Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Distributed State Space Minimization

Stefan Blom[1], **Simona Orzan**[2] ⋆

[1] CWI, The Netherlands, e-mail: `sccblom@cwi.nl`
[2] Eindhoven University of Technology, The Netherlands, e-mail: `s.m.orzan@tue.nl`

The date of receipt and acceptance will be inserted by the editor

Abstract We present a new algorithm, and its distributed implementation, for reducing labeled transition systems modulo strong bisimulation. The base of this algorithm is the Kanellakis-Smolka 'naive method', which has a high theoretical complexity, but is succesful in practice and well suited to parallelization. This basic approach is combined with optimizations inspired by the Kanellakis-Smolka algorithm for the case of bounded fanout, which has the best known time complexity. The distributed implementation is improved with respect to previous attempts by a better overlap between communication and computation, which results in an efficient usage of both memory and processing power. We also discuss the time complexity of this algorithm and show experimental results with sequential and distributed prototype tools.

1 Introduction

There is currently a lot of interest in building distributed model checking tools, both symbolic [11,2] and enumerative [24, 16,15,1]. The symbolic tools manipulate a compressed representation of the state space. The enumerative tools explicitly compute all the states and transitions of the state space and can be sub-divided into on-the-fly and full-generation. An on-the-fly tool will compute the transitions leading from a state on demand, while it is checking a property. A full-generation tool will first compute the whole state space and only then start checking the property. The main advantage of on-the-fly tools is that if the property can be proved or dis-proved exploring only a small part of the state-space, the unnecessary generation of the rest of the state space is avoided. However, if proving a property requires visiting the whole state space then full-generation tools have an important advantage: after being generated, a state space can be reduced modulo an equivalence that preserves the properties to be checked. The reduction can considerably reduce the size of the state space that needs to be verified. This is particularly important in cases where the original state space is too big to be verified on a single machine, like in two recent case studies with the μCRL toolset: a cache coherence protocol [20] and a model of JavaSpaces [21]. In both cases, verification on a single machine was possible for the reduced state space only. Moreover, state space reduction could only be performed using the distributed tool.

This paper proposes a new distributed solution for the problem of state space reduction modulo strong bisimulation equivalence. This is a widely used system equivalence which preserves all properties expressible as μ-calculus formulas.

We choose clusters of workstations as target architecture because it is the most common environment able to offer the memory and processing power required by model checking industrial applications. So, we are interested in message-passing algorithms that can handle very large problem instances on a comparatively small number of processors and that work well for the specific type of labeled transition systems representing state spaces. State spaces have bounded fanout and usually a small depth. The states are distributed evenly among the network nodes (workers), and the transitions are managed by the worker that owns their initial states.

History The strong bisimulation reduction is found in the literature under the name *multiple relational coarsest partition problem* (MRCPP): given a set S and a number of relations on S, $\rho_1 \cdots \rho_r$, find a partition of S into subsets $S_1 \cdots S_s$ such that for any two subsets S_i, S_j and any relation ρ_k, either all or none of the elements of S_i are in the relation ρ_k with an element of S_j. Moreover, the coarsest partition with this property is required, that is the one with the least number of subsets. MRCPP is an immediate generalization of the *relational coarsest partition problem* (RCPP), which

⋆ Partially supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009.

treats the case when the set of relations is a singleton. Let N, M the sizes of the set S and of the relation, respectively. The most well known solutions for RCPP are the $\mathcal{O}(MN)$ one proposed by Kanellakis and Smolka [14] and the later $\mathcal{O}(M \log N)$ one by Paige and Tarjan [19]. [14] also contains an $\mathcal{O}(N \log N)$ solution for the restricted case of bounded fanout. Long before these, Hopcroft described an $\mathcal{O}(N \log N)$ algorithm for the deterministic case [12], when the relation is a function.

Towards a good distributed solution The main challenge in building good distributed tools is dividing the computation in such a way that communication is triggered rather infrequently, but in the same time avoiding large idle times. Ideally, workers have equal computation loads and they rarely need access to remote data. In our case, workers should be able to compute as much as possible from the states and transitions that they own.

All the algorithms above are based on partition refinement and they vary in how the refinement step is defined. In the naive algorithm of [14], the refinement consists in putting in different blocks any two states that can be distinguished with respect to the current partition. To distinguish states, it suffices to compare the set of transitions going out, i.e., the set of pairs (label, destination block). This ensures an independent treatment of the states, very suitable for parallelization. In the other, theoretically better, algorithms, it is essential that the states in the same block could be easily retrieved. To achieve this, extra administration is necessary (like sorting), which can be very expensive in a distributed setting.

A distributed implementation of the Kanellakis-Smolka naive algorithm has been presented in [5]. However, the question remains how to distribute one of the theoretically better algorithms, or at least use some of their tricks. Therefore in this paper we propose a new algorithm that keeps the simplicity and symmetry of the naive algorithm, while employing some optimizations similar to those used in the bounded fanout Kanellakis-Smolka [14] or Paige-Tarjan [19]. We will refer to the new algorithm as "optimized".

Naive versus optimized In our implementations, a unique ID (an integer) is assigned to each block and partitions are represented as arrays of IDs. The signature of a state x with respect to a partition is a set of pairs of labels and IDs, such that a pair (a, id) is in the set if and only if there is a transition with the label a from the state x to another state belonging to the block with the ID id. Two states are distinguishable with respect to a partition if they have different signatures with respect to that partition.

The naive algorithm [5] computes the signatures of all states in every iteration and randomly assigns IDs to each signature. It terminates when the number of signatures becomes stable.

The optimized algorithm doesn't recompute the signatures on each iteration. Instead, it modifies the old signatures. While this recomputation goes on, the states with modified signatures are marked. Next, we assign new IDs to the signatures of marked states as follows: if only some of the states in a block are marked then the signatures of the marked states all get new IDs and the unmarked states keep their old ID; if all states in a block are marked then the old ID is reused for the signature which occurs most often and new IDs are assigned to the others. This is similar to the strategy used in the Hopcroft or the Paige-Tarjan algorithm, which always split with respect to the smallest block. The algorithm terminates if there are no more changes.

A close variation of the optimized algorithm achieves a time complexity of $\mathcal{O}(N \log N)$. Namely, this is the case if the old ID is always given to the most often occurring signature, not only when all states of the old block are unstable. However, to implement this, all the states belonging to a given block should be easily retrievable and this is not trivial to arrange in a distributed setting.

Related work Parallel versions of Kanellakis-Smolka and Paige-Tarjan have been proposed [25,22], with time complexities $\mathcal{O}(N^{1+\epsilon})$ using $\frac{M}{N^\epsilon}$ CREW PRAM processors (for any fixed $\epsilon < 1$), and $\mathcal{O}(N \log N)$ with $\mathcal{O}(\frac{M}{N})$ CREW PRAM processors, respectively. These algorithms are designed for shared memory machines and they are difficult to translate efficiently to a distributed memory setting. It would however be interesting to see how they work on a virtual shared memory. We expect that the latency of the shared memory simulation would seriously affect their performance.

There exist on-the-fly algorithms for *bisimilarity checking*, both sequential [18] and distributed [13]. They are based on solving boolean equation systems and can be used to compare two state spaces with respect to an equivalence notion, while generating them. Our problem is rather to find the equivalence classes of a given state space, which is quite different and cannot be immediately solved by these algorithms. In fact, we are not aware of any algorithm that attempts to solve on-the-fly bisimilarity reduction.

Overview The next section introduces some definitions and formalizes the problem of reduction modulo strong bisimulation equivalence. In section 3 the optimization of the naive algorithm by using a marking procedure is discussed and it is justified that the (sequential) algorithm thus obtained is still correct. Also the theoretical complexity of the new algorithm is discussed. Then, in section 4, the distributed implementation of this new optimized algorithm is briefly explained. Some performance data is presented in section 5 and some concluding remarks in section 6.

2 Bisimilarity checking, bisimulation minimization and the Relational Coarsest Partition Problem

Let Act be a fixed set of labels, representing actions. A labeled transition system (LTS) is a triple (S, T, s_0) consisting of a set of states S, a set of transitions $T \subseteq S \times \mathsf{Act} \times S$ and an initial state $s_0 \in S$. When T is understood, we will use the notation $p \xrightarrow{a} q$ for $(p, a, q) \in T$.

Definition 1. (strong bisimulation)
Let (S, T, s_0) be an LTS. A binary relation $R \subseteq S \times S$ is a *strong bisimulation* if for all $p, q \in S$ such that $p\ R\ q$:

– if $p \xrightarrow{a} p'$ then $\exists q' \in S : q \xrightarrow{a} q' \ \wedge\ p'\ R\ q'$ and
– if $q \xrightarrow{a} q'$ then $\exists p' \in S : p \xrightarrow{a} p' \ \wedge\ p'\ R\ q'$

If a strong bisimulation R exists, such that $p\ R\ q$, then we say that p and q are *bisimilar states*. (S^1, T^1, s_0^1) and (S^2, T^2, s_0^2), with $S^1 \cap S^2 = \emptyset$, are *bisimilar labeled transition systems* if their initial states s^1 and s^2 are bisimilar in the compound LTS $(S^1 \cup S^2, T^1 \cup T^2, s_0^1)$.

The problem that we focus on, *bisimulation minimization*, is to find the equivalence classes of the largest strong bisimulation on the states of a given LTS. Or, in other words, given an LTS, find the LTS that is strongly bisimilar to it and has the minimal number of states.

A related problem is that of *bisimilarity checking*: given an LTS $\mathcal{S} = (S, T, s_0)$ and two states $p, q \in S$, decide whether p and q are strongly bisimilar. This problem reduces to the minimization problem, since it suffices to check whether p and q are in the same equivalence class of the largest bisimulation relation definable on the states of $\mathcal{S}$. The way of deciding whether two transition systems represent the same behavior is to apply a bisimulation minimization algorithm to the compound LTS $(S^1 \cup S^2, T^1 \cup T^2, s_0^1)$ and see whether s_0^1 and s_0^2 end up in the same class.

The bisimilarity minimization problem is equivalent to the *relational coarsest partition problem*. For an LTS (S, T, s_0), a *partition* of the elements of S is a set of disjoint blocks $\{B_i \mid i \in I\}$ s.t. $\cup_{i \in I} B_i = S$. An equivalence relation can be represented as a partition with a block for every equivalence class. The relational coarsest partition problem is to find, for a given LTS and a given initial partition π_0 of S, a partition π s.t.:

- π is a refinement of π_0
- $\forall p, q \in B \in \pi :\ \forall a \in \mathsf{Act} :\ \forall B' \in \pi :$
  $\exists p' \in B' :\ (p, a, p') \in T$
  $\iff\ \exists q' \in B'\ (q, a, q') \in T$
- π has the fewest blocks among partitions satisfying the above two conditions.

(A partition π' is a refinement of π if every block of π' is contained in a block of π: $\forall C \in \pi' :\ \exists B \in \pi :\ C \subseteq B$).

The algorithm discussed in this paper solves the bisimulation minimization problem by solving the Relational Coarsest Partition Problem with $\pi_0 = \{S\}$.

3 The optimized algorithm

In [5], an algorithm is presented that uses the set of all outgoing transitions (signatures) as criteria to distinguish states, as opposed to theoretically more efficient algorithms that reason in terms of blocks. The *signature* of a state s with respect to a partition π is the set of all transitions going from s to blocks of π:

$$sig_\pi(s) = \{(a, B) \mid s \xrightarrow{a} s' \text{ and } s' \in B \in \pi\}$$

```
1.  E := 1;  c(0) := |S|;  U := S;
2.  for all x ∈ S
        ID(x) := 0 ; sig(x) := ∅;
3.  while U ≠ ∅
4.    for all x ∈ U
          sig(x) := {(a, ID(y)) | x -a-> y};
5.    Reusable := {i | 0 ≤ i < E
                  ∧ c(i) = |U ∩ {x | ID(x) = i}|}
6.    ST := ∅; νU := ∅;
7.    for all x ∈ U
8.        oid := ID(x);
9.        if (sig(x), i) ∈ ST
          then
10.           ID(x) := i
          else
11.           if ID(x) ∉ Reusable
              then
12.               c(E) := 0;
13.               ID(x) := E;
14.               E := E + 1;
              else
15.               Reusable := Reusable − {ID(x)};
16.           ST := ST ∪ {(sig(x), ID(x))};
17.       if oid ≠ ID(x)
          then
18.           νU := νU ∪ {y ∈ S | y -a-> x}
19.           c(oid) := c(oid) − 1;
20.           c(ID(x)) := c(ID(x)) + 1;
21.   U := νU;
22. for all x ∈ S   ID^f(x) := ID(x);
```

Figure 1. (OSBR) The optimized algorithm

Two states are distinguishable with respect to a partition if and only if they have different signatures with respect to that partition. A partition π is called *stable* if every two states of every block in that partition are undistinguishable, i.e. have the same signature with respect to π.

While performing successive refinements, the signature-based bisimulation minimization algorithm keeps the states with the same signature in the same block. The correctness of this method follows from two facts. First, a stable partition is a bisimulation relation (states are equivalent if they are in the same block). Second, bisimilar states have the same signature with respect to every computed refinement. Hence, the computed bisimulation relation is the coarsest one.

We indicate the current partition by a function $\mathsf{ID} : S \longrightarrow Nat$ that assigns block identifiers to states. The naive algorithm proceeds roughly as follows (E is the number of equivalence classes in the current partition, $o\mathsf{E}$ is the number of

equivalence classes in the previous partition):

$\forall x \in S\ \mathsf{ID}(x) := 0;\ \mathsf{E} := 1; o\mathsf{E} := 0;$
while $\mathsf{E} \neq o\mathsf{E}$
$\forall x$ compute $sig(x)$
assign new IDs to states, s.t.
$\mathsf{ID}(x) = \mathsf{ID}(y) \iff sig(x) = sig(y)$
$o\mathsf{E} := \mathsf{E}; \mathsf{E} :=$ no. of new IDs used

In this scheme, all signatures are recomputed in every iteration, which can be an unnecessary and costly effort in the case of large input LTSs with a structure that needs a lot of iterations to stabilize and where very few partition blocks can be split per iteration (very few signatures actually change).

The main idea of our new optimized approach is to mark, in every iteration, those states that might have suffered a signature change, i.e. the states that have an outgoing transition to a state whose ID changed in the current iteration. Then, in the next iteration, only the signatures of the marked states need to be recomputed, since the other signatures do not change. Moreover the unmarked states do not even need to be checked in the next refinement step. They just keep their old block identifier.

We will refer to the marked states as *unstable*. Note that, unlike other algorithms, that mark whole blocks (like bounded fanout Kanellakis-Smolka), we insist on reasoning about unstable *states* and not requesting that the states belonging to the same block should be easily retrievable. Extra attention has to be paid to ensure the correctness of the splitting procedure, but it pays off, since the ability to work directly on states provides parallel/distributed workers with a high(er) degree of independence.

Figure 2 shows an example of how the bounded fanout Kanellakis-Smolka and the optimized algorithms refine the same state space with transitions labeled a (the thin ones) and b (the thick ones). Since Kanellakis-Smolka is actually defined for a single relation, we considered a straightforward labels treatment: refine alternatively with respect to a and b, until the partition stabilizes. Note the difference in the aggressivity of splitting and, consequently, in the number of iterations.

The optimized algorithm OSBR is presented in Figure 1 and uses the following notations and data structures:
- $\mathcal{U}$, $\nu\mathcal{U}$ - the unstable states set for the current and the next iteration, respectively
- E - the number of blocks in the current partition. Throughout the algorithm, the invariant is kept that the blocks of the current partition are numbered $\{0 \ldots \mathsf{E}-1\}$.
- $\mathsf{c} : \{0 \ldots \mathsf{E}-1\} \longrightarrow Nat$ - the number of states in each block
- Reusable - the set of block identifiers that can be reused in the next iteration, since all the states belonging to those blocks are marked unstable. These identifiers should be reused in order to preserve the above mentioned invariant. Moreover, the identifier of every block should be reused for one of its own sub-blocks, to ensure termination of the iterations series.
- ST - a signatures hashtable used to map the signatures of

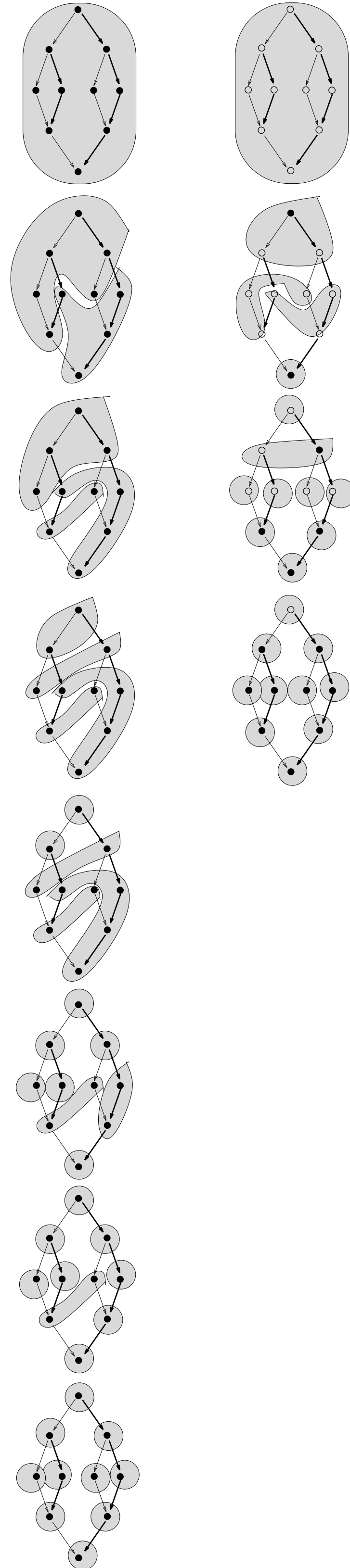

Figure 2. A refinement example, as performed by the Kanellakis-Smolka bounded fanout algorithm (left) and our optimized algorithm (right). The two types of transitions represent two different labels. The unfilled circles represent the unstable states at the beginning of each iteration.

the current iteration to new IDs (the block identifiers of the next iteration).

ID^f is the final partition, the blocks of which represent the states of the minimized LTS. The termination and correctness of OSBR follow from a few simple properties listed below.

Lemma 1. *Let $\mathcal{U}^n$, sig^n, ID^n, E^n denote the set of unstable states, the signatures mapping, the ID mapping, and the number of equivalence classes at the beginning of the n-th iteration of the optimized algorithm (i.e. before the n-th execution of step 3) — the count starts at 0. The following properties hold,for any $n \geq 0$:*

1. $(\forall x \in S)\ \mathsf{ID}^n(x) < \mathsf{E}^n$.
   (The blocks of the current partition are numbered $\{0 \cdots \mathsf{E}-1\}$.)
2. $(\forall 0 \leq i < \mathsf{E}^{n-1})\ \exists x \in S$ *s.t.* $\mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x) = i$.
   $(\forall 0 \leq i < \mathsf{E}^n)\ \exists x \in S$ *s.t.* $\mathsf{ID}^n(x) = i$.
   (Every identifier in the set $\{0 \cdots E-1\}$ is really used. Old blocks pass their identifiers to subblocks of their own.)
3. $(\forall x \in S\ \mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x))$ *iff* $\mathcal{U}^n = \emptyset$
   (A partition is final iff the unstable set becomes empty)
4. $(\forall x, y \in S)$
   $\mathsf{ID}^n(x) = \mathsf{ID}^n(y)$ *iff* $sig^n(x) = sig^n(y)$ *and*
   $sig^n(x) \neq sig^n(y) \implies sig^{n+1}(x) \neq sig^{n+1}(y)$.
   (The block identifiers are given correctly. The newer partitions are refinements of the older.)
5. $\mathsf{E}^{n-1} \leq \mathsf{E}^n$.
   $\mathsf{E}^n = \mathsf{E}^{n-1}$ *iff* $(\forall x \in S\ \mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x))$.
   (The number of blocks is increasing with the refinements.. until it stops because the final partition is reached.)

Proof. The properties 1, 3 and 4 will be proved independently, by induction on n. Property 2 relies on 4 and property 5 relies on 1 and 2.

1. $(\forall x \in S)\ \mathsf{ID}^0(x) = 0 < 1 = \mathsf{E}^0$. In every iteration, the only place where fresh values are introduced for ID is step 13 (in step 10 an old value is used). But E is also immediately increased (step 14), therefore the invariant stays true.

2. For $n = 0$ the property is obviously true, since $\mathsf{ID}^0(x) = 0$ for all states x. Suppose it is true for E^{n-1}, ID^{n-1} and let us look at how E^n and ID^n are computed, in the $n-1$th iteration. First, the set Reusable is constructed (step 5), containing the identifiers of the blocks whose states are *all* marked unstable. Let i be any identifier $0 \leq i < \mathsf{E}^n$. We distinguish three cases:
- $i \in \mathsf{Reusable}$. Then all states x with $\mathsf{ID}^{n-1}(x) = i$ (according to the induction hypothesis, there is at least one) must be in $\mathcal{U}^{n-1}$. Let y be the first of these states that is handled in the step 7 of the algorithm. $sig^n(y)$ cannot be already in ST, since this would mean that there exist a state z from another block ($\mathsf{ID}^{n-1}(z) \neq \mathsf{ID}^{n-1}(y)$) with the same signature ($sig^{n-1}(z) = sig^{n-1}(y)$), which contradicts point 4 of this lemma. Therefore, y will only be affected by steps 15 and 16, that do not modify the value of ID. Thus, $\mathsf{ID}^n(y) = \mathsf{ID}^{n-1}(y) = i$.
- $i \notin \mathsf{Reusable} \wedge i < \mathsf{E}^{n-1}$. Then there must be a state x for which $\mathsf{ID}^{n-1}(x) = i$ and $x \notin \mathcal{U}^{n-1}$. It follows, since the steps sequence $7-20$ does not regard x, that $\mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x) = i$.
- $\mathsf{E}^{n-1} \leq i < \mathsf{E}^n$. This means that i is "created" in the steps $12-14$ as identifier for a new block. In step 13 the ID^n of the first state of this block is explicitly defined as being i.

3. Let us consider an iteration n that satisfies $\forall x \in S\ \mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x)$. This means that in the iteration $n-1$, the condition in the step 17, that compares exactly $\mathsf{ID}^n(x)$ and $\mathsf{ID}^{n-1}(x)$ was never satisfied, thus $\nu\mathcal{U}$ remains $\emptyset$, that is $\mathcal{U}^n = \emptyset$. The inverse is also true: if $\mathcal{U}^n = \emptyset$ then $\nu\mathcal{U}$ ended up empty in the previous iteration. This could only happen if the condition on line 17 was never met, that is the value of ID was not changed for any state. Formally, $\forall x \in S\ \mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x)$.

4. We prove this by induction on $n \geq 0$. The case $n = 0$ follows from the fact that $(\forall x)\ sig^0(x) = 0$ and $(\forall x)\ \mathsf{ID}^0(x) = 0$. To prove the first half of the invariant for an arbitrary n, we consider three cases:
- $x, y \in \mathcal{U}^{n-1}$. In this case, both $sig^n(x)$ and $sig^n(y)$ are inserted in the hashtable ST, which ensures the same ID value for (and only for) equal signatures.
- $x, y \notin \mathcal{U}^{n-1}$. Then the sigs and IDs do not change, i.e. $sig^n(x) = sig^{n-1}(x)$, $\mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x)$ and $sig^n(y) = sig^{n-1}(y)$, $\mathsf{ID}^n(y) = \mathsf{ID}^{n-1}(y)$. From the induction hypothesis, it follows that $sig^n(x) = sig^n(y)$ iff $\mathsf{ID}^n(x) = \mathsf{ID}^n(y)$.
- $x \in \mathcal{U}^{n-1}$ and $y \notin \mathcal{U}^{n-1}$. Then there must be a state z that has caused the instability of x, i.e. there is a transition $x \xrightarrow{a} z$ with $\mathsf{ID}^{n-1}(z) \neq \mathsf{ID}^{n-2}(z)$. Then $\mathsf{ID}^{n-1}(z) = i \geq \mathsf{E}^{n-2}$, therefore the pair (a, i) cannot be in $sig^{n-1}(y)$. And since $sig^n(x)$ is recomputed and $sig^n(y)$ not, it follows that $sig^n(x) \neq sig^n(y)$. It remains to prove that $\mathsf{ID}^n(x) \neq \mathsf{ID}^n(y)$ as well. Let us first notice that

$$\text{if } \mathsf{ID}^n(x) \neq \mathsf{ID}^{n-1}(x) \text{ then } \mathsf{ID}^n(x) \geq \mathsf{E}^{n-1}. \tag{1}$$

If $sig^{n-1}(x) = sig^{n-1}(y) = i$, then $i \notin \mathsf{Reusable}$ (since $y \notin \mathcal{U}^{n-1}$), thus $\mathsf{ID}^n(x) \neq i$, thus (1) $\mathsf{ID}^n(x) \geq \mathsf{E}^{n-1}$, while $\mathsf{ID}^n(y) = \mathsf{ID}^{n-1}(y) < \mathsf{E}^{n-1}$. If, on the contrary, $sig^{n-1}(x) \neq sig^{n-1}(y)$, then $\mathsf{ID}^{n-1}(x) \neq \mathsf{ID}^{n-1}(y)$ (induction hypothesis). $\mathsf{ID}^n(x)$ is computed in the fragment $7-20$ and the outcome can be $\mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x) \neq \mathsf{ID}^{n-1}(y) = \mathsf{ID}^n(y)$ or $\mathsf{ID}^n(x) \neq \mathsf{ID}^{n-1}(x)$. In the latter case, $\mathsf{ID}^n(x) \geq \mathsf{E}^{n-1}$ (1), while $\mathsf{ID}^n(y) = \mathsf{ID}^{n-1}(y) < \mathsf{E}^{n-1}$.

And now we prove the second half of the property. Let x and y be two states for which $sig^n(x) \neq sig^n(y)$. Then (w.l.o.g.) there is some pair $(a, \mathsf{ID}^{n-1}(z)) \in sig^n(x)$ and $\notin sig^n(y)$. If $sig^n(y)$ does not contain any pair (a, j) then clearly $sig^{n+1}(x) \neq sig^{n+1}(y)$. Otherwise, let $y \xrightarrow{a} t$ be any of the a-transitions from y. Then $(a, \mathsf{ID}^{n-1}(t)) \in sig^n(y)$ and $\mathsf{ID}^{n-1}(t) \neq \mathsf{ID}^{n-1}(z)$, which means (induction hypothesis) that $sig^{n-1}(t) \neq sig^{n-1}(z)$ and, further, $sig^n(t) \neq sig^n(z)$. Above we have proved that this is equivalent to $\mathsf{ID}^n(z) \neq \mathsf{ID}^n(t)$. Thus, $sig^{n+1}(x)$ contains the pair $(a, \mathsf{ID}^n(z))$ and $sig^{n+1}(y)$ does not.

5. From the points 1 and 2 of this lemma it follows that $(\forall n)\ \mathsf{E}^n$ is exactly the number of different values for ID^n. Therefore, if $\forall x \in S\ \mathsf{ID}^n(x) = \mathsf{ID}^{n-1}(x)$ then obviously $\mathsf{E}^n = \mathsf{E}^{n-1}$.

We will now prove the inverse statement. Let n be so that $\mathsf{E}^n = \mathsf{E}^{n-1}$ and suppose there exist an $x \in S$ with $\mathsf{ID}^n(x) = i \neq \mathsf{ID}^{n-1}(x)$. The property 2 says that there exists $y \in S$ such that $\mathsf{ID}^n(y) = \mathsf{ID}^{n-1}(y) = i$. But this would mean $\mathsf{ID}^n(x) = \mathsf{ID}^n(y)$ and $\mathsf{ID}^{n-1}(x) \neq \mathsf{ID}^{n-1}(y)$, which comes in contradiction with property 4. □

Theorem 1. *(termination and correctness of* OSBR *) For any LTS (S,T,s_0),* OSBR *terminates and the equivalence relation $\sim$ determined by ID^f ($x \sim y \iff \mathsf{ID}^f(x) = \mathsf{ID}^f(y)$) is the largest strong bisimulation on S.*

Proof. It is easy to see that for any iteration $n > 0$, $\mathsf{E}^n \geq \mathsf{E}^{n-1}$. It is also clear that $\mathsf{E}^n > \mathsf{E}^{n-1}$ can happen only finitely often, since from the points 1,2 of Lemma 1 follows that $\forall n\ \mathsf{E}^n \leq |S|$. Hence eventually $\mathsf{E}^n = \mathsf{E}^{n-1}$ and then the algorithm stops (3,5 of Lemma 1 and the exit condition of the loop at step 3). This proves termination.

We will now justify that $\sim$ is a strong bisimulation on S. Let n_f be the number of the last iteration, that is $\mathsf{ID}^f = \mathsf{ID}^{n_f}$. Let x and y be any two equivalent states and let $x \xrightarrow{a} z$ be any transition from x. To prove that $\sim$ is a strong bisimulation, we have to prove that there exists a transition $y \xrightarrow{a} t$ with $t \sim z$. From $\mathsf{ID}^{n_f}(x) = \mathsf{ID}^{n_f}(y)$ and property 4 of Lemma 1 it follows that $sig^{n_f}(x) = sig^{n_f}(y)$. Then, since $(a, \mathsf{ID}^{n_f-1}(z))$ must be in sig^{n_f}, there is a state t with $\mathsf{ID}^{n_f-1}(t) = \mathsf{ID}^{n_f-1}(z)$ and $(a, \mathsf{ID}^{n_f-1}(t)) \in sig^{n_f}(y)$. But n_f is the final iteration, thus $\mathcal{U}^{n_f} = \emptyset$, that implies (Lemma 1, property 3) $\mathsf{ID}^{n_f}(t) = \mathsf{ID}^{n_f-1}(t)$ and similarly for z. Thus, $\mathsf{ID}^{n_f}(t) = \mathsf{ID}^{n_f}(z)$, or in other words $t \sim z$.

Finally, to prove that $\sim$ is the coarsest strong bisimulation, let $\sim'$ be any other strong bisimulation and show that $\forall x, y \in S\ x \sim' y \implies x \sim y$. To this end, we prove by induction that $\forall n \geq 0\ x \sim' y \implies \mathsf{ID}^n(x) = \mathsf{ID}^n(y)$. The base case $n = 0$ is immediate. Suppose the statement is true for $n-1$ and let x,y be two states such that $x \sim' y$. Then $\forall x \xrightarrow{a} z \exists y \xrightarrow{a} t$ with $z \sim' t$, and thus also (induction hypothesis) $\mathsf{ID}^{n-1}(z) = \mathsf{ID}^{n-1}(t)$. According to the signature definition, this means that $sig^n(x) = sig^n(y)$. From property 4 of Lemma 1, $\mathsf{ID}^n(x) = \mathsf{ID}^n(y)$. □

3.1 Remarks on the time complexity

The optimized algorithm uses a much more careful refinement strategy than the naive algorithm, which results in a better practical performance in many cases. This is supported by the experiments presented in Section 5.

The theoretical time complexity is a point that deserves some attention. Since in the worst case, the optimized algorithm behaves just like the naive, the complexity of OSBR is not worse than $\mathcal{O}(MN)$. With the Example 1 we show that this is a tight bound for the implementation presented in Figure 1. However, with a very small but significant modification, the optimized algorithm reaches $\mathcal{O}(N \log N)$. In Figure 3 we show this modified version and below we prove its complexity. We show by Example 2 that this bound is strict as well.

$\forall x \in S\ \mathsf{ID}(x) := 0;\ \mathsf{E} := 1; o\mathsf{E} := 0; \mathcal{U} := S;$
while $\mathsf{E} \neq o\mathsf{E}$
  $\forall B \in \pi$ s.t. $B \cap \mathcal{U} \neq \emptyset$
  – $\forall x \in B \cap \mathcal{U}$ compute $sig(x)$
  – split $B \cap \mathcal{U}$into $B_1 \cdots B_p$
  – assign the old identifi er of B
   to the largest block in $B - \mathcal{U}, B_1 \cdots B_p$
  – assign new IDs to the other new blocks
  – $\mathsf{E} := \mathsf{E}; \mathsf{E} :=$ no. of new IDs used

Figure 3. A $\mathcal{O}(N \log N)$ algorithm (not implemented)

Theorem 2. *(complexity) For any LTS (S,T,s_0) with bounded fanout the algorithm in Figure 3 terminates in $\mathcal{O}(N \log N)$ steps, where N is the size of S.*

Proof. When a state changes its ID, the size of its new block is at most half of the size of its old block. Otherwise, the old ID would be kept. This means that any state changes at most $\log N$ times.

Let (s,t) be any transition. Since t can change its ID at most $\log N$ times, this transition will be used for an update request at most $\log N$ times. That is, the maximal number of update requests due to state t is $\log N$. Since there are N states, the total number of update requests during the whole run of the algorithm is at most $N \log N$. Every signature check is triggered by one or more update requests, therefore we will have at most $N \log N$ signature checks.

The signature checks are expensive in the general case, but they are possible in constant time when we consider bounded fanout. Hence, the total complexity of this is $\mathcal{O}(N \log N)$. □

In order to achieve this bound in practice, a completely different implementation is needed, with carefully chosen data structures, that allow the fast retrieval of states from the same block. But this is exactly what we tried to avoid, because this operation is not too natural in a distributed setting. Therefore, we have chosen for the theoretically less competitive but practically succesful implemntation presented in Figure 1, where when a block is split, the old identifier is passed on to the stable subblock, which is not necessarily the largest subblock.

Example 1. The optimized algorithm can really reach $\mathcal{O}(MN)$ computation steps. Consider for example the state space in Figure 4, with N states and N transitions. After the first iteration, two blocks are identified, one containing the states with the signature $\{0,0\}$, the other $\{1,0\}$. The first has $N/2-1$

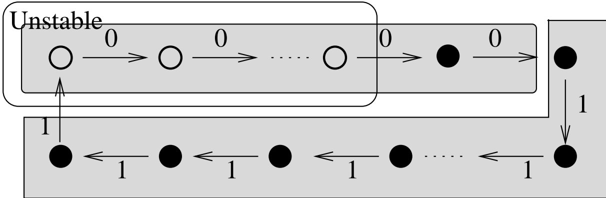

Figure 4. Example 1, showing that the bound $\mathcal{O}(N^2)$ for the optimized algorithm is tight.

states, the latter $N/2 + 1$. Therefore, the second keeps the identifier 0 and the first block gets a new identifier. Consequently, $N/2-2$ states are marked unstable and the last state in the 0-chain is not, which means that all the $N/2-2$ states will change their identifier in the next iteration. This leads to the first $N/2-3$ states in the 0-chain being marked unstable. In the next iterations, the same effect is repeated. Over the whole run, there will have been $(N/2-2)+(N/2-3)+\cdots 2+1$ unstable markings (i.e., signature recomputations). This means $\mathcal{O}(N^2)$.

Example 2. Consider the state space in Figure 5. It is a mirrored binary tree, with transitions labelled a in the upper part, and transitions labelled 0 and 1 in the mirrored part. This shape can be easily extended to obtain a state space with $\mathcal{O}(N)$ states, $\mathcal{O}(\log N)$ of which are placed on the mirror line. Note that every state has a unique trace to the final bottom state (deadlock). This means that if we apply the minimization algorithm, every state ends up in its own block. After the first iteration, the states in the upper part of the state space are all put together in one block, and the lower part is split into 3 blocks: the bottommost state, that has no outgoing transitions, and two other blocks, one of the states having 0 as outgoing transition, the other 1. Due to the symmetry of the construction, these two large blocks are equal. In every further iteration, every large block splits again into a singleton block (the bottom most state of the old block) and two other blocks, equal in size. (The argument for this is too tedious and we omit it.) Therefore, the reduction needs $\mathcal{O}(N \log N)$ steps.

4 The distributed implementation

4.1 Framework, assumptions

Our target architecture is a cluster whose nodes are connected by a high bandwidth network (Distributed Memory Machine). We assume that both the nodes and the network are reliable (no nodes failure, no message loss) and that the order of messages between nodes is preserved. Processes communicate by executing $send$ destination-process message (non-blocking) and $receive$ message (blocking). We assume that messages with the same source and destination keep their order. A message is a structure with a tag field (newsig, newid, update etc.) and data whose meaning depends on the tag.

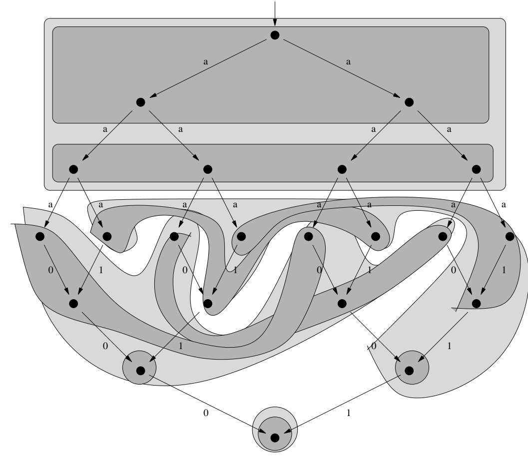

Figure 5. Example 2, showing that the bound $\mathcal{O}(N \log N)$ for the algorithm in Figure 3 is tight.

The input of our algorithm is an LTS (S, T, s_0) with N states and M transitions, and it has bounded fanout, which is a reasonable assumption for state spaces. An important hypothesis is that the input size (given by N and M) is much bigger than the number of processors available. That is what makes our framework different from other parallel implementations that assign a processor for each state and each transition.

4.2 Description

There are W workers, each consisting of two threads: a *segment manager*, that maintains a part (a segment) of the LTS and computes the signatures of the unstable states, and a *signatures server*, that maintains a part of the signature table ST and computes the new IDs. The data structures occurring in Figure 1 get distributed as follows:

- worker i, actually the segment manager i, is responsible for a subset S_i of S. $S_i \cap S_j = \emptyset, \forall i \neq j$ and $\bigcup_i S_i = S$. The function $SM : S \longrightarrow \{0 \ldots W-1\}$ maps every state to its base segment manager.
- transition set T generates for every segment manager i the sets

$$\mathsf{In}_i = \{(x, a, y) \mid y \in S_i \wedge x \xrightarrow{a} y\}$$
$$\mathsf{Out}_i = \{(x, a, \mathsf{ID}(y)) \mid x \in S_i \wedge x \xrightarrow{a} y\},$$

  where ID identifies the current partition.
- the unstable states sets $\mathcal{U}, \nu\mathcal{U}$ are maintained by managers in the form of $\mathcal{U}_i = \mathcal{U} \cap S_i$ and $\nu\mathcal{U}_i = \nu\mathcal{U} \cap S_i$, respectively.
- the set of block identifiers $\{0 \ldots \mathsf{E}-1\}$ gets divided into the disjoint sets $\mathsf{IDS}_0 \ldots \mathsf{IDS}_{W-1}$ and distributed to the W signatures servers by a mapping SS. Server j also maintains the part of the counts array c and of the signature table ST corresponding to IDS_j:

$$\mathsf{ST}_i = \{(oid, s, Lx) \mid SS(oid) = i \wedge Lx = [x \in S \mid \mathsf{ID}(x) = oid \wedge sig(x) = s]\}$$

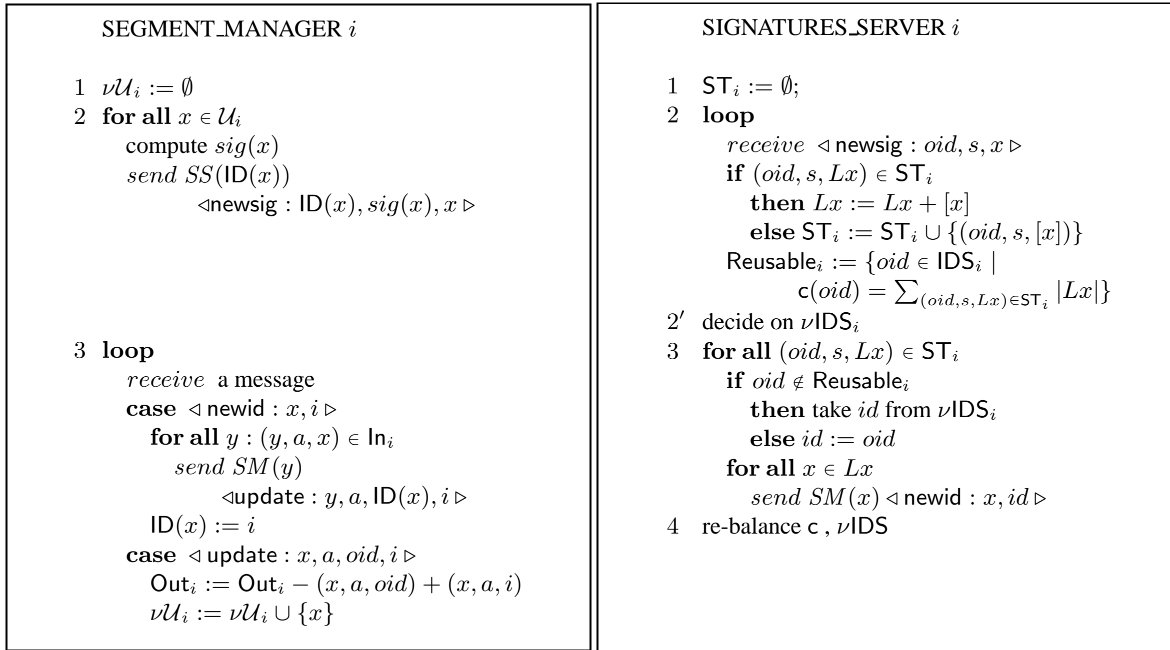

Figure 6. (ODBR) A distributed iteration

Here Lx is the list of all unstable states that have s as signature. Lx is necessary because unlike in the sequential implementation, in the distributed it is not possible to generate the new ID at the moment of signature insertion.

In the distributed algorithm , like in the sequential, a series of iterations is executed. In between iterations, workers synchronize in order to decide whether the final partition has been reached. The computation inside an iteration is asynchronous and directed only by messages, as sketched in Figure 6. There are five phases distinguishable within an iteration:
- managers compute the signatures of the unstable states and send them (newsig) to the appropriate servers
- servers receive the signatures (newsig) and insert them in their local ST
- servers compute new IDs for the unstable states and send them (newid) back to the managers
- managers receive the new IDs for their unstable states (newid) and send messages to the parent states of the own states that changed the ID (update)
- managers receive and process the update messages (update)

Due to the division of tasks between managers and servers, the first and the second phase happen in parallel (step 2 in Figure 6). Also the last three (step 3) are overlapped. The overlapping limits the amount of CPU idle time, by allowing computation and communication to proceed in parallel. For instance, the servers can already proceed with inserting signatures in the table while managers prepare and send more signature messages. In the actual runs of the program, a worker (manager + server) will use one processor.

The main advantage of overlapping the phases is memory gain: since the consumers and producers of messages are active in the same time, the messages don't have to be stored. Thus, less memory is used.

4.3 Correctness argument

Lemma 2. *The following properties hold for this distributed algorithm:*

1. *in every iteration, the signatures of the states in the same block are sent to the same server*
2. *every time a block splits, one of the new blocks gets the old id*
3. *in every iteration, finitely many* newsig *and* newid *messages are generated*
4. *in every iteration, a received* newid *message generates finitely many* update *messages.*
5. $(\forall n > 0)$ *if* ID_d^n *is the state partition at the beginning of iteration* n *of* ODBR *, and* ID_s^n *is the state partition at the beginning of iteration* n *of* OSBR *, then* $(\forall x, y \in S)\ \mathsf{ID}_d^n(x) = \mathsf{ID}_d^n(y) \iff \mathsf{ID}_s^n(x) = \mathsf{ID}_s^n(y)$

Proof. *1.* Indeed, if $\mathsf{ID}(x) = \mathsf{ID}(y) = i$, both $sig(x)$ and $sig(y)$ are sent (step 2 in the segment manager) to the signature server responsible for i, $SS(i)$.

2. Consider a block with the identifier i. If there are states $x \in S_j - \mathcal{U}_j$ with $\mathsf{ID}(x) = i$, then it's clear: all these states are not touched this iteration, i.e. they keep their old ID. If, on the contrary, all the states x with $\mathsf{ID}(x) = i$ are in some unstable set $(\forall x \in S\ \mathsf{ID}(x) = i\ \exists j x \in \mathcal{U}_j)$ then all

signatures will be computed and sent to the same server (step 2 in the segment manager). At the signature server side, all these signatures get inserted in ST_i and counted – and i is added to the $\mathsf{Reusable}_i$ set. Further, in step 3, when the first triple (i, s, Lx) is encountered, all the states in Lx get i as new ID.

3. The number of newsig and newid messages is limited by the total size of the sets $\mathcal{U}_i$, i.e. by $|S|$.

4. For each $\lhd$newid : $x, i \rhd$ messages, $|\mathsf{In}_i|$ messages (that is $\leq |S|$) with the tag update are sent.

5. By induction on n. □

Theorem 3. *(termination and correctness of* ODBR *) For any LTS (S, T, s_0),* ODBR *terminates and the* ID^f_d *function computed is the same as the* ID^f *computed by* OSBR .

Proof. The properties (1), (2) from Lemma 2 ensure that the invariants from Lemma 1 are also true in the distributed implementation ODBR . (3),(4) ensure that the computation within an iteration terminates. The global termination is justified by the one-to-one mapping between iterations in the sequential algorithm OSBR and the iterations in the distributed implementation ODBR (5). From (5) and the correctness of OSBR (Theorem 1) it follows that the partition computed is indeed the correct one. □

5 Experiments

We implemented both the sequential and the distributed versions of the optimized algorithm and compared their performance with the naive ones. The experiments were done on an 8 node dual CPU PC cluster and an SGI Origin 2000.[1] The test set consists of state spaces generated by case studies carried out with the μCRL toolset [4]. Problem sizes before and after reduction can be found in Table 1. *1394-LL,1394-LE* are models of the firewire link layer [17] and of the firewire leader election protocol with 17 nodes [23]. *CCP-2p3t* is a cache coherence protocol model with two processes and 3 threads [20] and *CCP* is an older (and smaller) variant of it. *lift5*, *lift6* are models of a distributed lift system with 5 and 6 legs [10]. *token ring* is the model of a Token Ring leader election for 4 stations[2].

5.1 Sequential tools compared

In [5], the usefulness of the signature approach was proved, by analysis and performance comparisons of the naive algorithm with existing tools. In order to justify the good performance of the marking procedure, we first present a comparison between the naive and the optimized sequential implementations (Table 2). The tests were run on one of the cluster machines. It is clear from this table that the marking procedure (used for the optimized) can give significant gains in time – see the numbers for both cache coherence protocols. The sequential optimized implementation needs more memory than the naive, since it keeps both the straight and the inverse transition systems. On the other hand, the naive one consumes more memory for the hashtable – all signatures have to be inserted, while only some have to be considered by the optimized implementation. Therefore, we expect that the optimized will be less memory expensive than the naive when it comes to large examples. The distributed implementation confirms this idea.

problem	bcg_min		naive		optimized	
	time (s)	mem (M)	time (s)	mem (M)	time (s)	mem (M)
CCP	15.0	18	21.3	20	4.5	18
1394-LL	18.5	19	6.2	14	3.3	21
lift5	113	184	64	123	43	214
CCP-2p3t	-	-	4363	968	779	1187

Table 2. A comparison of single threaded tools. The times include the I/O operations.

problem	d. naive – 16 CPUs		d. opt – 16 CPUs	
	time (s)	mem (M)	time (s)	mem (M)
lift5	33	460	20	480
CCP-2p3t	550	4430	104	1658
token ring	120	10802	231	4508
lift6	702	5958	346	3834
1394-LE	555	15388	428	8737

Table 3. A comparison of distributed implementations. The times are without I/O.

5.2 Distributed tools compared

Table 3 shows a comparison of the naive and optimized distributed implementations on the cluster, for a number of large LTSs. The numbers listed for the memory usage represent the maximum total memory touched on all 8 workstations during a run.

The runs indicate that the optimized implementation outperforms the naive one most of the time. The optimized is designed to perform better when the partition refinement series needs a large number of iterations to stabilize, yet very few blocks split in every iteration. This is exactly the case for the CCP state space. On the other hand, for state spaces like the Token Ring protocol, where almost all blocks split in every iteration, and the whole process ends in just a few rounds, the naive works faster, since it doesn't waste time on administration issues. In all larger examples though, the memory gain is obvious – and for the bisimulation reduction problem, memory is a more critical resource than time.

To test how the optimized distributed algorithm scales, we ran on the cluster series of experiments using 1-8 ma-

[1] The cluster nodes are dual AMD Athlon MP 1600+ machines with 2G memory each, running Linux and connected by gigabit ethernet. The Origin 2000 is a ccNUMA machine with 32 CPUs and 64G of memory running IRIX, of which we used 16 MIPS R10k processors. On the cluster, we used LAM/MPI 6.5 On the SGI, we used the native MPI implementation.

[2] The original LOTOS model [8] was translated to μCRL by Judi Romijn and extended from 3 to 4 stations.

problem	original			minimized		
	states (in 10^6)	transitions (10^6)	disk space (MB)	states (10^6)	transitions (10^6)	number of iterations
CCP	0.21	0.68	15	0.077	0.24	66
1394-LL	0.37	0.64	15	0.034	0.076	73
lift5	2.2	8.7	101	0.032	0.14	86
CCP-2p3t	7.8	59	678	1.0	6.6	94
token ring	19.9	132	1513	8.4	51.1	6
lift6	33.9	165	1898	0.12	0.65	91
1394-LE	44.8	387	4430	1.1	7.7	51

Table 1. Problem sizes.

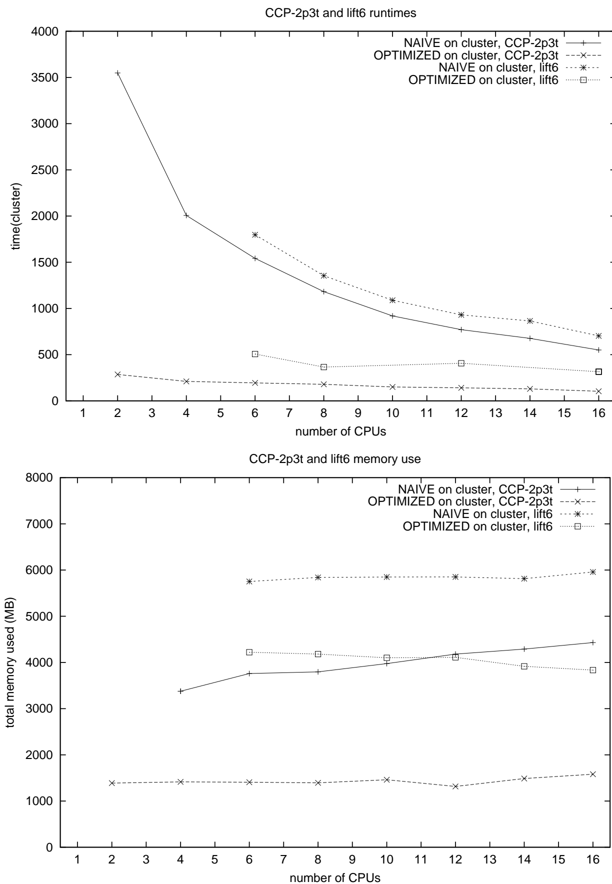

Figure 7. Runtimes and memory usage for *CCP-2p3t* and for *lift6*

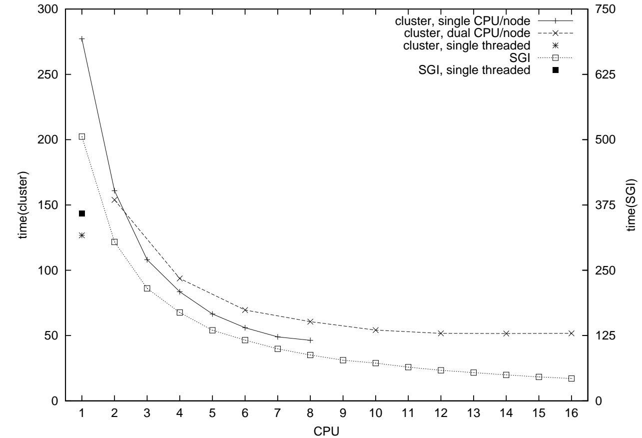

Figure 8. Runtimes for *lift5* on SGI and on the cluster

chines (2-16 processors). Figure 7 shows the runtimes (in seconds) needed to reduce *lift6* and *CCP-2p3t*. Since *lift6* is a real industrial case study with serious memory requirements, it couldn't be run single threaded on a cluster node or distributed on less than 3 nodes. We see that for both distributed implementations and both case studies presented, the memory usage scales well, i.e. the total memory needed on the cluster is almost constant, regardless the number of machines used. Hence, more machines available will mean less resources occupied on each machine.

On runtimes however, the naive implementation scales in a more predictable manner, while the optimized times don't seem to scale up as nicely. This is partly due to the nondeterminism present in the optimized implementation – signatures can arrive at servers in any order, the order influences the new IDs assignment to states, the new IDs determine how many unstable states are there in the next iteration, thus how much time will that iteration cost etc. It is also due to the possibly unbalanced distribution of signatures to servers, which introduces unpredictable idle times. Last, there is some latency due to the MPI implementation. We compared (Figure 8) the reduction of *lift5* on the cluster with the reduction on a shared memory machine that uses its native MPI implementation. It appears that the optimized algorithm does scale better on this other MPI.

5.3 The VLTS test suite

After analysing the behaviour of the two algorithms on some special case studies, we turn to "anonymous" state spaces from the VLTS benchmark [6]. Figure 9 shows the times and total memory usage of the optimized algorithm relative to those of the naive algorithm. Unlike the other measurements presented, the times considered now are total, that is the I/O operations are included. The 25 state spaces in this selection are small to medium size (between 0.06 and 12 million states, and between 0.3 million and 60 million transitions) and get reduced modulo strong bisimulation in less than 100 itera-

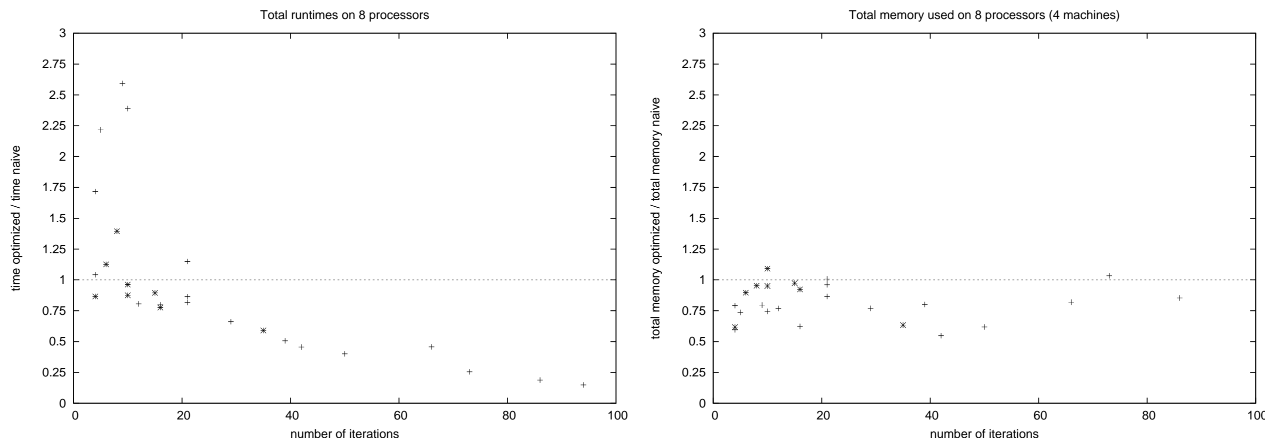

Figure 9. The VLTS test suite

tions. The stars mark the very small state spaces, i.e. those that get reduced in less than 5 seconds by both algorithms.

We present the state spaces ordered by the number of iterations in which the reduction procedure stabilizes. This is a relevant order only for the time performance, not for the memory usage.

As apparent from the figure, the relative time performance of the optimized is indeed influenced by the number of iterations and the size of the state space. This is roughly because, compared to the naive, it spends (much) more time on the initial setup - and this time pays back only if the reduction process has some length. Note that for very short reductions, it can be almost 3 times slower than the naive, but for lengthy ones it is usually much faster (up to 6 times faster).

Regarding the memory usage, we may notice that the optimized is indeed almost always an improvement. Exceptions are the small state spaces, where the fixed size buffers used by the optimized are significantly larger than needed. This could be fixed by using dynamic buffers.

6 Conclusions

We designed and implemented an optimized version of the algorithm described in [5]. The optimized version uses a marking technique for incremental computation of partitions and it allows communication and computation to proceed in parallel. The result is a distributed strong bisimulation reduction tool, which outperforms its straight forward counterpart in memory usage and in most cases also in time.

The optimized algorithm doesn't improve on the worst-case theoretical complexity ($\mathcal{O}(\frac{MN+N^2}{W})$, where M is the number of transitions and N the number of states), since there exist input LTSs where all states will be marked unstable, on every iteration. As the cost of marking is linear in the number of transitions, the complexitiy doesn't get worse either. In practice, the cost of processing a state using marking is typically twice the cost without. Hence, marking wins if on average less than 50% of the states are marked. On most practical state spaces this condition holds and marking visibly improves the performance.

The gain in time comes from the more elaborated treatment of partition refining. The gain in memory usage is the merit of two elements. First, the same improved refinement procedure makes sure that the hashtable accommodates less signatures, thus consumes less memory. The second and more important reason is that computation and communication are not separate phases anymore, but they are interleaved, saving this way the memory needed for storing intermediate results.

The concept of signature refinement also works for other equivalences, like branching bisimulation [9], weak bisimulation and τ^*a equivalence as defined in [7].

Another promising approach to generation, reduction and model checking of large state spaces is using the disk as extra storage. This technique would allow to increase the size of the models that can be verified without resorting to better equipment. There has not been much research yet in this direction yet.

References

1. J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL model-checking in SPIN. In *Proceedings SPIN'01*, volume 2057 of *LNCS*, pages 200–216, 2001.
2. G. Behrmann, T. Hune, and F.W. Vaandrager. Distributed timed model checking - How the search order matters. In *Proceedings CAV'00*, volume 1855 of *LNCS*, pages 216–231, 2000.
3. G. Berry, H. Comon, and A. Finkel, editors. *Proceedings CAV'01*, volume 2102 of *LNCS*, 2001.
4. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifi cations. In *Proceedings CAV'01*, volume 2102 of *LNCS*, pages 250–254, 2001.
5. S.C.C. Blom and S.M. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. In *Proceedings PDMC'02*, volume 68 of *ENTCS*, 2002.
6. CWI/SEN2 and INRIA/VASY. The VLTS benchmark. `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`.

7. J.-C. Fernandez and L. Mounier. Verifying bisimulations "on the fly". In *Proceedings FORTE'90*, 1990.
8. H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirection ring networks. *Science of Computer Programming*, 29(1–2):171–197, 1997.
9. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
10. J.F. Groote, J. Pang, and A.G. Wouters. Analyzing a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1–2):21–56, 2003.
11. O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In Berry et al. [3], pages 350–362.
12. J.E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
13. C. Joubert and R. Mateescu. Distributed on-the-fly equivalence checking. In *Proceedings PDMC'04*, ENTCS, 2004. To appear.
14. P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes and three problems of equivalence. In *Proceedings of 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.
15. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings SPIN'00*, volume 1680 of *LNCS*, 1999.
16. M. Leucker and T. Noll. Truth/SLC - A parallel verification platform for concurrent systems. In Berry et al. [3], pages 255–259.
17. S.P. Luttik. Description and formal specification of the Link Layer of P1394. In *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*, 1997.
18. R. Mateescu. A generic on-the-fly solver for alternation-free boolean equation systems. In *Proceedings TACAS'01*, volume 2619 of *LNCS*, pages 81–96, 2003.
19. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.
20. J. Pang, W.J. Fokkink, R. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. In *Proceedings FMPPTA'03*, 2003.
21. J.C. van de Pol and M. Valero Espada. Verification of JavaSpaces parallel programs. In *Proceedings ACSD'03*, pages 196–205, 2003.
22. S. Rajasekaran and I. Lee. Parallel algorithms for relational coarsest partition problems. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):687–699, 1998.
23. J.M.T. Romijn. A timed verification of the ieee 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001.
24. U. Stern and D. Dill. Parallelizing the Murϕ verifier. In *Proceedings CAV'97*, volume 1254 of *LNCS*, pages 256–278, 1997.
25. S. Zhang and S.A. Smolka. Towards efficient parallelization of equivalence checking algorithms. In *Proceedings FORTE'92*, volume C-10 of *IFIP Transactions*, pages 133–146, 1993.