

# A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces

Stefan Blom<sup>1</sup>, Simona Orzan<sup>2</sup>

<sup>1</sup> CWI, P.O.Box 94079, NL-1090 GB Amsterdam, The Netherlands, e-mail: [sccblom@cwi.nl](mailto:sccblom@cwi.nl)

<sup>2</sup> CWI, P.O.Box 94079, NL-1090 GB Amsterdam, The Netherlands, e-mail: [simona@cwi.nl](mailto:simona@cwi.nl)

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

**Abstract** It is a known problem that state spaces can grow very large, which makes operating on them (including reducing them) difficult because of operational memory shortage. In an attempt to extend the size of the state spaces that can be dealt with, we designed and implemented a bisimulation reduction algorithm for distributed memory settings using message passing communication. By using message passing, the same implementation can be used on both clusters of workstations and large shared memory machines. The algorithm performs reduction of large labeled transition systems modulo strong bisimulation. We justify its correctness and termination. We provide an evaluation of the worst-case time and message complexity and some performance data from a prototype implementation. Both theory and practice show that the algorithm scales up with the number of workstations.

---

## 1 Introduction

Model checking is a resource intensive application. The most common hardware architecture to provide these resources is the cluster of workstations. Thus, it is natural that a lot of interest exists in developing distributed model checking tools. These tools may be divided into three categories: symbolic, on-the-fly and enumerative. In all three approaches work is being done on distribution ([18], [9], [22], [5], [21], [4], [28]). We focus on the enumerative approach, followed by the  $\mu$ CRL toolset [7] and the CADP toolset [14]. There, one first generates the state space of a specification, reduces it modulo an equivalence relation and then model checks the reduced state space. State space generation has successfully been distributed [15], [11], [12] and with luck the reduced state space is small enough to be model checked on a single

machine. However, if the generated state space is too big to fit on a single machine then a distributed reduction tool is needed.

In this paper we present a distributed algorithm that reduces state spaces modulo strong bisimulation equivalence. Reducing the state spaces modulo strong bisimulation preserves all the interesting properties of the initial state space, that is all the properties expressible in  $\mu$ -calculus. The algorithm used for bisimulation reduction can be easily adapted to also solve the bisimulation equivalence problem. The worst-case time complexity of this algorithm is proved to be  $\mathcal{O}\left(\frac{MN+N^2}{W}\right)$ , where  $W$  is the number of processors used, and  $M$ ,  $N$  indicate the size of the input (number of transitions, respectively states of the LTS to be reduced).

The easiest solution to solving a memory shortage problem is to use a machine with more memory. However, machines with a really big amount of memory (16 times or more the amount of a normal workstation) are far less usual than the cluster of workstations. With luck the cluster has an efficient software/hardware shared memory implementation and one can run the old sequential program with only minor modification. However, most of the time the latencies of a software shared memory system are such that the algorithm has to be modified in order to have a decent performance. If the algorithm is really suitable to high latencies then of course there is also the possibility of using a disk as extra storage rather than remote memory. A nasty side effect of the modification is that the performance of the algorithm tends to get worse and that although there is no longer a memory shortage problem, the run times will be getting higher and higher. Hence what is needed is a distributed algorithm which scales up perfectly with respect to memory usage and which shows a limited scaling up with respect to run times. Because networking bandwidth grows faster than latencies decrease, it is best if the algorithm relies as little as possible on latency.

**Related work.** Very good sequential algorithms have been described for bisimilarity reduction and bisimilarity checking: [20], [24] and based on these, [13]. In [3], the bisimilarity checking problem was proved  $\mathcal{P}$ -complete, which means that it is hard to have it parallelized efficiently.

In [29],[26] Kanellakis-Smolka and Paige-Tarjan parallel algorithms were given. They were designed for shared memory machines and even though it is possible to simulate them on a distributed memory computer, it is unlikely that this will yield acceptable performance. Moreover, our setting is different: we are looking for a message-passing algorithm that would handle very large problem instances on a comparatively small number of processors and that would work well for the specific type of labeled transition systems representing state spaces, that is: multiple labels, but in a small constant number compared to the number of states; bounded fanout; a relatively small depth compared to the number of states.

[19] proposes a randomized parallel implementation of the Kanellakis-Smolka algorithm for bisimilarity checking, that works in linear time  $\mathcal{O}(N)$  on  $\mathcal{O}(N^2)$  processors. This solution does not consider the case of multiple labels, and it is not precise (has some small probability of error).

**Overview.** Our algorithm is a distributed version of a simple but surprisingly effective algorithm, called “the naive method” in [20]. After some preliminaries in section 2, we will discuss this simple algorithm in section 3, then we will present its distributed version in section 4. Section 6 contains a few statistics on the implementations. We conclude (section 7) with a summary of the results and a discussion of future work.

## 2 Preliminaries

In this section we fix a notation for labeled transition systems, we recall the definition of strong bisimulation and we state the bisimulation reduction problem. We also mention the general partition refinement scheme used by many known solutions.

Let  $\text{Act}$  be a fixed set of labels, representing actions.

**Definition 1.** (LTS) A labeled transition system (LTS) is a triple  $(S, T, s_0)$  consisting of a set of states  $S$ , a set of transitions  $T \subseteq S \times \text{Act} \times S$  and an initial state  $s_0 \in S$ .

**Definition 2.** (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 strongly bisimilar.

```

 $\pi := \pi^0;$ 
while  $\pi$  is not stable
  – pick some set of blocks  $\{B_1 \cdots B_n\} \subseteq \pi$ 
  –  $\pi^{new} := \pi - \{B_1 \cdots B_n\} \cup \{B_{1_1} \cdots B_{1_{m_1}}, B_{2_1} \cdots B_{n_{m_n}}\}$ 
    where  $\forall i \{B_{i_1} \cdots B_{i_{m_i}}\}$  is a partition of  $B_i$ 
    such that
       $\forall j \neq k \forall s \in B_{i_j} \forall t \in B_{i_k} : sig_\pi(s) \neq sig_\pi(t)$ 
  –  $\pi := \pi^{new}$ 
endwhile
    
```

**Figure 1.** Skeleton of a partition refinement algorithm using signatures

The problem that we focus on, *bisimilarity reduction*, is to find a smallest LTS which is strongly bisimilar to the given one. To do this, we have to find the equivalence classes of the largest strong bisimulation on a given LTS. The latter problem is also known as the Relational Coarsest Partition Problem for multiple relations.

A solution for this also offers a solution for the related problem of *bisimilarity checking*: given two LTSs  $(S^1, T^1, s^1)$  and  $(S^2, T^2, s^2)$  (with  $S^1 \cap S^2 = \emptyset$ ), decide whether they are strongly bisimilar, i.e. whether their initial states are strongly bisimilar in the compound LTS  $(S^1 \cup S^2, T^1 \cup T^2, s^1)$ .

For an LTS  $(S, T, s_0)$ , a *partition* (denoted by  $\pi$ ) of the elements of  $S$  is a set of disjoint blocks  $\{B_i \mid i \in I\}$  s.t.  $\cup_{i \in I} B_i = S$ . Most algorithms used to solve the bisimulation reduction problem are based on some form of partition refinement, i.e. they perform successive iterations in which blocks of the current partition are split in smaller blocks, until nothing can be split anymore. While splitting a block, states that cannot be distinguished are kept in the same block. Two states can be distinguished if one of the states allows a transition with a certain label to a state in a certain block and the other state does not have a transition with the same label to a state in the same block.

The *signature* of a state  $s$  with respect to a partition  $\pi$  is the set of  $s$ 's outgoing transitions to blocks of  $\pi$ :

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

Hence, by definition two states are distinguishable with respect to a partition if and only if they have different signatures with respect to that partition. A partition  $\pi$  is called *stable* if every two members of every block in that partition have the same signature with respect to  $\pi$ . A basic partition refinement algorithm using signatures is outlined in Fig. 1. The algorithm starts from a given initial partition and keeps states with the same signature in the same block. Its correctness 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.

For the purpose of LTS reduction, it is convenient to start with the initial partition containing one block that has all states,  $\{S\}$ . We will do so in the remainder of the paper.

### 3 The sequential algorithm

As starting point for our distributed implementation, we chose a very simple bisimulation reduction algorithm, namely the one called “the naive method” in [20]. From the point of view of theoretical complexity, it cannot compete with the Paige-Tarjan algorithm ( $\mathcal{O}(MN + N^2)$  vs.  $\mathcal{O}(M \log N)$ ), but in practice it performs quite well. Moreover, the Paige-Tarjan algorithm cannot be extended to branching bisimulation and weak bisimulation. For the naive algorithm this is a possibility.

Both the Paige-Tarjan algorithm and the naive algorithm use iterations. The Paige-Tarjan algorithm in each iteration carefully selects a number of states to work on. The naive algorithm works on all states independently. The data structures needed to make the selection cost a lot of memory in any case and require modification to allow an efficient distributed implementation. In contrast, the naive algorithm has a natural parallel implementation. For both algorithms the worst case number of iterations is the number of states. However, in practice the Paige-Tarjan algorithm needs this many iterations and the naive algorithm only needs a few. So the expected complexity of the two algorithms is the same. Another difference with practical advantages is that Paige-Tarjan uses three-way splitting of blocks, whereas the naive algorithm uses multi-way splitting.

In Fig. 2, we have described an implementation of the naive algorithm, for a given labeled transition system  $(S, T, s^0)$ . The idea is that signatures computed with respect to the current partition determine the next partition, i.e. the blocks of the new partition are sets of states with identical signatures. Keeping track of the current partition is done by assigning every block an unique identifier (natural number). The function  $ID : S \rightarrow \mathbb{N}$  indicates to which block every states belongs. Thus, the current partition is represented by the ID function and the signature of a state  $x$  w.r.t. it is  $\{(a, ID(y)) \mid x \xrightarrow{a} y\}$ . Since our signatures of interest are always computed w.r.t. the current partition, we will not index them; instead, we will make sure it’s always clear what the current partition is.  $ID^f$  denotes the final assignment for ID, i.e. the final partition. In step 5, new values are assigned to ID, such that  $\forall x, y \in S$ :

$$\begin{aligned} \text{the new value of } ID(x) &= \text{the new value of } ID(y) \\ \iff \\ sig(x) \text{ w.r.t. the old ID} &= sig(y) \text{ w.r.t. the old ID.} \end{aligned}$$

A hash table  $H = \{\langle oldsig, newID \rangle\}$  with signatures as hash values is used for this purpose.

```

1. for  $x \in S$   $ID(x) := 0$ ; endfor ;
2.  $oldcount := 0$ ;  $newcount := 1$ ;
3. while  $oldcount \neq newcount$ 
4.   for  $x \in S$   $sig(x) := \{(a, ID(y)) \mid x \xrightarrow{a} y\}$ ; endfor ;
5.   for  $x \in S$ 
       insert  $sig(x)$  in  $H$ 
       and get new value for  $ID(x)$ 
   endfor
6.    $oldcount := newcount$ ;
    $newcount := |\{ID(x) \mid x \in S\}|$ ;
   endwhile ;
7. for  $x \in S$   $ID^f(x) := ID(x)$ ; endfor ;

```

Figure 2. (SSBR)Single threaded implementation of the naive algorithm

Note that, unlike the general partition refinement scheme, SSBR doesn’t explicitly replace blocks of the old partition with new blocks. The only work done in an iteration of SSBR is computing the signatures of all states. The following lemma justifies that the partitions computed in this manner are indeed successive refinements, under the hypothesis that the initial partition is  $\{S\}$ :

**Lemma 1.** *For  $n \geq 0$ , denote  $ID_n$ ,  $sig_n$  the ID and sig functions as they are after execution of step 4 of the  $n$ th iteration of SSBR (first iteration has index 0). Then for every  $n > 0$  and for every  $x, y \in S$ :*

$$sig_{n-1}(x) \neq sig_{n-1}(y) \implies sig_n(x) \neq sig_n(y). \quad (1)$$

*Proof.* Let us first observe that step 5 of the  $n - 1$ th iteration ( $\forall n \geq 0$ ) takes care that, for every  $x, y \in S$ ,

$$ID_n(x) = ID_n(y) \iff sig_{n-1}(x) = sig_{n-1}(y). \quad (2)$$

We prove the lemma by induction on  $n$ . The initial partition is  $\{S\}$ , which means that  $\forall x, y \in S : sig_0(x) = sig_0(y)$ . Therefore the claim is true for  $n = 1$ . Let  $n$  be  $> 1$  and  $x, y \in S$  such that  $sig_{n-1}(x) \neq sig_{n-1}(y)$ . Then there must be a pair  $(a, ID_{n-1}(z))$  that (w.l.o.g.) occurs in  $sig_{n-1}(x)$  and doesn’t occur in  $sig_{n-1}(y)$ . We distinguish two cases: (i) There is no transition  $y \xrightarrow{a} t$ . Then  $sig_n(y)$  will not contain any pair of the form  $(a, -)$ , while  $sig_n(x)$  will have at least  $(a, ID_n(z))$ . (ii) There are transitions leaving from  $y$  and labeled with  $a$ . In this case, let  $y \xrightarrow{a} t$  be any of them.  $ID_{n-1}(t)$  must be  $\neq ID_{n-1}(z)$ , since  $(a, ID_{n-1}(z)) \notin sig_{n-1}(y)$ . Then, from (2) it follows that  $sig_{n-2}(t) \neq sig_{n-2}(z)$  and, by using the inductive hypothesis, that  $sig_{n-1}(t) \neq sig_{n-1}(z)$ . Further, we apply (2) one more time, to obtain that  $ID_n(t) \neq ID_n(z)$ . Thus, the set  $sig_n(x)$  will contain the pair  $(a, ID_n(z))$ , but  $sig_n(y)$  not.

## 4 The distributed algorithm

The obvious way to distribute a partition refinement algorithm is to distribute the data and keep the control flow centralized. More precisely, the workers perform iterations in which they independently do some refinement and then synchronize the results. This approach is fine in theory, but in practice it turns out that synchronization can take a lot of time. This is another reason to choose the naive algorithm: typically it needs far less iterations than Kanellakis-Smolka and Paige-Tarjan, thus less synchronizations.

### 4.1 Framework, assumptions

Our framework is a failure-free distributed memory machine (workers don't crash, messages don't get lost), where any two workers are connected by unbounded asynchronous channels. A message  $\langle \text{tagfield} : \text{data field} \rangle$  is a structure with a tag field (`hash_insert`, `hash_ID`, `update`, `endsig`) and some data whose meaning depends on the tag. The message passing happens by the execution of *send* and *receive* operations. We assume that messages with the same source and destination keep their order. *send* (*destination worker*, *message*) is non-blocking, *receive* (*message*) is blocking. We denote the number of workers by  $W$ .

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 workers available. This is why the algorithm is "distributed" as opposed to "parallel", which would mean assigning a processor for each state and each transition.

We also assume the existence of a hash function that allows the even distribution of the LTS to the  $W$  worker processes. Therefore, each worker will hold approximately  $\frac{N}{W}$  states and these states will have approximately  $\frac{M}{W}$  incoming and outgoing transitions. Note that the transitions from one worker to the other workers are not necessarily evenly distributed! As a matter of fact, we have seen differences of about an order of magnitude here. We assume that the distribution is given and we denote by  $S_i$  the set of states of worker  $i$ .  $S_0 \cup S_1 \cup \dots \cup S_{W-1} = S$  and  $S_i \cap S_j = \emptyset$ , for all  $i \neq j$ .

### 4.2 Description

Our distributed reduction algorithm (Fig. 3) is based on the sequential one (Fig. 2). As mentioned above, the states of the input LTS are evenly divided to the  $W$  workers. Worker  $i$  gets to be in charge of the set of states  $S_i$ . Every iteration, it has to compute the signatures of states in  $S_i$  and keep track of the ID of these signatures. It is also responsible for the administration of a part

```

1. read  $\text{In}_{j_i}(\forall j)$ , read  $\text{Out}_{i_j}(\forall j)$ ;
2.  $\text{newcount} := 1$ ;
3. loop
4.   for  $x \in S_i$ 
        $\text{sig}(x) := \{(a, p) \mid \langle x, a, p \rangle \in \text{Out}_{i_j}, 0 \leq j < W\}$ ;
     endfor ;
5.    $N_{\text{expected\_answers}} := 0$ ;
5'.   $\text{Send\_Signatures} \parallel \text{Handle\_Messages}$ 
6.    $\text{oldcount} := \text{newcount}$ ;
        $\text{newcount} := \sum_{i=0}^W \text{newcount}_i$ ;
6'.  if ( $\text{oldcount} = \text{newcount}$ ) break;
6''.  $\text{Update\_IDs}$ ;
     endloop ;
7. for  $x \in S_i$   $\text{ID}^f(x) = \text{ID}(x)$ ; endfor ;

```

Figure 3. (DSBR) Distributed version of SSBR.  $\text{WORKER}_i$

```

for  $x \in S_i$ 
   $\text{send}(\text{wrk}(\text{hash}(\text{sig}(x))),$ 
     $\langle \text{hash\_insert} : i, \text{sig}(x) \rangle$ );
   $N_{\text{expected\_answers}} ++$ ;
endfor ;
for  $j : 0 \leq j < W$ 
   $\text{send}(j, \langle \text{endsig} : i \rangle)$ ;
endfor ;

```

Figure 4. The *Send\_Signatures* routine of  $\text{WORKER}_i$

```

 $N_{\text{active\_workers}} := W$ ;
while  $N_{\text{active\_workers}} > 0 \vee N_{\text{expected\_answers}} > 0$ 
   $\text{receive}(msg)$ ;
  case  $msg$ 
     $\langle \text{hash\_insert} : j, s \rangle$ 
      insert  $s$  in  $H_i$  and compute  $\text{ID}(s)$ ;
       $\text{send}(j, \langle \text{hash\_ID} : s, \text{ID}(s) \rangle)$ ;
     $\langle \text{endsig} : j \rangle$ 
       $N_{\text{active\_workers}} --$ ;
     $\langle \text{hash\_ID} : s, sid \rangle$ 
       $\text{ID}(s) := sid$ ;
       $N_{\text{expected\_answers}} --$ ;
  endcase ;
endwhile ;
 $\text{newcount}_i := |H_i|$ ;

```

Figure 5. The *Handle\_Messages* routine of  $\text{WORKER}_i$

of the hash table used at step 5' (step 5 in SSBR). We denote  $i$ 's part by  $H_i$ .

Let  $T_{ij}$  be the indexed list of transitions having the source state in  $S_i$  and the destination state in  $S_j$ . About a transition in  $T_{ij}$ , worker  $i$  will wish to know its source state, its label and the current ID of the signature of its destination state, in order to be able to compute the source state's signature. It is worker  $j$ 's job to keep  $i$  informed about the current ID of the destination state.

```

newcount := newcounti
if 2i + 1 < W
  receive (◁count : x▷)
  newcount := newcount + x
if 2i + 2 < W
  receive (◁count : x▷)
  newcount := newcount + x
if i > 0
  send (0, ◁count : newcount▷)
  receive (◁count : newcount▷)
if 2i + 1 < W
  send (2i + 1, ◁count : newcount▷)
if 2i + 2 < W
  send (2i + 2, ◁count : newcount▷)

```

**Figure 6.** Code run by  $WORKER_i$  to compute

```

for j : 0 ≤ j < W
  send (j, ◁update : i, [ID(y) | y ∈ lnji]▷);
endfor ;
received := 0;
while received < W
  receive (◁update : w, IDList▷); received ++;
  update Outiw;
endwhile ;

```

**Figure 7.** The  $Update\_IDs$  routine of  $WORKER_i$

Therefore, the  $T_{ij}$ -concerned knowledge needed and maintained by workers  $i$  and  $j$  is captured in two lists ordered by the same index as  $T_{ij}$ :

$$\begin{aligned} \text{Out}_{ij} &= [\langle x, a, \text{ID}(y) \rangle \mid x \xrightarrow{a} y \wedge x \in S_i \wedge y \in S_j], \\ &\quad \text{residing in the memory of } i, \text{ and} \\ \text{ln}_{ij} &= [y \mid x \xrightarrow{a} y \wedge x \in S_i \wedge y \in S_j], \\ &\quad \text{in the memory of } j. \end{aligned}$$

Note that elements can occur repeatedly in a list – for instance, a state shows up twice in  $\text{ln}_{ij}$  if it is the destination of two transitions coming from states on  $i$ .  $\text{ln}_{ji}$  and the first two fields of the elements from  $\text{Out}_{ij}$  represent static information about the structure of the LTS. The data that change throughout the run of the algorithm are the functions  $sig$  and  $\text{ID}$ . Furthermore, workers need to know the number of different signatures of the states in  $S$  in the current and in the previous iteration, based on which it is decided whether the final partition has been reached.

The LTS is provided to workers in the form of lists  $\text{ln}_{ij}$ ,  $\forall i \neq j$  and  $\text{Out}_{ij}$ ,  $\forall i, j$ , the latter using a constant initial ID function:  $\text{ID}(x) = 0$  that reflects the initial partition  $\{S\}$ .

The step numbers in Fig. 3 and Fig. 2 illustrate the correspondence between the sequential and the distributed algorithm. Each iteration of DSBP consists of three phases:

1. signature computation (step 4),

2. computing globally unique IDs for signatures (step 5') and
3. exchanging ID information. (step 6'')

In the first phase (step 4) of every iteration, each worker computes the signatures of its own states.

In the second phase (steps 5, 5', detailed in Fig. 4 and 5), all signatures are inserted in the distributed hash table and are assigned unique IDs. The insertion is based on a hash function  $\text{hash} : 2^{\text{Act} \times \mathbb{N}} \rightarrow \mathbb{N}$  and the distribution of the hash table is done by a function  $\text{wrk} : \mathbb{N} \rightarrow \{0 \dots W - 1\}$ . We assume that  $\text{wrk}$  is capable of ensuring a balanced loading of signatures on workers.  $WORKER_i$  runs two threads. One is busy with sending each signature to the worker responsible for the part of the hash table where it should be inserted (determined using  $\text{wrk}$ ). When all signatures are sent, an  $\text{endsig}$  message is sent to all workers, to mark the end of the stream. The other thread handles the incoming messages. A request for inserting a signature in the local hash table ( $\text{hash\_insert}$ ) is handled by looking up the signature and fetching its ID, or, if not found, adding it to the table and assigning it a new ID. The ID is then returned to the owner of the signature. When receiving an answer ( $\text{hash.ID}$ ) to a request sent earlier by  $\text{Send\_Signatures}$ ,  $\text{Handle\_Messages}$  fills in the new ID value and decreases the counter of expected answers. Finally, on receiving an end-of-stream message ( $\text{endsig}$ ), it decreases the counter of workers that might still send  $\text{hash\_insert}$  requests. The  $\text{Handle\_Messages}$  thread terminates when all workers announced that they have no more signatures to send to  $i$  and all  $i$ 's requests have been answered.

After the second phase, we compute how many different signatures there are (steps 6). This operation is implemented as single call in the MPI library, but can also be implemented with messages (see Fig. 6). (The algorithm organizes the workers as a binary tree. The algorithm first sends count messages up the tree which hold the sum over the sub-tree. Thus the sum of all local counts is eventually computed by worker 0. Then worker 0 send this sum down the tree. This means that after  $\mathcal{O}(\log(W))$  steps all workers have the global sum.)

If the number of signatures did not increase w.r.t. the previous iteration, the stable partition has been reached and the computation must stop (6').

In the third phase (step 6'', shown in detail in Fig. 7), the lists  $\text{Out}_{ij}$  get updated. For every transition of the LTS, the new ID of the destination state's signature is sent to the owner of the source state. More precisely, every worker  $j$  sends  $\text{ID}(\text{ln}_{ij})$  to worker  $i$ , who will substitute this information on the last fields of its  $\text{Out}_{ij}$ . This happens correctly due to the fact that the lists  $\text{ln}_{ij}$  and  $\text{Out}_{ij}$  have the same index.

At the end of the **loop**, the IDs are the states of the reduced LTS and  $\cup_{i,j} \{\langle \text{ID}(x), a, p \rangle \mid \langle x, a, p \rangle \in \text{Out}_{ij} \}$  is its set of transitions. They can be dumped independently by the workers, after an eventual renumbering of IDs to consecutive numbers.

In the actual MPI implementation, the size of messages being sent in *Send\_Signatures* may vary, by buffering signatures. We have chosen to send multiple smaller messages rather than a single big one because this saves memory, although it requires a bit more work from the programmer. Moreover, by manually dividing the messages it is possible to start processing the already received parts while the rest of the message is still being received. For the update phase (6''), we issue all the *send* messages and *receive* requests, then wait for all *receive*'s to be completed.

The two threads from step 5' are implemented with explicit interleaving.

### 4.3 Analysis

We now justify that the algorithm described above is correct, i.e. it terminates and it produces the minimal LTS bisimilar to the input LTS. We also give an analysis of its performance in terms of time, memory and number of messages needed during the computation.

**Theorem 1.** (*correctness*) *Let  $\mathcal{S} \equiv (S, T, s^0)$  be an LTS. Then DSBR applied to any distribution of  $\mathcal{S}$  terminates and the resulting  $ID^f$  satisfies:*

$$\begin{aligned} &(\{ID^f(x) \mid x \in S\}, \\ &\{\langle ID^f(x), a, ID^f(y) \rangle \mid \langle x, a, y \rangle \in T\}, \\ &ID^f(s^0)) \end{aligned}$$

is the minimal LTS bisimilar to  $\mathcal{S}$ .

*Proof.* We first argue that every iteration (steps 4–6'') of DSBR terminates. For this, we take a closer look at the steps involving communication (5', 6 and 6''). The first thread of step 5' obviously terminates, since it only executes a finite number of *send* calls (that are always successful).

*Handle\_Messages*'s exit conditions of

$$N\_active\_workers = 0 \wedge N\_expected\_answers = 0$$

will eventually be satisfied.  $N\_active\_workers$  becomes 0 when  $W$  *hash\_ID* messages sent to  $i$  will have been received. Note that  $N\_active\_workers$  being 0 is a sign that all *hash\_insert* messages directed to  $i$  have been received, and also that all *hash\_insert* messages, originating from all workers, including  $i$  itself, have been sent. In particular, this means that when  $N\_active\_workers$  of  $i$  is 0,  $N\_expected\_answers$  of  $i$  will not increase anymore. This property rules out the undesired situation that the exit condition is fulfilled while messages for  $i$  are still pending. Computing the sum over the distributed counts ( $\sum_{i=0}^W newcount_i$ ) is implemented with a single MPI call. Hence, we can assume that it is correct. The termination of *Update\_IDs* is justified mainly by the fixed number of messages exchanged. Every worker successfully sends

exactly  $W$  messages (these messages can be very large, but this is not a problem, since we assumed unbounded channels), then picks up from the network the  $W$  messages addressed to it.

It can be easily proved by induction that DSBR mimics faithfully the sequential version SSBR, depicted in Fig. 2. That is, formally: for any  $r$ , if we consider  $ID_s =$  SSBR's ID, after step 5 of the  $r$ th iteration and  $ID_d^i =$  *WORKER<sub>i</sub>*'s ID, after step 5' of the  $r$ th iteration ( $\forall i$ ), then

$$\begin{aligned} &\forall i, j \forall x \in S_i, y \in S_j \\ &ID_d^i(x) = ID_d^j(y) \iff ID_s(x) = ID_s(y). \end{aligned}$$

From this and from the fact that the exit condition from DSBR and SSBR are identical, it follows that the step 3 of DSBR eventually terminates. Moreover, the LTS determined by the  $ID^f$  values is exactly the one found by SSBR, thus the solution of our problem.

To evaluate the performance of DSBR, we use the classic time/message complexity measures for distributed algorithms, as defined in [2].

**Theorem 2.** (*complexity*) *In the worst-case, time complexity of DSBR is  $\mathcal{O}(\frac{MN+N^2}{W})$  and message complexity is  $\mathcal{O}(N^2)$ .*

*Proof.* For computing the signatures, every state has to be considered and we assumed that the cost per state is linear in the number of outgoing transitions of that state. As workers do this computation independently and we assumed even distribution of states, the time needed is  $\mathcal{O}(\frac{M+N}{W})$ .

The number of signatures each worker has to insert into the global hash table is at most the number of states it processes:  $\lceil \frac{N}{W} \rceil$ . Assuming that *wrk* is a perfect hash function, each worker has to send  $\lceil \frac{\lceil \frac{N}{W} \rceil}{W} \rceil$  signatures to every other worker. Every worker therefore receives at most  $W \cdot \lceil \frac{\lceil \frac{N}{W} \rceil}{W} \rceil$  signatures. (The insertion in the local hash table takes constant time, as well as the computing of a new ID.) The same amount of replies must be sent back. Thus, the cost of computing globally unique identifiers for signatures is  $\mathcal{O}\left(W \cdot \lceil \frac{\lceil \frac{N}{W} \rceil}{W} \rceil\right)$ . Under the assumption that  $W \ll N$ , we can forget about rounding upwards and we get the cost of  $\mathcal{O}\left(\frac{N}{W}\right)$ .

To decide termination, we need to compute the total number of different signatures. The cost of this operation is  $W$ .

To exchange the new IDs, every worker has to prepare  $W$  buffers of total size  $\mathcal{O}(\frac{M}{W})$ , representing the total number of incoming transitions (see 4.1). It has to also receive and process  $W$  such buffers, from workers that are in charge of successor states.

Summing up, the cost of an iteration is  $\mathcal{O}(\frac{M+N}{W})$ . Because as many as  $N$  iterations might be needed, the worst case time complexity is  $\mathcal{O}(\frac{MN+N^2}{W})$ .

The message complexity is given by the total number of messages sent by all workers in the whole run of the algorithm. In the worst case, exchanging signatures takes  $N$  messages (if every signature has to be sent to another worker), and the update phase  $W^2$  messages. Synchronizing at step 6 takes always  $W$  messages. This results in at most  $N(N + W + W^2)$  messages over the whole run, that is  $\mathcal{O}(N^2 + NW^2)$ . Seen the fact that  $W$  is insignificant compared to  $N$ , we may conclude a message complexity of  $\mathcal{O}(N^2)$ .

The number of iterations is the most important factor in the performance of the algorithm. The worst case is that the number of iterations is the number of states. An example that has this worst case behavior is an LTS whose state are the numbers  $0..N$  and the transitions are  $i \rightarrow i + 1 (i = 0..(N - 1))$ . However, such a long series of events is not typical in state spaces. The typical phenomenon in state spaces is state space explosion: the system would consist of  $P$  processes each having  $N$  states that run in parallel. The size of the state space would then be  $N^P$ , which is a huge number for relatively small  $N$  and  $P$ . However, if the processes are completely independent then the reduction algorithm needs at most  $N + 1$  iterations. Of course neither the long thread nor the complete independence of processes occurs in practice, but they give some intuition about the worst case and why it is unlikely.

The memory needed by one worker can be estimated as follows:  $\mathcal{O}(\frac{M+N}{W})$  for the signature information,  $\mathcal{O}(\frac{M}{W})$  for the (destination ID, destination state) of incoming transitions,  $\mathcal{O}(\frac{M}{W})$  for the (source state, label, destination ID) of outgoing transitions. In total,  $\mathcal{O}(\frac{M+N}{W})$ , which is the best that can be achieved,  $W$  times less than the space used by the single-threaded implementation.

## 5 Implementation details

We have written two sequential implementations and one distributed implementation of the naive algorithm. The first sequential implementation follows the description in Fig. 2 faithfully. The second, optimized, marks in every iteration the unstable states – that is, the states with at least one transition to a state in a block that was split – and, in the next iteration, only recomputes signatures for those.

Note that our implementations do not keep the states of the same block clustered together, thus movement of states is not necessary. Instead, we update each state's *ID* information at every refinement round.

The distributed prototype implements the algorithm in Fig. 3.

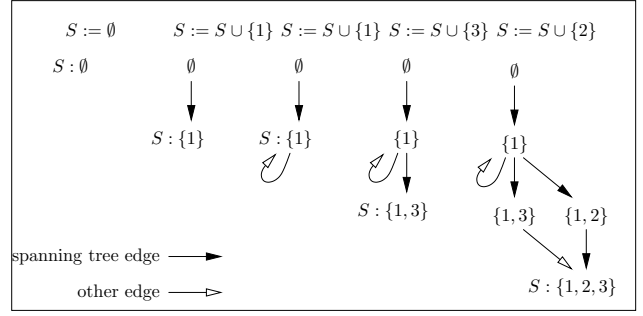


Figure 8. Evolution of a set data structure.

### 5.1 A set datatype for computing signatures

For computing signatures we have used a set datatype on which it is easy to add a single element and decide equality. The idea is to maintain a directed graph, whose vertices are sets and whose labeled edges are an 'obtained by insertion' relation. That is, an edge  $S \xrightarrow{e} S'$  is only allowed if  $S' = S \cup \{e\}$ . In order to efficiently decide if a certain set is present or not, we maintain the graph in such a way that the edges  $S \setminus \{max(S)\} \xrightarrow{max(S)} S$  form a spanning tree. By doing this the set corresponding to an ordered list can be found by starting in the empty set and then following the edges corresponding to the elements in the list. There may be other paths from the empty set to the same set, but if a set exists then this path exists. In Fig. 8, we have drawn the datastructure as it would look when starting with an empty set and adding the elements 1, 1, 3 and 2 in that order. Notice that adding 1 twice creates a cycle in the graph.

On this data structure, we can decide equality of sets in constant time. The complexity of inserting a single element into a set is linear in the size of the set the first time and constant afterwards. (The first time we have to create one or more edges and 0 or more vertices, afterwards we can find the edge in constant time using a hash table.)

Using this structure it is very easy to write code that computes signatures. However, the order in which the signatures are built matters for the performance of the algorithm. If a set is built in the same order every time then quadratic time is needed for the first build and linear time for every rebuild. The danger comes from the fact that quadratic time and memory may be used for every different order in which the signature is built. This means that to get decent performance, we have to sort the transitions ensuring that the amount of different orders is minimal.

## 6 Experimental results

In this section we present some experimental data obtained by testing the prototype implementations. To experiment with the distributed implementation, we used

**Table 3.** A comparison of single threaded and distributed runs.

problem	orig			reduced			cluster		distributed		single threaded	
	S	T	D	S	T	I	fast	gbit	time	mem	time	mem
cache coherence	7.8	59	678	1.0	6.6	94	1725	550	1249	5438	10480	1380
token ring	19.9	132	1513	8.4	51.1	6	355	120	299	13416	2505	4367
lift	33.9	165	1898	0.12	0.65	91	-	702	1513	5452	15355	2652
1394	44.8	387	4430	1.1	7.7	51	1600	555	1136	15372	12111	6566

**Table 4.** Separate timing of the three phases.

phase	firewire			cache coherence		
	fast	gbit	SGI	fast	gbit	SGI
compute	145	145	540	215	215	670
synchronize	375	100	250	1300	250	486
update	1000	230	180	170	65	55

problem	original		reduced	
	states	transitions	states	transitions
1394	$3.7 \cdot 10^5$	$6.4 \cdot 10^5$	$3.4 \cdot 10^4$	$7.6 \cdot 10^4$
cache	$2.1 \cdot 10^5$	$6.8 \cdot 10^5$	$7.7 \cdot 10^4$	$2.4 \cdot 10^5$
lift5	$2.2 \cdot 10^6$	$8.7 \cdot 10^6$	$3.2 \cdot 10^4$	$1.4 \cdot 10^5$

**Table 1.** Problem sizes.

problem	bcg_min		naive		optimized	
	mem	time	mem	time	mem	time
1394	19M	18.5s	14M	6.2s	21M	3.3s
cache	18M	15.0s	20M	21.3s	18M	4.5s
lift5	184M	113s	123M	64s	214M	43s

**Table 2.** A comparison of single threaded tools.

an 8 node dual CPU PC cluster and an SGI Origin 2000. The cluster nodes are dual AMD Athlon MP 1600+ machines with 2G memory each, running Linux and connected by both fast and gigabit ethernet. The Origin 2000 is a ccNUMA machine with 32 CPUs and 64G of memory running IRIX, of which we used 1-16 MIPS R10000 processors. On the cluster, we used LAM/MPI 6.5.6 On the SGI, we used the native MPI implementation.

### 6.1 A comparison of sequential tools

First, in order to validate the use of the naive algorithm, we have compared the memory use and run times of our single threaded implementations with those of the `bcg_min` reduction tool, which is part of the CADP toolset [14]. For this comparison, we used a problem set consisting of models of the firewire link layer [23], a cache coherence protocol [25] and a distributed lift system with 5 legs [17]. These problems are case studies carried out with the  $\mu$ CRL toolset [7]. Their sizes before and after

reduction can be found in Table 1. The test results can be found in Table 2. These tests were run on a PC running Linux with dual AMD Athlon MP1600+ CPUs and 2G memory. The version of `bcg_min` used was 1.3. From this table, we draw the conclusion that the performance of our sequential tools is comparable to that of `bcg_min`. Hence, using the naive algorithm is feasible.

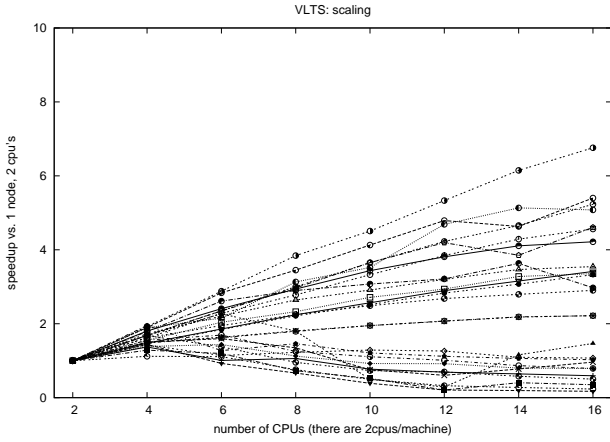
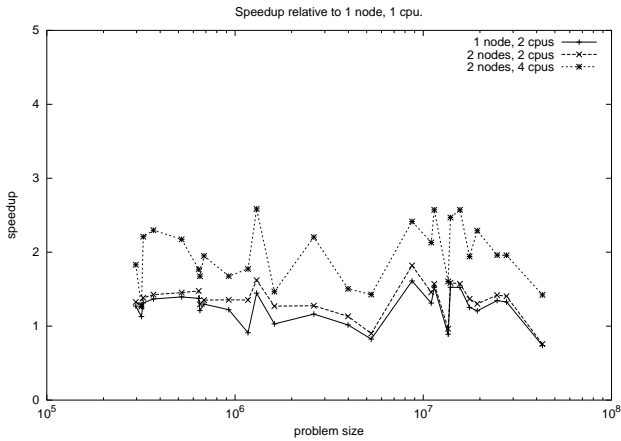
### 6.2 Comparing the sequential with the distributed implementation

In Table 3, we have shown the results of comparing the simple single threaded implementation against the distributed implementation running on 16 workers. The problem set consists of models of a version of the cache coherence model, token ring leader election for 4 stations (the original LOTOS model [16] was translated to  $\mu$ CRL by Judi Romijn and extended from 3 to 4 stations), the lift problem with 6 legs and firewire leader election for 17 nodes [27].

The table lists the problem size in  $10^6$  states (S),  $10^6$  transitions (T) and MB disk space (D), the reduced size in  $10^6$  states (S),  $10^6$  transitions (T) and the number of iterations needed (I), the run times on the cluster using the fast ethernet and the gigabit ethernet and the run times and memory use of the distributed and single threaded implementations on the SGI. It is clear from this table that gigabit ethernet outperforms fast ethernet. This difference can be seen even more clearly in Table 4, which splits the run times of two problems into the run times for the three phases of the algorithm. (See 4.2.)

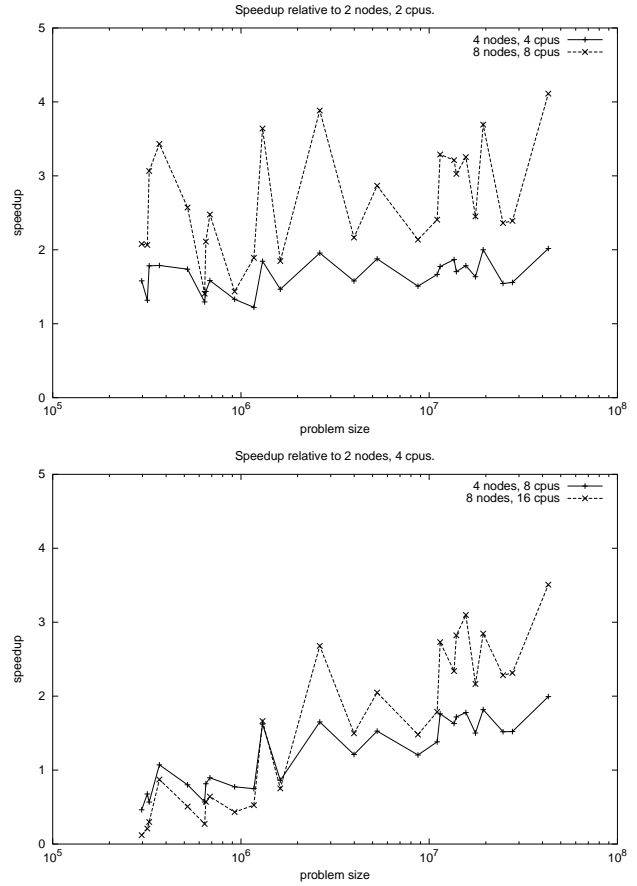
### 6.3 The distributed implementation: scalability

For the tests in this subsection we used the PC cluster. The inputs were 26 case studies from a preliminary version of the VLTS benchmark suite [1]. The selection criterion was no less than  $10^5$  transitions and small enough

**Figure 9.** Speedup comparison

**Figure 10.** Initial speedup


to be reduced on a single node. All the values presented are averaged from 5 runs. The speedup was computed from the real time (wall clock time) spent on the reduction only. That is, the time spent on doing IO operations for reading the LTS and writing the result is not included. The problem size is the number of transitions.

In Fig. 9, we have plotted the speedup for each of the problems. This picture shows clearly that for some problems we get good speedups and for others we get slow down. Many of the lines in this pictures curve downwards. This means that the efficiency decreases somewhat with the number of processors. Because we designed the algorithm especially for large transition systems, we were interested in how the problem size influence the speedup. First, we looked at the speedup we get from moving from a single CPU to a minimal distributed system. In Fig. 10, we show the speedup relative to the program running on a single CPU for three possibilities: a single dual CPU node, two single CPU nodes and two dual CPU nodes. The considerable amount of extra CPU power in the 2 node, 4 CPU system seems to have

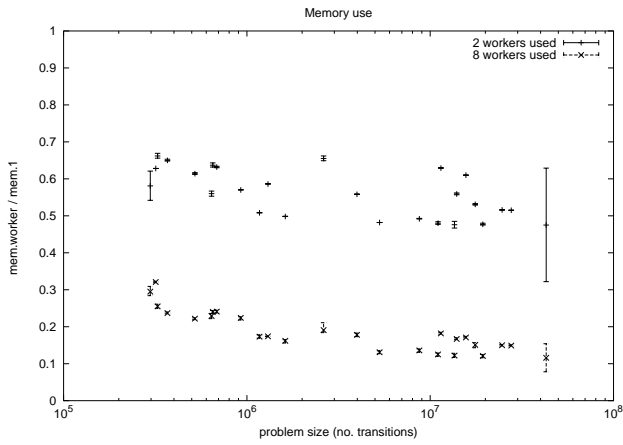
**Figure 11.** Additional speedup


had an effect, but apart from that moving from a single CPU to a minimal distributed system doesn't seem to have much of an advantage. Next, we looked at the speedup we got from moving from a minimal distributed system to larger distributed systems. In Fig. 11, we show the speedups achieved by using 4 and 8 nodes relative to using 2 nodes for both the single and dual cpu case. Even though the lines are pretty erratic, it is possible to see a tendency of the 4 node lines to converge to 2 and for the 8 node lines to converge to 4. In the dual CPU plot it is also very obvious that using too many CPUs hurts performance.

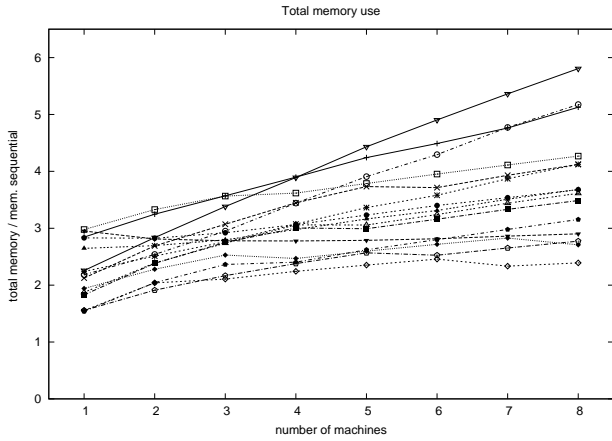
We also made memory measurements, in order to observe the efficiency of the distribution and the scalability of the total memory use.

Fig.12 shows data collected by measuring the memory use of each worker for distributed runs with 2 and with 8 workers. To facilitate comparisons and see the scaling up, we divided these values by the memory used by the distributed base run, i.e. the distributed implementation when run with one worker. For each problem, the plots show the minimum/average/maximum value thus obtained. We see in the figure that when 2 workers are used, memory consumption per worker drops to ap-

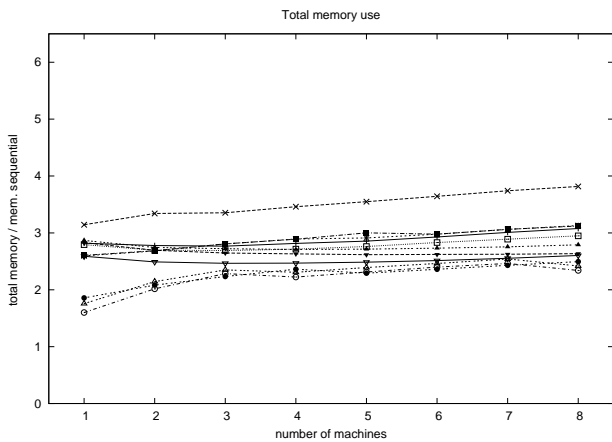
**Figure 12.** Memory use: distribution



**Figure 13.** Memory use: scaling, small LTSs ( $\leq 10^6$  trans.)



**Figure 14.** Memory use: scaling, big LTSs ( $> 10^6$  trans.)



proximately 0.6 of the distributed base run and when 8 are used, to approximately 0.2. Thus, the more workers the less memory needed per worker. Note also that only for two problems there is a noticeable difference between minimum and maximum. This means that the function we have used to distribute states across workers performs reasonably well: in most cases the workers need roughly the same amount of memory.

In Fig. 13 and 14 we have plotted the total memory use against the number of workers for small ( $\leq 10^6$  transitions) and large ( $> 10^6$  transitions) systems respectively. Here the memory is divided by the memory of the sequential implementation, in order to also get a distributed/sequential comparison. For small systems the memory use often increases quite rapidly with the number of workers, but for large problems the total memory use doesn't increase much. From these pictures it is quite clear that for most large problems the memory use of the distributed tool is between 2 and 3 times as much as the memory use of the sequential tool. This is not unexpected: at least two copies of all the signatures must be kept (a local copy and a global copy) and ID information is sent using buffers, whose size is linear in the number of transitions.

#### 6.4 The I/O operations

During our experiments with the 6 leg lift problem, we found that reducing the LTS is not the only problem. The ext3 filesystem as implemented in the Linux 2.4 kernels is not suitable for reading/writing multiple large files in parallel. As a result, reading the LTS from disk actually took more time than reducing it. For later experiments we have used PVFS (Parallel Virtual File System [10]) instead of NFS. This is a distributed file system, which uses the disks of multiple machines to present a large file system to the user. Per node the performance of PVFS was roughly equal to that of NFS, but the performance of PVFS scales linearly with the number of nodes so effectively it was 8 times better.

## 7 Conclusions

We took a simple algorithm for strong bisimulation reduction and designed and analyzed a distributed version of it. We implemented both the sequential and distributed version of the algorithm. With a few experiments, we showed that the performance of the simple algorithm is quite good. With a long series of experiments, we showed that although speedup is likely for large problems it is not guaranteed. We also showed that the scalability in terms of memory use is very good. Thus, we succeeded in our goal of writing a tool which can effectively reduce large state spaces.

The implementation of the distributed reduction tool can be improved in several ways. A conceptual improvement is to apply the marking technique that proved useful in the sequential case. Both the small cache coherence problem for the comparison with `bcg_min` and the large version for the comparison between distributed and single threaded, show a big decrease in run time, if the optimized single threaded version is used. (The large problem runs in 2258 seconds.)

Another point of attention is the fact that the three phases of the algorithm: compute signatures, exchange signatures and exchange partition information are now strictly separate. By overlapping these phases, we can get rid of the big transmit buffers for exchanging partition information for a gain in memory use. Because the processors show some idle time during the exchange phases, we can also expect a gain in time. Also, we distributed the states by using a random hash function. It might be useful to develop a hash function, which minimizes edges between different workers. This would improve the performance by minimizing communication.

By using different notion of signature, the algorithm can be adapted to other equivalences relations, like weak and branching bisimulation and  $\tau^*a$  equivalence. We already have a distributed implementation of branching bisimulation reduction (see [8]). The other equivalences and the problem of moving large state space between memory and disk are part of future work.

## References

1. [http://www.inrialpes.fr/vasy/cadp/resources/benchmark\\_bcg.html](http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html).
2. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.
3. J.L. Balcazar, J. Gabarro, and M. Santha. Deciding bisimilarity is  $\mathcal{P}$ -complete. *Formal Aspects of Computing*, 4(6A):638–648, 1992.
4. J. Barnat, L. Brim, and J. Stršbrná. Distributed LTL model-checking in SPIN. In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer-Verlag, 2001.
5. G. Behrmann, T. Hune, and F.W. Vaandrager. Distributed timed model checking - how the search order matters. In *Proceedings CAV 2000*. Springer-Verlag, 2000.
6. G. Berry, H. Comon, and A. Finkel, editors. *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *LNCS*. Springer-Verlag, July 2001.
7. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lissner, and J.C. van de Pol.  $\mu$ CRL: A toolset for analysing algebraic specifications. In Berry et al. [6], pages 250–254.
8. Stefan Blom and Simona Orzan. Distributed branching bisimulation reduction of state spaces. In Lubos Brim and Orna Grumberg, editors, *PDMC 2003: Parallel and Distributed Model Checking*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.
9. B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free  $\mu$ -calculus. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 543–558. Springer-Verlag, April 2001.
10. P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.
11. S. Caselli, G. Conte, and P. Marenzoni. A distributed algorithm for GSPN reachability graph generation. *Journal of Parallel and Distributed Computing*, 61(1):79–95, 2001.
12. G. Ciardo. Distributed and structured analysis approaches to study large and complex systems. In *Lectures on Formal Methods and Performance Analysis : First EEF/Euro Summer School on Trends in Computer Science, The Netherlands, July 2000*, volume 2090 of *LNCS*, pages 244–274. Springer-Verlag, 2001.
13. J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3):219–236, 1990.
14. J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, 1996.
15. H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In M.B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 217–234, Toronto, Canada, May 2001. Springer-Verlag.
16. H. Garavel and L. Mounier. Specification and Verification of Various Distributed Leader Election Algorithms for Unidirection Ring Networks. Technical Report 2986, INRIA Rhône-Alpes, 1996.
17. J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1-2):21–56, 2003.
18. O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for  $\mu$ -calculus. In Berry et al. [6], pages 350–362.
19. 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. Tainan, Taiwan.*, 1998.
20. 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, Montreal, Canada*, pages 228–240, 1983.
21. F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *Proceedings of the 5th International SPIN Workshop (SPIN'00)*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

22. M. Leucker and T. Noll. Truth/SLC - A parallel verification platform for concurrent systems. In Berry et al. [6], pages 255–259.
23. S.P. Luttkik. Description and formal specification of the Link Layer of P1394. In I. Lovrek, editor, *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*, 1997.
24. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.
25. J. Pang, W.J. Fokkink, R. Hofman, and R. Veldema. Model checking a cache coherence protocol for a java DSM implementation. In *Proceedings of the 8th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'03)*. IEEE Computer Society Press, 2003.
26. S. Rajasekaran and I. Lee. Parallel algorithms for relational coarsest partition problems. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):687–699, 1998.
27. J.M.T. Romijn. Model checking the HAVi leader election protocol. Technical Report SEN-R9915, CWI, June 1999.
28. U. Stern and D. Dill. Parallelizing the Mur $\phi$  verifier. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 256–278, 1997.
29. S. Zhang and S.A. Smolka. Towards efficient parallelization of equivalence checking algorithms. In *Proceedings of FORTE'92*, pages 133–146, 1992.