A Linear Parallel Algorithm to Compute Bisimulation and Relational Coarsest Partitions

A linear parallel algorithm to compute bisimulation and relational coarsest partitions

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

[1] *Eindhoven University of Technology, The Netherlands*
{j.j.m.martens, j.f.groote, l.b.v.d.haak, a.j.wijs}@tue.nl
[2] *Vrije Universiteit Amsterdam, The Netherlands*
pieter@cs.vu.nl

May 26, 2021

Abstract

The most efficient way to calculate strong bisimilarity is by finding the relational coarsest partition on a transition system. We provide the first linear time algorithm to calculate strong bisimulation using parallel random access machines (PRAMs). More precisely, with n states, m transitions and $|Act| \leq m$ action labels, we provide an algorithm on $\max(n, m)$ processors that calculates strong bisimulation in time $\mathcal{O}(n + |Act|)$ and space $\mathcal{O}(n+m)$. The best-known PRAM algorithm has time complexity $\mathcal{O}(n \log n)$ on a smaller number of processors making it less suitable for massive parallel devices such as GPUs. An implementation on a GPU shows that the linear time-bound is achievable on contemporary hardware.

1 Introduction

The notion of *bisimilarity* for Kripke structures and Labeled Transition Systems (LTSs) is commonly used to define behavioural equivalence. Deciding this behavioural equivalence is important in the field of modelling and verifying concurrent systems [4, 15]. Kanellakis and Smolka proposed a partition refinement algorithm for obtaining the bisimilarity relation for Kripke structures [11]. 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 graph. Later, a more sophisticated refinement algorithm running in $\mathcal{O}(m\ log\ n)$ steps was proposed by Paige and Tarjan [16].

In recent years the increase in the speed of sequential chip design 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. [14] 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 paper, we propose such an algorithm to decide bisimilarity.

Deciding bisimilarity is P-complete [1], which suggests that bisimilarity is an inherently sequential problem. This fact has not withheld the community from searching efficient parallel algorithms for deciding bisimilarity of Kripke structures. In particular, Lee and Rajasekaran [13] proposed a parallel algorithm based on the Paige Tarjan algorithm that works in $\mathcal{O}(n\ log\ n)$ time complexity using $\frac{m}{\log n} \log\log n$ Concurrently Read and Concurrently Write (CRCW) processors.

In this work, we improve on the best known theoretical bound for PRAM algorithms using a higher degree of parallelism. The proposed algorithm improves the run time complexity to $\mathcal{O}(n)$ on $max(m, n)$ processors and is based on the sequential algorithm of Kanellakis and Smolka [11]. 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 straightforward to incorporate in an efficient way (cf. for instance [21]), we discuss how our algorithm can be extended to take action labels into account. This leads to an algorithm with a run time complexity of $\mathcal{O}(n + |Act|)$, with Act the set of action labels.

Our algorithm has been designed for and can be analyzed with the CRCW PRAM model, following notations from [20]. This model is an extension of 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, but 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 in CUDA and experimented with an NVIDIA Titan RTX, showing that our algorithm performs mostly in line with what our PRAM algorithm predicts.

The paper is structured as follows: In Section 2, we recall the necessary preliminaries on the CRCW PRAM model and state the partition refinement problems this paper focuses on. In Section 3, we propose a parallel algorithm to compute 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 Section 4, we discuss the details for an implementation with multiple action labels, thereby supporting LTSs, which forms the Bisimulation Coarsest Refinement Problem (BCRP). In Section 5 we discuss the results of the implementation and in Section 6 we address the usage of weaker PRAM models. Finally, in Section 7, we discuss related work.

2 Preliminaries

2.1 The PRAM Model

The *Parallel Random Access Machine* (PRAM) is a natural extension of the normal Random Access Machine (RAM), where an arbitrary number of parallel programs can access the memory. Following the definitions of [20] we use a version of PRAM that is able to Concurrently

Read and Concurrently Write (CRCW PRAM). It differs from the model introduced in [8] 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. We call the model from [8] an Concurrent Read, Exclusive Write (CREW) PRAM model.

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. A PRAM also has a function $P : \mathbb{N} \to \mathbb{N}$ which defines a bound on the number of processors given the size of the input.

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

Initially, given input consisting of n data elements, the CRCW PRAM assumes that the input is stored in the first n registers of the common memory, and starts the first $P(n)$ processors $P_0, P_1, \ldots, P_{P(n)-1}$.

We need to define what the behaviour of the machine will be whenever a concurrent write happens. The way to handle this memory contention in concurrent writes is usually by assuming one of the following:

- **(Common)** All processors try to write the same value and succeed, otherwise, the writes are not legal and fail;
- **(Arbitrary)** Only one arbitrary attempt to write succeeds;
- **(Priority)** Only the processor with the lowest index succeeds in writing.

The algorithm proposed in this paper works if we make either the *arbitrary* or the *priority* assumption. In Section 6 we explain how we can adapt it to work under the *common* assumption.

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 $P \cdot T \in \mathcal{O}(S)$.

The computational complexity of these models is well studied and there is a close relation between circuit complexity and the complexity of PRAM algorithms [20].

2.2 Strong Bisimulation

To formalise concurrent system behaviour, we use LTSs.

Definition 1 (Labeled Transition System)**.** *A Labeled Transition System (LTS) is a three-tuple $A = (S, Act, \to)$ where S is a finite set of states, Act a finite set of action labels, and $\to \subseteq S \times Act \times S$ the transition relation.*

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

Kripke structures differ from LTSs in the fact that the states are labelled as opposed to the transitions. In the current paper, 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, [9]).

Definition 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:*

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

Whenever we refer to bisimulation we mean strong bisimulation. Two states $s, t \in S$ in an LTS A are called *bisimilar*, denoted by $s \leftrightarrows t$, iff there is some 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 $R = \{(s, t) \mid \exists B \in \pi . s \in B \wedge t \in B\}$. 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 1. *[16] Stability is inherited under refinement, i.e. given a partition π of S and a refinement π' of π, then if π is stable under $U \subseteq S$, then π' is also stable under U.*

The main problem we focus on in this work is called the bisimulation 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**) [11, 13, 16]. 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 is a bisimulation relation.

It is known that BCRP is not significantly harder than RCPP as there are intuitive translations from LTSs to Kripke structures [7, Dfn. 4.1]. However, some non-trivial modifications can speed-up the algorithm for some cases, hence we discuss both problems separately. In

```
1  π := π_0;
2  Unstable := π;
3  while Unstable ≠ ∅ do
4  |  foreach B ∈ Unstable do
5  |  |  Unstable := Unstable \ {B};
6  |  |  S' := {s ∈ S | ∃t ∈ B.s → t};
7  |  |  foreach B' ∈ π with ∅ ⊂ B' ∩ S' ⊂ B' do
   |  |  |  // Split B' into B' ∩ S' and B' \ S'
8  |  |  |  π := π \ {B};
9  |  |  |  π := π ∪ {B' ∩ S', B' \ S'};
10 |  |  |  Unstable := Unstable ∪ {B' ∩ S', B' \ S'};
11 |  |  end
12 |  end
13 end
```

Algorithm 1: Sequential algorithm based on Kanellakis-Smolka

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

3 Relational Coarsest Partition Problem

3.1 A Sequential Algorithm

In this section, we discuss a sequential algorithm based on one of Kanellakis and Smolka [11] for RCPP. This is the basic algorithm which we adapt to the parallel PRAM algorithm. 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.

The sequential algorithm, Algorithm 1, works as follows. Given are a set S, a 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 *Unstable*. In any iteration of the algorithm, if a block B of the current partition is not in *Unstable*, then the current partition is stable under B. If *Unstable* is empty, the partition is stable under all its blocks, and the partition represents the required bisimulation.

As long as some blocks are in *Unstable* (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 in the subset of states that reach S' and the subset of states that do not reach S' (lines 8-9). These two new blocks are added to the set of *Unstable* blocks (line 10). 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 *Unstable*, and so the algorithm terminates.

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 $|A| = 1$ and $|S| = n$ states, we assume that the states are labeled 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.

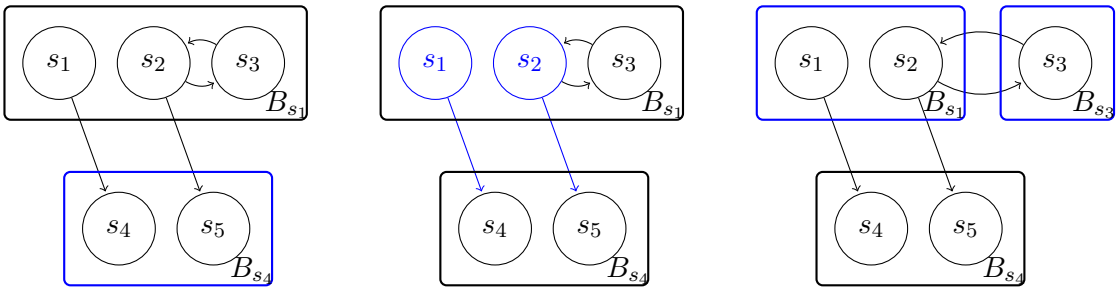
In a partition π, whenever a block $B \in \pi$ is split into two blocks B' and B'', the leader s of B which is part of B' becomes the leader of B', and for B'', a new state $t \in B''$ is elected to be the leader of this new block. Since the new leader is not part of any other block, the label of t is 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 [24, 23].

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) $unstable_i : \mathbb{B}$ indicating whether π is possibly unstable w.r.t. the block.

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 initialization, $unstable_i = \text{false}$ for all i that are not used as block label, and true otherwise.



Step 1: Select *current_block:*= B_{s_4} Step 2: Mark nodes s_1, s_2 Step 3: Split B_{s_1} into B_{s_1}, B_{s_3}

Figure 1: One iteration of Algorithm 2

3.2.3 The algorithm

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

The algorithm performs initialisation (lines 1-6), after which each block has selected a new leader (lines 3-4), ensuring that the leader is one of its own states, and the initial blocks are set to unstable. Subsequently, the algorithm enters a single loop that can be explained in three separate parts.

Splitter selection (lines 8-14), executed by n processors. Every variable $mark_i$ is set to false. After this, every processor with index i will check $unstable_i$. If block i is marked unstable the processor tries to write i in the variable C. If multiple write accesses to C happen concurrently in this iteration, then according to both the arbitrary and the priority PRAM model (see Section 2), only one process j will succeed in writing, setting $C := j$ as splitter in this iteration.

Mark states (lines 15-17), executed by m processors. Every processor i is responsible for the transition $s_i \to t_i$ 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.

Performing splits (lines 18-26), executed by 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 unstability of block $block_i$ is set to true (line 22). After that, all involved processors update the block index for their state (line 21) and update the stability of the new block (line 22).

The steps of the program are illustrated in Figure 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 unstable. 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 set to stable (line 19), all states compare their mark with the leader's mark, and the processor working on state s_3

```
1  if i < n then
       // Initialize all variables
2      unstable_i := false;
3      new_leader_{I_i} := i;
4      block_i := new_leader_{I_i};
5      unstable_{block_i} := true;
6  end
7  do
8      C := ⊥;
9      if i < n then
10         mark_i := false;
11         if unstable_i then
12             C := i;
13         end
14     end
15     if i < m and block_{target_i} = C then
16         mark_{source_i} := true;
17     end
18     if i < n and C ≠ ⊥ then
19         unstable_C := false;
20         if mark_i ≠ mark_{block_i} then
21             new_leader_{block_i} := i;
22             unstable_{block_i} := true;
23             block_i := new_leader_{block_i};
24             unstable_{block_i} := true;
25         end
26     end
27 while C ≠ ⊥;
```

Algorithm 2: The algorithm for each processor P_i in the PRAM with $i \in [0, \ldots, max(n, m)]$

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 unstable (lines 22 and 24).

The algorithm repeats execution of the **while**-loop until all blocks are marked stable.

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. Observe that 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.

A partition π induces a relation of 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 1 which is invariant throughout 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.

Lemma 1. *Let S be the input set of states, $\rightarrow\colon S \times S$ 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 iteration $k \in \mathbb{N}$:*

$$s \leftrightarrows_{\pi_0} t \implies \exists B \in \pi_k . s, t \in B$$

Proof. This is proven 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 $mark_s = mark_t = \text{false}$. Since they were in the same block, they will be in the same block in π_{k+1}.

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 $s \rightarrow s'$ with $s' \in C$. By Definition 2, there exists a $t' \in S$ such that $t \rightarrow t'$ 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 $mark_s = mark_t = \text{true}$. 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 2. *Let S be the input set of states with $\rightarrow\colon S \times S$, $L_B = S$ the block labels, and π_n the partition stored in the memory after 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 $unstable_i = \text{false}$. For a block index $i \in L_B$, $unstable_i$ 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 1, 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 $s \rightarrow s'$ 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 $t \rightarrow t'$ 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 1. *The partition resulting from executing Algorithm 2 forms the coarsest relational partition for a set of states S and a transition relation $\rightarrow\colon S \times S$, solving RCPP.*

Proof. By Lemma 2, the resulting partition is a bisimulation relation. Lemma 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 2. *RCPP on an input with m transitions and n states is solved by Algorithm 2 in $\mathcal{O}(n)$ time using $max(m, n)$ 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 unstable blocks $U_k \in \mathbb{N}$. 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 unstable is set to stable which causes the number of unstable blocks to decrease by one. In this iteration k the l_k blocks $B_1, \ldots, B_{l_k}$ are split, resulting in l_k newly created blocks. These l_k blocks are all unstable. A number of blocks $l'_k \leq l_k$ of the blocks $B_1, \ldots B_{l_k}$, were stable and are set to unstable again. The block C which was set to stable is possibly one of these l'_k blocks which were stable and set to unstable again. The total number of unstable blocks 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 $N_k = \sum_{i=0}^{k-1}(l_i) + |\pi_0|$ and unstable blocks $U_k = \sum_{i=0}^{k-1}(l_i + l'_i) - k + |\pi_0|$. 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 unstable 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 [18]. 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 $s \xrightarrow{a} s'$ to some state s'. 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 that have the same set of outgoing action labels. This is valid, since two bisimilar states must have the same outgoing actions. That two states of the same block have the same set of action labels is then an invariant of the algorithm, since in the sequence of produced partitions, each partition is a refinement of the previous one. 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 this is the following extra information:

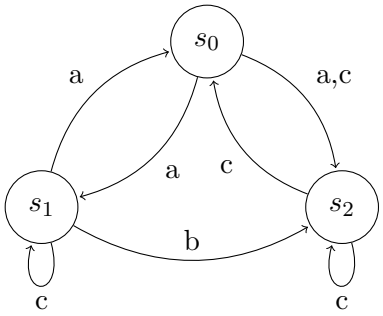
1. Each action label has an index $a \in \{0, \ldots, |Act| - 1\}$.
2. The following is stored in lists of size m, for each transition $s \xrightarrow{a} t$ with transition index $i \in \{0, \ldots, m-1\}$:
   (a) $a_i := a$
   (b) $order_i : \mathbb{N}$, the order of this action label, with respect to the source state s. E.g., 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 lists $mark_s$ for all states $s \in S$. Essentially, we have for each state $s \in S$:
   (a) $off(s) : \mathbb{N}$, the offset to access the marks of a given state s in $mark$.
   (b) $mark_{off(s)} : [\mathbb{B}]$, a list of marks (the list starting at position $off(s)$ in $mark$), where each mark indicates if the state can reach the current block with the corresponding action. We also refer to this list as $mark_s$. E.g., if state s has actions $[1, 3, 6]$ and only actions 1 and 6 can reach the current block, this list has the contents $[\text{true}, \text{false}, \text{true}]$.
   (c) nr_marks_s, the number of marks this state has, thus the length of list $mark_s$.
4. $mark\_length$: The total length of all the $mark_s$ lists together, i.e., the sum of all the nr_marks_s. 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 $split_s$. This indicates if the state will be split off from its block.

With this extra information, we can alter 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 splitted off when it differs for any action label from the block leader.
5. Line 24: If a state was tagged to be splitted off in the previous step, it should split off from its leader.
6. Line 29: If any block was split, the partition may not be stable w.r.t. the splitter.

```
1  if i < n then
2  |  unstable_i := false;
3  |  unstable_{block_i} := true;
4  end
5  do
6  |  C := ⊥;
7  |  if i < mark_length then
8  |  |  mark_i := false;
9  |  end
10 |  if i < n then
11 |  |  split_i := false;
12 |  |  if unstable_i then
13 |  |  |  C := i;
14 |  |  end
15 |  end
16 |  if i < m and block_{target_i} = C then
17 |  |  mark_{off(source_i)+order_i} := true;
18 |  end
19 |  if i < m and mark_{off(source_i)+order_i} ≠ mark_{off(block_{source_i})+order_i} then
20 |  |  split_{source_i} := true;
21 |  end
22 |  if i < n and C ≠ ⊥ then
23 |  |  unstable_C := false;
24 |  |  if split_i then
25 |  |  |  new_leader_{block_i} := i;
26 |  |  |  unstable_{block_i} := true;
27 |  |  |  block_i := new_leader_{block_i};
28 |  |  |  unstable_{block_i} := true;
29 |  |  |  unstable_C := true;
30 |  |  end
31 |  end
32 while C ≠ ⊥;
```

Algorithm 3: The Algorithm for BCCP, the highlighted lines differ from Algorithm 2.



$source_i$	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_mark_s			2			3		1
off			2			5		6

Figure 2: An example LTS and its derived preprocessing information.

4.1.1 Preprocessing.

To use the above algorithm, 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 at line 15, 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 found in Algorithm 4.

```
15 if i < m and a_i = a then
16 |   mark_{source_i} := true;
17 end
```

Algorithm 4: Marking the source per action label a_i.

After executing Algorithm 4, each block can split in 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 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 others 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 [5]. In order to calculate $order_i$ and nr_marks_s, we first calculate $action\_switch_i$ for each transition i, which is done in Algorithm 5. See Figure 2 for an example. Now, $order_i$ can be calculated with a parallel segmented prefix sum [19] (also called a segmented scan) of $action\_switch$. A parallel segmented sum can be performed on $action\_switch$ to calculate nr_marks_s, 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 prefix sum on nr_marks_s. The code in Algorithm 5 takes $\mathcal{O}(1)$ time on m processors, and a parallel segmented (prefix) sum takes $\mathcal{O}(\log m)$ time [19].

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

4.2 Complexity & Correctness

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 produce a stable block.

Theorem 3. *Algorithm 3 on an input LTS with n states and m transitions will terminate in $\mathcal{O}(n + |Act|)$ steps.*

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 still executed in constant time. Using the structure of the proof of Theorem 2, we derive a bound on the number of iterations.

At the start of iteration $k \in \mathbb{N}$ the total number of blocks and unstable 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 in two blocks, resulting

```
if i ≤ m then
    if i = 0 or source_i ≠ source_{i−1} or a_i = a_{i−1} then
        action_switch_i = 0;
    else
        action_switch_i = 1;
    end
end
```

Algorithm 5: Preprocessing step needed for Algorithm 3. We calculate $action\_switch_i$, which is needed for the $order_i$ and nr_marks_s variables.

in l_k new blocks, meaning that $N_{k+1} = N_k + l_k$. All new l_k blocks are unstable and a number $l'_k \leq l_k$ of the old blocks that are split, were stable at the start of iteration k and now unstable. If $l_k = l'_k = 0$ there are no blocks split and the current block C becomes stable. 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 unstable blocks. In iteration k there are $U_k = \sum_{i=0}^{k-1}(l_i + l'_i) - \sum_{i=0}^{k-1} c_i + |\pi_0|$ unstable 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 unstable blocks 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 total run time complexity $\mathcal{O}(n + |Act| + \log m)$. Since the total number of transitions m is bounded by $|Act| \times n^2$, this simplifies to $\mathcal{O}(n + |Act|)$. □

Concerning correctness, we need to address two things. Firstly, as argued above, we start with a different partition compared to Algorithm 2, but it is a valid choice since states with different outgoing labels can never be bisimilar. Secondly, although the partition may not become stable w.r.t. the splitter, this will eventually occur, and the algorithm will only stop once the partition is stable w.r.t. all blocks. Therefore, the algorithm will produce the coarsest bisimulation relation.

5 Experimental Results

In order to validate the proposed algorithm, we implemented Algorithm 3 from Section 4. The implementation targets graphics processing units (GPUs) since a GPU closely resembles a PRAM and supports a large amount of parallelism. The algorithm is implemented in CUDA version 11.1 with use of the Thrust library.[1] As input, we chose all benchmarks of the VLTS benchmark suite[2] for which the implementation produced a result within 10 minutes. The

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

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

Benchmark name	n	m	$\|Act\|$	$\|Blocks\|$	$\#It$	T_{pre}	T_{alg}	T_{total}	$\#It/n$	$\#It/\|Blocks\|$	T_{total}/n	$T_{alg}/\#It$
Vasy_0_1	289	1,224	2	9	16	0.50	0.37	0.87	0.06	1.78	0.003	0.023
Cwi_1_2	1,952	2,387	26	1,132	2,786	0.63	56.5	57.1	1.43	2.46	0.029	0.020
Vasy_1_4	1,183	4,464	6	28	45	0.56	1.01	1.58	0.04	1.61	0.001	0.022
Cwi_3_14	3,996	14,552	2	62	122	0.63	2.68	3.30	0.03	1.97	0.001	0.022
Vasy_5_9	5,486	9,676	31	145	193	0.84	4.22	5.06	0.04	1.33	0.001	0.022
Vasy_8_24	8,879	24,411	11	416	664	0.70	13.9	15	0.07	1.59	0.002	0.021
Vasy_8_38	8,921	38,424	81	219	319	1.12	6.64	7.76	0.04	1.46	0.001	0.021
Vasy_10_56	10,849	56,156	12	2,112	3,970	0.73	82.0	82.7	0.37	1.88	0.008	0.021
Vasy_18_73	18,746	73,043	17	4,087	6,882	1.01	142	143	0.37	1.68	0.008	0.021
Vasy_25_25	25,217	25,216	25,216	25,217	25,218	159	519	678	1.00	1.00	0.027	0.021
Vasy_40_60	40,006	60,007	3	40,006	87,823	0.87	1,810	1,811	2.20	2.20	0.045	0.021
Vasy_52_318	52,268	318,126	17	8,142	15,985	2.52	338	340	0.31	1.96	0.007	0.021
Vasy_65_2621	65,537	2,621,480	72	65,536	98,730	12.2	10,050	10,060	1.51	1.51	0.154	0.102
Vasy_66_1302	66,929	1,302,664	81	66,929	91,120	6.70	5,745	5,752	1.36	1.36	0.086	0.063
Vasy_69_520	69,754	520,633	135	69,754	113,246	4.13	3,780	3,780	1.62	1.62	0.054	0.033
Vasy_83_325	83,436	325,584	211	83,436	148,012	4.41	3,093	3,097	1.77	1.77	0.037	0.021
Vasy_116_368	116,456	368,569	21	116,456	210,537	2.50	5,900	5,900	1.81	1.81	0.051	0.028
Cwi_142_925	142,472	925,429	7	3,410	5,118	4.85	238	243	0.04	1.50	0.002	0.047
Vasy_157_297	157,604	297,000	235	4,289	9,682	4.58	201	206	0.06	2.26	0.001	0.021
Vasy_164_1619	164,865	1,619,204	37	1,136	1,630	8.34	125	134	0.01	1.43	0.001	0.077
Vasy_166_651	166,464	651,168	211	83,436	145,029	6.13	5,710	5,720	0.87	1.74	0.034	0.039
Cwi_214_684	214,202	684,419	5	77,292	149,198	3.58	6,948	6,952	0.70	1.93	0.032	0.047
Cwi_371_641	371,804	641,565	61	33,994	85,858	4.72	4,050	4,050	0.23	2.53	0.011	0.047
Vasy_386_1171	386,496	1,171,872	73	113	199	7.38	14.0	21	0.00	1.76	0.000	0.070
Cwi_566_3984	566,640	3,984,157	11	15,518	23,774	16.0	3,707	3,723	0.04	1.53	0.007	0.156
Vasy_574_13561	574,057	13,561,040	141	3,577	5,860	71.5	3,770	3,841	0.01	1.64	0.007	0.643
Vasy_720_390	720,247	390,999	49	3,292	3,782	3.97	143	147	0.01	1.15	0.0002	0.038
Vasy_1112_5290	1,112,490	5,290,860	23	265	365	24.0	99.3	123	0.0003	1.38	0.0001	0.272
Cwi_2165_8723	2,165,446	8,723,465	26	31,906	66,132	37.0	23,660	23,700	0.03	2.07	0.011	0.358
Cwi_2416_17605	2,416,632	17,605,592	15	95,610	152,099	64.1	96,400	96,500	0.06	1.59	0.040	0.634
Vasy_6020_19353	6,020,550	19,353,474	511	7,168	12,262	221	11,690	11,910	0.002	1.71	0.002	0.954
Vasy_6120_11031	6,120,718	11,031,292	125	5,199	10,014	74.0	6,763	6,837	0.002	1.93	0.001	0.675
Vasy_8082_42933	8,082,905	42,933,110	211	408	660	281	1,149	1,429	0.0001	1.62	0.0002	1.739

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

VLTS benchmarks are LTSs that have been 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 1,024 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.

Our implementation is purely a proof of concept, to show that our algorithm can be mapped to actual hardware and to understand how the algorithm scales with the number of states and transitions.

In the implementation, we have to make a few adjustments, since a GPU differs in some aspects from a PRAM. To make memory updates globally visible, we need to synchronize at certain points of Algorithm 3, otherwise the changes in the memory are not consistent. We do this by splitting up the algorithm in different kernels (functions that execute in parallel on a GPU) since after a kernel run all processors (threads) are synchronized.

To be precise, in Algorithm 3 we need to synchronize after:

- **Line 15**: To make sure the *mark* and *split* lists are reset and the splitter (C) is the same for all threads.

- **Line 18**: To make sure every thread has the same view of the *mark* list.

- **Line 21**: To synchronize the *mark* list.
- **Line 25**: To make sure the next leader for states that split off ($new\_leader_{block_i}$) is chosen consistently among threads.

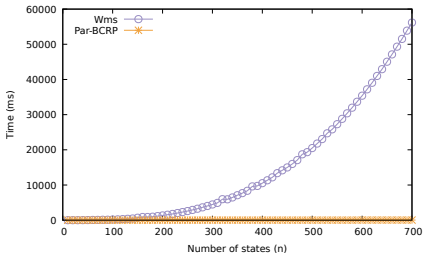
We have chosen to allow race conditions in our implementation, for instance at Line 6 where multiple blocks can mark themselves as current (C). Strictly speaking, this is not safe in the CUDA programming model, but it does work for 32 bit words. This can be easily adjusted using atomic instructions, although this will result in sequentializing write accesses to the same memory location, meaning that a write need not be in constant time anymore.

To ensure the implementation also works when n and/or m is larger than the number of threads d on the GPU, we encapsulate the **if-then** blocks at lines 1-4, 7-9, 10-15, 16-18, 19-21 and 22-31 of Algorithm 3 each in a **for**-loop, in which every thread accesses not only the data elements associated with its global index i, but also, if needed, the elements with index $i + d$, $i + 2d$, etc., as long as the indices are valid.

Table 1 shows the results of the experiments we conducted. 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), before all blocks became stable. The T_{pre} give the preprocessing times in milliseconds, which includes doing the memory transfers to the GPU, sorting the transitions and partitioning. The T_{alg} give the times of the core algorithm, in milliseconds. The T_{total} 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 initializes 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 proven in Section 4.2 ($k \leq 3n$) is indeed respected, $\#It/n$ is at most 2.20, and most of the time below that. The number of iterations is tightly related to the amount 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 stable, and all blocks must be checked on stability 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.

Concerning the GPU 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 column headed by $T_{alg}/\#It$, which corresponds to the time per iteration. The values are around 0.02 up to $300,000$ transitions, but for a higher number of states and transitions, the amount of time per iteration increases.



States	Run time Wms (ms)	Par-BCRP (ms)
10	1	1
100	182	5
200	1350	9
300	4463	13
400	10519	18
500	20508	22
600	35392	26
700	56183	30

Figure 3: Run times of Par-BCRP on the LTS Fan_out_n.

5.1 Experimental comparison

We compared our implementation (Par-BCRP) with an implementation of the algorithm by Lee and Rajasekaran (LR) [13] on GPUs, and the optimized GPU implementation by Wijs based on *signature-based* bisimilarity checking [2], with *multi-way splitting* (Wms) and with *single-way splitting* (Wss) [23]. Multi-way splitting indicates that a block is split in multiple blocks at once, which is achieved in signature-based algorithms 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.

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 non-deterministic 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 between parentheses. Furthermore, all the results are rounded with respect to the standard error of the mean. As a pre-processing step for the LR, Wms and Wss algorithms the input LTSs need to be sorted. We did not include this in the times, nor the reading of files and the first CUDA API call (which initializes the GPU).

This comparison confirms the expectation that our algorithm in all cases (except one small LTS) out-performs LR. This confirms our expectation that LR is not suitable for massive parallel devices such as GPUs. Furthermore, the comparison teaches that in most cases our algorithm (Par-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 us that our algorithm does not out-perform Wms. Wms employs multi-way splitting which is known to be very effective in practice. Contrary to our implementation, Wms is highly optimized 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 between Wms and our algorithm better, we analysed the complexity of Wms [23]. In general this algorithm is quadratic in time, and the linearity claim in [23] 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 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

Benchmark name	LR	Wms	Wss	Par-BCRP
Vasy_0_1	2.29	0.45	0.49	0.87
Cwi_1_2	17	21.8	18.8	57.1
Vasy_1_4	4.78	0.62	1.68	1.58
Cwi_3_14	60	3.72	3.80	3.30
Vasy_5_9	134	3.45	35.3	5.06
Vasy_8_24	277	3.03	31.5	15
Vasy_8_38	127	5.94	35.1	7.76
Vasy_10_56	860	4.6(0.2)	40.9	82.7
Vasy_18_73	1,354	21.7	211	143
Vasy_25_25	21,960	416	t.o.	678
Vasy_40_60	17,710	1,230	1,290	1,811
Vasy_52_318	11,855	152(20)	368	340
Vasy_65_2621	t.o.	1,230	27,000	10,060
Vasy_66_1302	480,600	240(20)	20,450	5,752
Vasy_69_520	94,800	35.4	16,090	3,780
Vasy_83_325	57,190	5,880	21,500	3,097
Vasy_116_368	80,900	2,930	6,360	5,900
Cwi_142_925	3,363	140(20)	220(30)	243
Vasy_157_297	1,058	579	1,240	206
Vasy_164_1619	8,173	46.8	470(30)	134
Vasy_166_651	80,210	9,560	29,660	5,720
Cwi_214_684	19,250	450(50)	440(30)	6,952
Cwi_371_641	26,940	1,548	6,970	4,050
Vasy_386_1171	334	34.8	30.6	21
Cwi_566_3984	98,200	2,200(200)	6,700	3,723
Vasy_574_13561	144,810	1,853	11,700	3,841
Vasy_720_390	2,454	183	1,633	147
Vasy_1112_5290	4,570	36.8	293	123
Cwi_2165_8723	140,170	1,965	9,700	23,700
Cwi_2416_17605	257,200	15,300	16,300(1100)	96,500
Vasy_6020_19353	107,900	19,230	34,000(2000)	11,910
Vasy_6120_11031	55,750	1,280	7,010	6,837
Vasy_8082_42933	17,272	2,030	5,530	1,429

Table 2: Comparison of the different algorithms with times in ms.

```
1  if mark_i ≠ mark_{block_i} then
2  |  new_leader_i := block_i;
3  |  if i < j and new_leader_i = new_leader_j then
4  |  |  new_leader_j := 0;
5  |  end
6  |  if new_leader_i ≠ 0 then
7  |  |  new_leader_{block_i} := i;
8  |  end
9  |  unstable_{block_i} := true;
10 |  block_i := new_leader_{block_i};
11 |  unstable_{block_i} := true;
12 end
```

Algorithm 6: Leader election for a *common* CRCW PRAM

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.

Figure 3 shows the results of calculating the bisimulation equivalence classes for Fan_out_n, with Wms and Par-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 amount of states. On the other hand, in conformance with our analysis, our algorithm scales linearly.

6 Weaker PRAM models

Algorithm 2 relies on concurrent writes to perform the constant time leader election and the choice of splitter. This means that the algorithm does not work on a weaker PRAM model. In this section we describe a modification for the *common* CRCW PRAM and a limitation for the ERCW PRAM.

It is shown in [12] that any *priority* CRCW PRAM using n processors and m memory cells can be simulated by a *common* CRCW PRAM with $\mathcal{O}(n^2)$ processors and $\mathcal{O}(m^2)$ memory cells. For our problem, a *common* CRCW PRAM with $\mathcal{O}(n^2)$ processors and no extra memory can solve leader election.

This leader election on the *common* CRCW PRAM is given in Algorithm 6. Every processor is indexed as $P_{i,j}$ for all $i, j \in \{0, \ldots, n-1\}$ for exactly n^2 processors. First, if $P_{i,j}$ has a state with index i that is eligible to be the leader of a new block (line 1), it writes $block_i$, i.e., the index of the block the state is currently a member of, to position i in a list new_leader. In the next step, $P_{i,j}$ replaces new_leader_j with 0 if $new\_leader_i = new\_leader_j$ and $i < j$. In other words, if $P_{i,j}$ encounters two states that can become the new leader, it selects the one with the smallest index. This is possibly a concurrent write, but all writes involve the same value 0, hence this is allowed by the common CRCW PRAM. Next, if for $P_{i,j}$, $new\_leader_i \neq 0$, it writes the value i to $new\_leader_{block_i}$ at line 7. For a given block $block_i$, the condition at line 6 only holds for the state with the largest index among the states that are split from $block_i$, hence there is at most one value is written.

Leader election on the ERCW PRAM is not possible in constant time, which follows from a result by Cook et al. [6, Thm 4.]. This result says that all functions that have a *critical* input are in $\Omega(log\ n)$ on ERCW PRAMs. A bit sequence I of size n is critical for a function

$f : \{0,1\}^n \to \{0,1\}$ iff for any I' obtained by flipping exactly one bit in I we have $f(I) \neq f(I')$. Leader election can be seen as a function $f : S \to \{0,1\}$, where $f(i) = 1$ iff i is elected as a new leader. This function has a critical input, namely the to be chosen leader.

7 Related work

In [13] Lee and Rajasekaran study RCPP. They implement a parallel version of Kanellakis-Smolka that runs in $\mathcal{O}(n\ log\ n)$ time on $\frac{m}{log\ n} log\ log\ n$ CRCW PRAM processors. In [17] they present a different algorithm based on Paige and Tarjan's algorithm [16] that has the same run time of $\mathcal{O}(n\ log\ n)$ but using only $\frac{m}{n} log\ n$ CREW processors. Jeong et al. [10] presented a linear time parallel algorithm, but it is probablistic in the sense that it has a non-zero chance to output the wrong result. Furthermore, Wijs [23] presented a GPU implementation of an algorithm to solve the strong and branching bisimulation partition refinement problems but although efficient for many practical cases, it has a quadratic time complexity.

In a distributed setting, Blom and Orzan studied algorithms for refinement [2]. Those algorithms use message passing as ways of communication between different workers in a network and rely on a small number of processors. Therefore, they are very different in nature than our algorithm. Those algorithms were extended and optimized for branching bisimulation [3].

8 Conclusion

We proposed and implemented an algorithm for RCPP and BCPP. We proved that the algorithm stops in $\mathcal{O}(n + |Act|)$ steps on $max(n, m)$ CRCW PRAM processors. We implemented the algorithm for BCPP in CUDA, and conducted experiments that show the potential to compute bisimulation in practice in linear time. Further advances in parallel hardware will make this more feasible.

For future work, it is interesting to investigate whether RCPP can be solved in sublinear time, that is $\mathcal{O}(n^\epsilon)$ for a $\epsilon < 1$, as requested in [13]. It is also intriguing whether the practical effectiveness of the algorithm in [23] by splitting blocks simultaneously can be combined with our algorithm, while preserving the linear time upperbound. Furthermore, it remains an open question whether our algorithm can be generalised for weaker bisimulations, such as weak and branching bisimulation [22, 9]. The main challenge here is that the transitive closure of so-called internal steps needs to be taken into account.

Acknowledgments

This work is carried out in the context of the NWO AVVA project 612.001751 and the NWO TTW ChEOPS project 17249.

References

[1] J. Balcázar, J. Gabarro, and M. Santha. Deciding bisimilarity is p-complete. *Formal aspects of computing*, 4(1):638–648, 1992.

[2] S. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. *Electronic Notes in Theoretical Computer Science*, 89(1):99–113, 2003.

[3] S. Blom and J. van de Pol. Distributed branching bisimulation minimization by inductive signatures. In Lubos Brim and Jaco van de Pol, editors, *Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009*, volume 14 of *EPTCS*, pages 32–46, 2009. URL: https://doi.org/10.4204/EPTCS.14.3, doi:10.4204/EPTCS.14.3.

[4] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. J. Wijs, and T. A. C. Willemse. 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, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, pages 21–39, 2019. doi:10.1007/978-3-030-17465-1_2.

[5] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[6] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[7] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In *LITP Spring School on Theoretical Computer Science*, pages 407–419. Springer, 1990.

[8] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, 1978.

[9] D.N. Jansen, J.F. Groote, F.J. A. Keiren, and A.J. Wijs. An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 12079, pages 3–20. Springer International Publishing, 2020. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-030-45237-7_1, doi:10.1007/978-3-030-45237-7_1.

[10] C. Jeong, Y. Kim, Y. Oh, and H Kim. A faster parallel implementation of kanellakis-smolka algorithm for bisimilarity checking. In *Proceedings of the international computer symposium*. Citeseer, 1998.

[11] P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43 – 68, 1990. URL: http://www.sciencedirect.com/science/article/pii/089054019090025D, doi:https://doi.org/10.1016/0890-5401(90)90025-D.

[12] L. Kučera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14(2):93–96, 1982.

[13] I. Lee and S. Rajasekaran. A parallel algorithm for relational coarsest partition problems and its implementation. In David L. Dill, editor, *Computer Aided Verification*, volume 818, pages 404–414. Springer Berlin Heidelberg, 1994. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/3-540-58179-0_71, doi:10.1007/3-540-58179-0_71.

[14] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495), 2020. URL: https://science.sciencemag.org/content/368/6495/eaam9744, arXiv:https://science.sciencemag.org/content/368/6495/eaam9744.full.pdf, doi:10.1126/science.aam9744.

[15] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. URL: https://doi.org/10.1007/3-540-10235-3, doi:10.1007/3-540-10235-3.

[16] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[17] S. Rajasekaran and I. Lee. Parallel algorithms for relational coarsest partition problems. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):687–699, 1998.

[18] M. A. Reniers, R. Schoren, and T.A.C. Willemse. Results on embeddings between state-based and event-based systems. *The Computer Journal*, 57(1):73–92, 2014.

[19] S. Sengupta, M. Harris, M. Garland, and J. Owens. Efficient Parallel Scan Algorithms for GPUs. In *Scientific Computing with Multicore and Accelerators*, chapter 19, pages 413–442. Taylor & Francis, 2011.

[20] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13(2):409–422, 1984.

[21] A. Valmari. Simple bisimilarity minimization in $O(m \log n)$ time. *Fundam. Informaticae*, 105(3):319–339, 2010. URL: https://doi.org/10.3233/FI-2010-369, doi:10.3233/FI-2010-369.

[22] R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[23] A.J. Wijs. GPU accelerated strong and branching bisimilarity checking. In *TACAS*, volume 9035 of *LNCS*, pages 368–383. Springer, 2015. doi:10.1007/978-3-662-46681-0_29.

[24] A.J. Wijs, J.-P. Katoen, and D. Bošnački. Efficient GPU Algorithms for Parallel Decomposition of Graphs into Strongly Connected and Maximal End Components. *Formal Methods in System Design*, 48(3):274–300, 2016.