One-Sided Communications for more Efficient Parallel State Space Exploration over RDMA Clusters

Camille Coti and Sami Evangelista and Laure Petrucci

LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité,
99, av. J.-B. Clément, 93430 Villetaneuse, France

Abstract. This paper investigates the use of one-sided communications in the context of state space exploration. This operation is often the core component of model checking tools that explores a system state space to look for behaviours deviating from its specification. It basically consists in the exploration of a (usually huge) directed graph whose nodes and edges represent respectively system states and system changes. We revisit the state of the art distributed algorithm and adapt it to RDMA clusters with an implementation over the OpenSHMEM library and report on preliminary experiments conducted on the Grid'5000 cluster. This asynchronous approach thus reduces the significant communication costs induced by process synchronisation in two-sided communications.

1 Introduction

Model checking [2] based on state space exploration is a prominent approach used to prove that finite-state systems match behavioural specifications. In its most basic form, it is based on a systematic exhaustive exploration of all system states (the state space) in the search for illegal behaviours violating the specification. This state space can be viewed as a graph capturing the behaviour of the system. Its nodes represent system states (e.g., program counters and content of variables and channels in the case of a distributed system) and its edges represent system changes (e.g., variable assignments or synchronisations). Despite the simplicity of this technique, its practical application is subject to the well-known state explosion problem [17]: the state space may be far too large to be explored in reasonable time or to fit within the available memory. Distributed verification thus arose [16] as a natural means to push the limits of model checking: distributing state space search allows to benefit from the aggregate memory and computational power of a machine network and hence to analyse larger models and/or reduce exploration times.

Although distributed algorithms have been proposed for various classes of properties, e.g., LTL (Linear-time Temporal Logic) properties [4,18], we focus in this work on the verification of safety properties, i.e., system invariants that can be verified using a simple enumeration of system states. Many interesting properties can however be expressed as system invariants.

An important characteristic of graph-based algorithms used in verification is that the graph is not known *a priori*. The model checker is instead provided with an initial state describing the system's initial configuration and a successor function that, from one

state, can generate its successors. Many verification algorithms are built upon this state space construction step. Therefore, the workload cannot be divided using traditional, static domain decomposition techniques. Moreover, the granularity of this step does not make it a good candidate for chunk-based approaches such as master-worker patterns.

The state-of-the-art algorithm that can be used for the verification of safety properties [16] distributes the search by partioning the state space among participating processes. A partition function maps state vectors (i.e., bit strings encoding states) to processes. Each process is then responsible of any state that is assigned to it: it stores it in a local state table, generates its successors and sends them to their owners that will later process these states in the same way.

To the best of our knowledge, all implementations of [16] are based on two-sided communications. In this distributed programming paradigm, two processes have to synchronise to exchange data. This means that, from a development perspective, the programmer has to explicitly mention in the code where processes have to wait for incoming data by invoking a receive statement. This constraint adds points of synchronisation in the code that makes each communication a big concern in terms of performance.

In this paper we redesign the algorithm of [16] to adapt it to one-sided communications. In such a model, a process can directly access remote memory segments of another process without the latter being aware of this access. The one-sided communication model is particularly interesting here, because when a process needs data located in another process's memory, the target process does not need to be aware that the source process needs it: the source process can get the remote data on its own.

In the more general context of model checking, [14] is the only work we are aware of, that proposes a distributed algorithm for Remote Direct Memory Access (RDMA) clusters. It can be used in the context of symbolic model checking, a different algorithmic approach than ours.

After an overview of the verification process by state space exploration considered in this work in Section 2 and a quick presentation of the communication and distributed memory model in Section 3, our new algorithm is described in Section 4. We present experiments conducted with this new implementation and compare it to the well known distributed model checker DiVinE [3] in Section 5.

2 Background

Model checking by state space exploration explores all the possible states of the system until it finds a counterexample of the property to be verified. If it can explore all possible states without finding a counterexample, it concludes that the property is always verified by the system. Therefore, it is of major importance to use an efficient algorithm for this state space exploration.

In this paper we assume a universe of system states S, an initial state $s_0 \in S$ and a mapping $succ : S \to 2^S$, that, from one state s, gives its set of successors. We want to explore the state space induced by these parameters, i.e., the smallest set $R \subseteq S$ of reachable states defined inductively as : $s_0 \in R \wedge (s \in R \Rightarrow succ(s) \in R)$.

Algorithm 1 is a sequential state space exploration algorithm usable for invariant checking. It operates on a queue Q of unexplored states and incrementally builds the

Algorithm 1 Sequential state space exploration

```
1: procedure exploreSequential is
2:     Q.init(s_0); R.init(s_0)
3:     while ¬Q.isEmpty() do
4:         s := Q.remove()
5:         for s' ∈ succ(s) do
6:             if ¬s'.checkInvariant() then
7:                 halt and report error
8:             else if ¬R.isIn(s') then
9:                 Q.insert(s'); R.insert(s')
```

reachability set R. Both initially contain the initial state. States are taken from Q (l. 4), their successors generated and put in R and Q (if not seen before) to be later processed (loop at ll. 5–9). The algorithm terminates when an erroneous state is found (ll. 6–7) or when the queue is empty, which is guaranteed to happen for finite-state systems.

The distributed algorithm of [16] that represents the core component of many distributed algorithms is given in Algo. 2. P exploration processes are used (l. 2). Each process i owns a local portion of the queue and the reachable states. The state space is partitioned among processes using a state hash function. Each exploration process basically acts as the sequential algorithm presented above except that when a state s' is reached, the process checks if it is the owner of this state (condition at l. 8). In that case, it is processed as in the sequential scenario. Otherwise it is sent to its owner and discarded by the current process. Similarly, only the owner of the initial state puts it in its local data structures (ll. 13–14). Processes also have to check for incoming messages (ll. 16–19). A state received is handled as would be any other new state owned by the process (i.e., ll. 18–19 and ll. 10–11 match).

Termination detection (not shown in the algorithm) is triggered by a unique process (e.g., node 0) when this one has been idle (i.e., it does not receive any messages and its queue is empty) for some amount of time. It then asks it peers if they are in the same situation and if all channels are empty (check made by counting messages sent and received) before notifying termination to other nodes if both conditions are met.

Algorithm 2 Distributed state space exploration algorithm usable for invariant checking

```
 1: procedure exploreDistributed() is
 2:     launch explore_0 || ... || explore_{P-1}
 3: procedure processQueue_i() is
 4:     s := Q.remove()
 5:     for s' ∈ succ(s) do
 6:         if ¬s'.checkInvariant() then
 7:             halt and report error
 8:         else if s'.hash()%P ≠ i then
 9:             s'.sendTo(s'.hash()%P)
10:         else if ¬R.isIn(s') then
11:             Q.insert(s'); R.insert(s')
12: procedure explore_i() is
13:     if s_0.hash()%P = i then
14:         Q.insert(s_0); R.insert(s_0)
15:     while ¬termination() do
16:         if stateReceived() then
17:             s := receiveState()
18:             if ¬R.isIn(s) then
19:                 Q.insert(s); R.insert(s)
20:         if ¬Q.isEmpty() then
21:             processQueue_i()
```

3 RDMA architectures and the OpenSHMEM specification

This section gives a brief presentation of the one-sided communication model we are using in this paper, and its implementation in the OpenSHMEM shared heap and communication interface.

3.1 RDMA and one-sided communications

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()`. A process can read (`get()`) and write (`put()`) in another process's memory. In practice, not all the process's memory can be reached from other processes, but only a specific, *public* area.

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. This fact makes one-sided communication more tricky to use in parallel, distributed programs compared to two-sided communications, and more prone to race conditions.

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.

3.2 The OpenSHMEM communication and memory model

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 it exhibits a public heap. One particularity of OpenSHMEM is that this heap is *symmetric*: every PE has a shared heap of the same size and that contains the same allocated objects and static global objects (Figure 1).

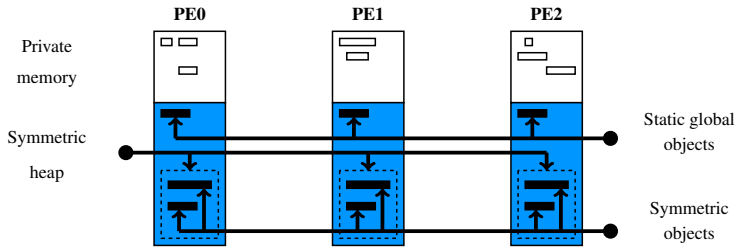


Fig. 1. OpenSHMEM memory model.

Symmetry is maintained between shared heaps through the use of dedicated memory management routines: `shmem_malloc()`, `shmem_realloc()`, `shmem_align()` and `shmem_free()` (or `shmalloc()`, `shrealloc()`, `shmemalign()` and `shfree()` until OpenSHMEM v1.2). 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 [8]. Besides, 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 fetch-and-add), collective operations, locks and synchronisation and ordering routines.

4 Distributed reachability analysis with one-sided communications

We now propose a distributed algorithm (see Algorithm 3) for state space exploration on RDMA clusters using one-sided communications. It assumes the following two procedures are provided by the communication layer:

- $getMem(i,o)$ returns the shared object o stored on PE i
- $putMem(i,o,data)$ stores $data$ in the shared object o of PE i.

These correspond in the OpenSHMEM API to `shmem_getmem` and `shmem_putmem`.

Our algorithm acts basically as the distributed algorithm presented in Sect. 2. PEs exchange states on the basis of a state space partition induced by the state hash function. These states are communicated through the shared memory space using remote put operations. Hence, we focus next on the specificities of our implementation.

A PE shares two objects with its peers: *buf*, an array of buffers containing states sent by other PEs ; and *free*, a boolean array used to prevent the PE from erasing states it has previously put in the *buf* object of another PE and which have not been consumed yet. Basically, it is an invariant property of the algorithm that $getMem(i,free[j]) = true$ implies that the buffer $getMem(j,buf[i])$ does not contain states put by PE i for PE j but not consumed by PE j yet.

Besides its private queue Q of states to process and its reachable states R, a PE also owns an array *sbuf* containing buffers of states to be sent to their owner and grouped together to avoid sending individual states.

In the main procedure (ll. 3–12), each PE periodically processes incoming states (ll. 7–8). This is done (ll. 36–43) by inspecting the *buf* array of its local shared memory space. All the states put by remote PEs are put in the private queue and in the reachable states set (ll. 42–43). Each time a buffer has been retrieved, the remote PE that sent these states is notified via the *free* array (l. 40) located in the shared memory of this remote PE. The implementation of *checkForIncomingStates* used to decide if input buffers must be inspected is discussed in Sect 5. As soon as its queue empties the process also has to flush its output buffers containing states destinated to remote PEs (ll. 9–10). This is mandatory to avoid a premature termination caused by all PEs being idle and ready to terminate whereas buffers still contain potentially new states to be processed. This is the purpose of procedures $flushOutBuffer_i$ described below and $flushOutBuffers_i$ (ll. 23–26) that simply flushes all non empty buffers.

Any state s belonging to another PE is processed by function $processOutState_i$ (ll. 13–16). The PE puts s in a private buffer containing states to be sent to their owner, i.e., the PE $j = s.hash\%PES$. This private buffer is $sbuf[j]$. If it becomes full, it has to be put in the shared space memory of PE j using procedure $flushOutBuffer_i$ (ll. 17–22). In this one, the PE first periodically polls its local shared memory to check whether the states it previously put in the shared memory of PE j have been consumed by this one. The condition at l. 18 evaluates to *false* as soon as PE j has completed its put statement at l. 40. Hence, we see that the purpose of the *free* shared array is to avoid communications when checking whether or not the states can be remotely put in the shared memory segment of its owner. Also note that, during polling, the process also has to process incoming states it may have received (l. 19). This is mandatory as, otherwise, a deadlock could occur. This would be the case, for instance, with two PEs, each PE waiting for the other to free its buffer, i.e., completing the put operation at l. 40, whereas it is blocked at ll. 18–19.

For termination detection (not shown in the algorithm to avoid overloading it) we adapted the algorithm of [16]. As soon as PE 0 has been idle for 100 ms it sends a token to PE 1. A PE receiving the token passes it to the next PE if it is idle, or destroys it otherwise. If PE 0 receives back the token, it asks all other PEs to participate to termination detection: a synchronisation barrier occurs, then all PEs process incoming states (if any) and publish in the shared memory their status (idle, i.e., without any state to process, or working). Termination occurs when all the processes are idle. The circulation of the token can be more efficient than a ring, for instance using Bruck's algorithm [6], which has a logarithmic number of steps. However, we have measured in the experimental evaluation of this algorithm that the termination phase is not significantly long with respect to the overall execution time. A more scalable algorithm can be used if this algorithm is meant to be executed on a large scale system.

Sketch of proof that all states in a buffer are indeed read. Let us assume a PE i has written states in the buffer of PE j. PE j can read them as long as they are not superseeded by other values, which could only be the result of PE i flushing a new version of the buffer. This operation is performed by $flushOutBuffer_i(j)$. Before PE i actually flushes the buffer at l. 21, it waits for $free[j]$ to become true (l. 18). This boolean value can only be set to true at l. 40 by PE j. This occurs after PE j reads the contents at l. 38. Note that PE i is also the only PE to set this variable to false, at l. 20, before writing the contents.

Therefore, it is not possible to write twice to a distant buffer without the corresponding process reading in between.

Sketch of proof that all states are processed. A state is created as the initial state s_0 at ll. 4–5, or as the successor of a state being processed. In this case, if it belongs to the same PE, it is inserted in the local queue at l. 35. Otherwise, *processOutState* is called, and the state is added to its PE buffer at l. 14, to be sent later. The buffer is sent when it is full (l. 16), or when the current PE has an empty queue (l. 10). In both cases, *flushOutBuffer* is eventually called, which puts the buffer in its associated PE memory. A PE checks its incoming states regularly, at l. 7 and l. 19. In both cases, the states read from the buffers are inserted in the local queue at l. 43.

Thus, all states are explored either processed locally or sent/received/processed.

Algorithm 3 Distributed state space exploration based on one-sided communications

Constant	PES : **int** := number of processing elements
Shared objects	buf : **state_list**$[PES]$:= {**empty**, ..., **empty**};
	$free$: **bool**$[PES]$:= {**true**, ..., **true**};
Private objects	Q, R : **state_set** := **empty**;
	$sbuf$: **state_list**$[PES]$:= {**empty**, ..., **empty**};

```
 1: procedure exploreDistributed() is
 2:     launch explore_0 || ... || explore_{PES-1}
 3: procedure explore_i() is
 4:     if s_0.hash()%P = i then
 5:         Q.insert(s_0); R.insert(s_0)
 6:     while ¬termination() do
 7:         if checkForIncomingStates() then
 8:             processInStates_i()
 9:         if Q.isEmpty() then
10:             flushOutBuffers_i()
11:         else
12:             processQueue_i()
13: procedure processOutState_i(j, s) is
14:     sbuf[j].append(s)
15:     if sbuf[j].full() then
16:         flushOutBuffer_i(j)
17: procedure flushOutBuffer_i(j) is
18:     while ¬getMem(i, free[j]) do
19:         processInStates_i()
20:     putMem(i, free[j], false)
21:     putMem(j, buf[i], sbuf[j])
22:     sbuf[j].empty()
23: procedure flushOutBuffers_i() is
24:     for j ∈ {0, ..., |PES| − 1} with j ≠ i do
25:         if ¬sbuf[j].isEmpty() then
26:             flushOutBuffer_i(j)
27: procedure processQueue_i() is
28:     s := Q.remove()
29:     for s' ∈ succ(s) do
30:         if ¬s'.checkInvariant() then
31:             halt and report error
32:         else if s'.hash()%P ≠ i then
33:             processOutState_i(s'.hash()%P, s')
34:         else if ¬R.isIn(s') then
35:             Q.insert(s'); R.insert(s')
36: procedure processInStates_i() is
37:     for j ∈ {0, ..., |PES| − 1} with j ≠ i do
38:         buf := getMem(i, buf[j])
39:         if ¬buf.isEmpty() then
40:             putMem(j, free[i], true)
41:             for s ∈ buf do
42:                 if ¬R.isIn(s) then
43:                     Q.insert(s); R.insert(s)
```

Sketch of proof that there is no livelock at l. 18. The only place where a PE could get stuck waiting forever is at l. 18. In this case, PE i is waiting for PE j to free the memory by reading it and setting the free boolean to true. This operation is done in $processInStates_j$, which reads all incoming buffers. Note that a PE cannot be stuck in *processInStates* nor calls any function from it. Function $processInStates_j$ is called either in the *while* loop at ll. 18–19 or from $explore_j$ at l. 8. PE j is thus handling its own states in the *while* loop at ll. 6–12, one by one, checking for any incoming state after processing one state. If it has no state to handle it flushes its buffers, and thus executes $processInStates_j$ at l. 19.

Hence no process gets stuck in the *while* loop of ll. 18–19.

5 Experiments

We have implemented the algorithm of the previous section in the Helena tool [9] (see `http://www-lipn.univ-paris13.fr/~evangelista/helena`). We experimen-

ted with our algorithm on models of BEEM [15], a database of models written in the DVE modelling language and used to benchmark model checkers.

Helena first compiles the model into a C library including state and transition definitions, the transition relation (successors computation), the initial state definition, and so on. This library is then linked with the model checking engine integrating search algorithms to produce a dedicated executable. This approach, adopted by many other model checkers, greatly speeds up the verification compared to model checkers that directly interpret the model without compiling it.

5.1 Experimental environment

Experiments presented in this paper were carried out using the Grid'5000 [7] 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 used the Graphene cluster, which is made of 144 nodes (although we could not experiment with more than 127 nodes), each of which features a quad-core Intel Xeon X3440 running at 2.53 GHz, 16 GiB of RAM and a 20G InfiniBand network interconnection. The nodes were running a 64-bit Linux 4.9 kernel. All the code was compiled using the GNU gcc 6.3.0 compiler with -03 optimization flag. We used the OpenSHMEM implementation provided by OpenMPI 2.0.1 and the InfiniBand communication libraries libverbs 1.2.1 and librdmacm 1.1.0.

Since the machines feature four cores, we executed four processes per node. Each experiment was run 5 times and plots present the average and standard deviation of the set of measurements. Each run consisted of a complete state space exploration, i.e., no property was checked.

5.2 Implementation details

We now address some implementation details that were left out in the description of the algorithm of the previous section.

First, at l. 7 in the main procedure, a process checks if it has received any new state to be processed. Such a check implies to look at all buffers of the shared memory space and must therefore not be done too frequently. The simple solution we adopted is to perform this check every $10\,000^{\text{th}}$ state processed. We experimented with other values and this one yielded the best performance on the average although we did not witness this parameter to have a large impact unless set to a too small value. It would however be relevant to experiment with a dynamic solution allowing this frequency to evolve during the search in order to try to maximise the state generation rate.

The SHMEM heap size was set to a number that allows buffers of 65 000 bytes which is close to the MTU of our network interfaces. Hence, a buffer becomes full (test at l. 15 of the algorithm) when it cannot store any more state (DVE states are encoded with a constant number of bytes). We did not intensively experiment with that parameter and leave this to future works.

5.3 Scalability

We evaluated the scalability of Helena on models of various sizes. The sizes (number of states and transitions) of these models are given in Table 1. The last column indicates the range of process numbers we experimented with on the model. Unless noted otherwise, the speed-up is computed as, by definition, the ratio between the execution time of the sequential implementation of Helena and the execution of the parallel implementation on a given set of processes, using one core per process.

	Name	States	Transitions	Processes used
Small size models (runnable on 1 node)	iprotocol.7	59 794 192	200 828 479	1–384
	peterson.5	131 064 750	565 877 635	1–384
	elevator.5	185 008 051	185 008 051	1–384
Medium size models ($< 10^9$ states)	lifts.9	266 445 936	846 144 885	16–384
	firewire_link.3	425 333 983	1 621 543 475	16–384
	leader_filters.8	431 401 020	1 725 604 080	32–384
	collision.5	431 965 993	1 644 101 878	32–384
	iprotocol.8	447 570 146	1 501 247 756	32–384
	anderson.8	538 699 029	2 972 732 133	32–384
Large size models ($\geq 10^9$ states)	public_subscribe.5	1 153 014 089	5 447 695 171	32–508
	lamport.9	1 436 848 880	7 025 053 020	48–508
	brp.8	1 526 547 707	3 207 513 490	32–508
	synapse.9	1 675 298 471	3 291 122 975	48-508
	szymanski.6	6 779 809 484	38 604 341 308	256–508

Table 1. Model characteristics

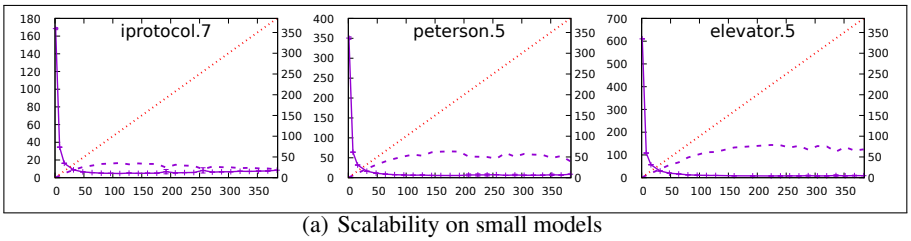
As expected, small size models (see Fig. 2(a)) can be run on a small number of cores, but they do not scale well beyond a certain number of processes, i.e., about 100–150 processes. Then the runtime tends to slightly increase. Indeed, as the number of processes grows, the number of states owned by each process decreases, meaning that queues often become empty. This causes an excessive number of flushes of partially filled buffers (l. 16 of the algorithm), synonym of an inefficient network usage.

When the size of the input model increases, Helena cannot be run on a single node. For medium and large size models, we computed the speed-up by normalizing using the execution time on the smallest number of processes we could get.

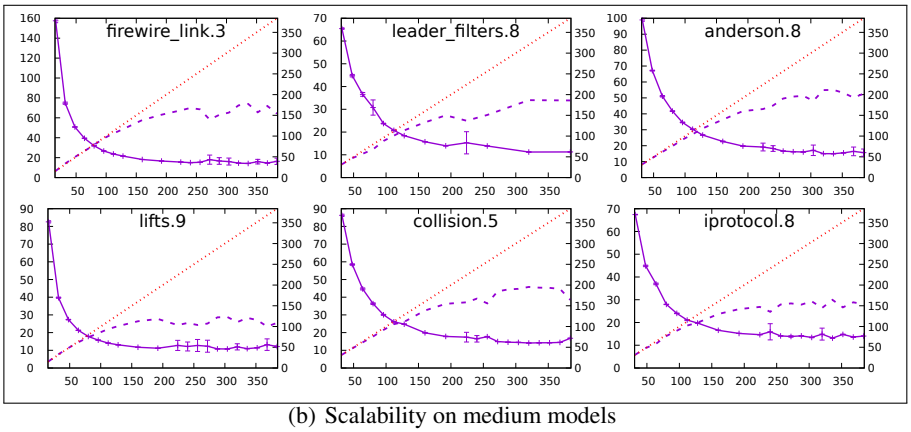
For medium size models (see Fig. 2(b)), the plots have the same shape, but the number of processes for which the execution time stagnates or increases is pushed to about 300 processes.

For four of the large models (see Fig. 2(c)) we did not observe any slow-down: they scale well on the full range of processes we were able to execute them on which is remarkable for a non-embarrassingly parallel application that communicates often.

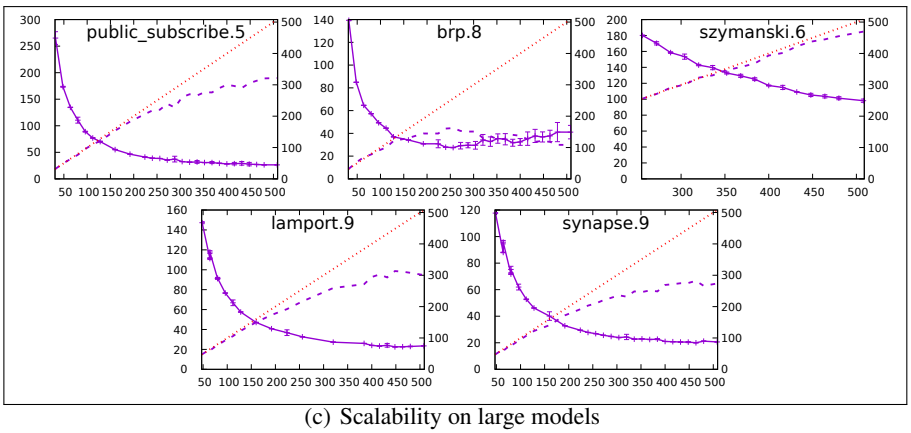
For model brp.8, we faced some unexpected behaviour described in Section 5.4 that explains the relatively bad speed-up observed. But beyond this problem, we conjecture that the high depth of this graph makes this model less appropriate for distributed model checking. The parallel exploration of such graphs is known to be less efficient.



(a) Scalability on small models



(b) Scalability on medium models



(c) Scalability on large models

Fig. 2. Scalability of Helena on the models of Table 1. On the X axis are the numbers of processes. On the left (resp. right) Y axis are execution times (speed-ups). The plain line with error bars gives execution times. The dashed one gives speed-ups. The dotted one gives the optimal theoretical speed-up (linear).

5.4 Process workload

We also studied the process workload to further investigate some issues revealed by Fig. 2 and make sure the load is balanced evenly among processes. Indeed, for some configurations (same model and number of processes) we noticed significant variations in the execution times of the five runs performed. This is especially visible for model brp.8 through the error bars. We thus recorded during each run, the number of states visited by each process during each second. The heat maps of Fig. 3 reproduce this data for two problematic runs of models brp.8 and firewire_link.3 (with 320 and 304 processes respectively) ; and, for the sake of comparison, for two "friendly" runs of models leader_filters.8 and public_subscribe.5 (with 240 and 384 processes respectively).

We first observe in all cases a slow start during which all processes have very few states to visit and spend most of their time idle, waiting for states coming from processes. This scenario is actually common to all models although the duration of this phase can vary, depending on — we conjecture — the structural characteristics of the state space graph. More specifically, the shape of the graph might be such that little parallelism can be extracted. The hash function distributes the few states between the processes and therefore, processes need to access only remote states. In the case of model brp.8 the long idle time at startup could indeed be explained by the important depth of its graph and the fact that very few states are gathered around the initial state. To remedy this issue we will investigate in future works the use of a small state cache used by a process to explore states it does not own in order to accelerate the discovery of its states, rather than waiting for other processes to send these states.

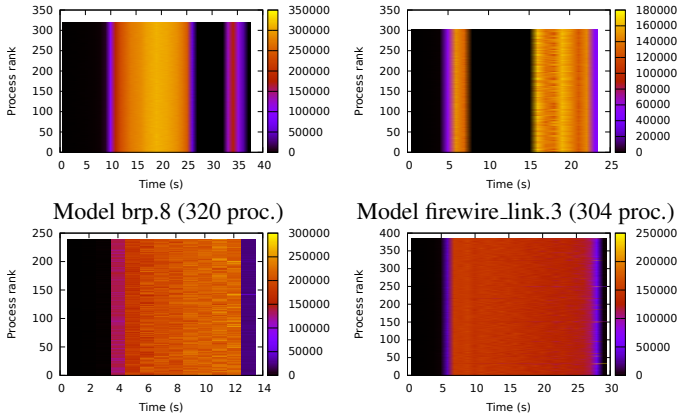


Fig. 3. Workload (number of states processed by second) of processes for four runs

In the case of models brp.8 and firewire_link.3, the heat maps also reveal that, after this slow start, the algorithm enters again a phase during which all processes are completely idle. This represents approximatively 5 and 7 seconds of the whole execution times of these two runs. Unfortunately, we are currently unable to explain this

phenomenon. We plan to profile the code to identify the source of this problem. Let us remark that this issue is actually the only source of the variations we observed during different runs with the same configuration. When the processes did not mysteriously halt this way during the search, we obtained remarkably stable performances.

Last, Fig. 3 also shows that the workload is well balanced among processes. This was however expected since all processes perform the same task and receive approximately the same amount of work, since states are distributed using the state hash function. Again, this observation can be generalised to all experiments we made.

5.5 Comparison with the DiVinE model checker

We also experimented with the DiVinE model checker [3], version 3.3, under the same conditions. DiVinE is a state-of-the-art verification tool that implements parallel algorithms for LTL model checking and reachability analysis using two-sided MPI communications [18]. Comparing these two tools can be viewed under two perspectives: speed, which depends highly on the speed of sequential computations, and parallel speed-up, which exhibits the efficiency of the parallel approach.

In this section, we are presenting both metrics. In their sequential implementation, Helena is slower than DiVinE, as we can see on the only models for which we were able to run sequential executions and presented Fig. 4 (top). We can see that, Helena has a higher speed-up and scales better than DiVinE. Although DiVinE is significantly faster when run sequentially, the two runtime curves cross each other quickly and Helena becomes faster. Therefore, our approach is efficient enough to make Helena faster when we use more than a handful of processes and the parallelism become non-trivial.

On very big models (public_subscribe.5, anderson.8), the difference between Helena and DiVinE is relatively small, especially at large scale. In our algorithm, the number of communication scales with the size of the model. Therefore, on large models, the parallel application performs a large number of communications. On DiVinE, we can expect that communicating often on all the processes reduces the penalty involved by the "forced" synchronisation between the processes and reduces the performance gap.

As explained in Section 5.3, when we cannot explore the state space with the sequential implementations, we normalize the speed-up using the execution time of the smallest possible parallel execution (using the same number of processes for DiVinE and Helena). Therefore, for larger models (Fig. 4, bottom), we normalize the speed-up beyond this cross-over between the execution time of DiVinE and Helena. But still, Helena scales better than DiVinE. We believe that the higher parallel efficiency of Helena is due to the less synchronous nature of the parallel algorithm for the state space exploration, which is made possible by the one-sided communication model.

6 Conclusion and perspectives

This paper is a first step towards the use of one-sided based communications in the context of distributed state model checking. Our experiments revealed that our distributed state space exploration algorithm can compete with the DiVinE model checker which is, to the best of our knowledge, the reference tool in distributed automated verification.

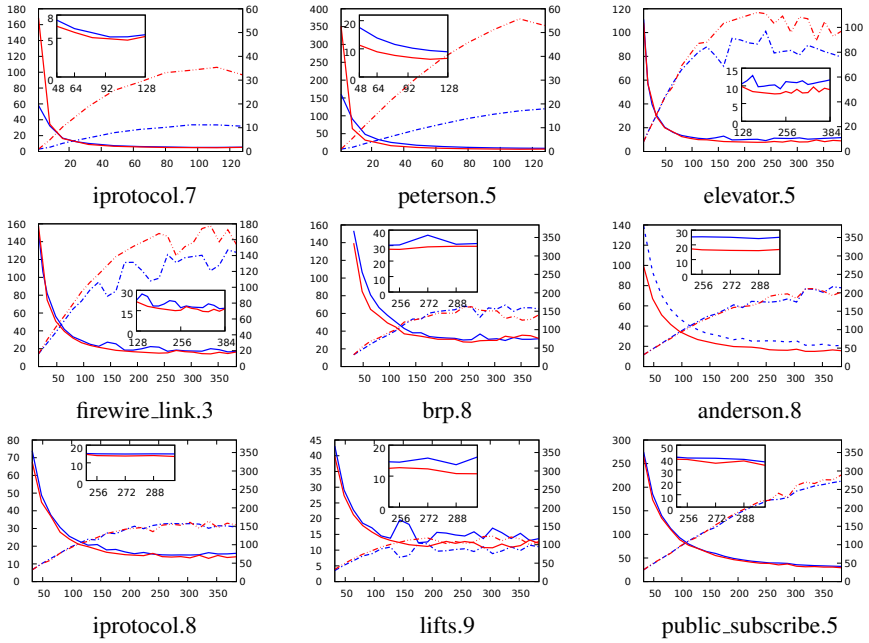


Fig. 4. Performance comparison between DiVinE and Helena on models of Table 1. On the X axis are the numbers of processes. On the left (resp. right) Y axis are execution times (speed-ups). Helena is represented by red lines, that are plain for the execution time and a pattern made of two dots and a dash for the speed-up. DiVinE is represented by blue lines, that are dashed for the execution time and a pattern made of a dot and a dash for the speed-up. Inside of each plot is a zoom on the execution time.

An immediate perspective is to experiment more thoroughly with our algorithm. The experiments have revealed some undesired behaviour that has to be investigated and we need to gain better understanding of the impact of some parameters such as the SHMEM shared heap size.

Our algorithm currently is a direct adaptation of the state of the art distributed algorithm for the one-sided communication model and it does not fully benefit from the primitives provided by the OpenSHMEM library (or any other library that falls in that category, such as MPI 3.0), such as, e.g., remote atomic compare-and-swap. We therefore plan, in future works, to study how to take advantage of the specificities of OpenSHMEM to efficiently implement distributed versions of state space reduction techniques such as the state compression technique of [11] based on distributed hash tables or other distributed state space exploration algorithms like the one we designed for multi-core architectures [10].

The adaptation of various optimisations proposed by the model checking community, such as load balancing [5,12], to the context of one-sided communications, is another perspective. Such techniques are especially required in the case of heterogeneous networks, which we did not consider nor experiment with in this work.

Last, we will consider the design of a multi-threaded version of our algorithm as done in the Eddy_Murphi tool [13] that separates state operations (e.g., successor computation, insertion in the hash table) performed by a first thread from communications done by second thread.

References

1. OpenSHMEM Application Programming Interface version 1.4. `http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf`, Dec 2017.
2. C. Baier and J-P Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. Z. Baranovà, J. Barnat, K. Kejstovà, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill. Model Checking of C and C++ with DIVINE 4. In *ATVA 2017*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017.
4. 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.
5. B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt. Industrial Strength Distributed Explicit State Model Checking. In *PDMC'2010*, 2010.
6. J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.
7. F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Vicat-Blanc Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, B. Quetier, and O. Richard. 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, Seattle, Washington, USA, November 2005. IEEE/ACM.
8. C. Coti. POSH: Paris OpenSHMEM: A High-Performance OpenSHMEM Implementation for Shared Memory Systems. In *Procedia Computer Science, special issue on the 2014 International Conference on Computational Science (ICCS 2014)*, volume 29, pages 2422–2431, 2014.
9. S. Evangelista. High Level Petri Nets Analysis with Helena. In *ATPN'2005*, 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. G.J. Holzmann. Recursive Indexing and Compression Training Runs. In *SPIN'1997*, 1997.
12. R. Kumar and E.G. Mercer. Load Balancing Parallel Explicit State Model Checking. *ENTCS*, 128(3):19–34, 2005.
13. I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in Eddy. *STTT*, 11(1):13–25, 2009.
14. W. Oortwijn, T. van Dijk, and J. van de Pol. Distributed Binary Decision Diagrams for Symbolic Reachability. In *SPIN'2017*, pages 21–30. ACM, 2017.
15. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'2007*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
16. U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *CAV'1997*, volume 1254 of *LNCS*, pages 256–278. Springer, 1997.
17. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.
18. K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *IPDPS'2009*, pages 1–12. IEEE, 2009.