Distributed Explicit State Space Exploration with State Reconstruction for RDMA Networks

Sami Evangelista and Laure Petrucci
LIPN, CNRS UMR 7030, Université Sorbonne Paris Nord,
99, av. J.-B. Clément, 93430 Villetaneuse, France {sami.evangelista,laure.petrucci}@lipn.univ-paris13.fr
Lars Michael Kristensen
Faculty of Engineering and Science, Western Norway University of Applied Sciences,
Inndalsveien 28, 5020 Bergen, Norway Lars.Michael.Kristensen@hvl.no

***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.**

I. 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 [1], [2]. 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], [4], [5].

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) which supports low-latency networking for cluster computing. Specifically, we consider RDMA as supported by the OpenSHMEM API [6]. In addition to being based on low-latency transfer of states, our proposed state space exploration algorithm provides 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, but 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. In addition, our algorithm relies on a concept of state stealing to distribute the state space exploration workload on the computing elements.

The rest of this paper is organised as follows. In section II 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 III illustrates the key elements in our proposed algorithm using a small example state space, and section IV provides a specification of the algorithm. In section V, we present results from an experimental evaluation of our approach. Finally, in section VI we sum up conclusions and discuss future work.

II. Background

To make the presentation of our algorithm independent of any particular modelling language for concurrent systems, we assume that $\mathcal{S}$ denotes the universe of syntactic system states and $\mathcal{E}$ denotes 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 in order to reconstruct a unique state from a sequence of events in the state reconstruction. Many modelling formalisms (including Petri nets) have deterministic transitions (events). As shown in [7], state reconstruction can be generalised to handle also non-deterministic events.

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. Successor states that have not been visited earlier are inserted into $\mathcal{R}$ and O.

A. State space exploration with 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 [8], [7], [9], [10] 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 which can also be represented using a hash table makes it possible to reconstruct full state descriptors when needed for comparison with newly generated states, *i.e.* to determine whether a duplicate state already included in the set of visited states has been generated.

Figure 1 illustrates state reconstruction. The top of fig. 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 fig. 1 is a linearised graphical representation of the hash table storing the set $\mathcal{R}$ of currently visited states. The dashed arcs represent references to parents in the 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.

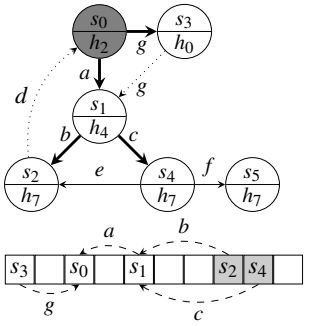


Fig. 1. State reconstruction example

When required, the full state descriptor for a state can be reconstructed by backtracking 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 fig. 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 both hashed to h_7. To decide whether s_2 is new, we reconstruct all nodes of the reconstruction tree that are also hashed to h_7 as these could potentially be the same state. These correspond to the grey cells of the hash table. For the first 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. For state s_5, we have to reconstruct s_2 using again $a.b$ as reconstructing sequence, and s_4 using the reconstructing sequence $a.c$. Since $succ(succ(s_0,a),b) \neq s_5$ and $succ(succ(s_0,a),c) \neq s_5$, then s_5 is new and is inserted in the hash table with a reference to its parent s_4 labelled with event f.

B. Multi-core state reconstruction with shared memory

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 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 running time of the algorithm.

C. RDMA architectures and the OpenSHMEM specification

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 another process's memory. Only a specific *public* memory area can be reached from other processes.

An attractive feature of one-sided communications is that only the process that initiates the communication needs to take active part in it. The process that owns the memory area it 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 NIC 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 [6]. 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 fig. 2.

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

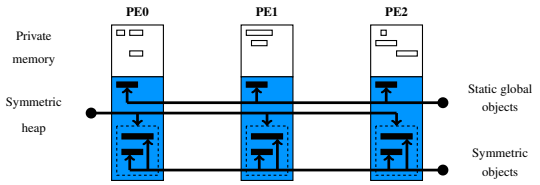


Fig. 2. OpenSHMEM memory model

specification also defines interfaces for atomic accesses (such as *compare-and-swap*), collective operations, or locks.

III. Illustration of our algorithm

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

A. Overview of the algorithm and its memory layout

The top of fig. 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 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 descriptor and execute a single event to retrieve the full state descriptor of s_8. It is straightforward to see that reconstructing sequences can be shortened compared to our multi-core algorithm [10] where all reconstructing sequences start at the initial state. Furthermore, the reconstruction starts from a single root in the forest. 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

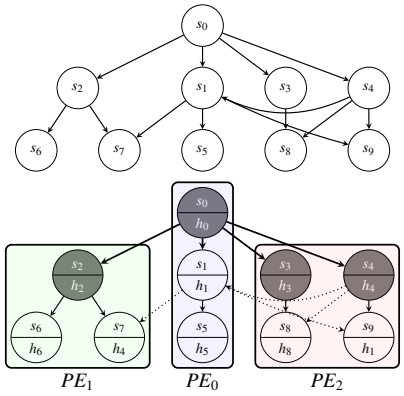


Fig. 3. Example state space (top) and a possible distribution on PEs (bottom)

[1] 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 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 puts 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 reconstructed it), the state is expanded. For instance, the distribution observed on fig. 3 may have been obtained after PE_1 and PE_2 stole s_2, and s_3 and s_4, respectively from PE_0.

B. Detailed execution of the algorithm

We now illustrate the steps performed by our algorithm to produce the distributed reconstruction tree in fig. 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 details 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 fig. 4.

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 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 fig. 5.

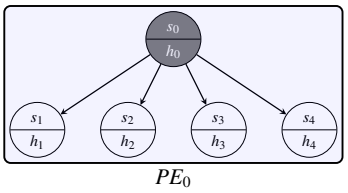


Fig. 4. After the expansion of s_0

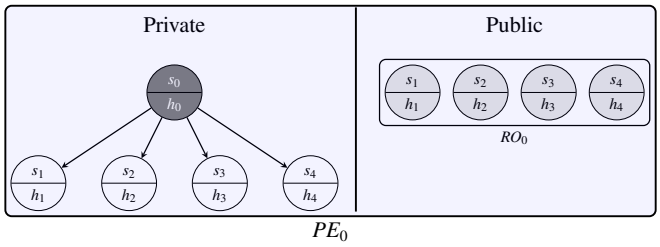


Fig. 5. After the reconstruction of $s_1, \ldots, s_4$ for their expansion

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 (see fig. 6 where dashed arrows represent network communications, as in subsequent figures). Note that the full state descriptors of candidates 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: 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. First, duplicate detection is triggered when the size of the candidate set has reached a predefined limit (set to bound the amount of memory used for the candidate set). In addition, an idle PE must periodically initiate duplicate detection to join its peers since this operation involves all PEs. Note that, as in [10], the passage from one step to another is controlled by the use of synchronisation barriers.

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 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.

First, 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 fig. 7, step 1.

After a synchronisation barrier, each PE retrieves the hash values of its peers. It then knows which 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 set. PE_0 would for instance reconstruct s_1 and put it in its public space, so that PE_2 can detect that 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 idea of *detection proxy* and the use of a mapping $dp : \mathcal{H} \to \mathcal{P}$ ($\mathcal{H}$ being the set of hash values, and $\mathcal{P} = \{0, \ldots |PE - 1|\}$). Basically, 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_p^i 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 also 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_p^i 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 observe the configuration of fig. 7, step 2.

Duplicate detection can then be performed locally. Each PE_i merges candidate states it received (*i.e.* blocks DC_i^j) in sets N_i^j from which it then removes reconstructed states it received (*i.e.* blocks R_i^j). Formally, sets N_i^j are defined as follows: $\forall i \in \mathcal{P}, \cup_{j \in \mathcal{P}} N_i^j = \cup_{j \in \mathcal{P}} DC_i^j \setminus \cup_{j \in \mathcal{P}} R_i^j$; $\forall i, j, k \in \mathcal{P}, j \neq k \Rightarrow N_i^j \cap N_i^k = \emptyset$; and $\forall i, j \in \mathcal{P}, N_i^j \subseteq DC_i^j$. In case a state s is present in both DC_i^j and DC_i^k (with $j \neq k$) — as, for instance, s_7 present in DC_0^0 and DC_0^1 — 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 will discuss 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 fig. 7, 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

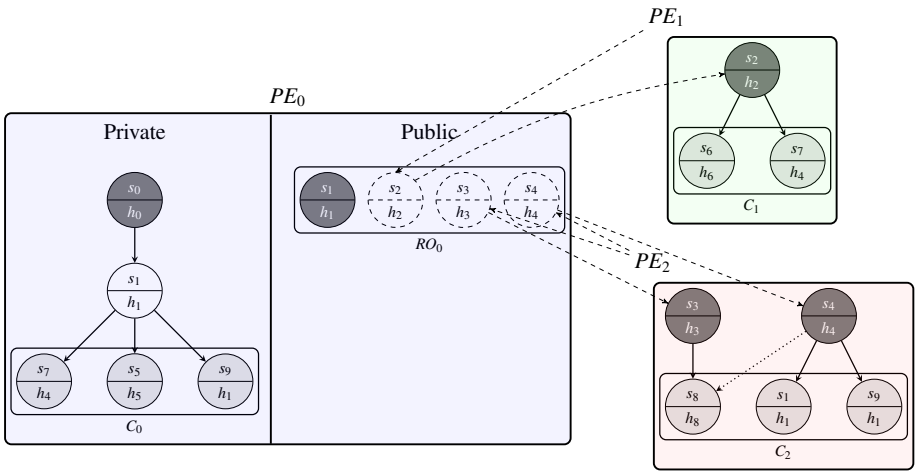


Fig. 6. After stealing and expansion of open states

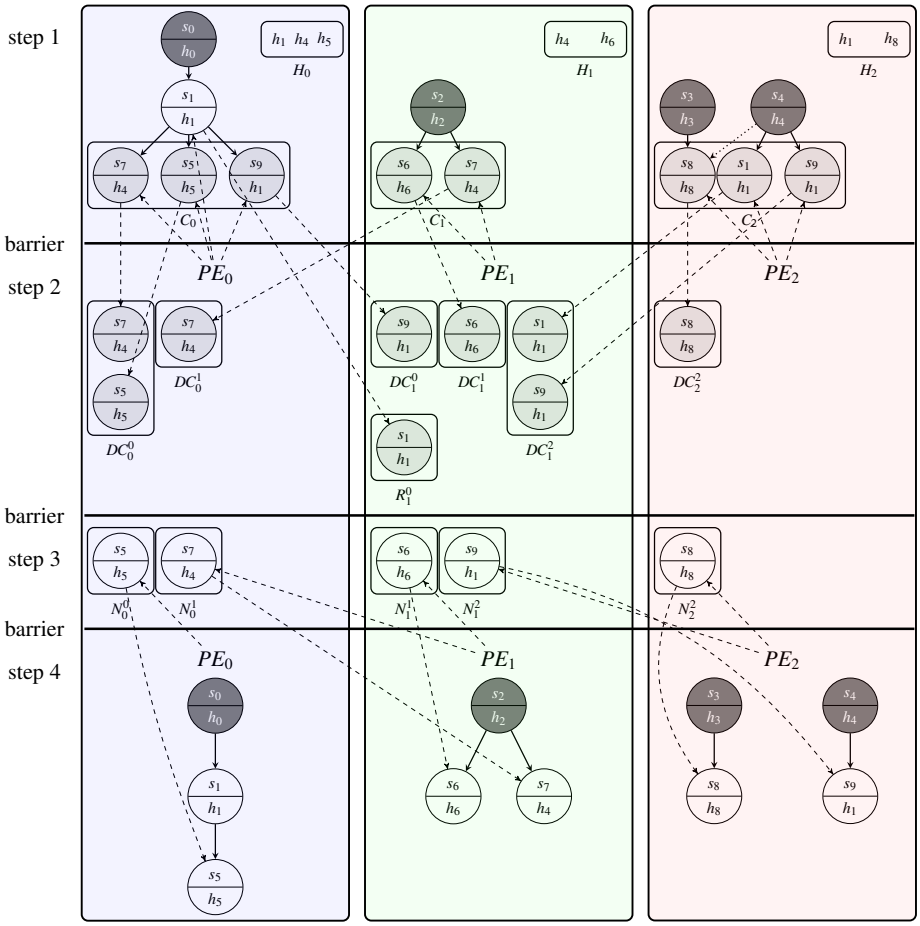


Fig. 7. The duplicate detection process. Step 1: publication of hash values of candidate states. Step 2: writing of candidates and reconstructed states in their detection proxy's public space. Step 3: publication of new states obtained after candidates merging and removal of reconstructed states. Step 4: PEs remotely read new states they are assigned to and insert these in their trees.

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 fig. 7, 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 proceed. As we will see in the next section this can be detected during duplicate detection.

IV. Specification of our algorithm

We give in this section an algorithmic specification of the core operations performed by our algorithm. The pseudo-code for our algorithm, described below, is given in Figure 8.

For the sake of clarity of the presentation we first detail some global data structures. These are private (*i.e.* in the private memory space of the PE) 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. 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. Synchronisation variables are also required to check for global termination (`TERM`) and level termination ($\texttt{LVL\_TERM}_{\texttt{ME}}$ and `LVL_TERM_ALL`).

In the main procedure (l. 17), 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 BFS level. The PE first reconstructs the open states it owns, then explores these before trying to explore open states owned by its peers. To control 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` is set to **true**. If open states of the current level generate new states, `TERM` may be reset to **false** during duplicate detection (l. 73).

Procedure `recons_open` (l. 26) recursively reconstructs the PE's open states and puts their full state descriptors in the set RO_{ME}. This set is bounded: whenever it fills up, the PE starts exploring its open states. 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. 79).

```
1  PES: constant set of int = {0, ..., number of PEs - 1}
2  ME: constant int = PE's identifier
3  C: map of (state → (int * event)) = empty
4  ROOTS: list of (int * state) = empty
5  TREE: map of (int → list(int * event)) = empty
6  RO_ME: public set of (int * state) = empty
7  H_ME: public set of int
8  DC^0_ME, ..., DC^{|PES|-1}_ME: public set of state
9  R^0_ME, ..., R^{|PES|-1}_ME: public set of state
10 N^0_ME, ..., N^{|PES|-1}_ME: public set of state
11
12 /* synchronisation variables */
13 TERM: bool = false
14 LVL_TERM_ME: public bool
15 LVL_TERM_ALL: bool
16
17 procedure main():
18   if ME == 0: add (0, s_0) to ROOTS, (0 → empty) to TREE
19   while not TERM:
20     LVL_TERM_ME = false
21     TERM = true
22     for (id, root) in ROOTS: recons_open(id, root)
23     explore_my_open()
24     explore_peers_open()
25
26 procedure recons_open(id: int, s: state):
27   if is_open(id):
28     add (id, s) to RO_ME
29     if RO_ME is full: explore_my_open()
30   for (idc, e) in TREE(id): recons_open(idc, succ(s, e))
31
32 procedure explore_my_open():
33   while RO_ME is not empty:
34     stolen = steal_from(ME)
35     for (id, s) in stolen: explore_open(id, s)
36
37 procedure explore_open(id: int, s: state):
38   for e in enab(s):
39     s' = succ(s, e)
40     if s' not in C: add (s' → (id, e)) to C
41     if C is full: dd()
42
43 procedure explore_peers_open():
44   LVL_TERM_ME = true
45   LVL_TERM_ALL = false
46   while not LVL_TERM_ALL:
47     stolen = steal_from(choose_victim_pe())
48     for (_, s) in stolen:
49       i = new_id()
50       add (i, s) to ROOTS, (i → empty) to TREE
51       explore_open(i, s)
52     if do_dd(): dd()
53
54 procedure dd():
55   /* step 1 of fig. 7 */
56   for pe in PES: DC^pe_ME = empty ; R^pe_ME = empty
57   H_ME = { hash(c) for (c → _) in C }
58   wait_for_peers()
59   /* check BFS level termination */
60   LVL_TERM_ALL = ∀ pe in PES: LVL_TERM_pe
61   /* step 2 of fig. 7 */
62   for (id, root) in ROOTS: recons_dd(id, root)
63   for (c → _) in C: add c to DC^ME_dp(hash(c))
64   wait_for_peers()
65   /* step 3 of fig. 7 */
66   for pe in PES: N^pe_ME = empty
67   for pe in shuffle(PES), c in DC^pe_ME: # merge candidates
68     if ∄ pe' st c in N^pe'_ME: add c to N^pe_ME
69   for pe in PES, r in R^pe_ME: # delete duplicates
70     if ∃ pe' st r in N^pe'_ME: del r from N^pe'_ME
71   wait_for_peers()
72   /* check global termination */
73   TERM = TERM and (∀ pe, pe' in PES: N^pe'_pe is empty)
74   /* step 4 of fig. 7 */
75   for (c → (id, e)) in C st c in N^ME_dp(hash(c)): # new states
76     nid = new_id()
77     add (nid → empty) to TREE and (nid, e) to TREE(id)
78
79 procedure recons_dd(id: int, s: state):
80   if ∃ pe st hash(s) in H_pe: add s to R^ME_dp(hash(s))
81   for (idc, e) in TREE(id): recons_dd(idc, succ(s, e))
```

Fig. 8. Pseudo-code of our algorithm

When exploring open states present in its own RO_{ME} set (procedure `explore_my_open` at l. 32), 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. 37) simply consists of putting all its successors in the candidate set — an operation that may trigger duplicate detection if the candidate set fills up.

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. 43). The PE first chooses another *victim* PE from which it tries to steal some state(s). 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. 52) 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 `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.

Finally, the duplicate detection procedure (procedure `dd` at l. 54) follows the different steps introduced in our example (see fig. 7). We draw the reader's attention to the operation that consists of merging candidates sent by peers (l. 67). 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, it allows for an easy detection of termination of both the current BFS level and the global search. First these termination checks (at l. 60 and 73) must be placed after barriers to avoid any race condition. The current BFS level may be considered as terminated (l. 60) if all PEs have executed the statement at l. 44. 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. Finally, global termination is detected if no candidate was identified as new (l. 73) during the exploration of the current BFS level.

V. Experimental Evaluation

We have implemented our algorithm in the Helena tool [11] and conducted experiments with models of concurrent systems specified in the DVE input language of the DiVinE model checker [12]. Models we have experimented with are listed in Table I, 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). The content of the last colum will be described herafter. All models come from the BEEM database [13].

A. Cluster computing infrastructure

Our experiments were carried out using the Grid'5000 [14] testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organisations (see https://www.grid5000.fr). We have conducted the experiments on the `grimoire` cluster of the Nancy site which features 8 nodes with 2 Intel Xeon E5-2630 v3 CPUs and 8 cores per CPU, connected through a 56 Gbps Infiniband network. We used the OpenSHMEM implementation provided by OpenMPI 2.0.1.

B. Algorithm configuration and implementation choices

In the previous section we deliberately omitted some implementation details, especially regarding the state stealing mechanism that we now address.

Stealing individual states appeared to be too costly communication-wise and unsuitable to achieve a good load balance. We instead implemented the stealing algorithm in such a way that blocks of states of size B can be stolen and we set $B = 100$ in our experiments. Some initial experiments showed that this parameter did not have a significant impact unless set to a too small or too large value (*i.e.* < 10 or $> 1,000$). Finally, our state stealing implementation relies on the atomic compare-and-swap routines provided by OpenSHMEM that are used to prevent race conditions between processes.

For the choice of the victim PE to steal from (l. 47 of fig. 8), we implemented the following: PE i first tries to steal from its right neighbour (*i.e.* PE $(i+1) \mod |PES|$) and keeps stealing from this one as long as its attempts are successful. It then repeats this process on following PEs: $(i+2) \mod |PES|$, $(i+3) \mod |PES|$, ... We have experimented with other strategies (*e.g.* always steal from the same PE) but found out that a ring-based strategy performs better overall.

The decision of initiating duplicate detection is determined by procedure `do_dd` (l. 52 of fig. 8) that may not indefinitely postpone the operation. We have implemented this decision in such a way that `do_dd` returns true only after $|PES| - 1$ successive failed attempts. Finally, we set the maximal sizes of candidate sets and sets of reconstructed open states to $100,000$.

C. Experimental results

We have launched a full state space exploration for 8, 16, 32, 48, ... 128 processes, *i.e.* for 1, 2, 4, 6, ... 16 processes per node. We were also able to perform a sequential execution for 6 of our 8 models. Each experiment was run 5 times and

Model	States	Arcs	Height	State size	Stolen states
collision.5	431 M	1,644 M	182	52	1.72%
firewire_tree.7	121 M	778 M	343	647	2.93%
iprotocol.8	447 M	1,501 M	353	45	1.78%
leader_election.7	235 M	1,712 M	248	281	3.52%
needham.7	806 M	3,546 M	44	98	3.87%
pgm_protocol.11	499 M	1,207 M	2,238	129	3.88%
public_subscribe.5	1,153 M	5,447 M	170	36	1.61%
synapse.9	1,675 M	3,291 M	88	58	3.28%

TABLE I
DVE models used in the experiments

plots present the average and standard deviation of the set of measurements. We checked that all runs on the same model resulted in the same number of states.

Execution times have been plotted in fig. 9. We observe that the execution times generally follow the same pattern. It decreases as more processes participate, but reaches a threshold with approximatively hundred of processes and then stagnates for largest configurations. Only on models needham.7 and public_subscribe.5 do we observe a continued decrease of the runtime. Model pgm_protocol.11 also stands out in that its best execution time is reached with 80 processes and more importantly, it grows significantly beyond that number.

To have a better understanding of the performances of our algorithm, we also investigated how the stealing mechanism distributes the workload upon processes. We have plotted in fig. 10 the workload of a single run with 128 processes for 4 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. It can be seen that for the models in fig. 10(a), (b), and (d) the discrepency between workload is well-controlled, but in fig. 10(c) there is some continued growth until the very end of the state space exploration. A possible cause for this may lie in the fact that the state space of the pgm_protocol.11 model is a long (and hence narrow) state space compared to the state space of the other models (cf. Table I).

To provide details on the state distribution we have plotted the number of states stored for each of the 128 processes in fig. 11. Here fig. 11(c) confirms the uneven distributed of states which in turn results in the uneven distribution of workload. Still, it can be seen that the majority of the processes have similar state distributions.

Last we observe that the proportion of stolen states (last column of Table I, value obtained by computing the average over the 5 runs with 128 processes and reported to the state space size) remains low. This is a crucial aspect of our algorithm since stolen states are fully stored in memory. Therefore the memory usage of our algorithm remains close to the one of our multi-core algorithm [10]. Moreover, although we cannot provide these data due to lack of space, this ratio does not seem to be increase with the number of processes.

VI. Conclusions and Future Work

We have presented a new state space exploration algorithm for cluster computing architectures that rely on the Remote Direct Memory Access (RDMA) paradigm for low-latency networking and communication between computing elements. The novelty of our algorithm lies in the concept of state reconstruction forests which can be seen as a generalisation of our earlier state reconstruction trees developed for multi-core architectures. As a further contribution, we introduce the concept of state stealing as a new approach to workload distribution. We have implemented our proposed algorithms and experimentally evaluated it on a large computing cluster using a variety of concurrent systems models.

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 narrow state spaces where the parallelisation potential is smaller. For future work, we have also observed instances of models where our heuristic for ensuring locality, *i.e.* to avoid many small reconstruction trees in the forest, needs further improvement. Experimentally exploring larger parts of the parameter space of our algorithms is also a task for future work.

We conjecture that the workload can be further balanced by using informed state stealing strategies that *e.g.* keep track of the number of states available for stealing on each PE. An even distribution of states among PEs seems the key for our algorithm to scale on larger networks.

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.

Acknowledgements: The work of L.M. Kristensen was partially funded by the SFI Smart Ocean NFR Project 309612/F40.

References

[1] U. Stern and D. L. Dill, "Parallelizing the Murphi Verifier," in *CAV'1997*, ser. LNCS, vol. 1254. Springer, 1997, pp. 256–278.

[2] H. Garavel, R. Mateescu, and I. Smarandache, "Parallel State Space Construction for Model-Checking," in *SPIN'2001*, ser. LNCS, vol. 2057. Springer, 2001, pp. 217–234.

[3] J. Barnat, L. Brim, and J. Stríbrná, "Distributed LTL Model-Checking in SPIN," in *SPIN'2001*, ser. LNCS, vol. 2057. Springer, 2001, pp. 200–216.

[4] L. Brim, I. Cerná, P. Krcál, and R. Pelánek, "Distributed LTL Model Checking Based on Negative Cycle Detection," in *FSTTCS'2001*, ser. LNCS, vol. 2245. Springer, 2001, pp. 96–107.

[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*, ser. LNCS, vol. 3312. Springer, 2004, pp. 352–366.

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

[7] M. Westergaard, L. Kristensen, G. Brodal, and L. Arge, "The Comback Method - Extending Hash Compaction with Backtracking," in *ATPN'07*, ser. LNCS, vol. 4546. Springer, 2007, pp. 445–464.

[8] S. Evangelista and J.-F. Pradat-Peyre, "Memory Efficient State Space Storage in Explicit Software Model Checking," in *SPIN'05*, ser. LNCS, vol. 3639. Springer, 2005, pp. 43–57.

[9] S. Evangelista, M. Westergaard, and L. Kristensen, "The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection," *ToPNoC*, vol. 5800, no. 3, pp. 189–215, 2009.

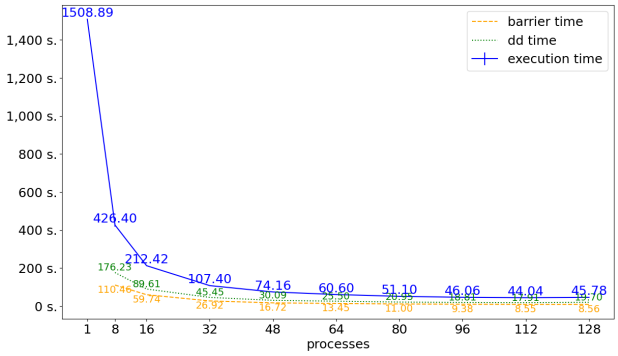
[10] S. Evangelista, L. M. Kristensen, and L. Petrucci, "Multi-threaded Explicit State Space Exploration with State Reconstruction," in *ATVA'2013*, ser. LNCS, vol. 8172. Springer, 2013, pp. 208–223.

[11] S. Evangelista, "High Level Petri Nets Analysis with Helena," in *ATPN'05*, ser. LNCS, vol. 3536. Springer, 2005, pp. 455–464.

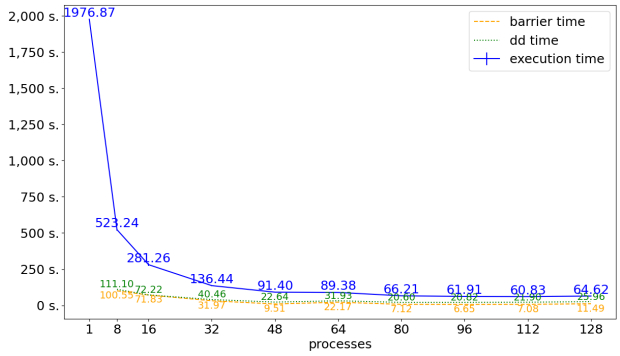
[12] Z. Baranovà, J. Barnat, K. Kejstovà, T. Kučera, H. Lauko, J. Mrázek, and P. R. a nd V. Štill, "Model Checking of C and C++ with DIVINE 4," in *ATVA 2017*, ser. LNCS, vol. 10482. Springer, 2017, pp. 201–207.

[13] R. Pelánek, "BEEM: Benchmarks for Explicit Model Checkers," in *SPIN'2007*, ser. LNCS, vol. 4595. Springer, 2007, pp. 263–267.

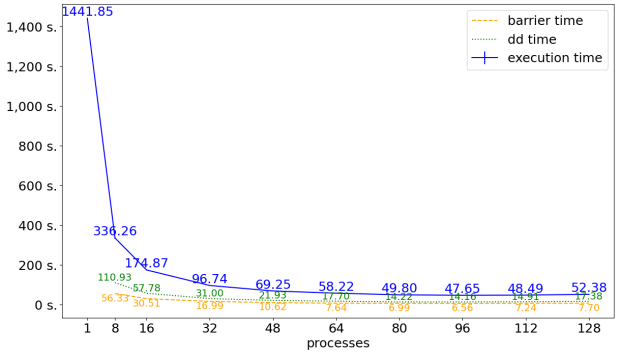
[14] F. C. et. al., "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD*. IEEE/ACM, 2005, pp. 99–106.



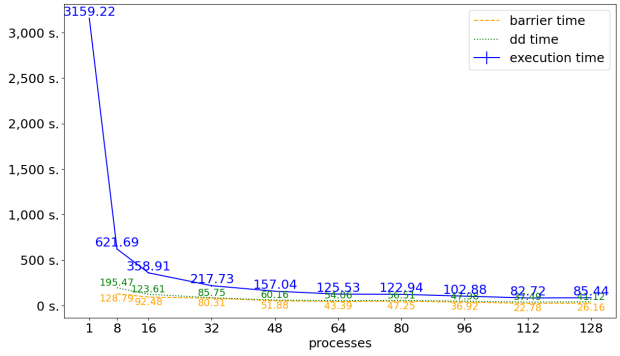
(a) collision.5



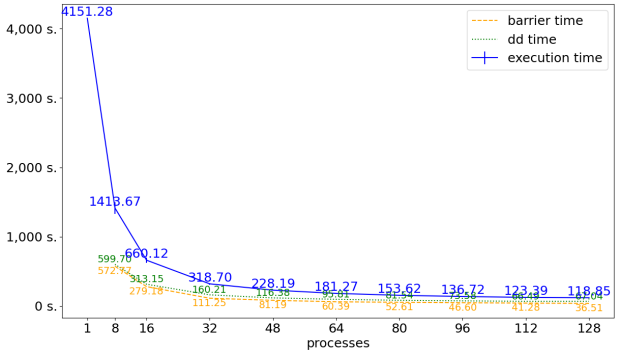
(b) firewire_tree.7



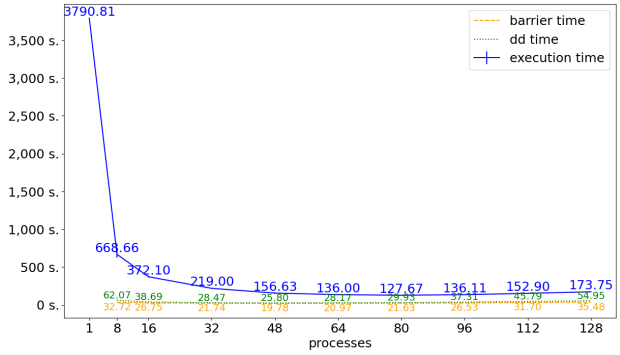
(c) iprotocol.8



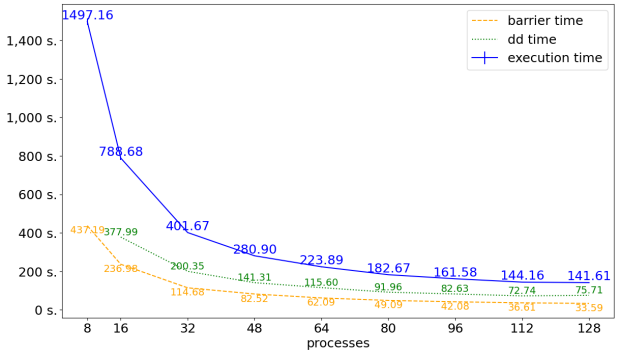
(d) leader_election.7



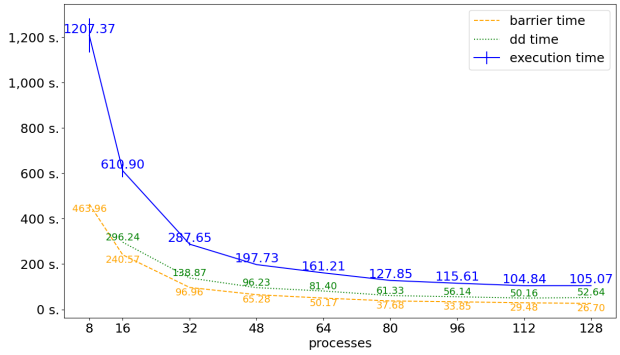
(e) needham.7



(f) pgm_protocol.11

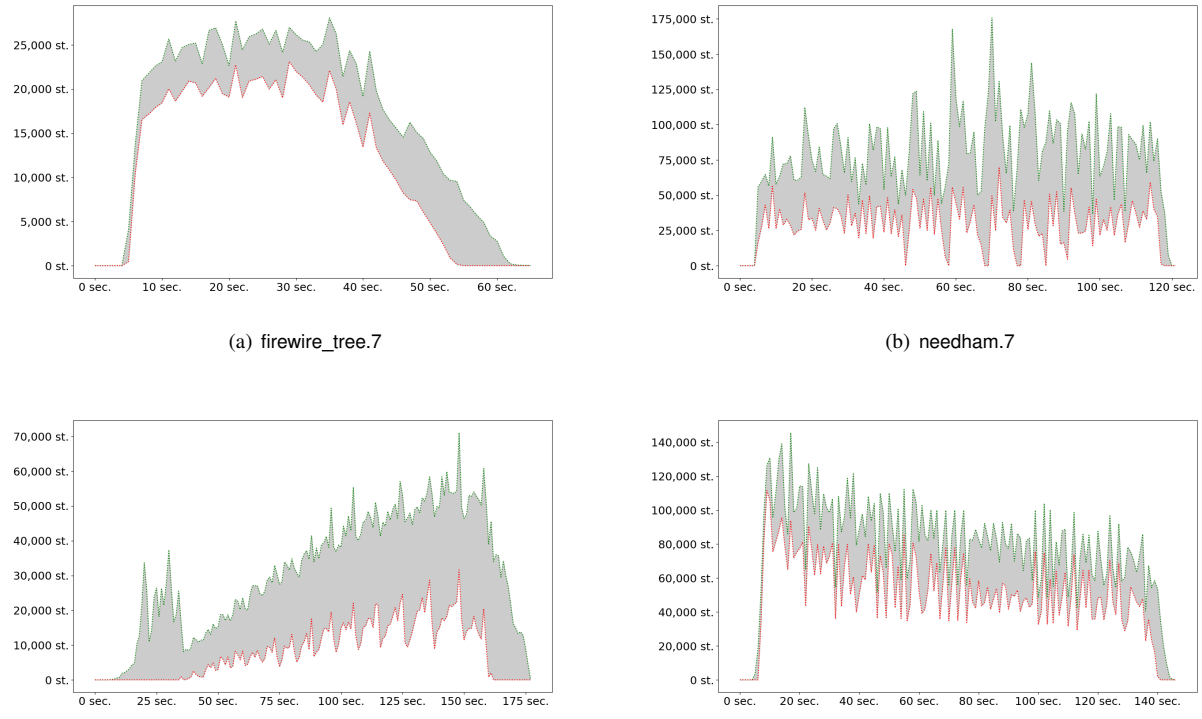


(g) public_subscribe.5



(h) synapse.9

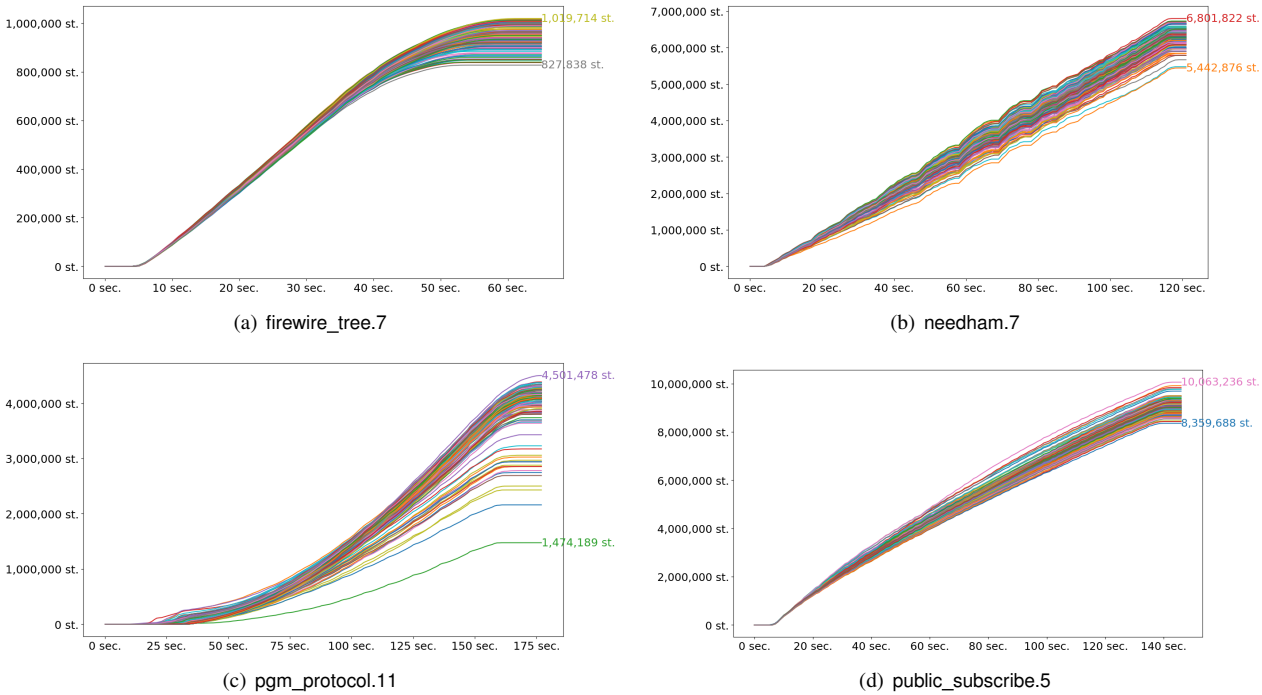
Fig. 9. Execution times



(a) firewire_tree.7 (b) needham.7

(c) pgm_protocol.11 (d) public_subscribe.5

Fig. 10. Difference between the least and most loaded process at each second for a single 128-process run



(a) firewire_tree.7 (b) needham.7

(c) pgm_protocol.11 (d) public_subscribe.5

Fig. 11. State distribution at each second for a single 128-process run