP. It is clear that the run time for Wms increases quadratically as the number of states grows linearly, already becoming untenable for a small number of states. On the other hand, in conformance with our analysis, our algorithm scales linearly.

5 Branching bisimilarity

When systems get more complex, abstractions are an essential tool to gain insight into the behaviour of the system. Often a special action label τ is used to model internal behaviour in LTSs. Branching bisimilarity, as introduced by Van Glabbeek

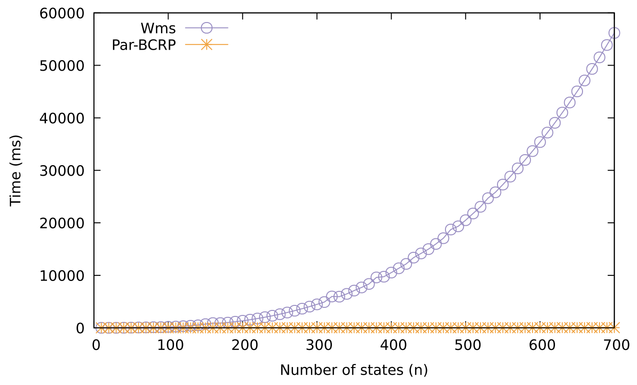


Fig. 2 Run times of BCRP and Wms on the LTS Fan_out_n

and Weijland [19] is an equivalence that takes these internal actions into account. It is defined as follows.

Definition 5.1 (*Branching bisimulation* [19, Definition 2.4]) Given an LTS $A = (S, \rightarrow, Act)$ a relation $R \subseteq S \times S$ is called a *branching bisimulation* relation iff it is symmetric and for all states $s, t \in S$ such that sRt and for all actions $a \in Act$: if $s \xrightarrow{a} s'$ then either

- $a = \tau$ and $s'Rt$, or
- there is a sequence $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'$ of zero or more τ-transitions such that sRt', $t' \xrightarrow{a} t''$ and $s'Rt''$.

Branching bisimilarity is the coarsest branching bisimulation relation, and we denote this as $\leftrightarrow_b$.

The problem we address in this section is taking into account branching bisimulation relations. We define the branching bisimulation refinement problem (**branching-BCRP**) as follows:

Input: An LTS $M = (S, Act, \rightarrow)$ with a dedicated internal action $\tau \in Act$.

Output: The partition π of S, which is the coarsest partition, i.e. has the smallest number of blocks, that forms a branching bisimulation relation.

This section explains how to adapt our parallel algorithm for strong bisimilarity (BCRP) to decide branching bisimilarity (branching-BCRP) in $O(n + |Act|)$ parallel time using $\max(n^2, m, |Act|n)$ parallel processors.

The section is structured as follows: In Sect. 5.1, we recall some preliminaries specific to branching bisimulations and establish the notation we use throughout this section. Next, in Sect. 5.2, we explain key ingredients from the sequential algorithm by Groote and Vaandrager [20]. After that, in Sect. 5.3, we explain the key modifications in a parallel setting. Then, in Sect. 5.4, we give the details of the algorithm. Lastly, in Sect. 5.5, we prove the correctness and complexity bounds of the parallel algorithm.

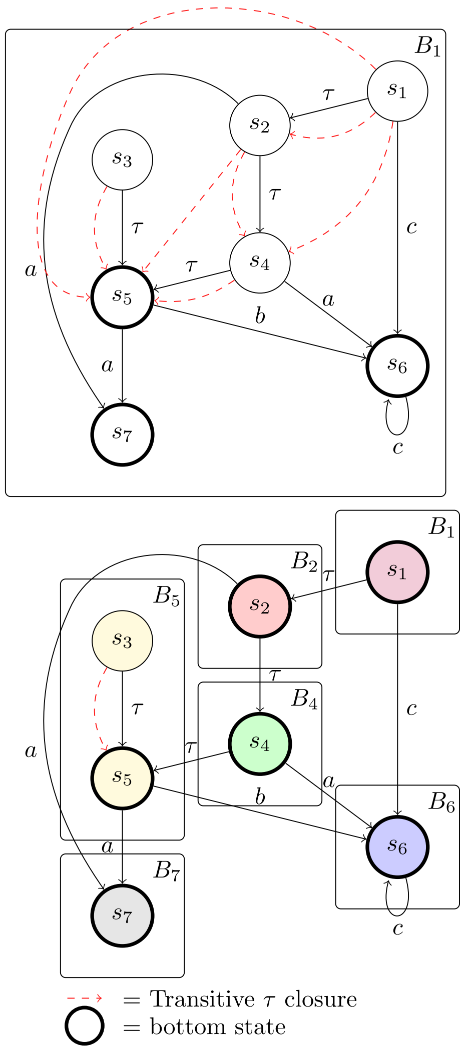


Fig. 3 Example LTS with internal transitions. At the top, all states are in one block. At the bottom, we grouped the states by branching bisimilar

5.1 Preliminaries on branching bisimulation

For the next section, we assume that in any LTS the internal τ-action is available, so $\tau \in Act$.

Fig. 3 shows an LTS at the top with all states in one block. We group branching bisimilar states at the bottom, where we denote the branching bisimilarity equivalence classes B_1 up to B_7.

Next, we define bottom states, which are states that do not have an outgoing τ-transition to another state in the same block.

Definition 5.2 (*Bottom state* [20]) Let $A = (S, Act, \rightarrow)$ be an LTS, π a partition of S, and $s \in B \in \pi$ where B is a block. We call s a *bottom* state iff there is no $s' \in B$ such that $s \xrightarrow{\tau} s'$.

In Fig. 3, at the top states s_5, s_6 and s_7 are bottom states. In the situation at the bottom every state except s_3 is a bottom state.

We call a τ-transition *silent* when both the source and target state are in the same block.

Definition 5.3 (*Silent τ-transition*) Let $A = (S, Act, \rightarrow)$ be an LTS. Let π be a partition of S. We call a τ-transition $s \xrightarrow{\tau} s'$ *silent* with respect to π iff $s, s' \in B \in \pi$.

So in Fig. 3 on the top side all τ-transitions are silent and on the bottom only $s_3 \xrightarrow{\tau} s_5$ is silent.

Below we define the *silent transitive closure* of $\xrightarrow{\tau}$ transitions, denoted by $\Rightarrow_\pi$, which relates each state s to all states that are reachable starting from s via a path of zero or more silent τ-transitions.

Definition 5.4 (*Transitive and reflexive τ-closure*) Let $A = (S, Act, \rightarrow)$ be an LTS. Let π be a partition of S. We denote $s \Rightarrow_\pi s'$ iff s can reach s' performing zero or more silent τ-transitions $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s'$ where for some $B \in \pi$ all $s, s', s_1, \ldots, s_n \in B$. We call $\Rightarrow_\pi$ the *transitive and reflexive τ-closure* with respect to π.

In Fig. 3, we indicate the transitive τ-closure for the two different partitions. To improve readability, we do not show the reflexive part.

The following lemma expresses that not only the first state t and the last state t' on a silent τ–path $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'$ are related but every intermediate state is related to t.

Lemma 5.5 (cf. [19, Lemma 2.5]) *Let $A = (S, \rightarrow, Act)$ be an LTS, and let for some $n > 0$, $t_0 \xrightarrow{\tau} t_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} t_n$ be a path of τ-transitions in A with $t_0 \leftrightarrow_b t_n$. Then for every $0 \leq i \leq n : t_0 \leftrightarrow_b t_i$.*

In order to talk about the directly reachable non-silent transitions of a state, we define the direct markings of a state.

Definition 5.6 (*Direct markings*) Let $A = (S, Act, \rightarrow)$ be an LTS, and π a partition of S. Let $s \in S$ and $B \in \pi$. We define the *direct markings* of s to B as:

$$dir(s, B) = \{a \mid \exists t \in B. s \xrightarrow{a} t \wedge (a \neq \tau \vee s \notin B)\}.$$

For branching bisimulations, the stability condition changes slightly. Given an LTS $A = (S, Act, \rightarrow)$ and π a partition of S. A block B is called *stable modulo branching bisimulation* with respect to a block $B' \in \pi$, iff for all actions $a \in Act$ either for all states $s \in B$ there is an $s' \in B$ such that $s \Rightarrow_\pi s'$ and $a \in dir(s', B')$, or for all states $s \in B$, $a \notin dir(s, B')$.

A partition π is again called stable under a block B' iff each block $B \in \pi$ is stable under B'. The partition π is called stable iff it is stable under all its own blocks $B \in \pi$.

5.2 The sequential algorithm

In this section, we explain the key ideas used in the sequential Groote–Vaandrager algorithm [20] to decide branching bisimilarity. In this algorithm, similar to the sequential Kanellakis–Smolka algorithm as explained in Sect. 3.1, a partition is maintained and iteratively refined based on behaviour that distinguishes states. For strong bisimulation, a block is split into the set of states that can directly reach a splitter block. However, in the Groote–Vaandrager algorithm, states are also split if they can reach the splitter by first doing a sequence of silent τ actions. In [20], this initial silent path is called stuttering and comes from the related problem of deciding stuttering equivalence on Kripke structures [21].

Another subtle but essential difference between the two algorithms is that stability modulo branching bisimulation is not necessarily preserved under refinement, i.e. Fact 2.3 may not hold. This is also observed in [20, Lemma 5.2.3] and is addressed by always considering all blocks as potential splitters.

In summary, when solving branching-BCRP the two main modifications we need to address with respect to the parallel algorithm for BCRP.

1. **Modification 1:** A different splitting strategy, that takes into account initial τ-paths.
2. **Modification 2:** After refinement of partition π into π' check that blocks that are stable w.r.t π are still stable for π'.

5.3 The parallel algorithm

This section addresses the necessary modifications in the parallel setting such that we have a linear parallel algorithm to decide branching bisimilarity.

In the strong bisimilarity algorithm, we choose a block as a splitter. We check which actions can reach this splitter for each state by going over all transitions and storing a mark for each action label. We also store a marking for each action label for branching bisimulation, but we must consider the τ-paths. Therefore, to address **Modification 1**, we need to propagate the markings over the τ-paths. For example, in Fig. 3 any marking that state s_5 has can potentially propagate to state s_1, s_2, s_3 and s_4.

This method of propagating information backwards over τ-paths is also used in the sequential algorithms [11,20]. Likewise, the parallel algorithm [14] for branching bisimilarity propagates information from the bottom states backwards over the silent τ-paths. However, this approach is inherently sequential since, in the worst case, we propagate state information over long silent paths before we can calculate a refinement. For a better asymptotic complexity, we calculate a transitive closure in the style of [22,23].

In our parallel algorithm, we maintain the relation $\Rightarrow_\pi$ that is the transitive and reflexive closure of all silent τ-transitions in π. This closure changes throughout the algorithm, but we can update this information in constant parallel time. Lemma 5.9 states that if π refines into π', we only need to check the source and target of a silent τ-path to see whether it remains silent in π'.

For **Modification 2** regarding stability, we use the fact that a block not in the waiting list can only make the partition unstable if a state has become newly bottom, as Lemma 5.8 shows.

A state that is non-bottom can become bottom only once, and this happens if a silent τ-transition becomes non-silent. For example, in Fig. 3 the transition $s_1 \xrightarrow{\tau} s_2$ is non-silent at the bottom in the picture; thus, s_1 became a bottom state at some point.

Finally, we add a further modification. Throughout the algorithm, we ensure that each block has a bottom state as a leader. This is well defined since we consider LTSs with a finite number of states, and in the preprocessing phase, we remove τ-loops, and therefore, each block has at least one bottom state. This modification makes it easier to compare states since bottom states do not get extra marks via silent τ-steps.

5.3.1 Modification 1: splitting

We split blocks by considering all actions a state can reach via the silent transitive τ-closure. A similar approach is used in [20] where the set of states that can reach a certain block with a specific action label after an initial inert τ-path is considered. We initially mark all states with the actions they can take directly to the splitter block. Afterwards, we distribute all the marks via the silent transitive τ-closure. This results in a similar set of states as in [20] but not with respect to a specific action label but with respect to all action labels. Distributing the marks naively would mean we hand out up to $|Act|n^2$ marks since there can be $|Act|$ markings per state and up to n^2 elements in the transitive τ-closure. Thus, we opt only to distribute one element per state if a state should split, thus distributing n^2 elements maximally.

For a partition π of S, we consider a block $B \in \pi$ that acts as a splitter, and we let B_{s_b} denote any block in π that has the bottom state $s_b \in S$ as its leader. We define:

$$
\begin{aligned}
&\mathit{PreSplit}_\pi(B_{s_b}, B) = \\
&\qquad \{s \in B_{s_b} \mid \mathit{dir}(s, B) \nsubseteq \mathit{dir}(s_b, B)\} \cup \\
&\qquad \{s \in B_{s_b} \mid s \text{ is a bottom state and} \\
&\qquad\quad \mathit{dir}(s, B) \nsupseteq \mathit{dir}(s_b, B)\} \\
&\quad \mathit{Split}_\pi(B_{s_b}, B) = \\
&\qquad \{s \in B_{s_b} \mid \exists s' \in \mathit{PreSplit}_\pi(B_{s_b}, B) \\
&\qquad\qquad \text{such that } s \Rightarrow_\pi s'\}
\end{aligned}
$$

Informally $\mathit{PreSplit}_\pi(B_{s_b}, B)$ contains all the states of B_{s_b} that have a behaviour with respect to B that s_b cannot mimic, and it also contains the bottom states that cannot mimic the behaviour of s_b with respect to B. The first means that the state can reach B with an action a which s_b cannot. The second means that s is a bottom state, and s_b can reach B with an action a, which s cannot. The set $\mathit{Split}_\pi(B_{s_b}, B)$ contains all states that have silent τ-transitions to states that are in $\mathit{PreSplit}_\pi$. These states are also necessarily not branching bisimilar to s_b.

The states that are in the set $\mathit{Split}_\pi(B_{s_b}, B)$ are all the states that should split from the block B_{s_b} into a new block. Based on this function we derive a new partition $\mathit{Ref}(\pi, B)$ in which every block B_{s_b} is split into $\mathit{Split}_\pi(B_{s_b}, B)$ and the states not in $\mathit{Split}_\pi(B_{s_b}, B)$

$$
\mathit{Ref}(\pi, B) = \bigcup_{B_{s_b} \in \pi} \{\mathit{Split}_\pi(B_{s_b}, B), (B_{s_b} \setminus \mathit{Split}_\pi(B_{s_b}, B))\} \setminus \emptyset
$$

From the initial partition $\pi_0 = \{S\}$, we iteratively refine the current partition π by finding a splitter for which $\mathit{Ref}(\pi, B) \neq \pi$ and continue based on the partition $\pi' = \mathit{Ref}(\pi, B)$. The correctness relies on the fact that $\mathit{Ref}(\pi, B)$ never splits states that are branching bisimilar and the fact that if the partition π does not induce the relation $\leftrightarrows_b$ then there still is a splitter B such that $\mathit{Ref}(\pi, B) \neq \pi$.

In Sect. 5.5.1, we will prove that this splitting procedure is correct.

5.3.2 Modification 2: stability

The key property (Fact 2.3) does not hold for stability modulo branching bisimilarity. Counterexamples show that refinement can cause blocks to lose stability. See [20, Remark 3.4], where a counterexample is given. In the Groote–Vaandrager algorithm, the authors resolve this by reconsidering all blocks if new bottom states are added [20, Lemma 3.3]. This strategy would result in a quadratic worst-case algorithm in our parallel setting.

The following lemma expresses when a block can act as a splitter. We observe that only item 5.7 from Lemma 5.7 is not inherited under refinement.

Lemma 5.7 (cf. [20, Lemma 5.2.2]). *Let $A = (S, \rightarrow, Act)$ be an LTS, π a partition of S refined by $\leftrightarrows_b$ and blocks $B, B' \in \pi$. The block B' is a splitter of B ($\mathit{Split}_\pi(B, B') \neq \emptyset$) iff there is an $a \in Act$ such that:*

1. *$a \neq \tau$ or $B \neq B'$,*

2. *for some $s \in B$ and $s' \in B' : s \xrightarrow{a} s'$, and*
3. *there is some bottom state $s_b \in B$ such that for no $s' \in B' : s_b \xrightarrow{a} s'$.*

Proof sketch. The proof is similar to the one in [20, Lemma 3.2 and 5.2.2]; thus, a full proof is omitted. We provide a rough sketch since the splitting condition is different. We only need to prove that the conditions 1 to 3 hold iff $\mathit{PreSplit}_\pi(B, B') \neq \emptyset$, since $\mathit{Split}_\pi(B, B')$ is only empty when $\mathit{PreSplit}_\pi(B, B')$ is as well.

For the ($\implies$) part, we make a case distinction on the two sets that define $\mathit{PreSplit}_\pi(B, B')$ and it follows that we find a state and bottom state for which the three conditions of this lemma hold. For the ($\impliedby$) part, we make a case distinction on whether the bottom state in condition 3 of this lemma is the block leader of B. In both cases, we can prove that there are states in $\mathit{PreSplit}_\pi(B, B')$. □

In order to maintain our invariant that if a block is not marked as *waiting* the partition is stable modulo branching bisimulation under this block of states, we perform extra steps for each new bottom state to check if there is a block that now is a splitter conform Lemma 5.7. Since a bottom state never becomes a non-bottom state, this is maximally done at most once for each state.

The following lemma is a slight modification of Lemma 5.7. However, here we specify that a *new* bottom state must witness that the partition is not stable anymore with respect to a block. We define $\mathit{bottom}(\pi)$ as all states that are bottom in the partition π.

Lemma 5.8 *Let $A = (S, \rightarrow, Act)$ be an LTS. Let π be a partition of S and π' be a refinement of π and both are refined by $\leftrightarroweq_b$. Now suppose $B' \in \pi$ and $B' \in \pi'$ and that $\mathit{Ref}(\pi, B') = \pi$, thus π is stable with respect to B'.*

Now for some block $B \in \pi'$ we find that B' is a splitter of B ($\mathit{Split}_{\pi'}(B, B') \neq \emptyset$) iff for all actions $a \in Act$:

1. *$a \neq \tau$ or $B \neq B'$,*
2. *for some $s \in B$ and $s' \in B' : s \xrightarrow{a} s'$, and*
3. *there is some bottom state $s_b \in (\mathit{bottom}(\pi') \backslash \mathit{bottom}(\pi)) \cap B$ such that for no $s' \in B' : s_b \xrightarrow{a} s'$.*

Proof ($\impliedby$): This follows immediately from Lemma 5.7. ($\implies$): Suppose $\mathit{Split}_{\pi'}(B, B') \neq \emptyset$, then we know by Lemma 5.7 that we have for some $s \in B$ and $s' \in B' : s \xrightarrow{a} s'$, $a \neq \tau$ or $B \neq B'$, and there is a bottom $s_b \in B$ such that for no $s' \in B' : s_b \xrightarrow{a} s'$. We now need to show that $s_b \in \mathit{bottom}(\pi') \setminus \mathit{bottom}(\pi)$. We assume to the contrary that s_b is not part of this set and show that this leads to a contradiction.

If $s_b \notin \mathit{bottom}(\pi') \setminus \mathit{bottom}(\pi)$, this must mean that $s_b \in \mathit{bottom}(\pi)$. Since π' refines π, we must have a $C \in \pi$ such that $B \subseteq C$. Now $s_b \in C$ and there is a bottom state $s \in C$ and either $a \neq \tau$ or $C \neq B'$. Thus, we have found an s_b, and s to which Lemma 5.7 applies and we know that $\mathit{Split}_\pi(C, B') \neq \emptyset$. This means that $\mathit{Ref}(\pi, B) \neq \pi$ which is a contradiction. □

This lemma implies that a block B', which is not in the waiting set, is a new splitter for the partition iff there is a newly created bottom state in block B that is unable to reach this block B', while there is some state in the block B that can reach this block B'. A procedure to find these blocks B' for a newly created bottom state $s_b \in B \in \pi$, can be run in parallel (see Algorithm 7). This procedure can be summarised as follows.

- For every transition $s \xrightarrow{a} s'$, if $s \in B$ do:
  - Find block B' such that $s' \in B'$.
  - If $s = s_b$ mark that s_b reaches B' with a.
  - If $s \neq s_b$ check if s_b was able to reach B' with a if not split by B' immediately.

5.4 Algorithm

Based on the modifications, we mention in Sect. 5.3 and the parallel algorithm for strong bisimilarity, we construct an algorithm that decides branching bisimulation in $O(n + |Act|)$ time in parallel on a PRAM with $\max(n^2, m, |Act|n)$ processors. In Algorithm 4, we give the main structure of our algorithm. The algorithm is similar to the parallel algorithm for strong bisimilarity. We add the modifications mentioned in Sect. 5.3 to subroutines, which we give in Algorithms 5, 6 and 7.

In this subsection, we first define the additional data we store in memory. Next, we discuss preprocessing. Finally, we discuss the algorithm in more detail and give an example iteration.

5.4.1 Data structure

The main memory contains the following *extra* data for our algorithm for branching bisimulation:

- We indicate $s_b : S$ as the current bottom state we are inspecting for unstability. It has value $\perp$ if no new bottom state has been selected.
- For every index $0 \leq i < n$ of a state s_i we added the boolean values:
  - bottom_i indicating whether the state is a bottom state;
  - $\mathit{old\_bottom}_i$ showing whether the state was already bottom in the previous iteration;
  - $\mathit{new\_bottom}_i$ telling whether s_i is a new bottom state;
  - $\mathit{sure\_split}_i : \mathbb{B}$ expressing whether state s_i can take an action that its leader cannot;

  - $same\_marks_i$: $\mathbb{B}$ specifying whether the state s_i has the same direct markings as its leader.

- An array of booleans of size m such that for index $0 \leq i < m$ of a transition i the boolean value $silent_i$: $\mathbb{B}$ indicates whether the transition is silent.
- We use $t : \mathbb{N}$ to indicate the number of transitive τ-transitions.
- For every index $0 \leq i < t$ of a transitive τ-transition $(s \Rightarrow_\pi s')$:

  - $t\_source_i : S$ the source state (s);
  - $t\_target_i : S$ the target state (s');
  - t_silent_i : $\mathbb{B}$ which is true iff the transitive τ-transition is silent.

- We need to adjust the $mark_i$: $\mathbb{B}$ array to not only contain the marks for directly reachable actions, but also actions that can be reached after performing τ-steps. This is needed since states with the same reachable actions can be in the same block and must be able to compare marks with each other. Each state always has a mark for the τ-action, regardless of whether it has τ-actions. For example, state s_3 of Fig. 3 has markings for τ, a and b. Additionally, for each mark, we now keep track of

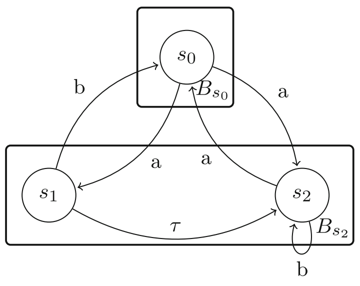
  - $mark\_source_i$: S the state to which this mark belongs;
  - $mark\_order_i$: $\mathbb{N}$ the order of the mark. For example, state s_3 of Fig. 3 has order 0, 1 and 2 for its τ, a and b marks, respectively.

- In the algorithm we use $order_i$: $\mathbb{N}$ such that a transition can set the correct mark of a state. Since $mark_i$ is different, we need to update this array as well.
- An array that has for each block B and action $a \in Act$ the boolean value $reachable_{B,a}$: $\mathbb{B}$. This is used for storing whether the current new bottom state under inspection can reach block B with a non-silent a action. We note that we can maximally have n blocks. Thus, this array has size $n \cdot |Act|$.

5.4.2 Preprocessing

First, we calculate the transitive τ-closure, which we can view as a pairwise shortest path computation. We can compute this using n^2 processors in $\mathcal{O}(n)$ [22,23]. If we consider the transitive τ-closure as transitions, we can end up with up to n^2 of these transitions. Thus, we need n^2 processors to go over them in $\mathcal{O}(1)$ time. In real-life examples of LTSs, especially when only considering the silent τ-steps, the number of these transitions is much smaller than n^2.

As in Groote and Vaandrager [20], we first remove τ-cycles since all states in a cycle end up in the same block of the final partition. We use the transitive τ-closure to remove the



$source_i$	0	0	1	1	2	2
a_i	a	a	τ	b	a	b
$action\_switch_i$	1	0	0	1	1	1
$order'_i$	1	1	0	1	1	2
$nr\_marks'_{source_i}$		2		2		3
$order_i$	1	1	0	2	1	2

$mark\_source_i$	0	0	1	1	1	2	2	2
$mark\_order_i$	0	1	0	1	2	0	1	2
$mark\_action_i$	τ	a	τ	a	b	τ	a	b
$nr\_marks_{mark\_source_i}$		2			3			3
$off(mark\_source_i)$		0			2			5

Fig. 4 An example LTS and its derived preprocessing information. The variables $action\_switch_i$, $order'_i$, $nr\_marks'_{source_i}$ are auxiliary and we explain them in Sect. 1 of Appendix

τ-cycles. We start a processor for each state s, and each state iterates over each other state s', and looks if both $s \Rightarrow_\pi s'$ and $s' \Rightarrow_\pi s$ hold. We note that $\pi = \{S\}$ here. We store the lowest state number, s_l, for which this holds, which we use to replace all references to s with s_l. All states in a cycle have one such number, namely the lowest. So this is consistent. In this way, we removed all τ-cycles in $\mathcal{O}(n)$ time using $\mathcal{O}(n)$ processors.

As opposed to the original algorithm, we now need to group states on directly possible actions and actions that a state can take after performing silent τ-steps. Therefore, we again use the base algorithm to split the sets of states into blocks based on the actions they can take and additionally let the transitive τ-closure distribute the marks for silent τ-paths. We describe this procedure in Appendix in Algorithm 8. Afterwards, we set all the blocks of this new partition to waiting, i.e. $waiting_{block_i} = \mathsf{true}$. Since we loop over all the actions and execute the loop body in constant time, this step takes $\mathcal{O}(|Act|)$ time.

Initially we mark all states as non-bottom ($bottom_i := \mathsf{false}$), all non-$\tau$-transitions as non-silent and all τ- and transitive τ-transitions as silent ($silent_i := \mathsf{true}$, $t\_silent_i := \mathsf{true}$). In the first partition, not every (transitive) τ-transition is actually silent anymore and we need to know which states are bottom. We use the procedure SetSilentAndBottom to determine this. To elect bottom states as leaders, we also execute ElectBottomLeaders. Both these procedures can be found in Algorithm 5. Finally, we mark all the values for

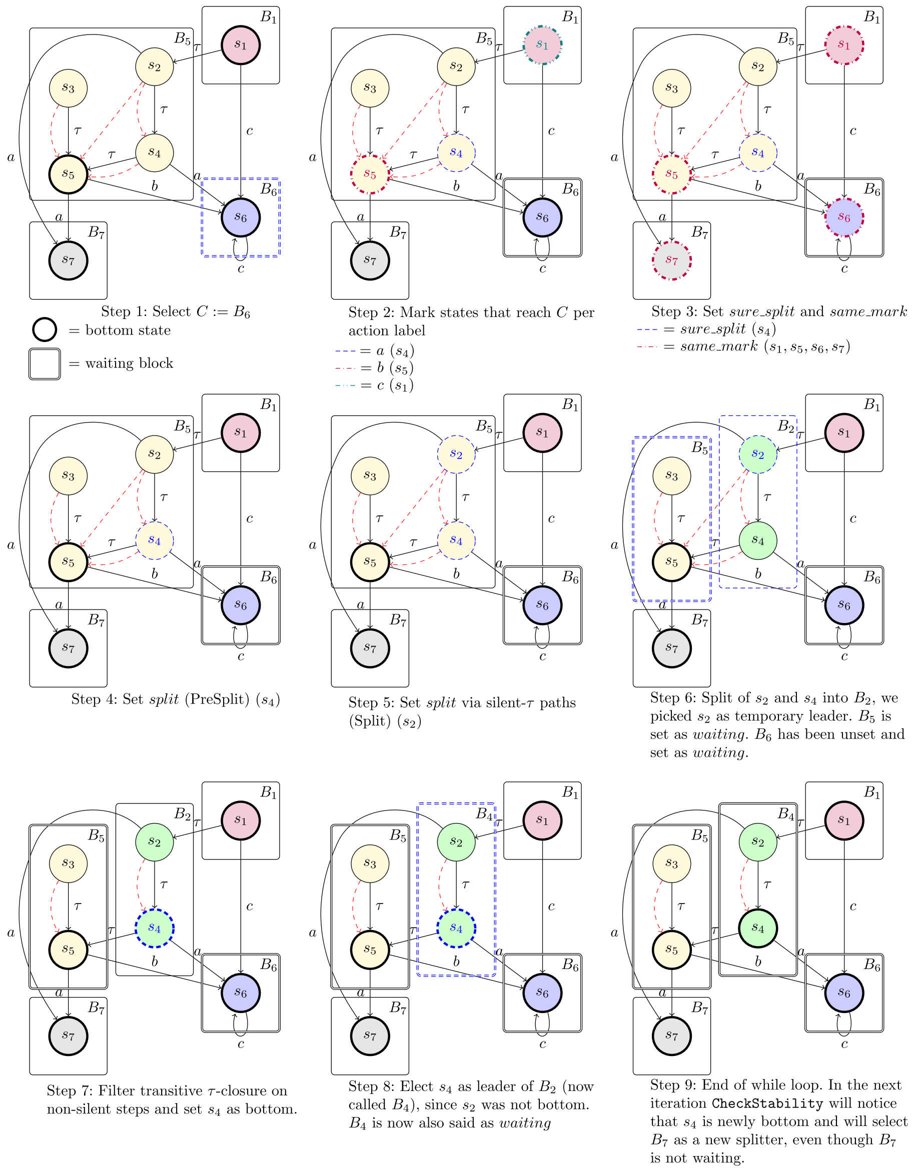

Step 1: Select $C := B_6$
= bottom state
= waiting block
Step 2: Mark states that reach C per action label
= a (s_4)
= b (s_5)
= c (s_1)
Step 3: Set *sure_split* and *same_mark*
= *sure_split* (s_4)
= *same_mark* (s_1, s_5, s_6, s_7)
Step 4: Set *split* (PreSplit) (s_4)
Step 5: Set *split* via silent-τ paths (Split) (s_2)
Step 6: Split of s_2 and s_4 into B_2, we picked s_2 as temporary leader. B_5 is set as *waiting*. B_6 has been unset and set as *waiting*.
Step 7: Filter transitive τ-closure on non-silent steps and set s_4 as bottom.
Step 8: Elect s_4 as leader of B_2 (now called B_4), since s_2 was not bottom. B_4 is now also said as *waiting*
Step 9: End of while loop. In the next iteration CheckStability will notice that s_4 is newly bottom and will select B_7 as a new splitter, even though B_7 is not waiting.

Fig. 5 An iteration of Algorithm 4 on example LTS from Fig. 3

new_bottom_i to false since we actually do not have to fix stability for these bottom states because every block is already waiting.

We can now remove any transitive τ-transitions which are not silent anymore. This step is unnecessary to guarantee correctness, but we expect it to significantly reduce the number of transitive τ-transitions. Therefore, this would improve the run time of the algorithm. Filtering these transitions takes $\mathcal{O}(\log t)$ time when using a parallel scan to put the elements in a consecutive array again. Note that we can write $\mathcal{O}(\log t)$ as $\mathcal{O}(\log n)$, since $t \leq n^2$.

Finally, we must preprocess the *mark* and *order* arrays differently. We only had a *mark* for each possible direct action in the original preprocessing. We now consider actions that a state can reach using silent τ-steps. Thus, a state can have a mark for an action that it cannot take directly. Since a bottom state only takes direct actions, and we always pick a bottom state as a block leader, we let each state look at its block leader to determine the needed marks. Each state also needs a mark for a τ-action. For more details, see Sect. 1 in Appendix. At most, this preprocessing step takes $\mathcal{O}(\log n + |Act|)$ time on $\max(n^2, m, |Act|n)$ processors.

Combining all the bounds on the complexity of the complete preprocessing procedure, we end up with $\mathcal{O}(n + |Act|)$ time using $\max(n^2, m, |Act|n)$ processors in the worst case.

Algorithm 4 Main loop for the branching bisimulation algorithm.

```
1: do
2:   $C := \bot$
3:   if $i < mark\_length$ then
4:     $mark_i :=$ false                                    ▷ Reset Marks
5:   end if
6:   if $i < n$ then
7:     $split_i :=$ false                                   ▷ Reset split
8:   end if
                            ▷ If not stable anymore, get a splitter here
9:   CheckStability()
10:  if $i < n$ and $waiting_i$ and $C = \bot$ then
11:    $C := i$                                ▷ Select splitter, if not picked
12:  end if
13:  if $i < m$ and $block_{target_i} = C$ and $\neg silent_i$ then
14:    $mark_{off(source_i)+order_i} :=$ true
15:  end if
16:  DetermineSplits()
17:  if $i < n$ and $C \neq \bot$ then
18:    $waiting_C :=$ false
19:    if $split_i$ then
20:      $waiting_C :=$ true                          ▷ If split, C waits
21:      $new\_leader_{block_i} := i$                      ▷ Elect a state
22:      $waiting_{block_i} :=$ true                     ▷ Old block waits
23:      $block_i := new\_leader_{block_i}$                    ▷ Do split
24:    end if
25:  end if
26:  SetSilentAndBottom()
27:  ElectBottomLeaders()
28:  if $i < n$ and $split_i$ then
29:    $waiting_{block_i} :=$ true                      ▷ New block waits
30:  end if
31: while $C \neq \bot$
```

Algorithm 5 Mark the states which are bottom, and for each block elect a bottom as block leader.

```
1: function SetSilentAndBottom()
2:   if $i < n$ then
3:     $old\_bottom_i := bottom_i$                      ▷ Safe old values
4:     $bottom_i :=$ true                       ▷ Initialize bottom value
5:   end if
6:   if $i < m$ and $a_i = \tau$ then
7:     if $block_{source_i} \neq block_{target_i}$ then
8:       $silent_i :=$ false
9:     else
                            ▷ Otherwise, source state is not bottom
10:      $bottom_{source_i} :=$ false
11:    end if
12:  end if
13:  if $i < t$ and $block_{t\_source_i} \neq block_{t\_target_i}$) then
14:    $t\_silent_i :=$ false
15:  end if
16:  if $i < n$ and $bottom_i \neq old\_bottom_i$ then
17:    $new\_bottom_i :=$ true
18:  end if
19: end function
20:
21: function ElectBottomLeaders()
22:   if $i < n$ and $\neg bottom_{block_i}$ then
23:     if $bottom_i$ then
24:       $new\_leader_{block_i} := i$                     ▷ Elect bottom
25:     end if
26:     $block_i := new\_leader_{block_i}$                    ▷ Set leader
27:   end if
28: end function
```

Algorithm 6 Determine for each state if it splits or stays in the block.

```
1: function DetermineSplits()
2:   if $i < n$ then
3:     $sure\_split_i :=$ false
4:     $same\_marks_i :=$ true                       ▷ Same as leader
5:   end if
6:   if $i < mark\_length$ then
7:     if $\neg mark_{off(block_{mark\_source_i})+mark\_order_i}$ and $mark_i$ then
                            ▷ Reaches C, but leader does not
8:       $sure\_split_{mark\_source_i} :=$ true
9:     end if
10:    if $mark_{off(block_{mark\_source_i})+mark\_order_i} \neq mark_i$ then
                            ▷ Mark is different than mark of leader
11:      $same\_marks_{mark\_source_i} :=$ false
12:    end if
13:  end if
14:  if $i < n$ and ($sure\_split_i$ or $bottom_i$ and $\neg same\_marks_i$ ) then
15:    $split_i :=$ true;                               ▷ PreSplit
16:  end if
17:  if $i < t$ and $t\_silent_i$ and $split_{t\_target_i}$ then
18:    $split_{t\_source_i} :=$ true                    ▷ Split via $\tau$-path
19:  end if
20: end function
```

5.4.3 Algorithm

In Fig. 5, an iteration of Algorithm 4 is performed on the example LTS from Fig. 3, starting at Line 13, after a splitter C has been selected.

Following Algorithm 4, the first lines up to Line 13 are to reset the variables and select a splitter. In the procedure `CheckStability`, the condition from Lemma 5.8 is used to determine whether there is a new bottom state that causes a split. If this procedure finds a splitter C, we use it. On the other hand, if this procedure does not yield a splitter, a splitter is selected uniformly among the blocks for which the partition is not considered stable yet (Line 9–10).

In Lines 13–15, the direct markings are set for each transition reaching the currently selected splitter C. After which the procedure `DetermineSplits`, given in Algorithm 6, marks states that are supposed to split. For each state $s \in B_{s_b} \in \pi$, the variable $split_i$ is set (Algorithm 6 Lines 14–16) to indicate whether the state is in the $PreSplit_\pi(B_{s_b}, C)$. Finally the transitive τ-closures are followed to also set the $split_i$ variable for states that are in the set $Split_\pi(B_{s_b}, C)$.

In lines 17–25, the new partition $Ref(\pi, C)$ is calculated. This is done in the same way as in the algorithm for strong bisimulation. The partition π is only considered stable with respect to C if $Ref(\pi, C) = \pi$, meaning not a single split happened.

The final step is updating the necessary information. The procedure `SetSilentAndBottom` updates all the silent

Algorithm 7 For all newly created bottom states check if they have a splitter.

```
1: function CheckStability()
2:   do
3:     $s_b := \bot$;
4:     if $i < n$ and $new\_bottom_i$ then
5:       $s_b := i$;                                  ▷ Elect a new bottom to check
6:     end if
7:     if $s_b \neq \bot$ then
8:       CheckBottom()                                ▷ Check for splitter
9:       if $C = \bot$ then
10:        $new\_bottom_{s_b}$ := false               ▷ Done for $s_b$
11:      end if
12:    end if
13:  while $s_b \neq \bot$ and $C = \bot$             ▷ Exit if splitter
14: end function
15:
16: function CheckBottom()
17:   if $i < m$ then
18:     if $source_i = s_b$ then
19:       reachable$_{block_{target_i}, a_i}$ := true  ▷ Mark the blocks $s_b$ reaches
20:     end if
21:     if $block_{source_i} = block_{s_b}$ and $\neg silent_i$ and
          $\neg reachable_{block_{target_i}, a_i}$ then
                                                   ▷ State can reach a block that $s_b$ cannot
22:       $C := block_{target_i}$
23:     end if
24:     if $source_i = s_b$ then
25:       reachable$_{block_{target_i}, a_i}$ := false ▷ Reset
26:     end if
27:   end if
28: end function
```

τ-paths by checking the conditions from Lemma 5.9. With all τ-steps now correctly marked silent, we can calculate the new bottom states, and the procedure `ElectBottomLeaders` ensures that for each new block, the leader is a bottom state. After we found the new leaders for split blocks, we set the newly split blocks to *waiting*.

5.5 Complexity and correctness

First, we proof that our splitting procedure of Sect. 5.3.1 is correct. Next, showing the correctness of the algorithm can be done in a similar fashion as in the previous algorithm. By first showing the resulting partition induces a branching bisimulation relation, and secondly showing it is the coarsest partition forming a branching bisimulation relation. Lastly, we prove the complexity of the algorithm.

5.5.1 Correct splitting

To establish that we have a valid splitting procedure, we need to prove that branching bisimilar states never end up in different blocks and that if we cannot split blocks anymore, the partition induces a branching bisimulation relation. We prove these two properties in the remainder of this subsection, but we first prove two auxiliary lemmas.

Firstly, the following auxiliary lemma shows that updating the silent transitive closure $\Rightarrow_\pi$ by only checking whether states are still in the same block is correct with the given splitting procedure.

Lemma 5.9 *Let $A = (S, \rightarrow, Act)$ be an LTS where S contains no τ-cycles, π a partition of S, $s, s' \in S$ two states, $C \in \pi$ a splitter, and π' the partition given by $\pi' = Ref(\pi, C)$. Then $s \Rightarrow_{\pi'} s'$ if and only if $s \Rightarrow_\pi s'$ and s and s' are in the same block $B \in \pi'$.*

Proof Given a partition π of an LTS $A = (S \rightarrow, Act)$, two states $s, s' \in S$ and a partition $\pi' = Ref(\pi, C)$, as mentioned in the lemma. We prove both directions:
($\Longrightarrow$) If $s \Rightarrow_{\pi'} s'$ then since there is a silent path we know that s and s' are in the same block $B \in \pi'$. Since π' is a refinement of π there is a block $B' \in \pi$ such that $B \subseteq B'$, and $s \Rightarrow_\pi s'$.
($\Longleftarrow$) Assuming $s \Rightarrow_\pi s'$ and there is a block $B' \in \pi'$ such that $s, s' \in B'$ we show that $s \Rightarrow_{\pi'} s'$. Since π' refines π there is a block $B_{s_b} \in \pi$ that contains s and s' and has the state s_b as leader. There is a set of states $s_1, \ldots, s_n \in B_{s_b}$ that witnesses the silent path $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s'$. The block B_{s_b} will either remain the same in π' or be split into $Split_\pi(B_{s_b}, C)$ and the remainder. If $s, s' \in Split_\pi(B_{s_b}, C)$ then by the transitive closure $s_1, \ldots, s_n \in Split_\pi(B_{s_b}, C)$, since every $s_1, \ldots, s_n$ reaches s' with inert τ steps. If $s, s' \notin Split_\pi(B_{s_b}, C)$ then also $s_1, \ldots, s_n \notin Split_\pi(B_{s_b}, C)$ since if for any $1 \leq i \leq n$ we have $s_i \in Split_\pi(B_{s_b}, C)$ then since $s \Rightarrow_\pi s_i$ it holds that $s \in Split_\pi(B_{s_b}, C)$. Thus, we conclude that $s_1, \ldots, s_n \in B'$ witnessing $s \Rightarrow_{\pi'} s'$. □

Secondly, the next auxiliary lemma shows that two bisimilar states can always reach branching bisimilar states within a block via silent τ-paths.

Lemma 5.10 *Let $A = (S, Act, \rightarrow)$ be an LTS. Given a partition π of S such that $\leftrightarroweq_b$ is a refinement of π. For two states s, t in the same block $B \in \pi$, if $s \leftrightarroweq_b t$ and $s \Rightarrow_\pi s'$ then there is a state $t' \in B$ such that $t \Rightarrow_\pi t'$, $s' \leftrightarroweq_b t'$.*

Proof We prove this using induction on i.
Induction Hypothesis: If $s \leftrightarroweq_b t$, $s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_i$ and $s_0, s_1, \ldots, s_i \in B$, then there is a state $t' \in B$ such that $t \Rightarrow_\pi t'$ and $s_i \leftrightarroweq_b t'$.

For $i = 0$, we have $s_0 = s_i$; thus, we can take t as t' and the induction hypothesis holds.

Now, assume that the induction hypothesis holds for i, we prove that it holds for $i+1$. We have $s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_i \xrightarrow{\tau} s_{i+1}$ and we want to find a t'' such that $t \Rightarrow_\pi t''$ and $s_{i+1} \leftrightarroweq_b t''$. From the induction hypothesis, we know that we have a t' such that $t \Rightarrow_\pi t'$, $s_i \leftrightarroweq_b t'$. We now have two cases.
Case 1: $s_i \leftrightarroweq_b s_{i+1}$. Then also $t' \leftrightarroweq_b s_{i+1}$ and if we take $t'' = t'$ we are done.
Case 2: $s_i \not\leftrightarroweq_b s_{i+1}$. We have $s_i \leftrightarroweq_b t'$ and $s_i \xrightarrow{\tau} s_{i+1}$. The first case of the definition of $\leftrightarroweq_b$ does not hold, so we know from

the second case that there exists a sequence $t' \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'''$ such that $t''' \leftrightarroweq_b s_i$, $t''' \xrightarrow{\tau} t''$ and $t'' \leftrightarroweq_b s_{i+1}$. Now all states of the sequence $t' \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'''$ are in B, because $\leftrightarroweq_b$ refines π and by Lemma 5.5 all intermediate states are related. Because $s_{i+1} \in B$ we know $t'' \in B$ also holds. Thus, we have $t \Rightarrow_\pi t''$ and $s_{i+1} \leftrightarroweq_b t''$ which proves the induction hypothesis for $i + 1$.

Thus, by mathematical induction, the lemma holds. □

We now prove that we never move two branching bisimilar states to different blocks.

Lemma 5.11 *Let $A = (S, Act, \rightarrow)$ be an LTS where S contains no τ-cycles. Given a partition π which is refined by $\leftrightarroweq_b$ and two states $s, t \in S$ in a block $B \in \pi$. Then for all $B' \in \pi$ if $s \leftrightarroweq_b t$ either $s, t \in Split_\pi(B, B')$ or $s, t \notin Split_\pi(B, B')$.*

Proof Let s_b be the leader state of B. This allows us to write B_{s_b} for B. Consider a partition π, and two branching bisimilar states $s, t \in B_{s_b} \in \pi$, i.e. $s \leftrightarroweq_b t$ as in the lemma. Assume that $s \in Split_\pi(B_{s_b}, B')$ for any $B' \in \pi$, we will show that $t \in Split_\pi(B_{s_b}, B')$. First we distinguish on whether $s \in PreSplit_\pi(B_{s_b}, B')$.

In the first case, we assume that $s \in PreSplit_\pi(B_{s_b}, B')$, and show that this implies $t \in Split_\pi(B_{s_b}, B')$. Assuming $s \in PreSplit_\pi(B_{s_b}, B')$, we can distinguish the following two cases:

- If it is the case that $dir(s, B') \not\subseteq dir(s_b, B')$, this means there is an action $a \in Act$ such that $s \xrightarrow{a} s'$ for some $s' \in B'$ and $s_b \not\xrightarrow{a} s''$ for any $s'' \in B'$. From the definition of dir, we know that it is not the case that $a = \tau$ and $s \leftrightarroweq_b s'$; thus, we can ignore the first case of the definition of $\leftrightarroweq_b$. Following the second case, since $s \leftrightarroweq_b t$ there is a sequence $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'$ such that $t \leftrightarroweq_b t'$, $t' \xrightarrow{a} t''$ and $t'' \leftrightarroweq_b s'$. Since $\leftrightarroweq_b$ refines π we know that $t' \in B_{s_b}$, $t'' \in B'$ and, using Lemma 5.5, $t \Rightarrow_\pi t'$. Therefore, with $dir(t', B') \not\subseteq dir(s_b, B')$, $t' \in PreSplit_\pi(B_{s_b}, B')$ holds, and hence, $t \in Split_\pi(B_{s_b}, B')$.
- For the other case where s is a bottom state and $dir(s_b, B') \not\subseteq dir(s, B')$, notice that there is at least one bottom state $t' \in B_{s_b}$ such that $t \Rightarrow_\pi t'$. By Lemma 5.10 as $s \leftrightarroweq_b t$ there has to be a $s' \in B_{s_b}$ such that $s \Rightarrow_\pi s'$ and $s' \leftrightarroweq_b t'$. Since s is a bottom state it has to be the case that $s \leftrightarroweq_b t'$. By the definition of branching bisimilarity and the states are bottom states it has to be the case that $dir(t', B') \subseteq dir(s, B')$. Since for every letter $a \in Act$ and $s'' \in B_{s_b}$, if $s \xrightarrow{a} s''$ then there is a $t'' \in B'$ such that $t' \xrightarrow{a} t''$ and $s'' \leftrightarroweq_b t''$. Since $dir(s_b, B') \not\subseteq dir(s, B')$ and $dir(t', B') \subseteq dir(s, B')$ we know that also $dir(s_b, B') \not\subseteq dir(t', B')$, and thus, $t' \in PreSplit_\pi(B_{s_b}, B')$, meaning $t \in Split_\pi(B_{s_b}, B')$.

This concludes that if $s \in PreSplit_\pi(B_{s_b}, B')$ it is the case that $t \in Split_\pi(B_{s_b}, B')$.

In the second case we assume $s \notin PreSplit_\pi(B_{s_b}, B')$ but s is part of the split, i.e. $s \in Split_\pi(B_{s_b}, B')$, and show that $t \in Split_\pi(B_{s_b}, B')$. In this case there is a state $s' \in B_{s_b}$ such that $s' \in PreSplit_\pi(B_{s_b}, B')$ and $s \Rightarrow_\pi s'$. According Lemma 5.10 there is a $t' \in B_{s_b}$ such that $t \Rightarrow_\pi t'$ and $t' \leftrightarroweq_b s'$, and since $s' \in PreSplit_\pi(B_{s_b}, B')$ by the previous argumentation we know that $t' \in Split_\pi(B_{s_b}, B')$ and since $t \Rightarrow_\pi t'$ also $t \in Split_\pi(B_{s_b}, B')$.

We are now ready to conclude that if $s \in Split_\pi(B_{s_b}, B')$ then also $t \in Split_\pi(B_{s_b}, B')$. Since s, t are chosen arbitrarily if $t \in Split_\pi(B_{s_b}, B')$ then also $s \in Split_\pi(B_{s_b}, B')$. This means that either both states $s, t \in Split_\pi(B_{s_b}, B')$, or both $s, t \notin Split_\pi(B_{s_b}, B')$, proving this lemma. □

Lastly, we show that if the current partition is not yet branching bisimilar, there is still a block that refines the partition with our splitting procedure.

Lemma 5.12 *Let $A = (S, \rightarrow, Act)$ be an LTS, and π a partition of S. If for all blocks $B \in \pi$ it holds that $Ref(\pi, B) = \pi$ then the relation R_π induced by the partition π is a branching bisimulation.*

Proof Let π be a partition of S for which every $B \in \pi$ it holds that $Ref(\pi, B) = \pi$ as in the lemma. Assume two states $s, t \in S$ are related $s R_\pi t$, i.e. there is a block $B_{s_b} \in \pi$, with leader $s_b \in S$ such that $s, t \in B_{s_b}$. For all actions $a \in Act$ and $s' \in B'$ such that $s \xrightarrow{a} s'$, if $a = \tau$ and $B' = B_{s_b}$ we know that $s', t \in B_{s_b}$. Hence, $s' \ R_\pi \ t$. If this is not the case then $a \in dir(s, B')$. We know that since $Ref(\pi, B') = \pi$, also $Split_\pi(B_{s_b}, B') = PreSplit_\pi(B_{s_b}, B') = \emptyset$. This means that $s \notin PreSplit_\pi(B_{s_b}, B')$, and hence, $dir(s, B') \subseteq dir(s_b, B')$. Additionally, we know that there is at least one bottom state $t' \in B_{s_b}$ such that $t \Rightarrow_\pi t'$, and thus, $t R_\pi t'$. Since t' is a bottom state and also $t' \notin PreSplit_\pi(B_{s_b}, B')$ we know that $dir(t', B') \supseteq dir(s_b, B') \supseteq dir(s, B')$. This means $a \in dir(t', B')$, so there is a state $t'' \in B'$ such that $t' \xrightarrow{a} t''$ and $s' \ R_\pi \ t''$. Since R_π is also symmetric, it follows that R_π forms a branching bisimulation. □

5.5.2 Correctness of algorithm

Theorem 5.13 *Let $A = (S, Act, \rightarrow)$ be an LTS that contains no τ-loops. The partition π resulting from executing Algorithm 4 with input A forms the coarsest partition inducing a branching bisimulation, solving branching-BCRP.*

Proof At the end of the iteration for all block labels $i \in \mathbb{N}, i \leq |S|$, $waiting_i$ = false. Since for all new bottom states we did the extra stability procedure in Algorithm 7 by Lemma 5.8 we know that the partition is stable modulo branching bisimulation. By Lemma 5.12 this means π is a branching bisimulation relation.

Finally by Lemma 5.11 it never happens that two branching bisimilar states are split. This means that π is the coarsest partition inducing a branching bisimulation, solving branching-BCRP. □

5.5.3 Complexity of algorithm

Theorem 5.14 *The algorithm for branching bisimulation terminates in $O(n+|Act|)$ parallel time on* $\max(n^2, m, |Act|n)$ *processors.*

Proof As indicated in the preprocessing section, the preprocessing takes $\mathcal{O}(n + |Act|)$ time using $\max(n^2, m, |Act|n)$ processors in the worst case.

In every iteration of the algorithm, either there is a splitter that causes the number of blocks to increase, or a block is removed from waiting. This means that the proof of Theorem 4.2 still holds with our modifications. So there are maximally $3n$ iterations of the main algorithm. Except for the procedure to maintain stability, every part of the algorithm runs in constant time. So our total running time is the number of iterations in addition to the number of iterations of `CheckStability` that does not result in a splitter. The stability check only happens at most once for every new bottom state since a bottom state can never become non-bottom.

We use $\max(n, m, t, mark\_length)$ processors. We observe that the size of the transitive closure t is maximally size n^2 and $mark\_length \leq |Act|n$. Therefore, we need $\max(n^2, m, |Act|n)$ processors.

Thus, the main part of the algorithm runs in $\mathcal{O}(n)$ time using $\max(n^2, m, |Act|n)$ processors.

Therefore, the total complexity of preprocessing and running the algorithm is $\mathcal{O}(n+|Act|)$ using $\max(n^2, m, |Act|n)$ processors. □

6 Related work

Deciding bisimilarity is P-complete [24], which suggests that bisimilarity is an inherently sequential problem. This fact has not withheld the community from searching for efficient parallel algorithms for deciding the bisimilarity of Kripke structures. In particular, Lee and Rajasekaran [12,25] proposed a parallel algorithm based on the Paige–Tarjan algorithm that works in $\mathcal{O}(n\ log\ n)$ time complexity using $\frac{m}{\log n} \log\log n$ Concurrent, Read Concurrent Write (CRCW) processors, and one using only $\frac{m}{n} log\ n$ Concurrent Read Exclusive Write (CREW) processors. Jeong et al. [26] presented a linear-time parallel algorithm, but it is probabilistic in the sense that it has a nonzero chance of producing an incorrect result.

Furthermore, Wijs [14] presented a GPU implementation of an algorithm to solve the strong and branching bisimulation partition refinement problems. But as shown in Sect. 4.3 it has only linear complexity in the number of states if the fan-out of an LTS is bounded and can otherwise have quadratic complexity. However, the multi-way splitting Wijs employs is effective in practice.

In a distributed setting, Blom and Orzan studied algorithms for refinement [18]. These algorithms use message passing to communicate between different workers in a network and rely on a small number of processors. Therefore, they are very different in nature from our algorithm. The algorithms of Blom and Orzan [18] were extended and optimised for branching bisimulation [27].

Parallel partition refinement algorithms are also studied in the problem of minimising deterministic finite automata (DFAs) [28]. These DFAs can be seen as the restricted LTSs where the transition relation is deterministic, and there is an initial partitioning distinguishing accepting and non-accepting states. This means our algorithm also works in the setting of DFAs. To our knowledge, this is the first linear-time parallel algorithm for DFA minimisation.

The sequential algorithm introduced in [11] requires time $\mathcal{O}(m \log n)$ compared to the original Groote–Vaandrager algorithm that requires $\mathcal{O}(mn)$ time [20]. The techniques in [11] are complex, using among others the *principle of the smaller half* [4]. These are therefore less suitable when aiming for maximal parallelism.

Recent work [6] provides a linear lower bound for the class of parallel partition refinement algorithms, matching the run time of our algorithm. The most efficient algorithms that decide bisimilarity use partition refinement; however, there is a possibility that other techniques give rise to algorithms that are faster than $\mathcal{O}(n)$.

Examples of these different techniques can be found in [29–31]. In [30], a linear-time sequential algorithm is given to decide bisimilarity in the restricted setting of deterministic automata with only a single action label. In the same setting, a parallel version of this algorithm is proposed in [31] with a time complexity of $\mathcal{O}(\log n)$. The run time complexity of these algorithms is fundamentally faster than the partition refinement algorithms. However, we do not think this approach can be generalised to a setting with multiple action labels or non-determinism.

The sequential algorithm proposed in [29] finds and merges bisimilar states in a way different from partition refinement. However, this algorithm performs in a run time complexity that is worse than that of partition refinement algorithms.

7 Conclusions and future work

In this work, we proposed linear-time PRAM algorithms to decide strong and branching bisimilarity. We implemented the strong bisimilarity algorithm in CUDA and conducted experiments that show the potential to compute bisimulation in practice in linear time on contemporary hardware. Further advances in parallel hardware will make this more feasible.

Other work suggests no significant improvement on the time complexity $\mathcal{O}(n)$ is possible using partition refinement [6]. On the other hand, the sequential run time of $\mathcal{O}((m+n)\log n)$ of the Paige–Tarjan algorithm suggests there might be an improvement in the number of processors needed. For future work, it would be interesting to see whether a linear algorithm deciding bisimilarity is feasible using fewer processors, for instance $\frac{m}{n}\log(n)$.

Another future direction is to optimise these algorithms for the available highly parallel hardware, such as GPUs. Optimising these algorithms should give a better insight into whether using an abundance of processors performs well in practice or other trade-offs should be made in practice. For instance, data locality and synchronisation play an important role in GPU performance but are not taken into account in a PRAM model.

Acknowledgements This work is carried out in the context of the NWO AVVA project 612.001751 and the NWO TTW ChEOPS project 17249. We thank the reviewers for their careful reviews and helpful comments.

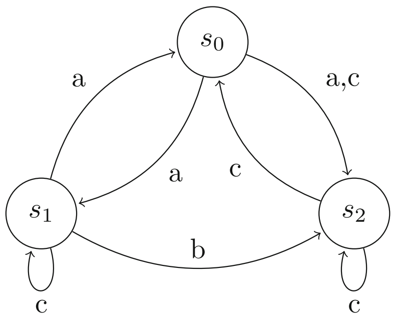
Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix A: Preprocessing for strong bisimulation

To use the Algorithm 3, we need to do two preprocessing steps. First, we need to partition the states w.r.t. their set of outgoing action labels. This can be done with an altered version of Algorithm 2. Instead of splitting on a block, we split on an action $a \in A$. We visit all transitions, and we mark the source if it has the same action label a. This can be done by the following PRAM pseudocode.

```
1: if $i < m$ and $a_i = a$ then
2:     $mark_{source_i}$ := true
3: end if
```

Each block can be split into two blocks: a block that contains states that have a as an outgoing action label and a block with states that do not have this outgoing action label. After



$source_i$	0	0	0	1	1	1	2	2
a_i	a	a	c	a	b	c	c	c
$action\_switch_i$	0	0	1	0	1	1	0	0
$order_i$	0	0	1	0	1	2	0	0
$nr\_marks_{source_i}$			2			3		1
$off(source_i)$			0			2		5

Fig. 6 An example LTS and its derived preprocessing information

doing this for all different action labels, we end up with a partition of blocks, in which all states of a block have the same set of outgoing action labels, and each pair of states from different blocks have different sets of outgoing action labels. Using m processors, this partition can be constructed in $\mathcal{O}(|Act|)$ time.

For the second preprocessing step, we need to gather the extra information that is needed in Algorithm 3. Only a_i is part of the input, the other information like *order* and *offset* need to be calculated. We start our preprocessing by sorting the transitions by $(source_i, a_i)$, which can be done in $\mathcal{O}(\log m)$ time with m processors, for instance using a parallel merge sort [32]. In order to calculate $order_i$ and nr_marks, we first calculate $action\_switch_i$ for each transition i, which is done as follows:

```
1: if i ≤ m then
2:     if $i = 0$ or $source_i \neq source_{i-1}$ or $a_i = a_{i-1}$ then
       $action\_switch_i = 0$;
3:     else $action\_switch_i = 1$;
4:     end if
5: end if
```

See Fig. 6 for an example. Now, $order_i$ can be calculated with a parallel segmented inclusive scan [17] of *action_switch*. A parallel segmented sum can be performed on *action_switch* to calculate *nr_marks*, where we make sure to set nr_marks_s to 0, if state s has no outgoing transitions. Finally, off_s, for the mark offsets, can be constructed as a list and calculated by applying a parallel exclusive scan on *nr_marks*. Calculating *action_switch* takes $\mathcal{O}(1)$ time on m processors, and a parallel (segmented) sum/scan takes $\mathcal{O}(\log m)$ time [17].

In total, the preprocessing takes $\mathcal{O}(|Act| + \log m)$ time using m processors.

Appendix B: Preprocessing for branching bisimulation

We need marks to keep track of all actions a state can directly take or can take after performing silent τ-transitions. After executing Algorithm 8, all states are now in blocks for which the reachable actions are the same. Each block now has a leader, which is a bottom state, and a bottom state cannot perform silent τ-steps. Thus, the leader has only direct actions for which we need to store markings. Therefore, we can first calculate how many marks we need for visible markings and afterwards, each state updates this by looking at its leader. See Fig. 4 for an example of the preprocessing we do in this section.

First we sort the transitions again on $(source_i, a_i)$, we note that we consider τ-actions and place them before action labels in the sorted list. Next, we need the help variable $action\_switch_i$ which keeps track of not counting the same action label for the same source state twice. It is defined exactly by the code below:

```
if i < m then
    if $i = 0$ and $a_i = \tau$ then
        $action\_switch_i = 0$                                  ▷ Initialize for $i = 0$
    else if $i > 0$ and $source_i \neq source_{i-1}$ and $a_i = \tau$ then
        $action\_switch_i = 0$                                  ▷ State switch
    else if $i > 0$ and $source_i = source_{i-1}$ and $a_i = a_{i-1}$ then
        $action\_switch_i = 0$                                  ▷ Same action
    else
        $action\_switch_i = 1$
    end if
end if
```

We cannot calculate *order* and *nr_marks* directly, so we calculate them first as a sort of approximation which we call *order*′ and *nr_marks*′, which only contain the correct values for bottom states.

We calculate a $order'$ by doing a parallel segmented inclusive scan on $action\_switch$. We calculate nr_marks' by performing a segmented sum on $action\_switch$ and adding 1 to each result. We note that states with no transitions still get a mark for τ. Now each state s determines its actual *nr_marks* by copying this from its leader: $nr\_marks_s := nr\_mark'_{block_s}$. Next, we calculate *off* by performing an exclusive scan on the *nr_marks* array and $mark\_length$ by calculating the sum of the *nr_marks* array. We can now calculate $mark\_source$ by starting $mark\_length$ processors and letting each processor do a binary search on *off*. Now, we create $mark\_order$ using a segmented exclusive scan on an array filled with ones and $mark\_source$ as a segment indicator. Lastly, we need to update $order_i$ again. First, by going over all transitions, we store the action label of a mark

Algorithm 8 Split the initial partition into blocks that have the same reachable actions.

```
if $i < n$ then
    $block_i := 0$                                  ▷ Put all states in a single block
end if
for all $a \in Act$ do                              ▷ Check $a = \tau$ last
    if $i < n$ then
        $mark_i :=$ false                           ▷ Reset marks
    end if
    if $i < m$ and $a_i = a$ and $\neg(a_i = \tau$ and $block_{source_i} = block_{target_i})$ then
                                                    ▷ Set marks for transitions except silent $\tau$ ones
        $mark_{source_i} :=$ true
    end if
    if $i < t$ and $mark_{t\_target_i}$ and $block_{t\_source_i} = block_{t\_target_i})$ then
                                                    ▷ Propagate marks via silent $\tau$-paths
        $mark_{t\_source_i} :=$ true
    end if
    if $i < n$ and $mark_i \neq mark_{block_i}$ then
        $new\_leader_{block_i} := i$                ▷ Elect new leader
        $block_i := new\_leader_{block_i}$          ▷ Split block
    end if
end for
```

in $mark\_action$ for transitions that start in a leader state (using $order'$). Next, we look if the $order$ indexes the correct action label in the $mark$ array for each transition. If this is not the case, we loop and increase the index where we look and see whether this is the correct action label. Here is the exact code:

```
if i < m then
    if $block_{source_i} = source_i$ then
        $mark\_action_{off(source_i)+order'_i} := a_i$
    end if
    $incr_i := 0$
    while $a_i \neq \tau$ and $a_i \neq mark\_$ do
        $action_{off(block_{source_i})+order'_i+incr_i}$
        $incr_i := incr_i + 1$
    end while
    $order_i := order'_i + incr_i$
end if
```

This procedure is correct since a state never has fewer marks than the original *nr_marks*′. Thus, we always find the correct action later in the mark array. This loop has at most $|Act|$ iterations.

Eventually, parallel sorting takes $\mathcal{O}(\log m)$ time, a parallel sum or scan also takes $\mathcal{O}(\log m)$ time, the binary search takes $\mathcal{O}(\log mark\_length)$ time, and the looping takes $\mathcal{O}(|Act|)$ time. Therefore, this preprocessing takes $\mathcal{O}(\log m + \log mark\_length + |Act|)$ time on $\max(n, m, t, mark\_length)$ processors. This can be simplified to $\mathcal{O}(\log n + |Act|)$ on $\max(n^2, m, |Act|n)$ processors, when observing that $m \leq |Act|n^2$, $t \leq n^2$ and $mark\_length \leq |Act|n$.

References

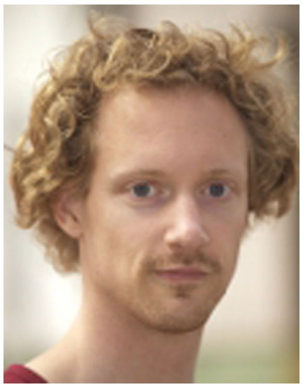
1. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A.J., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems—improvements in expressivity and usability. In: Tools and Algorithms for the Construction and Analysis of Systems—25th International Conference (TACAS 2019), Proceedings, Part II, pp. 21–39 (2019). https://doi.org/10.1007/978-3-030-17465-1_2
2. Milner, R.: A Calculus of Communicating Systems. Lecture Notes in Computer Science, vol. 92. Springer, Cham (1980)
3. Kanellakis, P., Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation **86**(1), 43–68 (1990). https://doi.org/10.1016/0890-5401(90)90025-D
4. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6), 973–989 (1987). https://doi.org/10.1137/0216062
5. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the top: What will drive computer performance after Moore's law? Science (2020). https://doi.org/10.1126/science.aam9744
6. Groote, J.F., Martens, J., de Vink, E.P.: Lowerbounds for bisimulation by partition refinement. pre-print (2022). arXiv:2203.07158
7. Valmari, A.: Simple bisimilarity minimization in $O(m \log n)$ time. Fundam. Informaticae **105**(3), 319–339 (2010). https://doi.org/10.3233/FI-2010-369
8. Stockmeyer, L., Vishkin, U.: Simulation of parallel random access machines by circuits. SIAM J. Comput. **13**(2), 409–422 (1984). https://doi.org/10.1137/0213027
9. Martens, J., Groote, J.F., Haak, L.v.d., Hijma, P., Wijs, A.: A linear parallel algorithm to compute bisimulation and relational coarsest partitions. In: International Conference on Formal Aspects of Component Software, pp. 115–133. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-90636-8_7
10. Fortune, S., Wyllie, J.: Parallelism in random access machines. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, pp. 114–118 (1978). https://doi.org/10.1145/800133.804339
11. Jansen, D.N., Groote, J.F., Keiren, F.J.A., Wijs, A.J.: An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, vol. 12079, pp. 3–20. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_1
12. Lee, I., Rajasekaran, S.: A parallel algorithm for relational coarsest partition problems and its implementation. In: Dill, D.L. (ed.) Computer Aided Verification, vol. 818, pp. 404–414. Springer, Cham (1994). https://doi.org/10.1007/3-540-58179-0_71
13. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: LITP Spring School on Theoretical Computer Science, pp. 407–419. Springer (1990). https://doi.org/10.1007/3-540-53479-2_17
14. Wijs, A.J.: GPU accelerated strong and branching bisimilarity checking. In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 368–383. Springer, Cham (2015). https://doi.org/10.1007/978-3-662-46681-0_29
15. Wijs, A.J., Katoen, J.-P., Bošnački, D.: Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components. Formal Methods Syst. Design **48**(3), 274–300 (2016). https://doi.org/10.1007/s10703-016-0246-7
16. Reniers, M.A., Schoren, R., Willemse, T.A.C.: Results on embeddings between state-based and event-based systems. Comput. J. **57**(1), 73–92 (2014). https://doi.org/10.1093/comjnl/bxs156
17. Sengupta, S., Harris, M., Garland, M., Owens, J.: Efficient Parallel Scan Algorithms for Manycore GPUs. In: Scientific Computing with Multicore and Accelerators, pp. 413–442. Taylor & Francis, Boca Raton (2011). Chap. 19. https://doi.org/10.1201/b10376
18. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. Electron. Notes Theor. Comput. Sci. **89**(1), 99–113 (2003). https://doi.org/10.1016/S1571-0661(05)80099-4
19. Van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. JACM **43**(3), 555–600 (1996). https://doi.org/10.1145/233551.233556
20. Groote, J.F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: International Colloquium on Automata, Languages, and Programming, pp. 626–638. Springer, Cham (1990). https://doi.org/10.1007/BFb0032063
21. Browne, M.C., Clarke, E.M., Grümberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**(1), 115–131 (1988). https://doi.org/10.1016/0304-3975(88)90098-9
22. Bošnački, D., Odenbrett, M.R., Wijs, A., Ligtenberg, W., Hilbers, P.: Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors. BMC Bioinform. **13**(1), 281 (2012). https://doi.org/10.1186/1471-2105-13-281
23. Han, Y., Pan, V., Reif, J.: Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In: Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 353–362 (1992). https://doi.org/10.1145/140901.141913
24. Balcázar, J., Gabarro, J., Santha, M.: Deciding bisimilarity is P-complete. Formal Aspects Comput. **4**(1), 638–648 (1992). https://doi.org/10.1007/BF03180566
25. Rajasekaran, S., Lee, I.: Parallel algorithms for relational coarsest partition problems. IEEE Trans. Parallel Distrib. Syst. **9**(7), 687–699 (1998). https://doi.org/10.1109/71.707548
26. Jeong, C., Kim, Y., Oh, Y., Kim, H.: A faster parallel implementation of Kanellakis–Smolka algorithm for bisimilarity checking. In: Proceedings of the International Computer Symposium. Citeseer (1998)
27. Blom, S., van de Pol, J.: Distributed branching bisimulation minimization by inductive signatures. In: Brim, L., van de Pol, J. (eds.) Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009. EPTCS, vol. 14, pp. 32–46 (2009). https://doi.org/10.4204/EPTCS.14.3
28. Tewari, A., Srivastava, U., Gupta, P.: A parallel DFA minimization algorithm. In: International Conference on High-Performance Computing, pp. 34–40. Springer, Berlin (2002)
29. Björklund, J., Cleophas, L.: Aggregation-based minimization of finite state automata. Acta Informatica **58**(3), 177–194 (2021)
30. Paige, R., Tarjan, R.E., Bonic, R.: A linear time solution to the single function coarsest partition problem. Theor. Comput. Sci. **40**, 67–84 (1985)
31. JáJá, J.F., Ryu, K.W.: An efficient parallel algorithm for the single function coarsest partition problem. Theor. Comput. Sci. **129**(2), 293–307 (1994)
32. Cole, R.: Parallel merge sort. SIAM J. Comput. **17**(4), 770–785 (1988)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Jan Martens is a PhD student in formal system analysis at Eindhoven University of Technology. His research interest is in theoretical computer science and parallel algorithms.

Jan Friso Groote (1965) is a full professor in formal system analysis since 1998 at Eindhoven University of Technology. His research interest is in modelling and analysing behaviour of programmed systems with as purpose of increasing its quality.

Lars B. van den Haak is a PhD student in formal system analysis at Eindhoven University of Technology. His research interest is in formal verification, parallel software, and programming languages.



Pieter Hijma worked as an assistant professor at the Vrije Universiteit Amsterdam and now works in the industry in the area of Open Source Hardware (OSH). His research focuses on the primary interface between humans and computers, namely the programming language, with a strong focus on parallel programming.

Anton Wijs is an assistant professor at Eindhoven University of Technology. His research interest is in model checking, particularly improving the algorithms to do model checking.