Large-scale Distributed Verification using CADP: Beyond Clusters to Grids

Hubert Garavel[1] Radu Mateescu[1] Wendelin Serwe[1]

Inria / LIG — CONVECS team
655, avenue de l'Europe, Inovallée Montbonnot, 38334 St Ismier Cedex, France

Abstract

Distributed verification uses the resources of several computers to speed up the verification and, even more importantly, to access large amounts of memory beyond the capabilities of a single computer. In this paper, we describe the distributed verification tools provided by the CADP (*Construction and Analysis of Distributed Processes*) toolbox, especially focusing on its most recent tools for management, inspection, and on-the-fly exploration of distributed state spaces. We also report about large-scale experiments carried out using these tools on Grid'5000 using up to 512 distributed processes.

Keywords: asynchronous systems, distributed verification, labeled transition system, model checking, on-the-fly verification, process calculus

1 Introduction

When analyzing concurrent systems using explicit-state verification methods in an action-based setting, the semantic models for state spaces are Labeled Transition Systems (LTSs). One approach consists in first building the LTS of the concurrent system under study; this LTS can then be minimized modulo a bisimulation relation to increase the efficiency of further analyses, such as model checking, equivalence checking, visual checking, etc. An alternative approach is to perform these analyses on-the-fly, i.e., during the construction of the LTS, so as to detect errors without constructing the entire LTS first. The latter approach is more suitable for early verification steps, where bugs are frequent and can be quickly detected, while the former approach is more

[1] `{Hubert.Garavel,Radu.Mateescu,Wendelin.Serwe}@inria.fr`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

efficient in the late phases of the design process, where the entire LTS must be explored to ensure its correctness.

Due to the state explosion phenomenon (prohibitive size of the LTS for systems containing many concurrent processes and/or complex data types), LTS generation can become a bottleneck in the verification process. In such case, if the LTS is too large to be constructed on a single machine, one may resort to distributed computing infrastructures, such as clusters and grids, which increase by several orders of magnitude the amount of memory available. The CADP verification toolbox [11] exploits this possibility by providing several tools for distributed verification, in particular DISTRIBUTOR and BCG_MERGE [13,12]. These tools respectively enable to construct a partitioned LTS (i.e., split into several fragments, each stored in a separate file, possibly on a different machine) and to convert it into a monolithic LTS (i.e., stored in a single file). To scale up the verification capabilities, it is sometimes beneficial to avoid the construction of a monolithic LTS and instead work as long as possible with a partitioned LTS.

In this paper, we present the CADP tools (some of which have been recently added) for manipulating partitioned LTSs. All these tools are based on distributed algorithms and are implemented using standard network communication primitives available on most clusters and grids. We also report about large-scale multi-cluster experiments on distributed LTS generation and on-the-fly reduction. These experiments have been carried out on the Grid'5000 computing infrastructure [9] using up to 512 distributed processes, and provide insight about the performance gains and scalability when the number of distributed processes increases.

The paper is organized as follows. Section 2 briefly describes the software library of CADP for handling network communications. Section 3 defines the PBG format for partitioned LTSs. Sections 4 and 5 present the new tools for creating and manipulating PBG files. Section 6 describes a new tool enabling an on-the-fly exploration of PBG files. Section 7 gives experimental measures about the use of these new tools on the Grid'5000 computing infrastructure [9] for the distributed generation and on-the-fly reduction of large LTSs. Section 8 gives a brief overview of related work. Finally, Section 9 concludes the paper and suggests directions for future work.

2 The Network Communication Library

A typical distributed application developed using CADP consists of $N + 1$ POSIX processes, namely N *workers*, possibly running on remote machines or on different processors/cores of multi-processor/multi-core machines, and one *master*, which supervises the execution of workers, runs on the user frontal machine, and interacts (inputs/outputs) with the user. The communications

```
rsh = ssh -q
rcp = scp -q
connect_timeout = 30
edel-42.grenoble.grid5000.fr
  directory=/home/serwe/1
edel-42.grenoble.grid5000.fr
  directory=/home/serwe/2
stremi-32.reims.grid5000.fr
  directory=/tmp/3
suno-41.sophia.grid5000.fr
  directory=/home/serwe/4
```

(a) sample GCF file

```
PBG 1.0
# PBG format by the SENVA team -- http://vasy.inria.fr/senva
# created by Distributor (C) INRIA/VASY -- http://cadp.inria.fr
grid: "grid_4.gcf"[0]
states: partitioned
edges: incoming
initiator: 1
fragments: 4
1: states: 4582872 fragment: "clh-1.bcg"[0] log: "clh-1.log"[0]
2: states: 4581049 fragment: "clh-2.bcg"[0] log: "clh-2.log"[0]
3: states: 4577666 fragment: "clh-3.bcg"[0] log: "clh-3.log"[0]
4: states: 4576262 fragment: "clh-4.bcg"[0] log: "clh-4.log"[0]
```

(b) sample PBG file

Fig. 1. Sample GCF and PBG files describing a partitioned LTS

between these processes rely on a dedicated library named `caesar_network_1` (or `network_1` for short), which currently implements two communication topologies: star (all workers connected only to the master) and fully-connected (complete graph between workers and master).

The configuration of the grid is described as a text file in the GCF (*Grid Configuration File*) format [7], which specifies the computing nodes on which workers have to execute, the way to access them (connection protocols and parameters for file transfer), and the directories in which workers have to run. A GCF format is application-independent and can be used for launching several distributed applications running on the same computing nodes. Figure 1(a) shows an example of an GCF file with four workers.

To ensure maximal portability, the `network_1` library is purposely based on standard operating system primitives (namely, TCP/IP sockets) and standard remote access protocols (namely, `rsh` or `ssh`), so that the distributed tools of CADP do not require additional communication libraries (such as MPI) to be specifically installed on each remote machine. Furthermore, the `network_1` library supports seamless multi-core, intra-cluster, and inter-cluster communication: apart from performance aspects, the end-user sees no functional difference between an application running on several cores of a single machine and an application running on several machines belonging to one or many clusters.

3 The PBG Format

To represent LTSs, the CADP toolbox provides the BCG (*Binary Coded Graphs*) file format and its associated software libraries. A BCG file stores the states, labels, and transitions of an LTS in a compact way using binary encoding and dedicated compression schemes that enable efficient representation and manipulation. BCG files can be handled using the existing CADP tools (e.g., inspection, visualization, label renaming, bisimulation minimization, on-the-fly exploration, etc.) or using custom tools developed using the

CADP libraries for reading and writing BCG files.

When dealing with distributed verification tools and LTSs stored on several machines, a single BCG file is no longer sufficient. The PBG (*Partitioned BCG Graph*) format [7,8] addresses this problem. This format is an outcome of the SENVA co-operation [2] between the former SEN2 team of CWI and the former VASY team of Inria Grenoble. Specifically designed for the purpose of distributed verification, the PBG format implements the theoretical concept of *Partitioned LTS* introduced in [13] and provides a unified access to an LTS distributed over a set of remote machines. A PBG file gathers a collection of BCG files, called *fragments* (one fragment per worker), which can be stored either in separate directories located on the (possibly remote) machines on which workers execute, or on a common file system shared (e.g., using NFS or Samba) by all workers. Taken altogether, these fragments form a partition of the LTS, the states and transitions of which are distributed across the various fragments as specified in [13], each fragment storing a set of states and the transitions going into these states. Note that, taken individually, each fragment is meaningless; for instance, it may be a disconnected graph, which is never the case with an LTS representing the reachable state space of a concurrent system.

Concretely, a PBG file is a text file containing references to the fragments and the GCF file used for constructing the partitioned LTS. The example of a PBG file shown in Figure 1(b) corresponds to an LTS partitioned in four fragments, the first of which contains the initial state of the LTS. For each fragment, the PBG file lists the number of states of the fragment and the files associated to this fragment (i.e., a BCG file and a log file containing error messages that may have been issued when building the fragment). There exists a simple code library for reading and writing PBG files.

It is worth noticing that the PBG format uses a number of fragments linear in the number of computing nodes; this is better than the competing SVC format [6], whose number of fragments is quadratic in the number of computing nodes.

4 Tools for PBG Creation

A partitioned LTS in the PBG format can be generated using the DISTRIBUTOR tool [13,12] of CADP. The tool works by launching several workers on (local and remote) computing nodes specified by a GCF file. Each worker is in charge of generating a fragment of the LTS, which is stored as a BCG file; a static hash function determines which state is explored by which worker. Upon termination, DISTRIBUTOR produces a PBG file gathering all these

[2] See `http://vasy.inria.fr/senva`

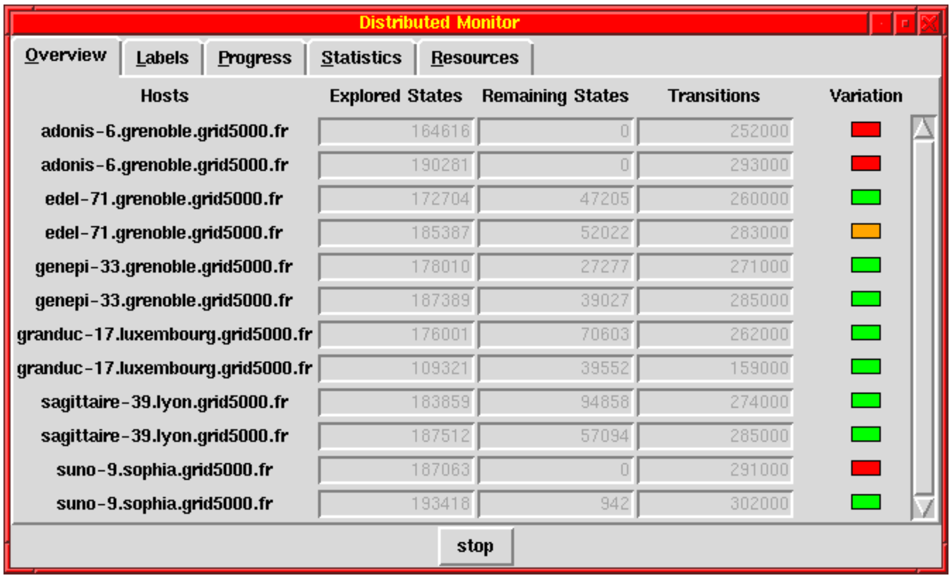


Fig. 2. Overview tab monitoring LTS generation using twelve workers distributed over six nodes of six clusters (`adonis`, `edel`, `genepi`, `granduc`, `sagittaire`, and `suno`) of Grid'5000, geographically located in Grenoble, Luxembourg, Lyon, and Sophia-Antipolis. In column "Variation", a green (respectively, orange) box indicates that the number of remaining states is increasing (respectively, decreasing or stable), and a red box indicates that the worker is idle; boxes are red for both workers on `adonis-6` and the first worker on `suno-9`, orange for the second worker on `edel-71`, and green for all other workers.

fragments. The progression of the distributed computation can be monitored in real-time (see Figure 2).

Besides the obvious advantages brought by distributed LTS generation (speeding up the generation and increasing the amount of memory far beyond what is made available by a single machine), DISTRIBUTOR also provides on-the-fly reductions (τ-compression and τ-confluence) that preserve branching bisimulation and may be useful when dealing with large LTSs that cannot be generated and minimized on a sequential machine.

5 Tools for PBG Inspection and Manipulation

Several tools are currently available in CADP for handling PBG files:

- PBG_MERGE (previously called BCG_MERGE [12]) converts a partitioned LTS represented in the PBG format into a monolithic LTS stored in a BCG file. The LTS fragments are merged into a single file, in which states are given a contiguous numbering that improves compactness of the resulting BCG file.
- PBG_CP, PBG_MV, and PBG_RM are new tools for copying, moving, and removing PBG files, keeping in mind that the fragments of these PBG files may be disseminated on a number of remote machines, possibly located in different countries. These tools facilitate standard operations on PBG files

and maintain consistency during these operations.

- PBG_INFO is a new tool for inspecting PBG files. It currently provides several functionalities, such as consistency checking (i.e., existence and readability of all fragment files), calculation the size of the corresponding LTS (number of states and transitions), display of the list of labels, and concatenation of remote log files (this is useful, e.g., to understand the reason why a PBG generation fails, and to compute global statistics about CPU and memory usage by the workers).

6 Tools for PBG On-the-fly Exploration

On-the-fly verification is an approach to fight state explosion by verifying an LTS during its construction rather than constructing the LTS first and verifying it afterward. So doing, on-the-fly verification may enable an early detection of errors even in presence of state explosion. The OPEN/CAESAR [10] environment of CADP provides a modular architecture for on-the-fly verification tools, which separates the language-dependent aspects (translation of a concurrent system description into an LTS) from the language-independent aspects (forward exploration of an LTS, e.g., for verification). OPEN/CAESAR defines a generic API for representing an LTS by its transition relation. OPEN/CAESAR also contains a set of libraries implementing various primitives and data structures dedicated to on-the-fly graph exploration (hash tables, stacks, etc.).

CADP currently provides OPEN/CAESAR-compliant compilers for several high-level description languages (LOTOS, LNT, and FSP) and low-level state machine formats (BCG, EXP, and SEQ). These compilers implement the OPEN/CAESAR API to explore the corresponding LTSs on the fly.

CADP also provides a set of on-the-fly verification tools based on the OPEN/CAESAR API: model checking MCL formulas (EVALUATOR), bisimulation checking (BISIMULATOR), partial order reduction (REDUCTOR), random exploration (EXECUTOR), regular sequence searching (EXHIBITOR), steady-state Markov chain simulation (CUNCTATOR), etc. All these tools can be applied to any description in an input language for which an OPEN/CAESAR-compliant compiler exists.

The newly developed PBG_OPEN tool is an OPEN/CAESAR-compliant compiler for the PBG format. The main advantage of PBG_OPEN is that it can use the memory of several machines to store the transition relation of a partitioned LTS. Therefore, PBG_OPEN can explore on-the-fly large partitioned LTSs that could not be explored using other tool combinations.

PBG_OPEN (see Figure 3) is a distributed tool consisting of a master and several workers, each associated to a BCG fragment referenced in the PBG file. Each worker is responsible for opening its fragment, initializing

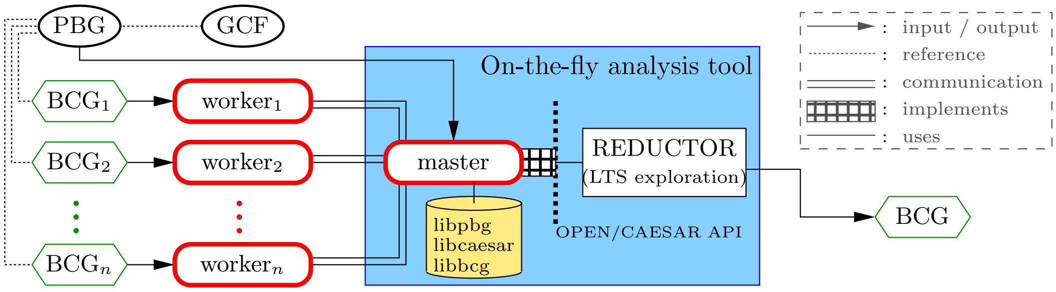


Fig. 3. Architecture of an OPEN/CAESAR on-the-fly application built using PBG_OPEN and the partial order reduction tool REDUCTOR of CADP

label information, and answering the master requests to compute the outgoing transitions of states belonging to that fragment.

The initialization phase consists in normalizing the transition labels of the fragments by assigning unique label numbers across all workers. This is necessary because the same label may be numbered differently in different fragments. During initialization, each worker sends its list of labels to the master, which assigns a unique number to each label and then sends back to each worker globally unique numbers for these labels. This preliminary step avoids the performance overhead that would occur if the master had to renumber labels in all transitions.

Each transition is represented by a triple $\langle s, a, s' \rangle$, where s, a, and s' are numbers encoding the source state, label, and target state; a is now a globally unique label number common to all fragments. Notice that fragments store the incoming transitions, i.e., a given worker stores all triples $\langle s, a, s' \rangle$ for a given s', whereas the triples $\langle s, a, s' \rangle$ for a given s may be distributed between workers.

When requested by an on-the-fly exploration tool to compute all transitions going out of a given state s, the master forwards the request to all workers. Each worker retrieves (using the BCG primitives) its transitions going out of state s and sends them back to the master. The master explores the transitions received from the workers in their order of arrival, which is nondeterministic.

Given that some on-the-fly analysis tools (e.g., the REDUCTOR tool) often explore several times the transitions going out of the same state, PBG_OPEN implements memoization using a cache. After receiving the list L of transitions going out of a state s, the master stores the couple $\langle s, L \rangle$ in the cache. When the cache is full, a couple $\langle s', L' \rangle$ present in the cache is selected according to the LRU (*Least Recently Used*) strategy, and is replaced by the new couple $\langle s, L \rangle$. In general, neither the number of transitions in L nor the maximal size of L are known in advance (they could be computed using a preliminary LTS traversal, but this would be too costly). Two variants of the cache have been implemented in PBG_OPEN using the `cache_1` library initially developed for state space caching [19]. These variants differ in the way

example	number of processes	state size (bytes)	direct		bcg_min -branching	
			states	transitions	states	transitions
TTAS	4	17	18,721	39,736	80	224
Burns&Lynch-4	4	43	769,244	1,367,318	3,023	11,244
Peterson_tree	4	102	7,205,545	12,692,584	2,361	8,352
Szymanski	4	60	9,243,653	18,859,330	3,090	10,356
Knuth	4	48	16,642,361	32,614,282	6,721	27,281
CLH	4	48	18,317,849	31,849,616	320	848
Burns&Lynch	5	63	39,796,190	75,024,550	35,734	167,747
Lamport	4	62	78,535,973	154,003,176	29,719	99,850
Anderson	5	49	166,488,027	345,843,975	1,712	4,880
Peterson	4	49	214,175,671	389,640,061	6,460	21,347
MCS	5	90	261,064,933	500,744,765	1,712	4,880
Dijkstra	4	57	289,120,985	542,886,005	41,513	163,538

Table 1
State space sizes

the couples $\langle s, L\rangle$ are stored: (a) the *variable-size* variant stores source states s in the cache and transition lists L in the heap, outside of the cache; (b) the *fixed-size* variant stores both source states s and transition lists L inside the cache; if L is long, it may replace one, several, or even all entries of the cache; if L is too long for the cache, $\langle s, L\rangle$ will not be stored in the cache. Variant (a) is simpler to implement, but variant (b) guarantees a statically bounded amount of memory.

7 Applications and Experiments

We experimented these tools on examples taken from a case study on the formal verification and performance analysis of various mutual exclusion protocols [17] specified in the LNT language. [3] We instantiated each protocol for four or five processes competing for the critical section and generated the corresponding LTSs [4] directly: so doing, we observed larger state spaces (see Table 1) than those obtained by the compositional approach described in [17].

The experiments ran on the Grid'5000 clusters geographically located in Grenoble, Luxembourg, Lyon, Reims, and Sophia-Antipolis, all of which were equipped with two processors per node (i.e., machine or server) and the same operating system (Debian Linux 6.0 "Squeeze"). The clusters differ however in the number of nodes, the type of processor, the number of cores per processor, and the amount of RAM per node (see Table 2 for details). In each cluster, the nodes are connected by 1 GBit/s links; all the clusters in Grenoble are connected to the same switch. Communication between sites uses a dedicated 10 GBit/s link. Each site provides a unique resource manager for all clusters of the site.

[3] These specifications will be included as examples in the next stable release of CADP.

[4] In fact, these LTSs are *interactive Markov chains* [15], which can be seen as particular forms of LTSs.

cluster	nodes	processor	cores/proc.	frequency	RAM	site
adonis	10	Xeon E5520	4	2.26 GHz	24 GB	Grenoble
edel	64	Xeon E5520	4	2.27 GHz	24 GB	Grenoble
genepi	32	Xeon E5420 QC	4	2.50 GHz	8 GB	Grenoble
granduc	22	Xeon L5335	4	2.00 GHz	16 GB	Luxembourg
sagittaire	65	Opteron 250	1	2.40 GHz	2 GB	Lyon
stremi	44	Opteron 6164 HE	12	1.70 GHz	48 GB	Reims
suno	34	Xeon E5520	4	2.26 GHz	32 GB	Sophia-Antipolis

Table 2
Characteristics of the used Grid'5000 clusters

Whenever possible, we used a separate node for the master and dedicated one core to each worker, i.e., we launched at most n workers on a node with a total of n cores. Execution time and memory consumption were measured using `memtime`[5], and, if possible, averaging several executions. In execution time, we include setting up the workers (creating working directories, copying files, etc.). Concerning memory consumption, we measure for each process (master and workers) the maximal amount of memory required by this process during its execution; we call *total memory* the sum of these maximal amounts for all processes, and we call *peak memory* the greatest value among all these maximal amounts.

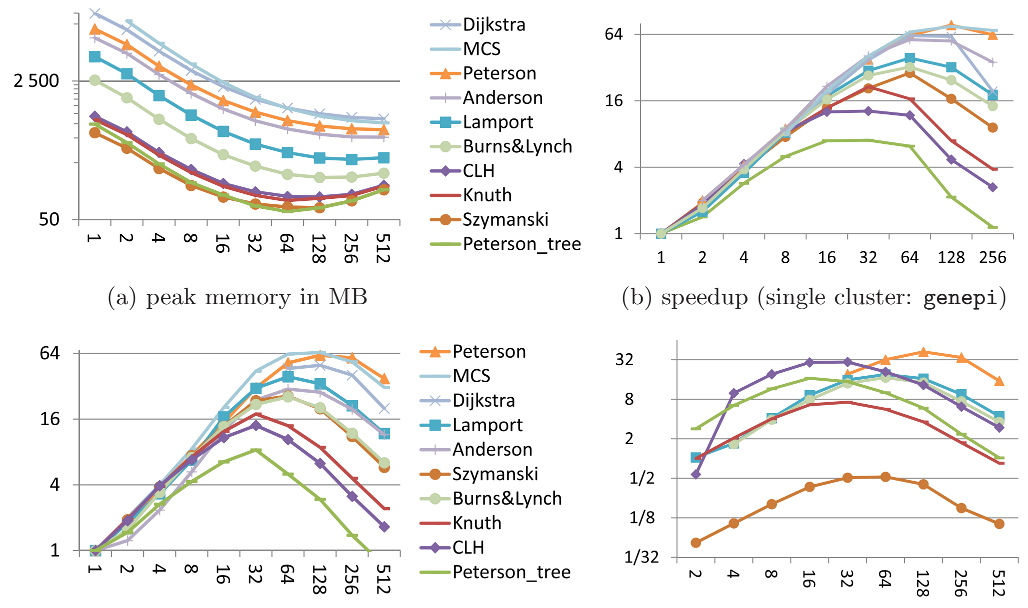
Because access to computing nodes of Grid'5000 is granted by resource managers, it took extra efforts to run a lot of experiments under the same conditions because we have no control on distribution of allocated nodes over the different clusters and/or switches, overall load on file servers and communication network, etc. For instance, we observed significant variation in execution time; to ensure consistent figures, we excluded the 20% extreme values before computing the average. Also, due to the rather small set of examples sharing similar characteristics (such as the average number of outgoing transitions or the number of labels), the experiments reported here illustrate tendencies; results may vary for other types of LTSs and/or grid configurations.

7.1 Performance Study of Distributed State Space Generation

For each mutual exclusion protocol example, we used DISTRIBUTOR to generate the corresponding PBG using a single cluster, multiple clusters at the same site, and multiple clusters at different sites. Figures 4 and 5 give memory consumption and execution time speedup for up to 512 workers; Figure 4 is for distributed execution and Figure 5 is for multi-core execution on a single server. Note that many axes of the figures use a logarithmic scale.

Peak memory consumption. Figure 4(a) shows that increasing the number of

[5] Downloadable at `http://www.update.uu.se/~johanb/memtime/`



(a) peak memory in MB

(b) speedup (single cluster: `genepi`)

(c) speedup (multiple clusters, single site)

(d) speedup (multiple sites)

Fig. 4. Distributed state space generation; all axes in logarithmic scale

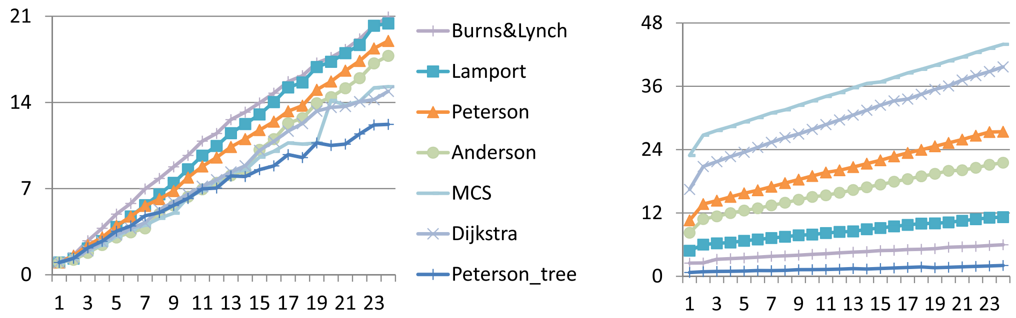
workers reduces the peak memory consumption [6], i.e., the maximum amount of memory required by the master or any of the workers: when using two workers, we observe reductions up to 40%. On the three largest examples ("Peterson", "MCS", and "Dijkstra"), this memory reduction enables LTS generation on the **genepi** cluster that provides only 8 GB RAM per node (using two or four nodes).

For each example there is an "optimal" number of workers which minimizes peak memory consumption: using more workers increases peak memory consumption. This can be explained by the constant memory requirements of the CADP communication library (table of nodes, communication buffers, sockets, etc). Figure 4(a) shows that the smaller the LTS, the smaller the optimal number of workers (for very small examples, distributed verification makes no sense and sequential LTS generation performs betters).

Figure 4(a) was obtained by experimenting with workers distributed over the three clusters in Grenoble; the figures observed using either a single cluster or clusters on different sites are the same.

Speedup. Figures 4(b) to 4(d) show the speedup for different combinations of clusters. Workers use their local disks rather than a shared file system. We take as the reference for the speedup the sequential GENERATOR tool, running on a machine of the **genepi** cluster for Figure 4(b) and running on a machine of the **edel** cluster for Figures 4(c) and 4(d). However, on Figure 4(b),

[6] For the total memory consumption, see Figure 5(b).



(a) speedup

(b) total memory in GB

Fig. 5. State space generation on a single machine of the `stremi` cluster

the large examples "Dijkstra", "MCS", and "Peterson" require more memory than available on single a machine: thus we take as reference the execution of DISTRIBUTOR with the smallest number of workers sufficient to generate the LTS (namely, two for "Peterson" and four for "Dijkstra" and "MCS").

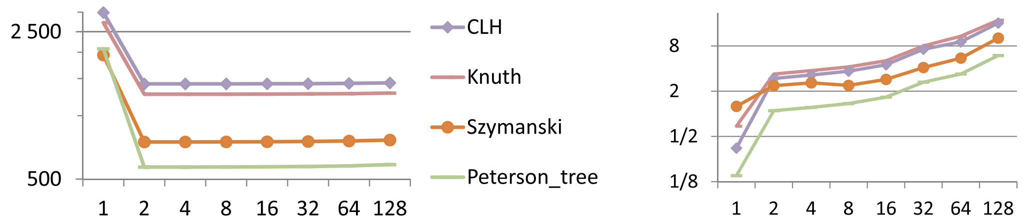
The observed speedups depend on the communication cost, but the tendencies are similar. All three graphs show an almost linear speedup as long as the number of workers is lower than the "optimum" value mentioned above. If the number of workers becomes too large, speedup drops: for small examples and many workers, distributed execution even becomes longer than sequential execution (speedup lower than one) due to the overhead of setting up the distributed application (currently implemented by sequentially copying all necessary files to the remote machines).

Concerning execution time (rather than speedup): the difference in execution time between the slowest and fastest examples for a given number of workers is reduced when the number of workers increases. For instance, this difference is more than one hour for sequential execution and only two minutes with 256 workers.

Multi-core execution. Figure 5 shows the speedup and total memory observed when running DISTRIBUTOR on one 24 core server, with at most one worker per core.

Concerning speedup, the almost linear speedups of Figure 5(a) show that DISTRIBUTOR also performs well on multi-cores even if the socket-based `network_1` library used by DISTRIBUTOR has not been specifically optimized for shared-memory architectures. When increasing the number of workers beyond the number of cores, the speedup drops drastically; because this is expected, it is not shown in Figure 5(a).

Concerning total memory consumption, Figure 5(b) shows that the memory consumption increases linearly with the number of workers, which can be explained by the constant memory cost per worker.



(a) peak memory in MB

(b) execution time in hours

Fig. 6. On-the-fly τ-confluence reduction using DISTRIBUTOR and PBG_OPEN/REDUCTOR

7.2 Performance Study of τ-confluence Reduction

Simple τ-confluence reduction. To measure the overall performance of PBG_OPEN on a particularly demanding on-the-fly application, we selected the REDUCTOR tool of CADP, which performs τ-confluence reduction using a sequential on-the-fly algorithm. We first generated PBG files using DISTRIBUTOR and then applied PBG_OPEN and REDUCTOR to reduce these distributed state spaces with respect to τ-confluence, after hiding all actions except entry and leave of the critical and non-critical sections. In such "stress tests" conducted on all but the largest examples[7], the complete state space, including very large components of τ-transitions, must be explored on-the-fly (as can be deduced from the small size of the state spaces after minimization for branching bisimulation — see the last two columns of Table 1).

Figure 6 summarizes the results of the experiments using the `granduc` cluster and with up to 128 workers. Figure 6(a) shows the overall peak memory consumption, i.e., the maximum amount of memory used by DISTRIBUTOR and PBG_OPEN/REDUCTOR; Figure 6(b) shows the overall execution time, i.e., the sum of the execution time of DISTRIBUTOR and PBG_OPEN/REDUCTOR (we did not compute the speedup because these experiments concern a sequential algorithm operating on a distributed state space). For these experiments, the caching mechanism of PBG_OPEN was deactivated, because the effects of caching will be studied later in section 7.3. The numbers for one worker correspond to the execution time of a sequential execution done directly at the level of the LNT source language, using the LNT.OPEN/REDUCTOR tools without DISTRIBUTOR.

As expected, for medium-size examples, with between two and 128 workers, peak memory consumption is lower than with LNT.OPEN/REDUCTOR because PBG_OPEN uses eight bytes per state, which is less than the state sizes with LNT mentioned in the second column of Table 1. Because the master executing the sequential on-the-fly algorithm of REDUCTOR is the

[7] The long execution times exceed the 62 hour limit of Grid'5000 jobs and are the reason why we did not experiment with the larger examples.

example	LNT.OPEN		BCG_OPEN		PBG_OPEN	
	time	memory	time	memory	time	memory
Peterson_tree	549	2,076	365	626	3,954	570
Szymanski	4,530	1,940	1,290	842	8,537	750
Knuth	2,470	2,764	1,642	1,428	12,174	1,264
CLH	1,267	3,087	1,060	1,585	10,615	1,415
Lamport	oom	oom	9,227	6,930	59,932	6,015

Table 3
Comparison of different reduction techniques; two workers for distributed tools; execution time in seconds, peak memory (MB, complete tool combination); "oom" means "out of memory"

bottleneck, the peak memory consumption is almost independent from the number of workers; however, memory consumption increases for all examples if the number of workers gets too high.

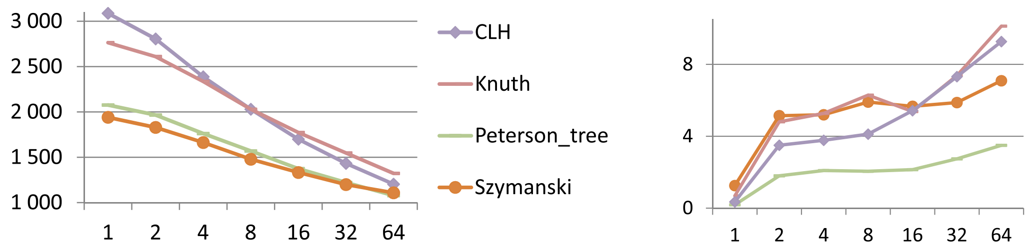
The reduction of the peak memory comes at the price of a (significant) increase in execution time due to the communication latency when collecting outgoing transitions from the distributed fragments: we observed a drop in CPU usage of REDUCTOR from 100% for LNT.OPEN down to 40% for PBG_OPEN (even down to 1% if the fragments are located on geographically distant sites).

Alternative τ-confluence reduction. We then compared the aforementioned τ-confluence reduction using DISTRIBUTOR, PBG_OPEN, and REDUCTOR against two other approaches, namely:

- the sequential execution of REDUCTOR directly at the level of the LNT source language (i.e., using LNT.OPEN and REDUCTOR on a single node),
- the distributed generation of a PBG file using DISTRIBUTOR, followed by an LTS merge using PBG_MERGE, and sequential execution of REDUCTOR on the resulting BCG file (i.e, using BCG_OPEN and REDUCTOR on a single node).

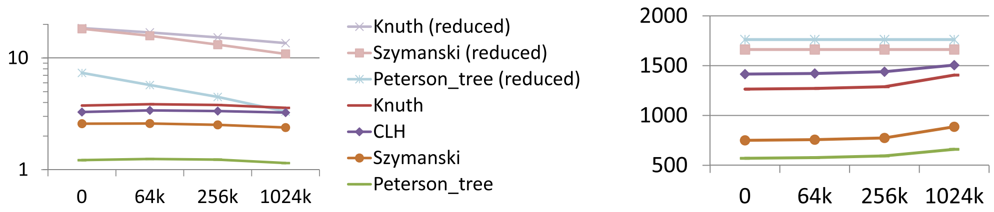
Table 3 shows execution time and peak memory consumption of these three approaches, confirming the claims of section 6: for all but the smallest examples, PBG_OPEN requires the least amount of memory. The direct connection to the source language requires the most memory, because states are larger (in particular when the LNT program contains complex data structures) than for the exploration of a PBG or BCG (where each state is represented by a number). BCG_OPEN, which loads the whole LTS in memory, requires more memory than PBG_OPEN, in which each worker uses its own memory to load only a fragment of the partitioned LTS.

Double τ-confluence reduction. For some examples, we also experimented the combination of two successive on-the-fly τ-confluence reductions, applying REDUCTOR on a PBG generated using DISTRIBUTOR with activated on-the-fly τ-confluence reduction. This double reduction yields LTSs that are up



(a) peak memory in MB (b) execution time in hours

Fig. 7. Two successive on-the-fly τ-confluence reductions using DISTRIBUTOR with option τ-confluence reduction followed by PBG_OPEN/REDUCTOR



(a) execution time in hours (b) peak memory in MB

Fig. 8. Cache with increasing number of cache entries: `granduc` cluster, four workers

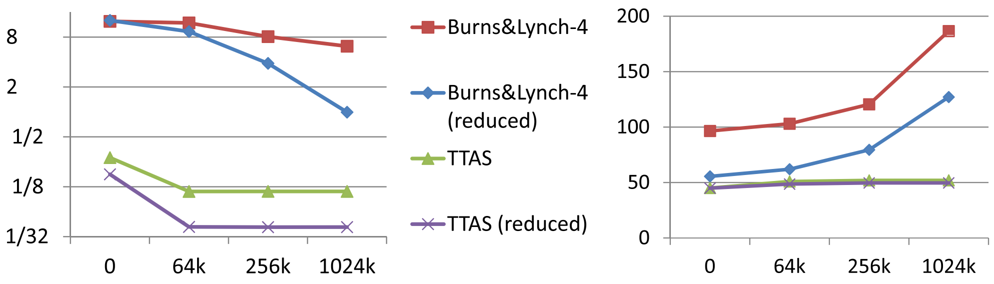
to a factor 1.9 smaller than those obtained by calling REDUCTOR only (but still about 500 times larger than the minimized LTS).

Figure 7 shows the results of these experiments with up to 64 nodes of the `granduc` cluster. Contrary to Figure 6(a), we observe that the peak memory consumption decreases when the number of workers increases. However, the peak memory consumption is higher than for the combination of REDUCTOR and DISTRIBUTOR without τ-confluence; the only exception is "CLH" for 64 or more workers. Indeed, because activating the on-the-fly τ-confluence reduction of DISTRIBUTOR yields a significantly smaller PBG, the bottleneck concerning peak memory consumption is not the master running REDUCTOR, but the workers running DISTRIBUTOR with activated τ-confluence reduction. We observed that for very large components of τ-transitions and few workers, DISTRIBUTOR with activated τ-confluence reduction requires almost as much memory as a single sequential execution of REDUCTOR, because the complete component is explored by (at least) one worker.

Note that the size (i.e., number of states and transitions) of the reduced partitioned LTS depends on the order in which states are explored; this order depends on both the number of workers and communication latencies.

7.3 Performance Study of Caching in PBG_OPEN

Fixed-Size Caching. Figures 8 and 9 show the effects of increasing the cache size in PBG_OPEN up to one million entries, when using the fixed-size cache



(a) execution time in hours

(b) peak memory in MB

Fig. 9. Cache with increasing number of cache entries: `borderline`, `granduc`, and `suno` clusters, eight workers

implementation on a cluster (four workers) and several clusters on different sites (eight workers); figures for other numbers of workers show the same tendencies.

As expected, caching reduces execution time if the cache is large enough to hold all required transitions. This can be seen on small examples, in particular those obtained by DISTRIBUTOR with τ-confluence reduction: these examples have significantly smaller components of τ-transitions, so that one observes a reduction in execution time already for small cache sizes.

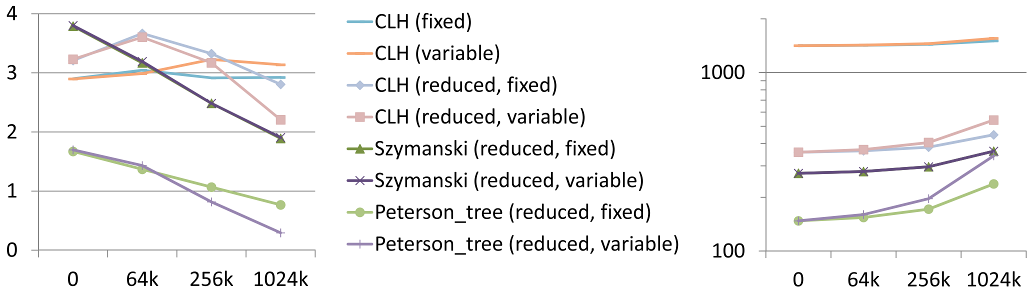
On the other hand, caching increases peak memory consumption, except for those examples where DISTRIBUTOR requires more memory than REDUCTOR. For small LTSs and many cache entries, memory consumption may become even larger than for sequential execution — namely, when the cache stores the entire LTS.

Comparison of Fixed- and Variable-Size Caching. Comparing both cache implementations with rigorously the same cache size is not simple because of the varying sizes of the lists of outgoing transitions. It is difficult to predict the memory consumption of the variable-size cache, as well as the number of states that can be stored in a fixed-size cache (because each fixed-size cache entry can store either a state or a transition). To simplify our comparisons, we chose to use the same number of cache entries (which is only an approximation for comparing caches with the same amount of memory).

Figure 10 compares both implementations for four selected examples, using two workers running PBG_OPEN/REDUCTOR on the `granduc` cluster. We observed a significant difference only for "Peterson_tree", where the variable-size cache is faster, but requires more memory.

8 Related Work

There are other approaches to parallel and distributed verification. For instance, DIVINE [4] supports distributed LTL model checking on both multi-



(a) execution time in hours

(b) memory in MB

Fig. 10. Comparison of cache implementations for increasing number of cache entries

core architectures [3], clusters, and grids [20]. The parallel extension of Spin [16] also supports multi-core LTL model checking, including partial order reduction. Another example is PREACH [5], which enables distributed reachability analysis of Murφ models. PREACH is built on top of the original sequential Murφ code, implementing distribution and communication in the distributed functional programming language Erlang. Note that flow control credit mechanism that is crucial for scalability of PREACH is built-in in our communication library, as it is provided by TCP/IP.

A key difference between these approaches and our approach is that CADP is action-based rather than state-based. Also, CADP offers tools for generating and handling distributed LTSs, which enables a wide spectrum of verification techniques to be used, including both model checking and equivalence checking (bisimulations). Let us also mention the work of Eric Madelaine and colleagues, who used DISTRIBUTOR and BCG_MERGE for the generation and on-the-fly reduction of large LTSs using the PACAGrid [8] infrastructure [14,2,1].

9 Conclusion

We presented the latest distributed verification tools recently added to the CADP toolbox in order to manipulate partitioned LTSs represented as PBG files. We experimented these new tools, together with the DISTRIBUTOR and BCG_MERGE tools previously available in CADP, on a large-scale grid involving several clusters geographically located in different places and different countries. Our experiments were intended to push the PBG machinery to its limits by using hundreds of workers and to study how this influences performance and scalability.

Our experiments confirm the finding of [13] that distributed state space generation using DISTRIBUTOR scales well up to the point where “too many cooks spoil the broth”.

[8] See `http://proactive.inria.fr/pacagrid`

We observed that the on-the-fly exploration of partitioned LTSs using PBG_OPEN avoids the memory bottleneck faced when using a single machine, and thus enables to take advantage of clusters and grids to handle much larger problems (as the peak amount of memory needed is roughly divided by the number of computing nodes).

Conversely, our experiments showed that PBG_OPEN may be slower than a sequential implementation (between two and nine times) when the on-the-fly application (e.g., REDUCTOR) is an intrinsically sequential state space exploration algorithm. We observed a significant impact of communication latencies between computing nodes, a problem that we addressed by introducing caches in PBG_OPEN.

We did not measure the consumed (peak) bandwidth: similar to the observations reported in [5], we never found bandwidth to be a bottleneck. The heterogeneity of the clusters clearly impacts the experiments. On the one hand, the worker with the smallest amount of available memory determines if the computation succeeds (because the static hash function distributes states uniformly over the workers). On the other hand, measuring speedup becomes imprecise if one worker runs on a faster or slower processor than the sequential reference execution. However, we found it interesting to experiment how a static work distribution can cope with this heterogeneity.

This work can be pursued along several directions. Firstly, the network communication library of CADP could be optimized to support parallel (rather than sequential) file transfers during the initialization, which would remove the overhead observed when using hundreds of workers. Secondly, static load-balancing techniques could be investigated by specializing the static hash function used for assigning states to workers. Thirdly, one could experiment PBG_OPEN with a truly parallel OPEN/CAESAR application (such as DISTRIBUTOR with its τ-confluence reduction algorithm, or the distributed version of the EVALUATOR 4.0 [18] on-the-fly model checker) rather than a sequential application (such as the REDUCTOR used in the present paper).

Acknowledgments. The experiments presented in this paper were carried out using the Grid'5000 experimental testbed [9] built by Inria with support from CNRS, RENATER, several Universities, and other funding bodies. We are also grateful to Iker Bellicot, Nicolas Descoubes, Jerôme Fereyre, Yann Genevois, and Rémi Hérilier for their contribution in testing and bug-hunting of the CADP distributed verification tools.

[9] See `http://www.grid5000.fr`

References

[1] Ameur-Boulifa, R., R. Halalai, L. Henrio and E. Madelaine, *Verifying Safety of Fault-Tolerant Distributed Components – Extended Version*, Rapport de recherche RR-7717, INRIA (2011). URL `http://hal.inria.fr/inria-00621264/en/`

[2] Ameur-Boulifa, R., L. Henrio and E. Madelaine, *Behavioural models for group communications*, in: *Proc. of the Int. Workshop on Component and Service Interoperability, WICS'10*, (2010).

[3] Barnat, J., L. Brim and P. Ročkai, *Scalable Multi-core LTL Model-Checking*, in: *Proc. of the 14th Int. SPIN Workshop*, LNCS **4595** (2007), pp. 187–203.

[4] Barnat, J., L. Brim, M. Češka and P. Ročkai, *DiVinE: Parallel distributed model checker*, in: *Proc. of the 9th Int. Workshop on Parallel and Distributed Methods in Verification PDMC 2010* (2010), pp. 4–7.

[5] Bingham, B., J. Bingham, F. M. de Paula, J. Erickson, G. Singh and M. Reitblatt, *Industrial Strength Distributed Explicit State Model Checking*, in: *Proc. of the 9th Int. Workshop on Parallel and Distributed Methods in Verification PDMC 2010* (2010), pp. 28–36.

[6] Blom, S., I. van Langevelde and B. Lisser, *Compressed and Distributed File Formats for Labeled Transition Systems*, in *Proc. of the 2nd Int. Workshop on Parallel and Distributed Model Checking PDMC'2003*, ENTCS **89** (2003), pp. 68–83.

[7] Cadp, *Distributor manual page* (2004). URL `http://cadp.inria.fr/man/distributor.html`

[8] Cadp, *Pbg_info manual page* (2012). URL `http://cadp.inria.fr/man/pbg_info.html`

[9] Cappello, F., E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet and O. Richard, *Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform*, in: *Proc. of the 6th Int. Workshop on Grid Computing GRID'2005* (2005), pp. 99–106.

[10] Garavel, H., *Open/Cæsar: An open software architecture for verification, simulation, and testing*, in: *Proc. of the 1st Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS **1384** (1998), pp. 68–84.

[11] Garavel, H., F. Lang, R. Mateescu and W. Serwe, *Cadp 2011: A toolbox for the construction and analysis of distributed processes*, STTT (2012), DOI 10.1007/s10009-012-0244-z.

[12] Garavel, H., R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm and G. Stragier, *Distributor and bcg_merge: Tools for distributed explicit state space generation*, in: *Proc. of the 12th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, LNCS **3920** (2006), pp. 445–449.

[13] Garavel, H., R. Mateescu and I. Smarandache, *Parallel state space construction for model-checking*, in: *Proc. of the 8th Int. SPIN workshop*, LNCS **2057** (2001), pp. 217–234.

[14] Henrio, L. and E. Madelaine, *Experiments with distributed model-checking of group-based applications*, Technical report, INRIA Sophia-Antipolis. Presented at the Sophia-Antipolis Formal Analysis Group 2010 Workshop, SAFA2010 (2010).

[15] Hermanns, H., *Interactive Markov Chains and the Quest for Quantified Quality*, LNCS **2428** (2002).

[16] Holzmann, G. J., *Parallelizing the Spin Model Checker*, in: *Proc. of the 19th Int. SPIN workshop*, LNCS **7385** (2012), pp. 155–171.

[17] Mateescu, R. and W. Serwe, *Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols*, Science of Computer Programming (2012). URL `http://dx.doi.org/10.1016/j.scico.2012.01.003`

[18] Mateescu, R. and D. Thivolle, *A model checking language for concurrent value-passing systems*, in: *Proc. of the 15th Int. Symp. on Formal Methods FM 2008*, LNCS **5014** (2008), pp. 148–164.

[19] Mateescu, R. and A. Wijs, *Hierarchical adaptive state space caching based on level sampling*, in: *Proc. of the 12th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2009*, LNCS **5505** (2009), pp. 215–229.

[20] Verstoep, K., H. E. Bal, J. Barnat and L. Brim, *Efficient Large-Scale Model Checking*, in: *Proceedings of the 23rd Int. Symp. on Parallel and Distributed Processing IPDPS 2009* (2009), pp. 1–12.