Evaluation of a Distributed Explicit State Space Exploration Algorithm with State Reconstruction for RDMA Networks

Sami Evangelista[1*], Lars Michael Kristensen[2] and Laure Petrucci[1]

[1] Laboratoire d'Informatique de Paris Nord, CNRS UMR 7030, Université Sorbonne Paris Nord, 99, av. J.-B. Clément, Villetaneuse, 93430, France.

[2] Faculty of Engineering and Science, Western Norway University of Applied Sciences, Inndalsveien 28, Bergen, 5020, Norway.

*Corresponding author(s). E-mail(s): sami.evangelista@univ-paris13.fr;
Contributing authors: lars.michael.kristensen@hvl.no; laure.petrucci@univ-paris13.fr;

Abstract

The inherent computational complexity of validating and verifying concurrent systems implies a need to be able to exploit parallel and distributed computing architectures. We present a new distributed algorithm for state space exploration of concurrent systems on computing clusters. Our algorithm relies on Remote Direct Memory Access (RDMA) for low-latency transfer of states between computing elements, and on state reconstruction trees for compact representation of states on the computing elements themselves. For the distribution of states between computing elements, we propose a concept of state stealing. We have implemented our proposed algorithm using the OpenSHMEM API for RDMA and experimentally evaluated it on the Grid'5000 testbed with a set of benchmark models. The experimental results show that our algorithm scales well with the number of available computing elements, and that our state stealing mechanism generally provides a balanced workload distribution.

Keywords: model checking, state explosion problem, distributed state space exploration, RDMA, state reconstruction

1 Introduction

Model checking of concurrent and distributed systems based on state space exploration is a highly compute- and storage intensive task requiring considerable computing resources when applied to real-life systems. This has prompted research into algorithms that are able to exploit networked computing elements [14, 18]. In addition to specialised algorithms for the exploration of the state space itself, these approaches typically require tailored algorithms for the verification of properties, including LTL-based model checking [3–5].

Our proposed state space exploration algorithm proceeds by forward traversal of the state space. It provides a compact state representation via distributed state reconstruction trees. This implies that for a substantial number of the visited states, the full state descriptors are not explicitly stored in memory. Instead, their implicit representation via the state reconstruction tree allows the full state descriptor to be retrieved on-demand when required for comparison with newly generated states during duplicate detection. As our algorithm is based on state reconstruction trees, we rely on a concept of state stealing instead of conventional hash functions to distribute the state space exploration workload on the computing elements.

The application of multiple computing elements for model checking entails a non-trivial amount of

overhead due to synchronization and networked transfer of states between the computing and storage elements. In this paper we consider the use of Remote Direct Memory Access (RDMA) to provide low-latency networking for cluster computing. As such, our algorithm is not tied to the RDMA paradigm, but can be implemented on any platform supporting distributed shared memory. However, the RDMA paradigm is well-suited for efficient implementation of state stealing strategies. In our concrete implementation, we exploit RDMA as supported by the OpenSHMEM API [1] for low-latency transfer of states between computing elements. OpenSHMEM is supported by a consortium comprising several industry, research, and university partners and has multiple implementation.

The present paper is a journal extension of [11], which additionally contains a complete presentation of the algorithm, includes a proof of correctness of our algorithm, and provides a comprehensive experimental evaluation including the comparison of different implementation strategies and also assessing the performance of our algorithm on two different modelling formalisms.

The rest of this paper is organised as follows. In Section 2 we introduce the basic concepts associated with state reconstruction and duplicate state detection, and explain the memory and communication paradigms of RDMA architectures and open shared memory. Section 3 illustrates the key elements in our proposed algorithm using a small example state space. Section 4 provides a specification of the algorithm and a proof of correctness. In Section 5, we present the strategies implemented for state stealing and duplicate detection. The results from the experimental evaluation of our algorithm implementation on DVE models of the DiVinE model checker [2] and on Petri net models of the Helena tool [9] are presented and discussed in Section 6. Finally, in Section 7 we sum up conclusions and discuss future work.

2 Background

To make the presentation of our algorithm independent of any particular modelling language for concurrent systems, we let $\mathcal{S}$ denote the universe of syntactic system states and let $\mathcal{E}$ denote the set of possible events. The system is given through an initial state $s_0 \in \mathcal{S}$, a mapping $enab : \mathcal{S} \to 2^{\mathcal{E}}$ associating with each state a set of enabled events, and a mapping $succ : \mathcal{S} \times \mathcal{E} \to \mathcal{S}$ used to generate a successor state from a state and one of its enabled events.

This definition of the *succ* function implies that events are assumed to be deterministic, i.e., the state resulting from executing an enabled event in a given state is uniquely defined. This assumption allows us to reconstruct a unique state from a sequence of events in the state reconstruction. It is important to note that we only assume that the execution of individual events is deterministic. The model may still have multiple enabled events in a given state and there may be a non-deterministic choice between which of the enabled events are being executed. Furthermore, deterministic events do not imply that events are assumed to be reversible, i.e., backward executable. Many modelling formalisms (including Petri nets) have deterministic transitions (events). Process algebra supports non-deterministic events as we may have several enabled events with the same label in a given state. As shown in [19], state reconstruction can be generalised to handle also non-deterministic events, and this generalised approach can also be adapted to our setting.

State space exploration is concerned with computing the set of states reachable from s_0, *i.e.* states s such that there exist $e_0, \ldots, e_{n-1} \in \mathcal{E}$, $s_1, \ldots, s_n \in \mathcal{S}$ with $s = s_n$ and, for all $i \in \{0, \ldots, n-1\}$: $e_i \in enab(s_i)$ and $succ(s_i, e_i) = s_{i+1}$. State space exploration in its basic form maintains a set $\mathcal{R}$ of *visited states*, and a set O of currently *open states* for which successor states have not yet been computed. The algorithm iterates until there are no more open states. In each iteration, an open state s is selected and *state expansion* is performed by exploring all events enabled in s. Duplicate detection is performed on the successor states of s in order to identity states already stored in $\mathcal{R}$. Successor states that have not been visited earlier (new states) are inserted into $\mathcal{R}$ and O.

2.1 State reconstruction

The set $\mathcal{R}$ of visited states in basic state space exploration is typically implemented as a hash table of full state descriptors to make it easy to determine whether a newly generated state has already been visited. State space exploration with state reconstruction [10, 12, 13, 19] trades space for time and maintains instead a *state reconstruction tree* which constitutes an *inverse spanning tree* rooted in the initial state. The state reconstruction tree makes it possible using a

[1] http://www.openshmem.org

backward lookup step to identify a sequence of events which when executed forward from the initial state (root) allow us to reconstruct full state descriptors when needed for duplicate detection, *i.e.* when determining whether a newly generated state is already included in the set of visited states. The state reconstruction tree can also be represented using a hash table.

Figure 1 illustrates state reconstruction. The top of Figure 1 shows the state space where the upper part of each node is the full state descriptor, the bottom part is its hash value, and the thick edges are edges representing references in the spanning tree. The lower part of Figure 1 is a linearised graphical representation of the hash table storing the set $\mathcal{R}$ of currently visited states (s_0 - s_4). The dashed arcs in the linerarised graphical representation in the lower part are references to parent nodes in the state reconstruction tree and are labelled by corresponding generating events. Note that full state descriptors appear in the table for the sake of clarity, but they are not (explicitly) stored in memory.

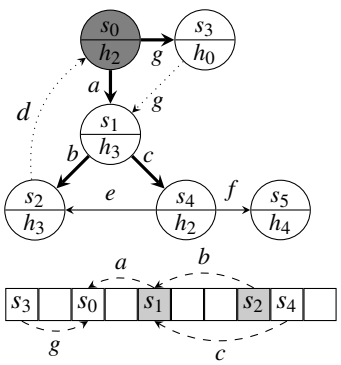


Fig. 1 State reconstruction example

When required, the full state descriptor for a state can be reconstructed by backtracking (following the dashed references) up to the root node (initial state) for which we have the full state descriptor, and then forward execute the *reconstructing sequence* of generating events on the path leading from the initial node (state) to the node in question. This is performed each time the algorithm generates a successor state s' from an open state s and needs to determine whether s' has already been visited.

As an example, consider Figure 1 and assume that the algorithm has explored states s_0 to s_3 and is expanding s_4 corresponding to the exploration of the two thin edges. The expansion of s_4 generates s_2 and s_5 hashed to h_3 and h_4, respectively. Since no currently visited states are mapped to h_4 we know that s_5 is a new state. To determine whether the newly generated (duplicate of) s_2 is new, we reconstruct all nodes of the reconstruction tree that are also hashed to h_3 as these could potentially be the same state. These correspond to the grey cells of the hash table. For the first cell (s_1) we have to follow the reference labelled a back to the initial state (the root of the reconstruction tree) and execute the reconstruction sequence a starting from the initial state. We obtain state s_1 which is different from the newly generated s_2. For the second cell (s_2), we have to follow references labelled b, a to the initial state and finally execute the reconstruction sequence $a.b$ starting from the initial state. Since this execution produces state s_2, we conclude that executing e from s_4 does not generate a new state.

It can be noted that in the example, the reconstruction sequence for s_1 is a prefix of the reconstruction sequence for s_2 which makes it possible to reduce the amount state recontruction. In general, all states to be reconstructed may not have shared prefixes in their state reconstruction sequence.

2.2 Multi-core state reconstruction

In earlier work [10], we developed and experimentally evaluated a state space exploration algorithm with state reconstruction for shared memory multi-core architectures. This earlier algorithm operates in rounds and is based on barrier synchronization to allow concurrently executing threads to perform state space exploration in three phases within each round. In the first phase the threads traverse the reconstruction tree in order to generate a frontier set of states comprising the next layer of states. The algorithm also stores open states in the reconstruction tree (instead of full descriptors of open states) which further reduces memory usage. In the second phase, duplicate detection is performed on the frontier set generated in the first phase resulting in a merged candidate set of potentially new states. In the third phase, the threads perform state reconstruction to determine the candidate states that are new. Any new states are then added to the reconstruction tree. The algorithm employs delayed duplicate detection and grouping of state reconstructions as means to reduce the number of state reconstructions and hence reduce the execution time of the algorithm.

A main contribution of this paper is to lift the idea of state reconstruction from the multi-core setting of

our earlier work [10] to a distributed setting. To do so, since states are no longer residing in the same memory space, we have to ensure some locality in the distribution of states. The latter is what motivates the introduction of state stealing.

2.3 RDMA and OpenSHMEM

RDMA (Remote Direct Memory Access) is a communication mechanism that implements one-sided inter-process communication. It relies on two basic communication primitives `put()` and `get()` used by a process to write and read in memory of another process. Only a specific *public* memory area can be reached from other processes.

An attractive feature of one-sided communication is that only the process p that initiates the communication needs to take active part in it. The process that owns the memory area that p is reading from or writing into is not participating to the communication, nor is it even aware that this communication is happening. Fast cluster interconnection networks such as InfiniBand implement RDMA communications with zero-copy, meaning that the network interface card transfers data directly from one process's memory into the other process's memory, and, in particular, without involving the other process's operating system.

The RDMA paradigm is implemented in the OpenSHMEM shared heap and communication interface. OpenSHMEM is an API for parallel programs. It defines a set of one-sided RDMA communication routines, designed specifically for clusters featuring low-latency networks [1]. The processes are called *Processing Elements* (PEs). Each PE has its own (private) memory, and exhibits a public heap. OpenSHMEM features a *symmetric* heap: every PE has a shared heap of the same size, which contains the same allocated objects and static global objects as illustrated in Figure 2.

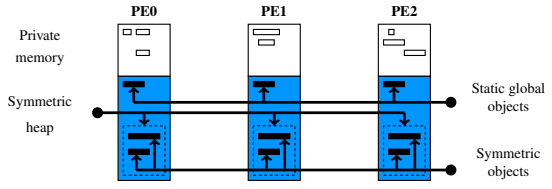


Fig. 2 OpenSHMEM memory model

Symmetry is maintained between shared heaps through the use of dedicated memory management routines (such as `shmem_malloc()` or `shmem_free()`). The OpenSHMEM specification states that these routines are *collective* routines and must end by something semantically equivalent to a barrier. Hence, every object is allocated at the same offset from the beginning of the buffer on all the PEs. Global and static variables are also located in the shared heaps and therefore remotely accessible by other PEs. The OpenSHMEM specification also defines interfaces for atomic accesses (such as *compare-and-swap*), collective operations, and locks.

3 Illustration of the algorithm

In this section we first present the general memory layout that our algorithm relies on: a state space scattered as a forest of trees distributed upon PEs (Processing Elements). We then informally illustrate the execution of our algorithm step by step on a small example state space to show how this forest is obtained. Section 4 contains the formalisation of our algorithm.

3.1 Overview and memory layout

The top of Figure 3 shows the graphical representation of a small example state space, and the bottom part depicts a possible outcome of our algorithm when distributed over three PEs (PE_0, PE_1 and PE_2). As in earlier figures, each state is split into its identity (top part) and its hash value (bottom part).

The general principle of the memory layout of our algorithm is to partition the reconstruction tree upon the different PEs in such a way that each PE holds in its private memory a forest of trees. Root states of these trees (states with a dark gray background on the figure) are fully stored in memory, *i.e.* their full state descriptor is available to the PE storing them. Other states are stored as a reference to their parent in the reconstrution tree together with a transition label that allows reconstructing the state from its parent. Dotted arcs represent links (edges) between states that are not part of the reconstruction tree: they were identified to lead to duplicate states by the duplicate detection procedure. Arcs in the reconstruction tree linking states on different PEs are indicated as thick arcs. The main advantage of this layout is that state reconstruction can be done locally since the reconstructing sequence of a referenced state starts at the root of the tree this state belongs to — root of which the PE has the full state descriptor — and only contains states located on the same PE. For instance, if PE_2 needs to reconstruct s_8 it will backtrack to s_3, recover its full state

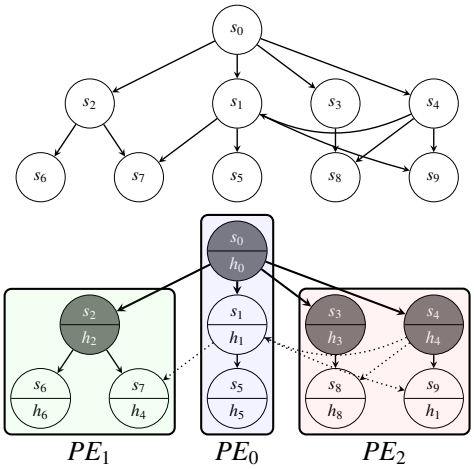


Fig. 3 Example state space (top) and a possible distribution on PEs (bottom)

descriptor and execute a single event to retrieve the full state descriptor of s_8. It is straightforward to see that reconstructing sequences are shortened compared to our multi-core algorithm [10] where all reconstructing sequences start at the initial state. As an obvious counterpart, the total memory consumption of our distributed algorithm exceeds the one of our multi-core algorithm since the latter only requires the initial state to be fully stored in memory. On the other hand, since the memory is distributed, we generally have more memory available.

This layout raises the question of how states are distributed upon PEs. A common approach in distributed model checking [18] is to use a partitioning function (*e.g.* based on the state hash function) mapping states to PEs. Such an approach does not seem relevant in our context since these functions usually favour even distributions (to balance as much as possible memory usage and workload) against locality (*i.e.* the fact that the partition function is designed to gather as much as possible states with their parents). The latter is our main concern: states must be gathered as much as possible in local trees for our technique to provide a good memory reduction (remember that only the root of a local tree is fully stored in memory).

To distribute the reconstruction tree upon PEs, our algorithm relies on a work stealing strategy. In a nutshell, during the expansion of a level of unexpanded states, a PE first proceeds by reconstructing all these states (starting from the roots it possesses) and makes their full state descriptors publicly available to all PEs (including itself). An idle PE then looks in the pools of states published by its peers and steals some of these to "plant" new trees in its private memory. After a PE has stolen a state from one of its peers (or simply took it back if it is the PE that constructed it), the state is expanded. For instance, the distribution observed on Figure 3 may have been obtained after PE_1 and PE_2 stole s_2, and s_3 and s_4, respectively, from PE_0.

3.2 Detailed execution of the algorithm

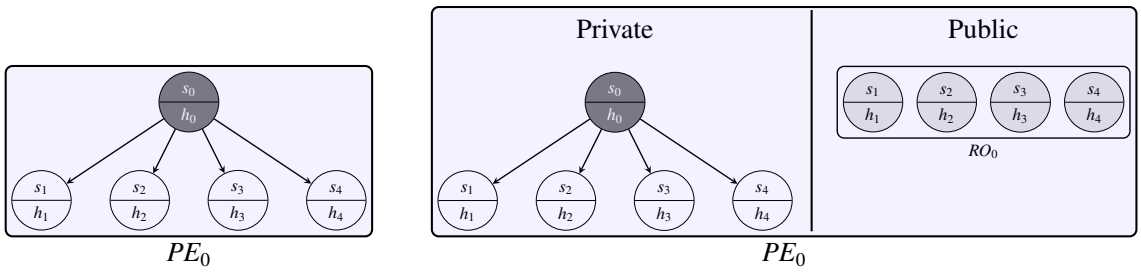
We now illustrate the steps performed by our algorithm to produce the distributed tree in Figure 3.

State expansion phase

Initially, one process, say PE_0, owns the initial state. It generates its successors $s_1, \ldots, s_4$ and puts their full state descriptors in a private hash table C_0 storing candidate states in a similar way as our multi-core algorithm. After this expansion step, duplicate detection is performed. This phase normally involves all PEs and will therefore be explained in greater detail later. At this point, only PE_0 can take part in this operation and since all outgoing edges of s_0 lead to new states, these can be directly inserted in the reconstruction tree of PE_0 leading to the configuration of Figure 4(a).

Then starts the expansion of level 1. From s_0, PE_0 traverses the tree to reconstruct $s_1, \ldots, s_4$. We do not detail this process here, but state expansion as well as duplicate detection rely on the same tagging principle as used by our multi-core algorithm: a state to be reconstructed is marked as such by flipping a specific bit and, using backward pointers, all states up to the root are tagged as well to indicate that they lead to a state to be reconstructed. Once reconstructed, the state descriptors of $s_1, \ldots, s_4$ are published by PE_0 in the RO_0 (reconstructed open states) shared data structure, reaching the configuration in Figure 4(b).

At that point, PE_1 and PE_2 which have not yet participated will see states published by PE_0 in RO_0 that contains open states to be expanded. We assume here that PE_1 steals s_2 while PE_2 steals s_3 and s_4, which leaves s_1 to PE_0. The implementation of state stealing will be described in more detail in the next section. Each PE_i generates the successor states of the open states it stole (or still has in its RO_i data structure) and puts these successors in its private candidate set C_i. This is illustrated in Figure 5 where a dashed arrow between two states indicates that the state in question was transferred (stolen) while a dashed arrow from PE_i to a state indicates a request from PE_i to steal the



(a) (b)

Fig. 4 After the expansion of s_0 4(a) ; and after the reconstruction of $s_1, \ldots, s_4$ for their expansion 4(b)

state. In subsequent figures we will indicate network communications in a similar manner.

Note that the full state descriptors of candidate states are kept in the candidate set (depicted with a light gray background) along with information needed to insert the states in the reconstruction tree in case the candidate is actually new: that is a reference to its parent state coupled with the event labelling the arc between the two states. The three PEs have now expanded all open states they own and can then proceed to duplicate detection.

Duplicate detection phase

Moving from the state expansion phase to the duplicate detection phase occurs in two situations. The first situation is when duplicate detection is triggered by the size of the candidate set reaching a predefined limit set to bound the amount of memory used for the candidate set. The second situation is when idle PEs periodically initiate duplicate detection to join its peers as this operation must involve all PEs. Note that, as in [10], the passage from one step to another is controlled by the use of synchronisation barriers.

Considering Figure 5, then PE_2 already noticed that the edge (s_4, s_8) led to a duplicate state since s_8 has been already reached from s_3 and is thus already present in the candidate set C_2 when expanding s_4. The duplicate detection phase still has to identify several duplicates (s_7 present in both C_0 and C_1 ; s_9 in C_2 and C_0 ; and s_1 in C_2 and in the private tree of PE_0) and resolve the corresponding hash conflicts (on h_1 and h_4) before each PE can insert its candidates that are actually new in its trees. Several operations are required to achieve this goal.

The identification of duplicates may naturally imply the reconstruction of some previously generated states (*i.e.* that are not present in any candidate set) as it is the case here for s_1. Thus, in order to identify these states, duplicate detection first starts by PEs exchanging hash values of candidate states. To do so, each PE_i puts in its public area the hash values of its candidates in set H_i. This is illustrated in Figure 6, step 1.

After a synchronisation barrier, each PE retrieves the hash values of its peers. It then knows which of the states it owns need to be reconstructed. A direct solution to detect duplicates would then be, for each PE, to publish in its public space the states it owns (candidates or reconstructed) that share the same hash values as those published by its peers in their H_i set. PE_0 would for instance reconstruct s_1 and put it in its public space, so that PE_2 can detect that the edge (s_4, s_1) leads to a duplicate state. However, since states sharing the same hash value may be found on each PE, this solution would involve a polynomial number of communications. Our solution rather relies on the use of a *detection proxy* (dp) based on a mapping $dp : \mathcal{H} \rightarrow \mathcal{P}$, where $\mathcal{H}$ is the set of hash values, and $\mathcal{P} = \{0, \ldots \mathsf{PE} - 1\}$). The basic idea is that if $dp(h) = i$, then PE_i will be responsible, during duplicate detection, of storing any candidate or reconstructed state s such that $hash(s) = h$.

Going back to our example, after each PE_i has retrieved the hash values published by its peers, it first writes each of its candidate states c in the public space of $p = dp(hash(c))$ in the DC^i_p state block (DC standing for distant candidates). Then, each state s present in its tree, and of which the hash value is present in the H_i of one of its peers, must be reconstructed and written in the public space of its proxy. With our example, PE_0 thus has to reconstruct s_1. Any reconstructed state s is then written in the R^i_p state block (again with $p = dp(hash(s))$). Assuming that $dp(h_4) = dp(h_5) = 0$, $dp(h_1) = dp(h_6) = 1$ and $dp(h_8) = 2$, we obtain the configuration of Figure 6, step 2.

Duplicate detection can then be performed locally. Each PE_i merges candidate states it received (*i.e.*

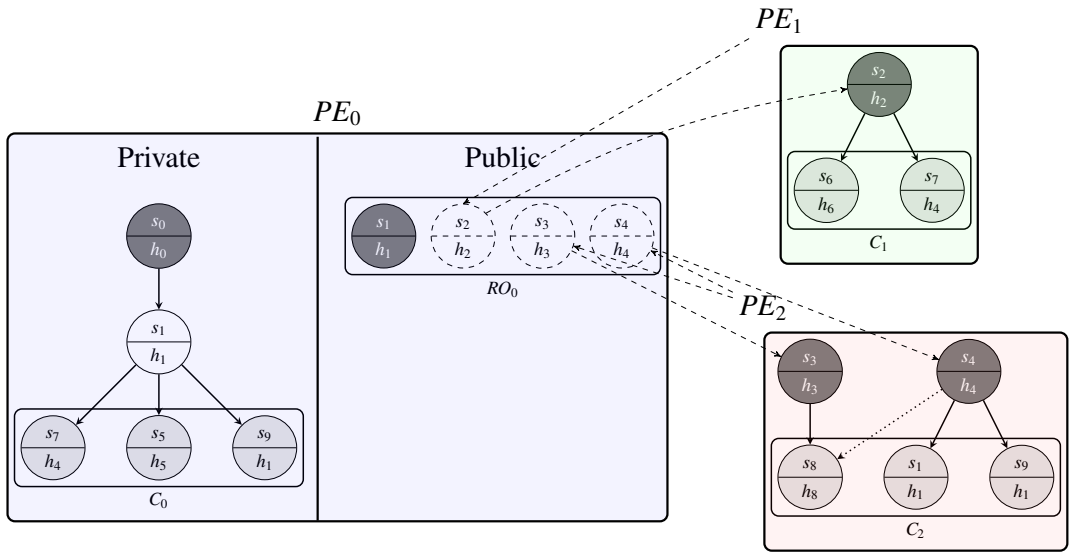


Fig. 5 After stealing and expansion of open states

blocks DC_i^j) in sets N_i^j. It then removes from these sets reconstructed states it received (*i.e.* blocks R_i^j). Formally, the sets N_i^j are defined such that:

- All distant candidates except reconstructed states are covered: $\forall i \in \mathcal{P} : \cup_{j\in\mathcal{P}} N_i^j = \cup_{j\in\mathcal{P}} DC_i^j \setminus \cup_{j\in\mathcal{P}} R_i^j$
- Sets are disjoint: $\forall i,j,k \in \mathcal{P} : j \neq k \Rightarrow N_i^j \cap N_i^k = \emptyset$
- Only distant candidates are included in the sets: $\forall i,j \in \mathcal{P} : N_i^j \subseteq DC_i^j$

In case a state s is present in both DC_i^j and DC_i^k (with $j \neq k$) (e.g., s_7 is present in both DC_0^0 and DC_0^1), then PE_i has to choose which of its two peers, PE_j or PE_k, will store s. In our example we made some assumptions on how these new states are distributed upon PEs, but we present in the next section the actual implementation of this process. As an example, the local duplicate detection performed by PE_1 thus leads to first merge DC_1^0, DC_1^1 and DC_1^2 hereby removing one occurence of s_9 and secondly to remove state s_1 present in R_1^0 (obtained by reconstruction from PE_0). After these merging and deletion steps, sets of new states are published by PEs in their public space leading to the configuration of Figure 6, step 3.

After a last synchronisation barrier, the duplicate detection phase ends with PEs processing new states: each PE_i remotely reads $N_0^i, \ldots N_{\|\mathcal{P}\|-1}^i$ and inserts all recovered states in its private tree. Note that it is not necessary to store full state descriptors in the sets N_i^j since the goal of duplicate detection is ultimately to insert reference states in the reconstruction tree (*i.e.* a pair identifying the parent state and a transition label). Hence, it is preferable, in order to reduce communication times, to only store this meta information in the N_i^j sets. At the end of duplicate detection, we reach the configuration of Figure 6, step 4.

The operations described above will be reiterated on the newly discovered states. Termination occurs when PEs do not have any new open states to prosess. As we will see in the next section this can be detected during duplicate detection.

4 Specification of the algorithm

We give in this section an algorithmic specification of the core operations performed by our algorithm.

4.1 Data structures

For the sake of clarity of the presentation we first detail some global data structures listed in Figure 7. These are private (*i.e.* in the private memory space of the PEs) unless specified with the **public** keyword.

- `ROOTS` contains the list of root states of all the trees owned by the PE. Full state descriptors of these roots are stored and each such root is associated an integer identifier.
- `TREE` encodes the tree structure. It maps any state identifier to the list of its successors (*i.e.* the successor identifier and the event that generated it) that are part of its tree.

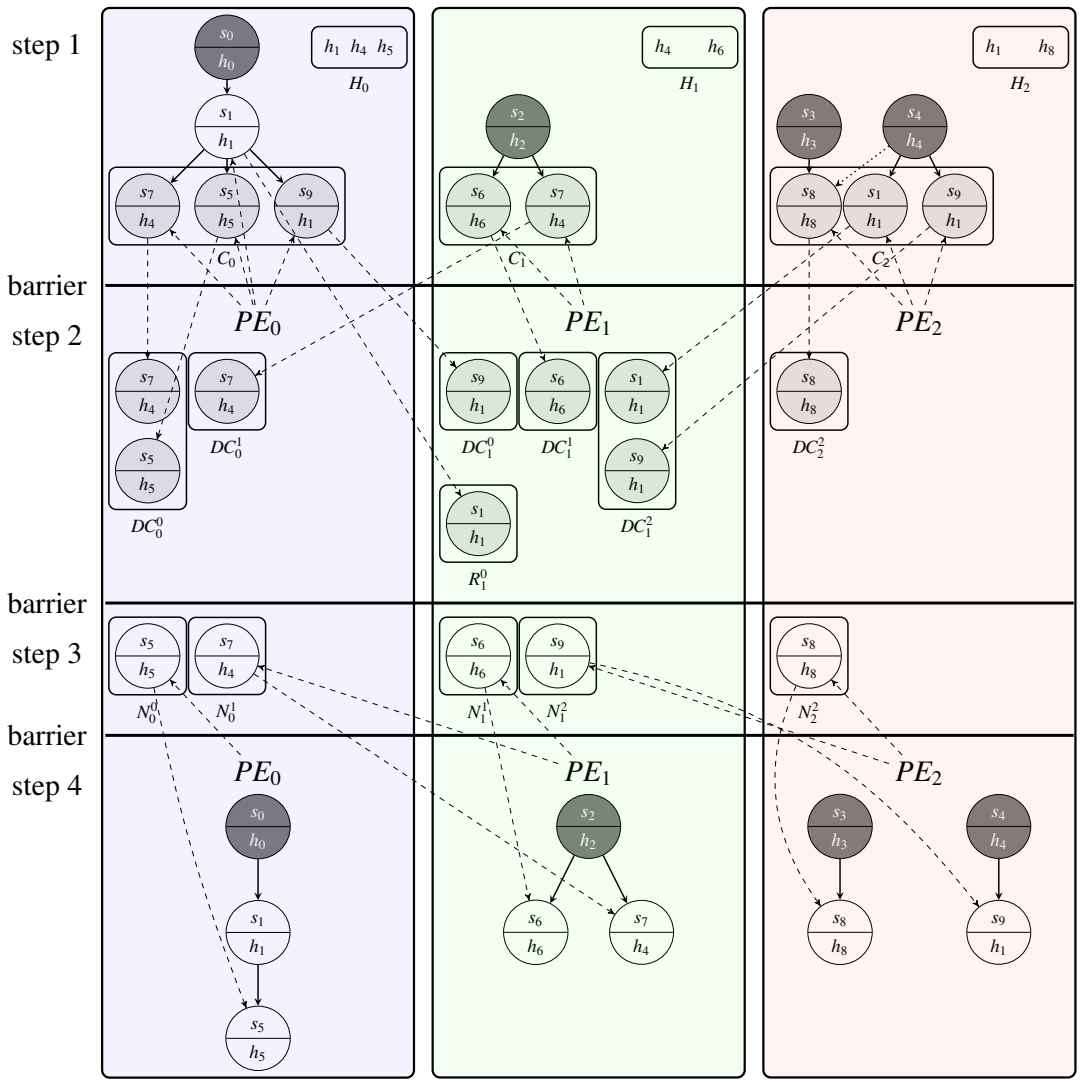


Fig. 6 The duplicate detection process. Step 1: publication of hash values of candidate states. Step 2: writing of candidates and reconstructed states in their detection proxy's public space. Step 3: publication of new states obtained after candidates merging and removal of reconstructed states. Step 4: PEs remotely read new states they are assigned and insert these in their trees.

As in the previous section, `C`, `RO`, `H`, `DC`, `R` and `N` denote, respectively: the set of candidate states, reconstructed open states, hash values, distant candidates, reconstructed states, and new states. The candidate set `C` maps any candidate state c to a pair (id, e) where id is an integer identifying the predecessor state p of c, and e is the event such that $succ(p, e) = c$. The algorithm has to memorise this information since the candidate may ultimately be inserted in the `TREE` structure if it is found to be new.

Three synchronisation variables are used to check for global termination (`TERM`), and level (iteration) termination ($\texttt{LVL\_TERM}_{\texttt{ME}}$ and `LVL_TERM_ALL`).

4.2 Main algorithm

In the main procedure (see Figure 8), PE_0 first plants a new tree rooted in s_0. The PEs then explore the state space in a breadth-first manner. Each iteration of the main loop processes one breadth-first search (BFS) level until termination.

The PE first reconstructs the open states it owns at l. 13, then explores these (l. 15) before trying to explore open states owned by its peers (l. 16).

To control the level and global termination, the synchronisation variable $\texttt{LVL\_TERM}_{\texttt{ME}}$ is initialised to **false**, indicating that the PE has not finished processing this level, and the global termination flag `TERM`

```
1  PES: constant set of int = {0, ..., number of PEs - 1}
2  ME: constant int = PE's identifier
3
4  ROOTS: list of (int * state) = empty
5  TREE: map of (int → list(int * event)) = empty
6
7  C: map of (state → (int * event)) = empty
8  RO_ME: public set of (int * state) = empty
9  H_ME: public set of int
10
11 DC^0_ME, ..., DC^(PES-1)_ME: public set of state
12 R^0_ME, ..., R^(PES-1)_ME: public set of state
13 N^0_ME, ..., N^(PES-1)_ME: public set of state
14
15 TERM: bool = false
16 LVL_TERM_ME: public bool
17 LVL_TERM_ALL: bool
```

Fig. 7 Variables and data-structures used by our algorithm

```
1  proc main():
2
3    if ME == 0:
4      add (0, s_0) to ROOTS
5      add (0 → empty) to TREE
6
7    while not TERM:
8
9      LVL_TERM_ME = false
10     TERM = true
11
12     for (id, root) in ROOTS:
13       recons_open(id, root)
14
15     explore_my_open()
16     explore_peers_open()
```

Fig. 8 Main procedure of our algorithm

is set to **true** at the beginning of each loop iteration. If open states of the current level (iteration) generate new states, TERM will be set to **false** during duplicate detection (ll. 41, 42 in Figure 10).

4.3 State expansion

Let us now consider Figure 9 that contains the code of the state expansion phase.

Procedure recons_open (l. 1) recursively (l. 11) reconstructs the PE's open states and puts their full state descriptors in the set RO_{ME} (l. 5). This set has a bounded size: whenever it fills up, the PE starts exploring its open states (l. 8).

Although, as presented, the reconstruction implies a traversal of all nodes owned by the PE, our algorithm makes use of the same tagging mechanism as used in [10] to prune the tree and only visit tree nodes leading to open states. We have hidden this process for the sake of simplicity. The same remark applies to the reconstruction process used during duplicate detection (l. 55 in Figure 10).

When exploring open states present in its own RO_{ME} set (procedure explore_my_open at l. 13), a PE tries to pick one of these in order to explore it. Since, meanwhile, another PE may also steal some states from that same set, the PE must not carelessly pick states from it: synchronisations are required through the use of the steal_from procedure to guarantee a proper consumption of reconstructed states.

The exploration of an open state (procedure explore_open at l. 22) simply consists in putting all its successors in the candidate set — which may trigger duplicate detection if the candidate set fills up (l. 32).

Once a PE has finished exploring its own states it tries to steal and process states from its peers (procedure explore_peers_open at l. 34, called at l.16 of the main algorithm in Figure 8).

The PE first chooses another *victim* PE from which it tries to steal some state(s) (ll. 41, 42). States stolen this way are also explored using procedure explore_open with the difference that they must also be inserted in ROOTS. As a requirement, this procedure must also periodically invoke duplicate detection (l. 50) meaning that procedure do_dd that checks if duplicate detection has to be initiated locally must eventually return **true** (we leave the actual implementation of this procedure to the next section). Indeed, other PEs may be waiting for their peers to perform duplicate detection so this operation must eventually be triggered by all PEs. Moreover the exit from

tion

```
1  proc recons_open(id: int, s: state):
2
3    if is_open(id):
4
5      add (id, s) to RO_ME
6
7      if RO_ME is full:
8        explore_my_open()
9
10   for (idc, e) in TREE(id):
11     recons_open(idc, succ(s, e))
12
13 proc explore_my_open():
14
15   while RO_ME is not empty:
16
17     stolen = steal_from(ME)
18
19     for (id, s) in stolen:
20       explore_open(id, s)
21
22 proc explore_open(id: int, s: state):
23
24   for e in enab(s):
25
26     s' = succ(s, e)
27
28     if s' not in C:
29       add (s' → (id, e)) to C
30
31     if C is full:
32       dd()
33
34 proc explore_peers_open():
35
36   LVL_TERM_ME = true
37   LVL_TERM_ALL = false
38
39   while not LVL_TERM_ALL:
40
41     v = choose_victim_pe()
42     stolen = steal_from(v)
43
44     for (_, s) in stolen:
45       i = new_id()
46       add (i, s) to ROOTS
47       add (i → empty) to TREE
48       explore_open(i, s)
49
50     if do_dd():
51       dd()
```

Fig. 9 The exploration procedure

explore_peers_open — which, in turn, means moving to the next BFS level — is also conditioned by the outcome of the duplicate detection procedure as explained below.

4.4 Duplicate detection

Finally, the duplicate detection procedure (procedure dd in Figure 10) follows the different steps introduced in our example (see Figure 6). We draw the reader's

```
1  proc dd():
2
3    /* step 1 of fig. 6 */
4    for pe in PES:
5      DC^pe_ME = empty
6      R^pe_ME = empty
7
8    H_ME = { hash(c) for (c → _) in C }
9
10   barrier()
11
12   /* check BFS level termination */
13   LVL_TERM_ALL = ∀ pe in PES: LVL_TERM_pe
14
15   /* step 2 of fig. 6 */
16   for (id, root) in ROOTS:
17     recons_dd(id, root)
18
19   for (c → _) in C:
20     add c to DC^ME_dp(hash(c))
21
22   barrier()
23
24   /* step 3 of fig. 6 */
25   for pe in PES:
26     N^pe_ME = empty
27
28   /* merge candidates */
29   for pe in shuffle(PES), c in DC^pe_ME:
30     if ∄ pe' st c in N^pe'_ME:
31       add c to N^pe_ME
32
33   /* delete duplicates */
34   for pe in PES, r in R^pe_ME:
35     if ∃ pe' st r in N^pe'_ME:
36       del r from N^pe'_ME
37
38   barrier()
39
40   /* check global termination */
41   TERM = TERM
42     and (∀pe, pe' in PES: N^pe'_pe is empty)
43
44   /* step 4 of fig. 6 */
45   for (c → (id, e)) in C:
46
47     if c in N^ME_dp(hash(c)):
48       nid = new_id()
49       add (nid → empty) to TREE
50       add (nid, e) to TREE(id)
51
52   C = empty
53
54
55 proc recons_dd(id: int, s: state):
56
57   if ∃ pe st hash(s) in H_pe:
58     add s to R^ME_dp(hash(s))
59
60   for (idc, e) in TREE(id):
61     recons_dd(idc, succ(s, e))
```

Fig. 10 The duplicate detection procedure

attention to the operation that consists in merging candidates sent by peers (l. 29). In order to balance the workload as much as possible, the algorithm does not consider PEs one by one starting from PE_0 — which would result in processes having a workload that decreases with their identifier — but instead in a random way.

Duplicate detection being a rendez-vous between all PEs, allows for an easy detection of termination of both the current BFS level and the global search. The termination checks (at l. 13 and 41) are placed after barriers to avoid any race condition. The current BFS level may be considered as terminated (l. 13) if all PEs have executed the statement at l. 36. If this holds, then it is guaranteed that any open state of the current BFS level either has been explored by its owner, or has been stolen and processed by another PE. Global termination is detected if no candidate state was identified as new (l. 41) during the exploration of the current BFS level.

4.5 Correctness of the algorithm

We now prove that our algorithm, upon termination, has explored all states reachable from the initial state (completeness); that only states reachable from the initial state are computed (soundness); and that visited states are stored only on a single PE.

The propositions and proofs of correctness below are directed by the observation that each iteration of the algorithm explores the next BFS level. The proof of termination is split into a proof of termination of each iteration of the algorithm, and a proof of global termination when there are no longer open states to be explored. We denote by BFS_i the set of states at BFS level i in the state space, with BFS_0 containing only the initial state. $BFS_{0..i}$ denotes the set of states up to BFS level i, i.e., $BFS_{0..i} = \cup_{0 \leq j \leq i} BFS_j$. For the proofs of termination it is important to note that not only is each PE required to terminate, but termination must be detected by all PEs (global termination). In the formulation of propositions and proofs we assume that the state space being explored is finite, i.e., we have a finite number of reachable states and a finite number of enabled events in each reachable state.

Proposition 1 (Soundness)**.** *Let s be a state stored in* `ROOTS` *or being reconstructible from* `TREE`*. Then s is reachable from the initial state.*

Proof The initial state is inserted on PE_0 at the beginning of the execution (Figure 8, ll. 4–5). The only states that are added to `ROOTS` and `TREE` are computed at l. 26 of Figure 9, as successors of an existing state. Hence such states (by transitivity) belong to the set of reachable states. □

Proposition 2 (Level Completeness and Termination)**.** *Assume that* `ROOTS` *and* `TREE` *prior to an iteration in l.7 of Figure 8 represent states $BFS_{0..i}$ for some $i \geq 0$ with each state being represented on a single PE. Then the iteration will terminate and upon termination* `ROOTS` *and* `TREE` *represent states $BFS_{0..i+1}$ with each visited state being represented on a single PE.*

Proof At each iteration, open states are explored, either by `explore_my_open` or by `explore_peers_open` (Figure 8, l. 15 and 16, respectively). Each of these procedures steals an open state (Figure 9, l. 17 and 42), which is then explored by the PE with `explore_open` (Figure 9, l. 20 and 48). This procedure only adds its successors, so does not go beyond a BFS level. If the new state fills the candidate set `C`, duplicate detection is launched, which ends in cleaning the candidate set at l. 52 of Figure 10, so that the exploration can continue. Hence, all open states in the BFS level are explored.

An iteration ends with the call to `explore_peers_open`, described in Figure 9. The PE gets at least once in the **while** loop. At the end of this loop, the duplicate detection (procedure `dd`, Figure 10) is called as procedure `do_dd` must eventually return **true**. A concrete implementation of `do_dd` which ensures this is provided in Section 5.2. All PEs must synchronise, as per the barrier on l. 10. When passing this barrier, variable $\texttt{LVL\_TERM}_{\texttt{ME}}$ of all individual PEs is used to check for the global termination of the BFS level. It is **true** when exploring open states of peers, but **false** when exploring the owned states of the PE. Thus, $\texttt{LVL\_TERM}_{\texttt{ME}}$ is **true** for all peers when all the local open states have been explored and no state remains to steal, thereby ensuring termination.

The duplicate detection deletes duplicates at l. 36, keeping only the copy in R^{pe}_{ME}. This copy has been added by l. 58 in a single set R^{pe}_{ME}. Therefore, a state is owned by (represented at) a single PE. □

Proposition 3 (Global Completeness and Termination)**.** *The main algorithm in Figure 8 eventually terminates and upon termination each reachable state s is represented by* `ROOTS` *and* `TREE` *of exactly one PE.*

Proof As we assume that the state space is finite, then there will be a finite number of BFS levels. Hence, eventually there will be no more open states to be explored. Termination

happens at the end of the **while** loop in the `main` procedure, *i.e.* after exploring a BFS level, when `TERM` is **true**. This value is set at the beginning of the loop and can only be modified at l. 41 in Figure 10. When set to **false** by one of the PEs, other PEs can execute this line without changing the value. To first make it **false**, a PE must see that one of the sets of new states `N` is not empty. The modification of these sets occurs only at step 3 of procedure `dd`, between two barriers. Therefore, all PEs perform the same computation. The only deletion of states occurs when finding duplicates at l. 36, Figure 10. Hence, all states of the state space are present on a single PE as per Prop.2. □

5 Implementation

We have implemented our algorithm in the Helena tool [9]. We provide in this section implementation details regarding the state stealing mechanism and the triggering of the duplicate detection procedure. The implementation of these two mechanisms were deliberately left unspecified in the previous section in order to make the presentation of the algorithm as general as possible.

5.1 State stealing strategies

A key point of our algorithm is the state stealing mechanism aiming at providing a balanced workload between processes during state space exploration. We designed four strategies to implement the `choose_victim_pe` and `steal_from` functions (see Figure 9) that decide from which victim peer a PE should try to steal states and how these should be stolen. The four strategies are:

- rnd where a PE simply picks its victim in a random way.

- ring where a PE_i first tries to steal from its right neighbour (*i.e.* $(i+1) \mod |PES|$) and keeps stealing from this neighbour as long as its attempts are successful. It then repeats this process on following PEs: $(i+2) \mod |PES|$, $(i+3) \mod |PES|$, ...

- info which takes advantage of duplicate detection being a rendez-vous, thereby making it an appropriate moment for PEs to exchange information. In this strategy, PEs exchange their numbers of reconstructed states (*i.e.* $|\mathrm{RO_{ME}}|$ in Figure 7). Peers are then sorted locally according to these numbers in a decreasing order and choose their victims following that order.

- frnd (fully randomized) that proceeds as rnd, except that a PE, not only randomly selects its victim but it also randomly selects a block of states to steal from it. This differs from other strategies in which a PE always tries to steal the first block of states from the victim it selected. Indeed, on each PE_i, the set $\mathrm{RO_i}$ is organised as a ring of blocks of states. Hence, strategy frnd should reduce conflicts between stealing PEs and enhance concurrency.

5.2 Duplicate detection triggering

The triggering of duplicate detection is controlled by function `do_dd` that, as we have specified is required to eventually return **true** to preserve termination. To implement this decision procedure we designed and implemented two strategies:

- steal where an idle PE (*i.e.* one that does not own anymore a state to expand) will enter the duplicate detection step after $|PES| - 1$ unsuccessful steal attempts. The intuition behind this strategy is that, at that point, it is likely that there is not anymore available work on the current BFS level. Triggering duplicate detection will then close this level allowing to move to the next one.

- time consists of triggering the duplicate detection after a fixed amount of time. It is therefore parametrised by the number of milliseconds representing that amount. Hereafter, we denote by time(ms) this strategy parametrised by waiting time ms.

6 Experimental Evaluation

We conducted experiments with models of concurrent systems specified in the DVE input language of the DiVinE model checker [2] and in the Petri Net (PN) formalism of the Helena tool [9]. We have conducted two set of experiments. In the first set of experiments, we explore configurations of our algorithm to investigate the impact of its parameters on performance. In the second set of experiments, we evaluate the relative performance of our algorithm to related multi-core and distributed algorithms.

6.1 Cluster computing infrastructure

Our experiments were carried out using the Grid'5000 [8] testbed, supported by a scientific

interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organisations[2]. We used the OpenSHMEM implementation provided by OpenMPI 2.0.1. For the first set of experiments we used the `grimoire` cluster of the Nancy site which features 8 nodes with 2 Intel Xeon E5-2630 v3 CPU and 8 cores per CPU, connected through a 56 Gbps Infiniband network. The second set of experiments was conducted on the `uvb` cluster of the Sophia site which features 4 nodes with 2 Intel Xeon X5670 CPU and 6 cores per CPU, connected with a 40 Gbps InfiniBand network.

6.2 Input models

The models we have experimented with are listed in Table 1, together with the number of states and arcs in their state space, their height (*i.e.* the number of BFS levels in their state space) and the size of their state vector (in bytes). All DVE models come from the BEEM database [17] and all PN models come from the Petri Net Model Checking Contest (MCC) [16] model database. We have selected models having state spaces with different structural characteristics in terms of width, depth, and size. This has been done to ensure that our experimental evaluation covers a representative set of models which in turn contributes to the validity our conclusions.

Model	States	Arcs	Height	State size
DVE models				
collision.5	431 M	1,644 M	182	52
firewire_tree.7	121 M	778 M	343	647
iprotocol.8	447 M	1,501 M	353	45
leader_election.7	235 M	1,712 M	248	281
needham.7	806 M	3,546 M	44	98
pgm_protocol.11	499 M	1,207 M	2,238	129
public_subscribe.5	1,153 M	5,447 M	170	36
synapse.9	1,675 M	3,291 M	88	58
Petri Net models				
aslink(1,a)	189 M	957 M	418	53
des(1,a)	109 M	1,213 M	105	84
discovery(8,a)	214 M	2,183 M	74	33
shieldRVT(2,b)	165 M	1,248 M	156	38

Table 1 Models used in the experiments

6.3 Algorithmic configurations

For the first set of experiments we have launched a full state space exploration for 32, 48, 64, ..., 128 PEs, (*i.e.* for 4, 6, ..., 16 PEs per node) and with 64 algorithmic configurations. An algorithmic configuration is a tuple (C, RO, BS, ST, DD) where:

- C is the size of private candidate sets (*i.e.*, if the candidate set reaches this threshold, the PE immediately triggers duplicate detection) ;
- RO is the size of the shared reconstructed open states data structure in which a PE puts reconstructed states available for stealing ;
- BS is the number of states a block of stolen states can hold ;
- and ST and DD are the state stealing and duplicate detection triggering strategies as described in Section 5.

The 64 algorithmic configurations are obtained by a cartesian product of the following values:

- $C \in \{10^4, 10^5\}$
- $RO \in \{10^4, 10^5\}$
- $BS \in \{10^2, 10^3\}$
- $ST \in \{$ rnd, ring, info, frnd $\}$
- $DD \in \{$ steal, time(10) $\}$

Overall, our experiments resulted in 5 376 runs for a cumulated runtime of approximately 17 days. We checked that all runs on the same model resulted in the same number of states.

6.4 Observations on execution times

Figures 11 to 13 graphically represent execution times for the 12 models considered and for the 10 best algorithmic configurations of each model. By 10 best configurations, we mean those that obtained the 10 smallest execution times on a 128 processes run. The configurations depicted are ordered according to execution time, and the values depicted are execution times observed by the best configuration.

For DVE models (Figures 11 and 12) we observe that the execution times generally follow the same pattern. It decreases as more processes participate, but reaches a threshold with approximatively a hundred PEs and then stagnates for the largest configurations. Model pgm_protocol.11 stands out in that performance degrades significantly beyond 96 processes. We will focus later on this particular case.

For PN models (Figure 13) this observation is less valid as we observe a continuous decrease in the execution times throughout our process range. We conjecture that it is due to the higher complexity

[2] https://www.grid5000.fr

tion

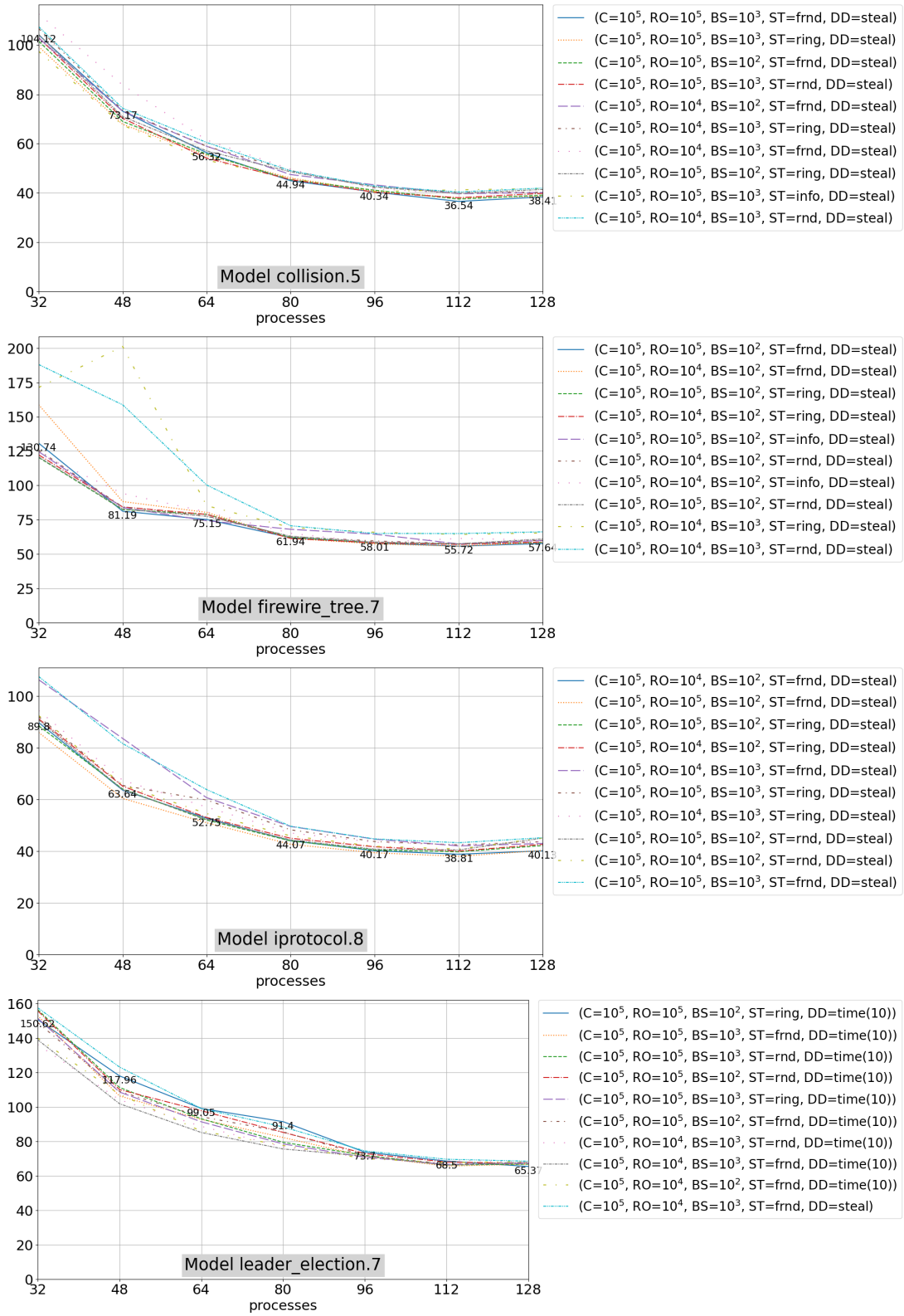


Fig. 11 Execution time in seconds (4 DVE models)

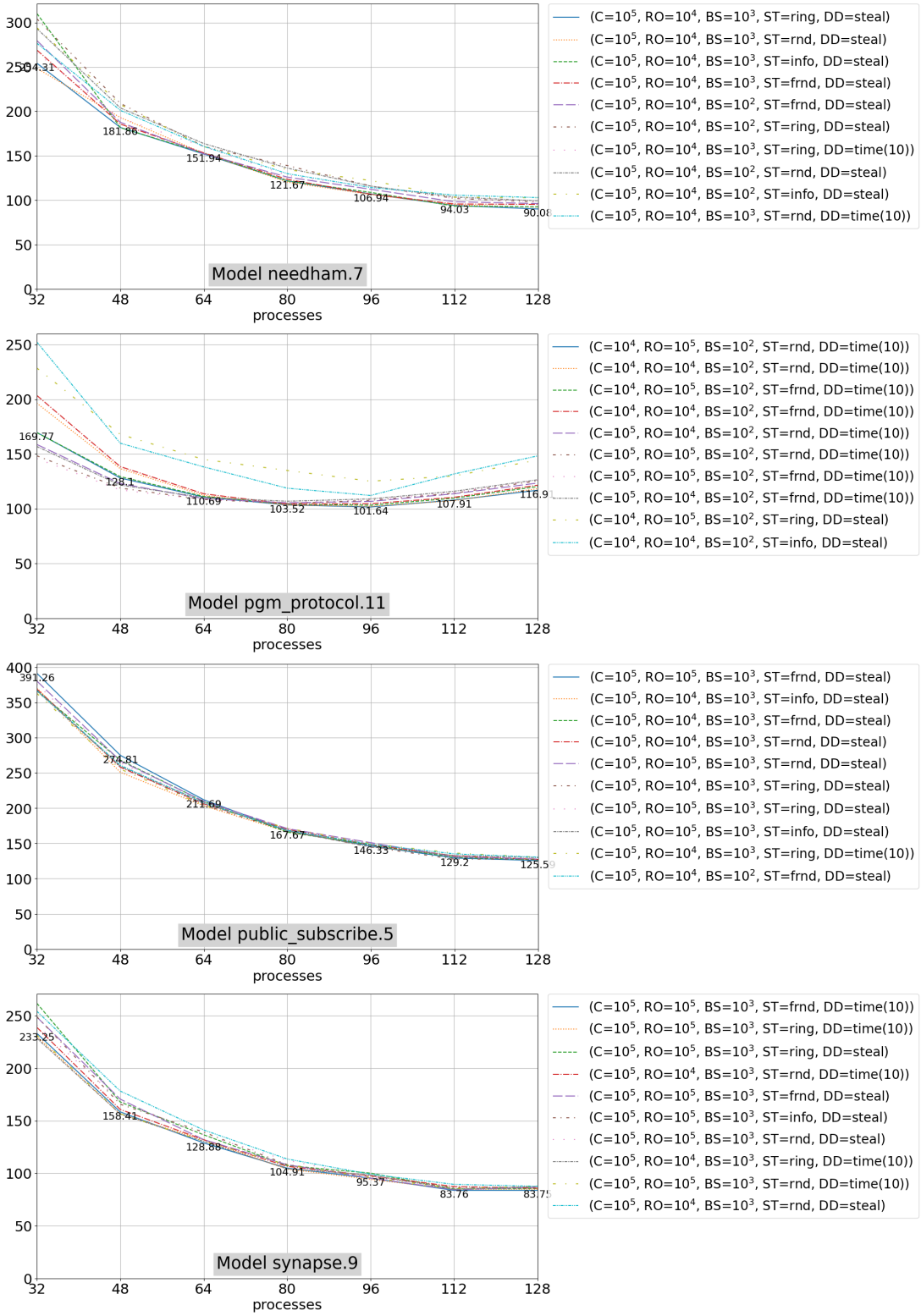


Fig. 12 Execution time in seconds (4 DVE models)

tion

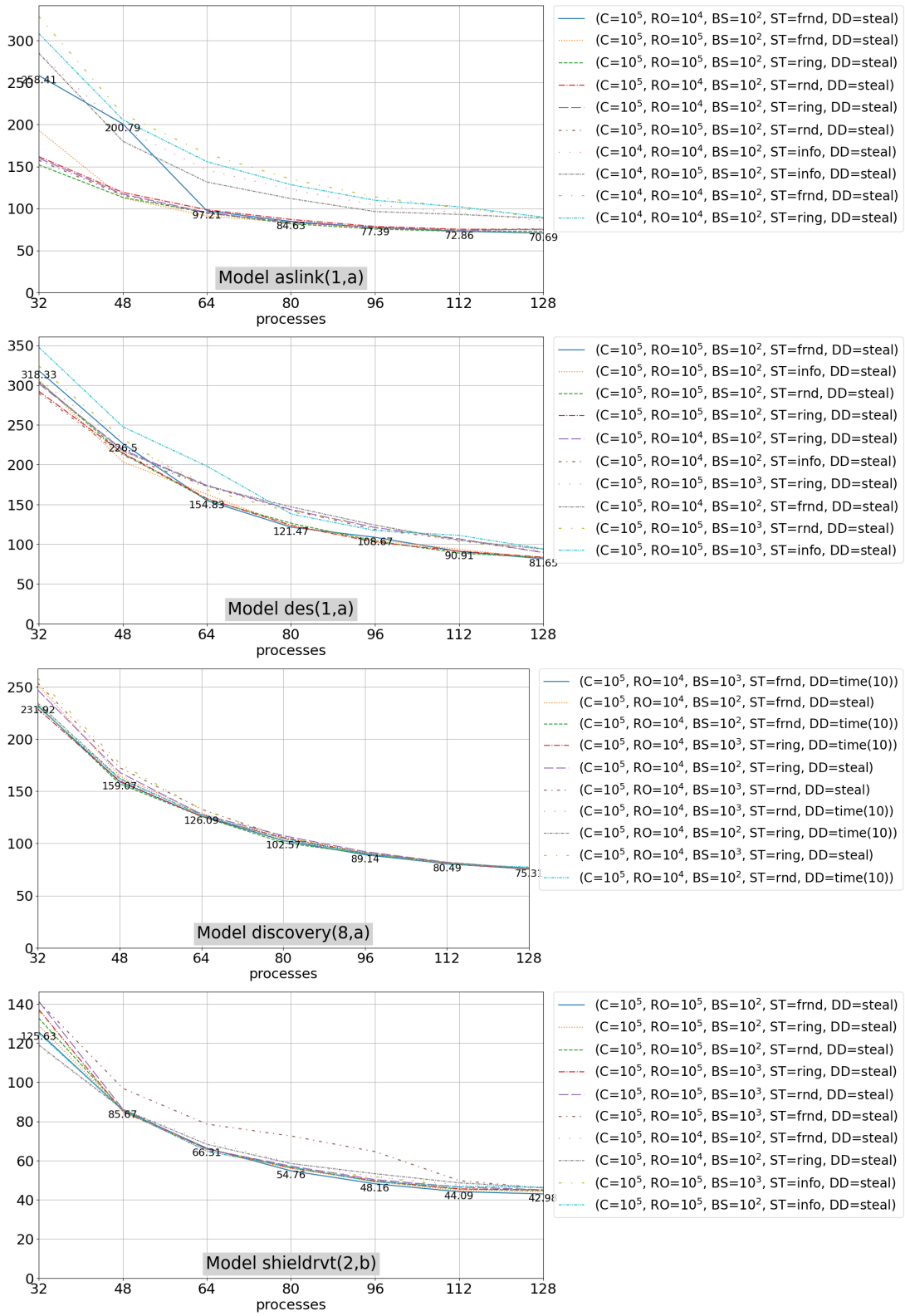


Fig. 13 Execution time in seconds (4 PN models)

of internal operations performed by PEs: computation of successors and state encoding and decoding involve more costly computations for PN than for DVE models. Hence, in the case of PN, processes spend more time in internal computations rather than in synchronisations.

It should be noted that when running our algorithm using only a single processing element it becomes equivalent to the sequential (reference) algorithm with state compression presented in [13]. As part of our experiment we did execute our algorithm in this configuration, and the execution time obtained aligns with the trend shown in Figures 11 to 13, *i.e.* with a single processing element it has a higher execution time than when using two or more processing elements.

Having a closer look at best configurations, it appears that the size of the candidate set C is the parameter having the largest impact. All the best configurations have this size set to 10^5 except for model pgm_protocol.11 that, once again, stands out. This is not surprising as a larger candidate set implies less frequent duplicate detections and in turn less synchronisations. The second parameter that seems the most discriminating is the duplicate detection strategy *DD*. Best configurations usually rely on the steal strategy.

We observe that generally switching the stealing strategy has a weak impact. It seems that randomisation based strategy (frnd and rnd) performs slightly better while info performs worse. The same remark applies to parameter *RO*. Last, it should be noted that large blocks (parameter *BS*) are preferable for large state spaces (see, e.g., models needham.7, public_subscribe.5 and synapse.9 on Figure 11), which was to be expected since, then, the amount of stealable states is more important and large blocks naturally mean fewer communications.

As pointed out earlier, model pgm_protocol.11 refutes most of our previous observations. We conjecture that the particular structure of its space — especially its large height (more than 2 000 BFS levels, see Table 1) — can explain the differences observed. Its large height implies frequent duplicate detections and, hence, more synchronisations. The steal strategy for duplicate detection triggering seems to be less efficient for that model since small BFS levels naturally imply fewer states to steal. Hence, processes have to poll all their peers (usually unsuccessfully) before entering duplicate detection. We thus observe some form of starvation. It seems that this model is a challenging case for our algorithm. Releasing the BFS strict order policy might improve its performances in that case.

6.5 Observations on workload

To have a better understanding of the performances of our algorithm, we also investigated how the stealing mechanism distributes the workload upon processes. The left column of Figure 14 contains plots graphically representing the workload of a single run with 128 processes for 5 selected models: on the x-axis is the time in seconds and on the y-axis is the number of states processed during the last elapsed second by the most (dashed green curve) and least (red dotted curve) loaded process for that second. On the right column of this same figure, we have plotted, for the same models, the number of states stored for each of the 128 processes.

It can be seen that the discrepancy between workload (left column) tends to be well-balanced and controlled. Model synapse.9 stands out as we observe important differences of workloads for that model. This is confirmed by the final state distribution in Figure 14 (right) of states upon processes. For other models, it can be seen that the majority of processes have similar state distributions. The relatively bad performance of model synapse.9 is somehow surprising since it has large and wide (*i.e.* with a small height) state spaces. Therefore, there should be opportunities for state stealing. This tends to show that there is still room for improving our stealing strategies. On the contrary, the workload and state distribution observed with model pgm_protocol.11 seem to indicate that the work distribution operates pretty well but, as noted above, that our algorithm, as such, is not suited to it.

6.6 Observations on memory usage

A crucial aspect of our algorithm is also the proportion of stolen states as these are fully stored in memory. Hence, the higher this proportion, the lower the memory reduction. It is also desirable that this proportion should not be affected in a negative way by the increase of participating processes. Figure 15 gives the proportion of stolen states for all runs of the best configuration of each model. We observe that this proportion hardly increases as we increase the number of processing elements, and in all cases remains below 7%.

This shows that our algorithm has very little overhead in terms of state transfer (state stealing). In particular, it shows that the number of explicitly stored states (stolen states) remains low, and scales well when increasing the number of processing elements. Combined with the results of Figure 14(right)

tion

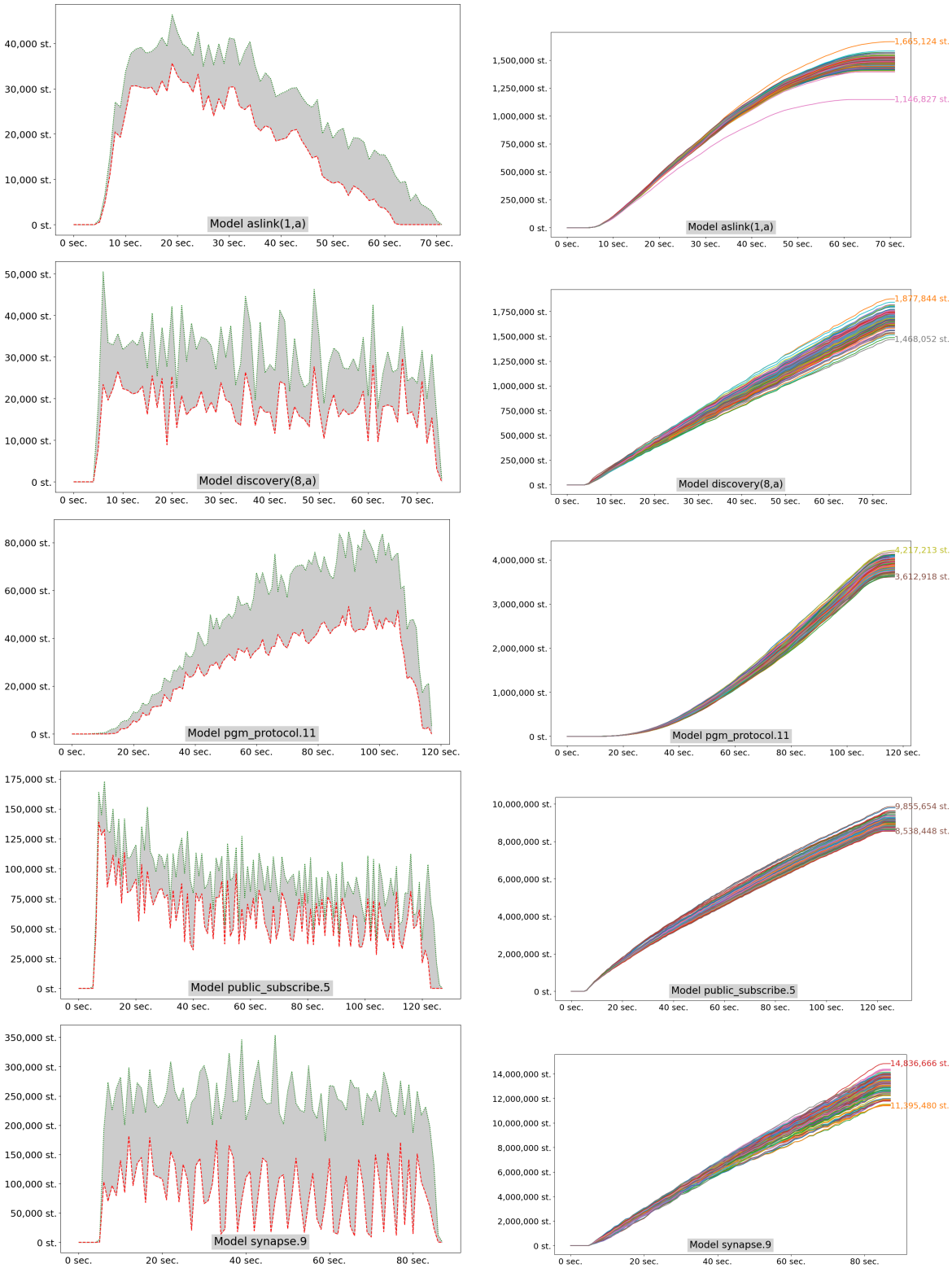


Fig. 14 Difference between the least and most loaded process (left column) and state distribution (right column) at each second for a single 128-process run at each second for a single 128-process run

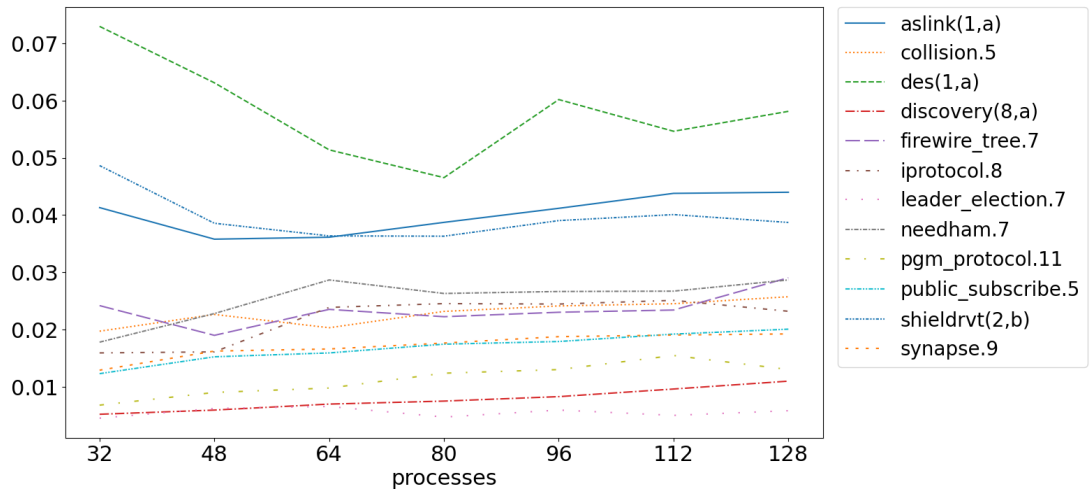


Fig. 15 Proportion of stolen states for the best configuration of each model

demonstrating that the memory consumption is fairly well distributed among the processing elements, this implies that the memory consumption on each processing element is close to M/n where M is the memory consumption of the sequential algorithm with state reconstruction, and n is the number of processing element.

6.7 Comparison with other parallel algorithms

In the second set of experiments, we compared our algorithm to three other ones that are also implemented in Helena:

- the multi-core algorithm of [10], also based on the principle of state reconstruction ;
- the distributed algorithm of [6] that is a distributed BFS tailored for RDMA architectures but without any state compression technique implemented ;
- and the distributed algorithm of [7], built on top of [6], that implements a state compression mechanism inspired by Spin's collapse compression technique [15].

Algorithms [6], [7] and our new algorithm could be run on 4 to 48 processes, while algorithm [10] could be run on 1 to 16 cores. For our algorithm, we used the configuration $(C = 10^5, RO = 10^4, BS = 10^3, ST = \mathsf{rnd}, DD = \mathsf{steal})$ which our first set of experiments identified as one of the best configurations on the average. Likewise for other algorithms, we used appropriate configurations as described in the mentioned papers. Each test (*i.e.* a triple consisting of an algorithm, a model, and a number of threads/processes) was run 5 times. Numbers reported below are averages. For all tests, either the 5 runs succeeded, or none did.

Table 2 summarises our results in the second set of experiments. Each cell reports the relative performances of our algorithm compared to one of the three listed above. For each comparison, either our algorithm could be run and terminated for the 5 runs whereas its competitor could not (marked with a ✓ symbol) ; or our algorithm could not terminate while the other could (marked with a × symbol) ; or none of the two could (marked with a = symbol) ; or both could. In the latter case, the cell contains two ratios with the first number being the run-time of our algorithm divided by the run-time of the other algorithm while the second number relates memory usage in the same way.

Several reasons can explain that an algorithm could not be run or could not terminate. For instance, algorithm [7] assumes state vectors with fixed size, making it unusable for Petri nets, while in some configurations, the algorithm ran out of memory even in a distributed setting. For several models, algorithm [6] ran out of memory. However, we could still estimate its memory usage by simply multiplying the state vector size by the number of reachable states. This is represented in the table by a ✓ symbol coupled with a ratio relating memory usage. For instance, for model firewire_tree.7 using 4 processes, our algorithm terminated successfully while [6] did not and our algorithm consumed 5% of the memory that [6] would have theoretically required.

tion

Model		Threads/Processes				
		4	8	16	32	48
aslink(1,a)	[6]	1.88 / 0.52	2.02 / 0.52	2.06 / 0.52	3.10 / 0.53	4.09 / 0.53
	[7]	✓	✓	✓	✓	✓
	[10]	1.58 / 1.24	1.46 / 1.25	1.01 / 1.26		
collision.5	[6]	4.99 / 0.52	4.43 / 0.52	3.33 / 0.53	2.57 / 0.53	2.12 / 0.53
	[7]	4.13 / 2.09	3.93 / 2.10	3.27 / 2.11	2.86 / 2.12	2.56 / 2.12
	[10]	1.22 / 1.23	0.81 / 1.24	0.44 / 1.25		
des(1,a)	[6]	1.48 / 0.34	1.46 / 0.34	1.39 / 0.35	1.43 / 0.36	1.47 / 0.37
	[7]	✓	✓	✓	✓	✓
	[10]	1.26 / 1.28	1.17 / 1.31	0.92 / 1.34		
discovery(8,a)	[6]	2.80 / 0.82	2.69 / 0.82	2.42 / 0.82	2.33 / 0.82	2.42 / 0.82
	[7]	✓	✓	✓	✓	✓
	[10]	1.14 / 1.23	0.93 / 1.23	0.74 / 1.23		
firewire_tree.7	[6]	✓ / 0.05	✓ / 0.06	✓ / 0.07	✓ / 0.08	✓ / 0.09
	[7]	✓	✓	✓	✓	✓
	[10]	1.48 / 1.54	1.44 / 1.79	0.85 / 1.99		
iprotocol.8	[6]	4.86 / 0.60	4.66 / 0.61	3.39 / 0.61	2.66 / 0.61	2.17 / 0.61
	[7]	3.74 / 2.71	3.77 / 2.72	2.98 / 2.73	2.54 / 2.74	2.16 / 2.75
	[10]	1.08 / 1.23	0.76 / 1.24	0.50 / 1.24		
leader_election.7	[6]	✓ / 0.11	✓ / 0.11	✓ / 0.12	✓ / 0.12	✓ / 0.12
	[7]	1.54 / 0.47	1.57 / 0.51	1.49 / 0.53	1.54 / 0.54	1.68 / 0.55
	[10]	1.11 / 1.36	0.88 / 1.45	0.65 / 1.52		
needham.7	[6]	=	✓ / 0.29	✓ / 0.29	✓ / 0.29	✓ / 0.29
	[7]	×	3.53 / 1.22	2.63 / 1.22	2.33 / 1.24	2.46 / 1.25
	[10]	×	0.96 / 1.28	0.59 / 1.28		
pgm_protocol.11	[6]	3.43 / 0.23	10.11 / 0.24	12.59 / 0.24	13.38 / 0.24	16.01 / 0.27
	[7]	✓	✓	✓	✓	✓
	[10]	1.18 / 1.32	1.01 / 1.39	1.21 / 1.39		
public_subscribe.5	[6]	5.24 / 0.75	4.75 / 0.75	3.69 / 0.75	2.59 / 0.75	2.71 / 0.75
	[7]	4.32 / 2.70	4.04 / 2.70	3.45 / 2.71	2.71 / 2.71	2.99 / 2.71
	[10]	1.10 / 1.23	0.76 / 1.23	0.57 / 1.23		
shieldrvt(2,b)	[6]	2.04 / 0.71	2.04 / 0.72	1.92 / 0.72	2.08 / 0.72	2.30 / 0.72
	[7]	✓	✓	✓	✓	✓
	[10]	1.08 / 1.23	0.72 / 1.24	0.71 / 1.24		
synapse.9	[6]	×	4.24 / 0.47	2.74 / 0.48	1.77 / 0.48	✓ / 0.48
	[7]	×	3.92 / 2.11	2.83 / 2.12	2.07 / 2.14	✓ / 2.16
	[10]	×	0.90 / 1.24	0.57 / 1.25		

Table 2 Relative performances of our algorithm with respect to other parallel algorithms

Comparing the multi-core algorithm [10] with our algorithm reveals that our algorithms consumes slightly more memory (because the data structure used to represent compressed state is slightly more complex in the distributed setting) while it usually scales better. A possible explanation is that threads in [10] can perform redundant work, exploring part of the state space already explored by other threads.

When it could be used, algorithm of [7] was usually at the same time faster and more memory efficient than ours. However, our algorithm still has the noteworthy advantage of working for all input models (provided their transition relation is deterministic) while, as said above, [7] assumes fixed size state vectors ; and it has a fairly predictable memory usage (since it compresses input states in a constant

number of bytes) while [7] is more sensitive in that perspective, see, *e.g.* model leader_election.7.

Overall, our results show that our algorithm scales better than the others up to 32 processes. It then reaches threshold where it performes 2–3 times slower than [6] and [7]. Once again, models with long and narrow state spaces (*e.g.* pgm_protocol.11 and, to a lesser extent, aslink(01,a) refute this pattern).

7 Conclusions and Future Work

We have presented a new state space exploration algorithm for cluster computing architectures, and used the Remote Direct Memory Access (RDMA) paradigm for low-latency networking and communication between computing elements. The main novelty of our algorithm lies in the concept of a state reconstruction forest. This can be seen as a generalisation of our earlier state reconstruction trees for compact state representation developed for multi-core architectures such that they now also apply to distributed computing architectures. As a further novel contribution, we introduced the concept of state stealing as a new approach to workload distribution. The use of state stealing was motivated by the need for locality in the state distribution which cannot be gauranteed by the use of a hash function for state distribution onto the processing elements. We have implemented our proposed algorithms and experimentally evaluated it on a large computing cluster using a variety of concurrent systems models in the DVE and PN modelling languages.

Our experimental results generally show that our algorithm performs well in terms of speed-up and workload distribution on state spaces with a large diameter. In contrast, our algorithm shows less impressive performance on the few instances with narrow state spaces where the parallelisation potential is smaller. Our experimental results show that 1) our algorithm generally achieves a balanced workload distribution among processing elements throughout state space exploration; and 2) that the number of stolen states which induces fully stored root states is kept low. Furthermore, our experimental evaluation have demonstrated that our algorithm is able to explore state spaces that cannot be handled with related parallel algorithms.

We have experimentally explored a large configuration space for our algorithm, including four state stealing strategies and two heuristics for triggering duplicate detection. Our results show that the duplicate detection heuristics has the most significant impact on performance, while the particular stealing strategy is of less importance. The latter implies that it is relevant as part of future work to possibly develop better stealing strategies to further improve the performance of our algorithm.

Verification of safety properties will be relatively straightforward for a forest of state reconstruction trees as we can easily reconstruct full state descriptors in parallel on the computing elements. A more challenging future line of research resulting from the present work is to investigate how CTL and LTL model checking algorithms in general can be developed that are able to operate on the distributed forest of reconstruction trees that we have introduced in this work. Another area of future work would be to evaluate the effect of relaxing the BFS level synchronisation barrier allowing processing elements to continue exploring states in subsequent levels. This may reduce waiting time for individual processing elements at the expense of possible undertaking redundant work.

References

[1] OpenSHMEM Application Programming Interface version 1.5. http://www.openshmem.org, Jun 2020.

[2] Z. Baranovà, J. Barnat, K. Kejstovà, T. Kučera, H. Lauko, J. Mrázek, and P. Ročkai a nd V. Štill. Model Checking of C and C++ with DIVINE 4. In *ATVA 2017*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017.

[3] J. Barnat, L. Brim, and J. Stríbrná. Distributed LTL Model-Checking in SPIN. In *SPIN'2001*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[4] L. Brim, I. Cerná, P. Krcál, and R. Pelánek. Distributed LTL Model Checking Based on Negative Cycle Detection. In *FSTTCS'2001*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.

[5] L. Brim, I. Cerná, P. Moravec, and J. Simsa. Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In *FMCAD'04*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.

tion

[6] C. Coti, S. Evangelista, and L. Petrucci. One-Sided Communications for More Efficient Parallel State Space Exploration over RDMA Clusters. In *Euro-Par 2018*, volume 11014 of *LNCS*, pages 432–446. Springer, 2018.

[7] Camille Coti, Sami Evangelista, and Laure Petrucci. State Compression Based on One-Sided Communications for Distributed Model Checking. In *ICECCS 2018*, pages 41–50. IEEE, 2018.

[8] F. Cappello et. al. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD*, pages 99–106. IEEE/ACM, 2005.

[9] S. Evangelista. High Level Petri Nets Analysis with Helena. In *ATPN'05*, volume 3536 of *LNCS*, pages 455–464. Springer, 2005.

[10] S. Evangelista, L. M. Kristensen, and L. Petrucci. Multi-threaded Explicit State Space Exploration with State Reconstruction. In *ATVA'2013*, volume 8172 of *LNCS*, pages 208–223. Springer, 2013.

[11] S. Evangelista, L. Petrucci, and L. M. Kristensen. Distributed explicit state space exploration with state reconstruction for RDMA networks. In *26th International Conference on Engineering of Complex Computer Systems, ICECCS 2022*, pages 107–116. IEEE, 2022.

[12] S. Evangelista and J.-F. Pradat-Peyre. Memory Efficient State Space Storage in Explicit Software Model Checking. In *SPIN'05*, volume 3639 of *LNCS*, pages 43–57. Springer, 2005.

[13] S. Evangelista, M. Westergaard, and L.M. Kristensen. The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. *ToPNoC*, 5800(3):189–215, 2009.

[14] H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN'2001*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001.

[15] G. J. Holzmann. State Compression in Spin: Recursive Indexing and Compression Training Runs. In *SPIN'1997*, 1997.

[16] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, G. Ciardo, S. Dal Zilio, P. G. Jensen, C. He, D. Le Botlan, S. Li, A. Miner, J. Srba, and Y. Thierry-Mieg. Complete Results for the 2020 Edition of the Model Checking Contest. http://mcc.lip6.fr/2020/results.php, June 2020.

[17] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'2007*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[18] U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *CAV'1997*, volume 1254 of *LNCS*, pages 256–278. Springer, 1997.

[19] M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The Comback Method - Extending Hash Compaction with Backtracking. In *ATPN'07*, volume 4546 of *LNCS*, pages 445–464. Springer, 2007.