

Efficient Large-Scale Model Checking*

Kees Verstoep, Henri E. Bal

Dept. of Computer Science, Fac. of Sciences
VU University, Amsterdam, The Netherlands
{versto,bal}@cs.vu.nl

Jiří Barnat, Luboš Brim

Dept. of Computer Science, Fac. of Informatics
Masaryk University, Brno, Czech Republic
{barnat,brim}@fi.muni.cz

Abstract

Model checking is a popular technique to systematically and automatically verify system properties. Unfortunately, the well-known state explosion problem often limits the extent to which it can be applied to realistic specifications, due to the huge resulting memory requirements. Distributed-memory model checkers exist, but have thus far only been evaluated on small-scale clusters, with mixed results. We examine one well-known distributed model checker, DiVinE, in detail, and show how a number of additional optimizations in its runtime system enable it to efficiently check very demanding problem instances on a large-scale, multi-core compute cluster. We analyze the impact of the distributed algorithms employed, the problem instance characteristics and network overhead. Finally, we show that the model checker can even obtain good performance in a high-bandwidth computational grid environment.

1 Introduction

One of the main challenges in the field of computer science is to provide formalisms, techniques, and efficient tools for assessing the correctness or other functional properties of increasingly complex computer systems. One such technique is model checking, which systematically (and automatically) checks whether a model of a given system satisfies a desired property. This automated technique for verification and debugging has developed into a mature and widely used approach.

Conventional sequential model checking techniques have high memory requirements and are very computationally intensive; they are thus unsuitable for handling real-world systems that exhibit complex behaviors which cannot be captured by simple models having a small or regular state space. Various authors have proposed ways of solving

this problem by either using powerful shared-memory multiprocessors (e.g., multi-core machines) or by distributing the memory requirements over several machines (e.g., on a cluster of workstations).

Memory requirements are often the bottleneck in being able to solve a problem at all. Therefore, it can still be beneficial to use algorithms with a slightly higher computational complexity, provided they can be distributed effectively using a large distributed memory. A prominent example in this category is the DIVINE [1] system, which we will focus on in this paper. As DIVINE is especially targeted on model specifications that induce very large state spaces, an important question is to what extent it scales to a large number of compute nodes. Previous research has shown that the different distributed algorithms included in DIVINE can have widely diverse execution times, depending on the model characteristics [1]. We will closely examine two different algorithms that previously were shown to have the best overall performance, and we will analyze their behavior on model instances that require significantly more memory than the ones tackled before.

Models with large search spaces arise naturally from a straightforward formalization of a system under development. To make complete (finite) analysis of such models possible, often simplifying assumptions have to be introduced, with the unfortunate risk of certain inconsistencies escaping analysis. Typically, also, models have to be made amenable for analysis by putting an artificial boundary on the number of resources or processes involved. By scaling the model up from small instances to more realistic proportions, gradually more trust can be gained in the verification results. However, seemingly simple, restricted specifications can still give rise to unexpectedly huge search spaces, also known as the *state explosion* problem. Although abstraction techniques exist which restrict models to their essential core (without losing behavioral characteristics that do require checking), large-scale analysis is often still a necessity in practical cases. For example, the checking of routing protocols for mobile ad hoc networks [29] resulted in verification of various scenarios, several of which could

*This work has been supported in part by the Czech Science Foundation, grants No. 201/09/1389 and 201/09/P497.

not be verified using the efficient (sequential) SPIN [16] model checker, due to their very large state spaces. As DIVINE also supports SPIN specifications [28], additional scenarios can now be verified using a cluster.

The contributions of this paper are as follows. We describe and analyze several optimizations for the DIVINE framework and two of its algorithms that together improve their performance up to 50%. We show that these optimizations allow the algorithms to scale well, up to at least 256 cores, and that they can efficiently exploit modern multi-core architectures. We compare the performance of both parallel algorithms on seven representative models having different characteristics, all exhibiting state spaces that are much larger than could be tackled before. We analyze the sensitivity of the algorithms to protocol overhead of the network used, as this can typically have a large impact on parallel performance. Finally, we show that DIVINE, which is largely implemented using asynchronous communication, can now even be run efficiently on a large-scale optical computational grid, despite the much higher (wide-area) latencies on such a platform.

The paper is structured as follows. In Section 2 we examine DIVINE and two of its main parallel algorithms, and we discuss their communication patterns. Section 3 discusses the optimizations we applied to DIVINE, and their effectiveness in improving the performance of both algorithms. Next, Section 4 contains a performance analysis of the optimized model checker on six additional realistic problems with search spaces up to 245 GB. In Section 5 we discuss related work and conclude.

2 Distributed-Memory Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. The check itself is in principle an exhaustive search in the model. The main technical problem in model checking is the *state explosion* which can occur if the system being verified has many components which make transitions in parallel. The size of the constructed model grows exponentially in the size of the system’s description.

Much attention has been paid to the development of approaches to battle the state explosion problem. Many techniques, such as abstraction, state compression, partial order reduction, symbolic state representation, etc., are used to reduce the size of the model, thus allowing a single computer to still process large systems. However, despite impressive progress on these reduction techniques, the memory required to handle large industrial models still exceeds the capacities offered by a single contemporary computer.

One possible approach is to increase the computational power and memory capacity of the system by using a com-

pute cluster, in which the compute nodes communicate via a message passing interface. The use of distributed-memory processing for model checking indeed has gained interest in recent years. Techniques have been developed for both explicit and symbolic model checking, analysis of stochastic and timed systems, equivalence checking and other verification methods.

2.1 LTL Model Checking

In this paper we consider one particular model-checking procedure, namely *enumerative LTL model checking*. In LTL model checking, the properties are specified in Linear Temporal Logic, which is a temporal logic suitable to express properties about the future of executions of the system model, e.g., that a condition will eventually be true, or that a condition will be true until another fact becomes true, etc. An efficient procedure to decide LTL model checking problems is based on automata and was introduced by Vardi and Wolper [26]. In this approach, both the model and the LTL formula are associated with an *automaton*, and the LTL model-checking problem is reduced to detecting an *accepting cycle* (i.e., a cycle in which one of the vertices is marked “accepting”) in the combined *automaton graph*.

The optimal sequential algorithms for accepting cycle detection use depth-first search (DFS) strategies. The individual algorithms differ in their space requirements, length of the counterexample produced, and other aspects. The well-known *Nested DFS* algorithm is used in many model checkers and is considered to be the best suitable algorithm for enumerative *sequential LTL* model checking. The algorithm was proposed by Courcoubetis et al. [9] and its main idea is to use two interleaved graph searches to detect reachable accepting cycles. The first search discovers accepting states, while the second (the nested one) checks for self-reachability. Another group of optimal algorithms are *SCC-based algorithms* originating in Tarjan’s algorithm for the decomposition of the graph into Strongly Connected Components (SCCs) [25]. While Nested DFS is more space efficient, SCC-based algorithms produce shorter counterexamples in general, which can thus be analyzed more conveniently. The time complexity of these algorithms is linear in the size of the graph, i.e., $O(m + n)$, where m is the number of edges and n is the number of vertices.

The effectiveness of the *Nested DFS* algorithm is achieved due to the particular order in which the graph is explored, also guaranteeing that vertices are not re-visited more than twice. In fact, all best-known algorithms rely on the same exploration principle, namely the *postorder* as computed by the DFS. It is a well-known fact that the post-order problem is P-complete and, consequently, a scalable parallel algorithm which would be directly based on DFS postorder is unlikely to exist.

An additional important criterion for a model checking algorithm is whether it works *on-the-fly*. On-the-fly algorithms generate the automaton graph gradually as they explore vertices of the graph. An accepting cycle can thus be detected before the complete set of vertices is generated. On-the-fly algorithms usually assume the graph to be given *implicitly* by the function F_{init} giving the initial vertex and by the function F_{succ} which returns immediate successors of a given vertex.

2.2 Parallel Algorithms for LTL Model Checking

In many cases the algorithms as used traditionally are not appropriate to be adapted to parallel architectures. In the case of LTL model checking, all efficient algorithms build on depth-first search exploration of the state space. However, there is no known way to efficiently compute DFS postorder on parallel machines. New algorithms have to be invented to replace the classical ones. We briefly introduce two algorithms for accepting cycle detection that are (among others) implemented in DIVINE. The sequential complexity of these algorithms is worse than for those based on DFS, but both allow solving the LTL model-checking problem on parallel architectures much more efficiently. For a detailed survey on these and other algorithms implemented in DIVINE we refer to [1].

OWCTY: Topological Sort Algorithm

The main idea behind the OWCTY (One Way Catch Them Young) algorithm stems from the fact that a directed graph can be topologically sorted if and only if it is acyclic. The core of the cycle detection algorithm is thus an application of the standard linear topological sort algorithm to the input graph. Failure in topologically sorting the graph means the graph contains a cycle. Accepting cycles are detected with multiple rounds (iterations) of the topological sort. Every iteration consists of reachability and elimination procedures. The reachability procedure removes vertices unreachable from an accepting vertex (as these cannot belong to an accepting cycle) and computes indegrees for all remaining vertices. The succeeding elimination procedure recursively eliminates vertices whose predecessor count drops to zero. The algorithm does not work on-the-fly, as the entire automaton graph has to be generated first. Also, the algorithm does not immediately give the accepting cycle; it only checks for its *presence* in the graph. However, the counterexample is easily generated using two additional linear graph traversals, like breadth-first search.

The time complexity of the algorithm is $O(h \cdot m)$ where h is the height of the SCC graph. Here the factor m comes from the computation of the *reachability* and *elimination*

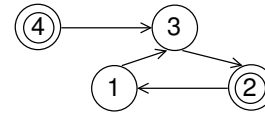


Figure 1. Undiscovered cycle

functions and the factor h relates to the number of external iterations. In practice, the number of external iterations is very small (up to 40–50), even for very large graphs. This observation is supported by experiments in [11]. Similar results are communicated in [22] where heights of SCC graphs were measured for several models. As reported, 70% of the models have heights smaller than 50.

A positive aspect of the algorithm is its extreme effectiveness for *weak automaton graphs*. A graph is weak if in each SCC all the states are accepting or none of them is. For weak graphs only one iteration of the algorithm is necessary to decide about accepting cycles, the algorithm works in linear time and is thus optimal. A study of temporal properties [8] has revealed that verification of up to 90% of LTL properties leads to weak automaton graphs.

MAP: Maximal Accepting Predecessors

The main idea behind the MAP algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own predecessor. The algorithm that would be directly derived from this idea requires expensive storing of all proper accepting predecessors for each (accepting) vertex. To remedy this, the algorithm instead stores only a single representative accepting predecessor for each vertex. We presuppose a linear ordering of vertices (given, e.g., by their memory representation) and choose the *maximal accepting predecessor*. For a vertex u we denote its maximal accepting predecessor in the graph G by $map_G(u)$. Clearly, if an accepting vertex is its own maximal accepting predecessor ($map_G(u) = u$), it lies on an accepting cycle. Unfortunately, the opposite does not hold in general. It can happen that the maximal accepting predecessor for an accepting vertex on a cycle does not lie on the cycle. This is exemplified in the graph given in Fig. 1. The accepting cycle $\langle 2, 1, 3, 2 \rangle$ is not revealed due to the greater accepting vertex 4 outside the cycle. However, as vertex 4 does not lie on *any* cycle, it can safely be deleted (marked as non-accepting) from the set of accepting vertices, and the accepting cycle still remains in the resulting graph. This idea is formalized as a *deleting transformation*.

Whenever the deleting transformation is applied to the automaton graph G with $map_G(v) \neq v$ for all $v \in V$, it shrinks the set of accepting vertices by those vertices that do not lie on any cycle. As the set of accepting vertices can change after the deleting transformation has been applied,

```

while (!synchronized()) {
  if ((state = waiting.dequeue()) != NULL) {
    state.work();
    for (tr = state.succs(); tr != NULL; tr = tr.next()) {
      tr.work();
      newstate = tr.target();
      dest = newstate.hash();
      if (dest == this_cpu) waiting.queue(newstate);
      else send_work(dest, newstate);
    }
  }
  else idle();
  process_messages(&waiting);
}

```

Figure 2. Distributed graph traversal skeleton

the maximal accepting predecessors must be recomputed. It can happen that even in the graph $del(G)$ the maximal accepting predecessor function is still not sufficient for cycle detection. However, after a finite number of iterations consisting of computing maximal accepting predecessors followed by application of the deleting transformation, an accepting cycle is certified (after which a counterexample can be reconstructed). For an automaton graph without accepting cycles, the application of deleting transformations results in an automaton graph without accepting vertices.

The time complexity of the algorithm is $O(a^2 \cdot m)$, where a is the number of accepting vertices. Here the factor $a \cdot m$ comes from the computation of the *map* function and the factor a relates to the number of iterations. Unlike the OWCTY algorithm, the MAP algorithm does work on-the-fly. Experimental evaluation of this algorithm demonstrated that accepting cycles were typically detected in a very small number of iterations. On the other hand, if there is no accepting cycle in the graph, the number of iterations tends to be very small compared to the size of the graph (up to 40–50). Thus, the algorithm exhibits near linear performance in practice.

2.3 DiVinE Tool

The DiVinE toolkit consists of several separate implementations of various LTL model checking algorithms such as described above. Although the algorithms are very different, they follow the same overall pattern, illustrated in Figure 2. All algorithms perform a strict-order independent repeated traversal of a directed graph. Vertices of the graph are very small (typically less than 1 KB), but there are many. To distribute work among compute nodes, the tool partitions the graph into (disjunct) sets of vertices such that each set is owned by one node. This partitioning is implemented using a hash function: every vertex is assigned to a compute node according to the hash value computed from its state representation. Due to the large number of vertices to be distributed, the hash-based partitioning scheme results in a

quite well-balanced workload, at the price of minimal locality. The probability that immediate descendants of a vertex belong to the same compute node as the vertex is $1/p$, where p is the number of compute nodes. This means in practice, that significant portions of edges of the graph are so called *cross edges*, i.e., edges whose incident vertices belong to different compute nodes. Basically, every cross edge results in a message to be sent from the compute node owning the source vertex of the edge to the node owning the target vertex of the edge. The message bears information about the explored edge plus a small amount of additional data that is dependent on the algorithm involved. As a result, a huge number of small messages is exchanged among compute nodes during the execution of a DiVinE tool.

Both OWCTY and MAP show a gradually increasing memory usage during their first exploration phase, where the state space is being expanded. During subsequent application phases, memory usage remains constant, as the algorithm-specific state meta-data is preallocated during the first phases. Overall, generating, hashing and comparing states is responsible for a large fraction of the applications’ runtime. Furthermore, large-scale graph algorithms like explicit-state model checking have a high data access to computation ratio compared to scientific computing applications [19].

Distributed Graph Traversal

As shown in Figure 2, the core of each graph traversal algorithm is a while loop over a queue of vertices (states) waiting to be processed. Each time a vertex is dequeued, edges (transitions) emanating from it are enumerated, and for each of them an algorithm-related action is performed. The target vertices are examined, and if they need to be stored locally, they are inserted back into the queue. Non-local vertices are wrapped into messages and sent to their owners. In the serial case the main loop terminates as soon as the queue becomes empty. For distributed algorithms, however, the processing of incoming messages produces new vertices to be inserted into the queue, thus introducing new work. Therefore, the parallel algorithm may terminate only if all local queues are empty and there is no message in transit. To detect this termination condition, Safra/Dijkstra’s distributed termination detection algorithm [10] is used; see also [21].

An important observation is that the communication among compute nodes is asynchronous: the algorithms described simply push work to other compute nodes, without triggering replies that require more processing. This aspect also enables an important optimization: work items sent to the same destination can be aggregated into larger messages, significantly reducing the communication overhead. DiVinE implements the communication using asyn-

chronous MPI primitives, which allows for efficient parallel processing on a wide variety of architectures. On the other hand, the use of asynchronous messages may increase the memory demands, both at the application and the communication layer. The more vertices are enqueued in a local queue, the longer it takes before incoming messages are actually received. Since the number of messages is limited by the number of edges, we can observe a shift in the space complexity of the algorithms. Unlike the serial case, where the space complexity of a graph traversal algorithm is asymptotically linear in the number of vertices, the distributed algorithms exhibit space complexity that is asymptotically linear in the number of vertices and edges, hence, up to asymptotically quadratic in the number of vertices. Experimental experience has shown that even for graphs with a relatively small number of transitions (with an average outdegree less than 10), the practical memory demands are significantly increased in the distributed case compared to the serial one, due to incoming message buffering.

To avoid increased memory demands during computation, DiVINE algorithms regularly check for incoming messages. If the content of an incoming message indicates further processing, the appropriate vertex is extracted from the message and it is enqueued to the local queue. If there is no further processing required for the incoming message, the message is discarded immediately.

3 Optimizing DiVINE's Performance

The performance of DiVINE was considered to be reasonably good, but had not yet been evaluated on large-scale parallel systems. The original evaluation [1] had investigated performance up to 20 compute nodes, which was the size of the cluster used for the development of DiVINE. For this paper, we were able to make use of the DAS-3 system (discussed below), allowing performance evaluation at a much larger scale. It should be noted that that the applications are far from trivially parallel, as they are very communication intensive, as shown later in this Section. It was soon determined that performance of several DiVINE algorithms did not scale well with a high number of nodes. It was unclear whether this was possibly caused by inherent scalability limitations of the underlying (distributed) algorithms. For example, in the elimination phase of the OWCTY algorithm, the states of the graph must be expanded in topological order. It was previously unclear, whether strictly following this order would force some cores to become idle due to an insufficient amount of work. As our results will show, this is not the case in practice.

An important reason for the initial scalability to drop, was found to be an inefficiency in DiVINE's timer management for its user-level messaging layer. Even though the associated system calls are highly optimized in the

Linux kernel, the extent to which they were used still seriously impeded performance when many compute nodes were used. By modifying the timer management to use a cached version of the current time where appropriate (optimization `TIMER`), large-scale performance was improved significantly.

3.1 Optimizations Applied

To investigate possibly remaining performance problems, we started with a bottom-up approach in which DiVINE's networking module (shared by the implementations of parallel algorithms, including MAP and OWCTY) was first instrumented for performance analysis. Every MPI invocation was wrapped in a low-overhead layer that maintained statistics about the call's overhead (e.g., to determine send and receive overhead), and about data transfer rates (both incoming and outgoing). The most important statistics were logged once every three seconds on every CPU core. The combined data was then graphically analyzed to hypothesize causes for the performance degradations, upon which action could be taken. The same approach was recently successfully applied on a distributed application from an entirely different domain (distributed game tree search) that showed a traffic pattern remarkably similar to DiVINE's [27].

DiVINE's receive primitive, called *process_messages* (see Figure 2), was an important target in several of our optimizations. Besides implementing polling, receiving and processing user messages, it is also responsible for timeout-based flushing of pending messages and the handling of distributed termination detection. The following optimizations (with acronyms for reference) were applied:

Auto-tune receive rate (RATE) – DiVINE's applications often performed much more polling than necessary. This aspect surfaced as a high overall `MPI.poll` failure rate. Straightforward reduction of the number of polls can already improve performance substantially, but statically determining the optimal polling rate is quite hard. Typically it depends on many factors (besides the host and network hardware, the messaging middleware, the application, the problem instance, etc.), but it can also change over the application's runtime. The optimized version of *process_messages* thus *dynamically* changes the polling rate, based on the actually experienced message arrival rate. Note that as the data transfers occur in an essentially unpredictable order, blocking receives at the MPI layer cannot be applied effectively, since this would introduce additional delays.

Prioritize I/O tasks (PRIO) – DiVINE implements timeout-based flushing of pending work to improve performance by providing other nodes with additional work in cases where message combining would otherwise postpone

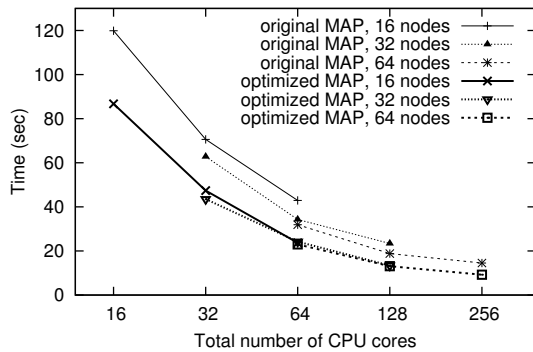


Figure 3. Optimization effects for MAP

transfers too long. Furthermore, distributed termination detection is a requirement for DIVINE’s correct functioning, but involves a separate MPI communication channel and its associated polling overhead. Both these tasks in *process_messages* were not timing-critical, yet originally were performed each time the primitive was invoked. In the optimized version, both overheads were reduced by only executing the associated code a suitably small fraction of the time.

Optimize message flushing (FLUSH) – Another aspect that was optimized, is the flushing of messages during the applications runtime, including distributed termination phases (as discussed in Section 2, each DIVINE algorithm has several of these phases). When running out of local work, the original implementation simply flushed *all* outstanding (non-full) messages in a fixed sequential order, which caused much congestion due to hotspots in the network and at the receivers. Also, as every message was flushed indiscriminately, the average message size during distributed termination detection phases dropped substantially, causing a relatively high overhead. Message flushing was optimized quite similarly to the Awari application [27]. In the new version, messages are now flushed from large to small – effectively spreading the traffic over the network – also taking care not to exceed a reasonable upperbound in the outgoing traffic rate – thus avoiding the syndrome of frantically sending tiny messages.

Pre-establish network connections (PRESYNC) – A final optimization is of a quite different nature. As will be discussed in Section 4.3, DIVINE is also suitable for running on wide-area distributed systems, but it was noticed that during large-scale grid experiments some endpoints would often fail to start communicating efficiently. Sometimes this situation could prolong such that this led to a huge backlog of MPI messages at the sender, eventually causing the application to fail due to excessive paging. The cause of the problem is that the MPI implementation used (Open MPI [14]) establishes TCP connections on-demand. This

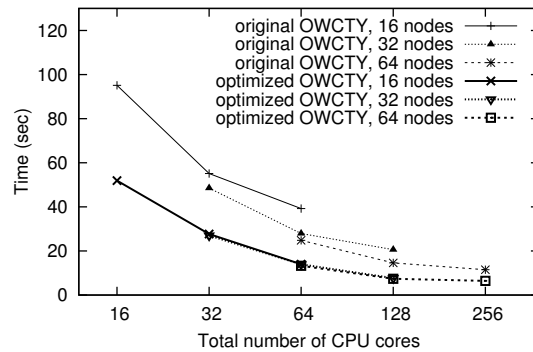


Figure 4. Optimizations effects for OWCTY

is often a useful feature, as it decreases large-scale MPI initialization time, and often only a small subset of the endpoints communicate point-to-point. However, DIVINE is atypical in the sense that it requires *every* endpoint to communicate with every other endpoint. Also, immediately after startup, it starts communicating at peak data rates. These data rates can be such that they can fill almost the entire capacity of the wide-area network between sites, making further connection-establishment very difficult due to timeouts. This issue was resolved by the addition of a (by purpose) naively implemented small all-to-all data exchange at the initialization of DIVINE’s runtime system, when the network is still uncongested. This forces all network connections to be readily available when the actual data transfers start.

3.2 Impact of the Optimizations

Our performance evaluation was done on the Distributed ASCI Supercomputer [7] (DAS-3), a wide-area distributed system for Computer Science research in the Netherlands. DAS-3 consists of five clusters distributed over four sites. DAS-3 uses Myri-10G networking technology from Myri-com both as an internal high-speed interconnect as well as an interface to remote DAS-3 clusters. DAS-3 is largely homogeneous: every cluster uses dual-CPU AMD Opteron nodes, but with different clock speeds and/or number of CPU cores. For the single-cluster performance evaluations in this paper we used the DAS-3 cluster at VU University, since it has the largest number of compute nodes and cores (85 nodes with a dual-cpu, dual-core 2.4 GHz AMD Opterons).

Figures 3 and 4 show the optimization effects for MAP and OWCTY on an increasing number of DAS-3/VU compute nodes, using Myri-10G’s native MX layer for communication. Results are shown for 1, 2 and 4 cores per compute node. We used LTL problem instance 6 of the Anderson specification from the BEEM benchmark set [23] (this

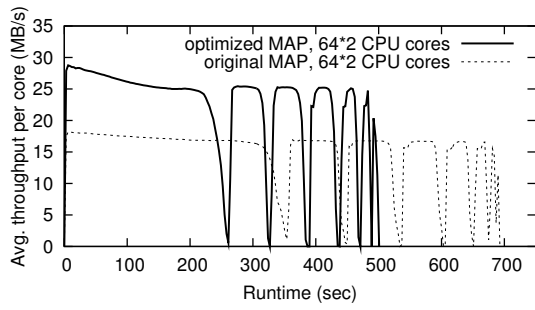


Figure 5. Per-core throughput of MAP

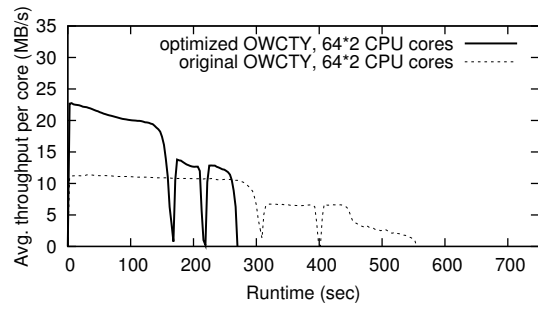


Figure 6. Per-core throughput of OWCTY

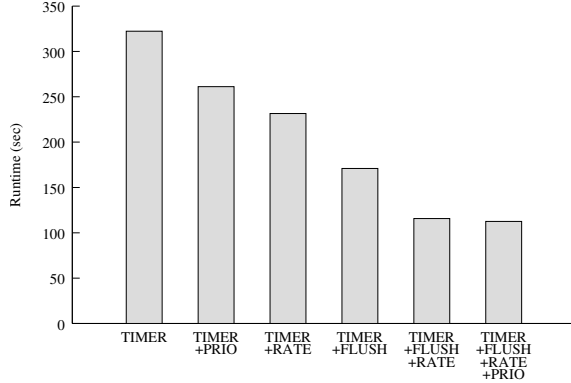


Figure 7. Impact of the individual optimizations for OWCTY on 64*4 cores

specification concerns the correctness of a mutual exclusion algorithm). In this case the state space has to be searched completely, but it is still small enough to fit into the memory of 16 compute nodes, making a scalability comparison feasible. There are several conclusions to be drawn from these figures:

- The performance for both the MAP and the OWCTY implementation has increased substantially: they are about 30–50% faster than before;
- The scalability of both algorithms is quite good: by increasing the number of cores with a factor 16 (from 16 to 256 cores), both MAP and OWCTY run about a factor 10 faster;
- Due to reduced overheads, the performance of the optimized version is almost insensitive to the placement of multiple processes on the same node, unlike the original version.

Figures 5 and 6 illustrate the effect of the optimizations on the applications' throughput as a function of their runtime. In this case a larger instance of the Anderson problem was used to obtain a higher resolution graph.

Nodes	Total cores	Runtime (s)		Efficiency	
		MAP	OWCTY	MAP	OWCTY
1	1	956.8	628.8	100%	100%
16	16	73.9	42.5	81%	92%
16	32	39.4	22.5	76%	87%
16	64	20.6	11.4	73%	86%
64	64	19.5	10.9	77%	90%
64	128	10.8	6.0	69%	82%
64	256	7.4	4.3	51%	57%

Table 1. Efficiency of MAP and OWCTY

In the case of MAP (Figure 5), the throughput graph clearly shows the application-specific phases. The first phase (originally taking 354 seconds, optimized 261 seconds), the state space is constructed on-the-fly, besides applying the MAP algorithm itself (see Section 2.2). It is therefore taking significantly longer than the subsequent phases. The graph clearly shows that peak throughput is maintained almost throughout the application's runtime, and that the optimizations have let the average per-core throughput increase from 15.0 to 22.1 MByte/s.

Likewise, the throughput graph for OWCTY (Figure 6) shows the application-specific phases. As in the case for MAP, in the first phase (lasting resp. 309 and 168 seconds) the state space is constructed on-the-fly. After that, OWCTY here only requires two smaller phases to complete (as is true for many specifications, see Section 2.2). However, note the long tail of the last phase, which is due to inefficiencies in the original termination detection implementation. For OWCTY, the average per-core throughput increases from 7.9 to 16.4 MByte/s.

To investigate the *relative* impact of the optimizations, we constructed a version of DIVINE where combinations of individual optimizations discussed above could be enabled dynamically at runtime. The results for OWCTY using 256 cores on the larger instance of the Anderson model are shown in Figure 7. The version with TIMER optimization was used as a baseline, since this modification is re-

Table 2. Large-scale models used

Model	Description	Specification	State space
Elevator	Elevator controller correctness	elevator.4.prop2 (scaled)	123.8 GB
Publish-subscribe	Groupware protocol	public-subscribe.5.prop1	209.7 GB
AT	Timing based mutual exclusion	at.7.prop2	245.0 GB
Le Lann	Token ring leader election	lann.8.prop1	> 320 GB
GIOP	CORBA General Inter-Orb Protocol	scenario 1, property 3	203.8 GB
Lunar	Ad hoc routing protocol	scenario 4d; two properties	181.6 GB

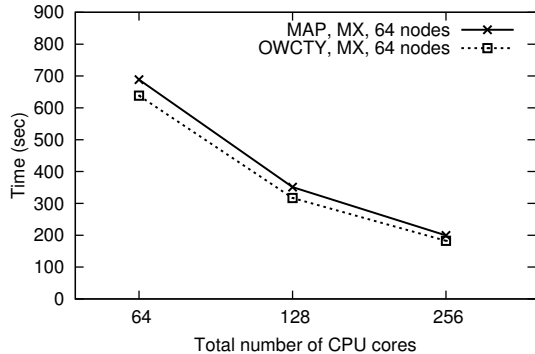


Figure 8. Publish-subscribe on 64 nodes

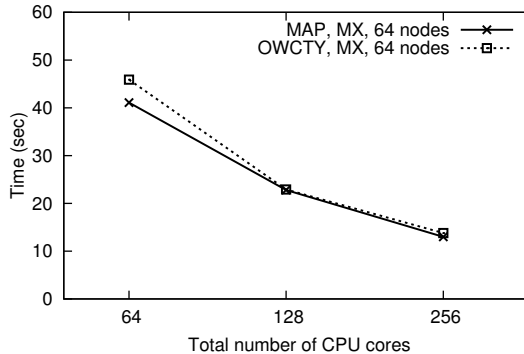


Figure 9. Lunar on 64 nodes, property 1

quired to still get reasonable speedup as the number of CPU cores grows. As shown, optimization FLUSH has the highest impact on performance, but RATE and PRIO also have significant impact. Optimization PRESYNC (not shown) only has impact when the network contains a bottleneck, such as for grid configurations discussed in Section 4.3.

It should be noted that some of the optimizations are not independent: optimization RATE can by itself reduce the polling rate such that optimization PRIO becomes less effective (or required). Interestingly, both polling optimizations turn out to have less impact for MAP, but this is explained by the fact that MAP already applies an (ad-hoc) polling rate reduction at the application level. However, for both OWCTY and MAP, when a network with higher host overhead is employed (e.g., a TCP/IP network as discussed later in this paper), enabling both RATE and PRIO is still required to obtain good performance.

To estimate the efficiency of MAP and OWCTY, we ran single-core versions on a special DAS-3/VU node equipped with 16 GB memory (the regular compute nodes have 4 GB, which is insufficient to store the state space there). The results are shown in Table 1. Considering the high data exchange rates of the fine-grained applications, and the highly demanding traffic pattern (irregular all-to-all), these results can be considered quite good. Also note that this is still a reasonably small problem: DIVINE’s efficiency increases further with problem size as the relative impact of synchronizations between the application phases then lessens.

4 Scalability of Optimized DIVINE

Besides being able to analyze medium-scale models efficiently, another important use case for DIVINE is dealing with problems that simply are too large to fit into the main memory of (typical) small-scale computer systems. In this section we will therefore focus on the performance of MAP and OWCTY on a diverse set of large-scale models. We will look into scalability aspects, and also examine the impact of network overhead.

An overview of the model specifications used is shown in Table 2. The first four models (Elevator, Publish-subscribe, AT, and Le Lann) are written in DIVINE’s native modeling language “DVE”, and are taken from the on-line BEEM database [23]. The last two models are examples of realistic specifications written in Promela, the SPIN modeling language. These models are related to protocols for the CORBA architecture (General Inter-Orb Protocol, GIOP [17]) and Ad hoc routing (Lunar [29]), for which we examine two different LTL properties. In DIVINE, Promela specifications are handled using the embedded “NIPS” module. NIPS is a complete reimplementa-tion of the original SPIN tool, by means of a specially developed model-checking *virtual machine* [28].

In two models, AT and Le Lann, the LTL formula being verified is false, i.e., there exists a counterexample that has to be found by the DIVINE tool. In all other cases, the LTL formula provided is valid in the model, as a result of

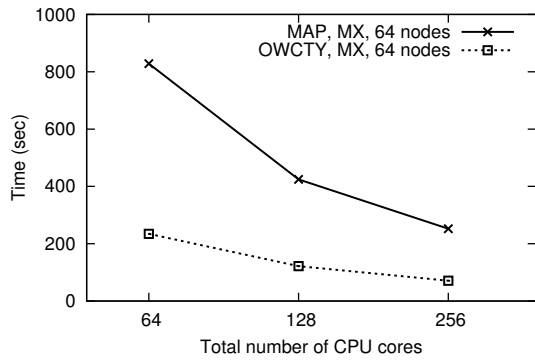


Figure 10. Elevator on 64 nodes

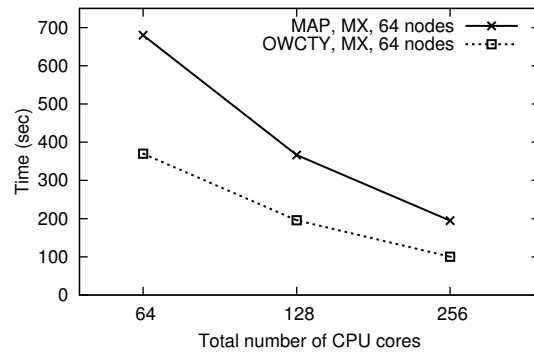


Figure 11. GIOP on 64 nodes

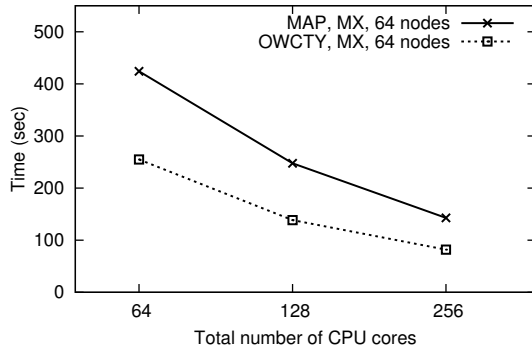


Figure 12. Lunar on 64 nodes, property 2

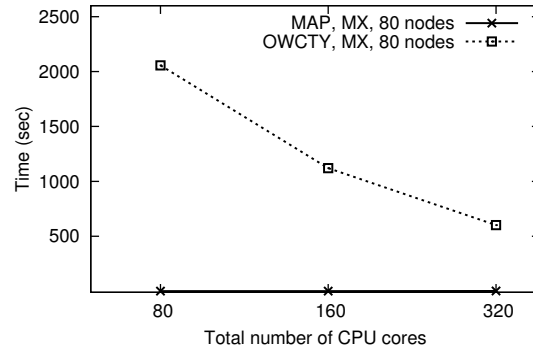


Figure 13. AT on 80 nodes

which the entire state space has to be built and analyzed for the presence of accepting cycles. The state space memory requirements shown are the ones reported by OWCTY; the algorithm-dependent per-state memory overhead for MAP is somewhat lower, reducing its overall memory requirements on average by 14%.

4.1 DAS-3 Cluster Performance

We will now show results of 1-, 2- and 4-core configurations of the DAS-3/VU cluster, using MX over Myri-10G like in the previous section. We used 64 nodes unless (when noted) the search space was so big that 80 nodes were required to store it in main memory.

The specifications requiring a full space search are shown in Figures 8 – 12. Interestingly, two groups of specifications with similar performance patterns can be identified: for Publish-subscribe and the first Ad hoc routing specification, OWCTY and MAP are about equally fast. In contrast, for Elevator, GIOP and the second Ad hoc routing specification, OWCTY is much faster than MAP, but both show good scalability when increasing the number of cores. The reason is that in the case of Publish-subscribe, MAP only requires 9 very short cycle-searching phases after the

first on-the-fly one. For the second Ad hoc routing specification, the property is even known without either OWCTY or MAP having to start any additional phases, therefore their performance is about equal. This should be contrasted with the second group of specifications, where, e.g., MAP on Elevator requires 21 longer phases and OWCTY only needs three phases.

The two inconsistent specifications show a rather different picture. As seen in Figure 13, MAP is extremely quick in finding the counterexample: it is at the bottom of the graph, taking only a few seconds. OWCTY requires a very expensive preparation phase constructing the entire search space, which is large enough that 80 compute nodes are required. The Le Lann specification (graph not included) even has a search space too large to fit on DAS-3/VU so OWCTY is unable to find the counterexample. On the other hand, like for AT, MAP is able to find it in a matter of seconds.

4.2 Network Impact

In this section we compare DIVINE's performance using Myri-10G's native MX interface with performance using TCP/IP over the same network. We use TCP/IP to provide insight into DIVINE's performance on a higher-overhead

MPI primitive	MX	MX	MX	TCP	TCP	TCP
	64 cores	128 cores	256 cores	64 cores	128 cores	256 cores
MPI_Isend	12.5	13.1	13.4	18.6	19.4	19.8
MPI_Recv	7.9	8.3	8.8	7.7	7.6	7.3
MPI_lprobe (failed)	1.9	2.6	4.6	38.7	51.2	87.7
MPI_lprobe (success)	4.2	4.5	5.1	3.2	3.7	4.9

Table 3. Average MPI host overhead in μ s on DAS-3/VU

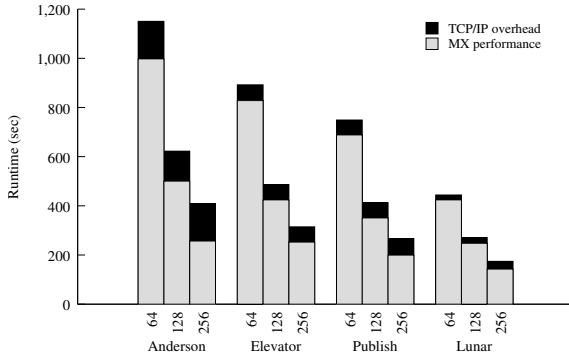


Figure 14. TCP/IP overhead for MAP

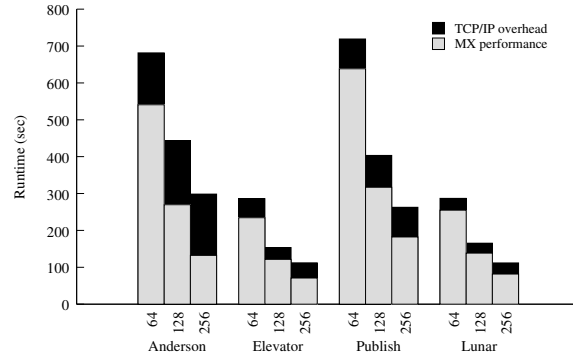


Figure 15. TCP/IP overhead for OWCTY

network, much as would be the case on a general purpose 1Gb/s or 10Gb/s Ethernet. We use the same version of Open MPI for both MX and TCP/IP, as Open MPI conveniently allows selection of a specific network backend at runtime.

Table 3 shows host-level overheads using MX and TCP/IP for the most important MPI primitives used in DiVINE. The overheads were measured using the MAP application (results using OWCTY are very similar). Receives and successful probes have about the same overhead, but send overhead is about 50% higher on TCP/IP. The biggest difference is for failing probes, however, where TCP/IP is about a factor 20 more expensive than MX. In addition, kernel-based TCP/IP receive processing is a source of overhead, but this is harder to measure as it occurs interrupt-driven, asynchronous to the application. Note that differences in end-to-end latency and peak throughput are less relevant, as they have little impact on application performance given DiVINE’s asynchronous communication style.

Figures 14 and 15 show a quite consistent pattern in the impact of using TCP/IP instead of MX. The TCP/IP interface with higher send and receive overhead does increase the runtimes, but interestingly enough this increase is almost independent of the number of cores used. It should be noted that in DiVINE the communication rate *per core* is in principle independent of the total number of cores, so one would expect that doubling the number of cores would on average halve the total number of sends and receives per core. Unfortunately, the overhead of a TCP/IP-based network is significantly more dependent on the number of remote endpoints than MX, as shown above, which counter-

balances the gain due to the reduced total number of transfers per core.

4.3 DAS-3 Grid Performance

Given DiVINE’s consistent use of asynchronous communication throughout its execution, an interesting question is to what extent its overall performance is truly latency independent. However, running the distributed model checker at a large scale does pose very high demands on the wide-area network bandwidth, as every compute node indiscriminately needs to transfer a large portion of its protocol messages to nodes at other clusters. Fortunately, DAS-3 provides the opportunity to examine this aspect in detail since it features a dedicated wide-area network called Star-Plane [24], built out of multiple optical 10G links. The impact of using a single or multiple optical links on distributed application performance (including DiVINE) is discussed in another recent paper [20]; here we will use a static configuration of two 10G links between the sites.

We use a DAS-3 grid configuration of 160 compute nodes distributed over 4 clusters, located at 3 sites in the Netherlands (VU University, University of Amsterdam and Leiden University). The one-way latencies over TCP/IP between these clusters range between 0.37 and 0.98 milliseconds, which should be contrasted with an intra-cluster one-way TCP/IP latency on DAS-3/VU of 26 microseconds, i.e., up to a factor 38 difference.

Figures 16 and 17 show the results for running an increasingly large instance of the Elevator specification on

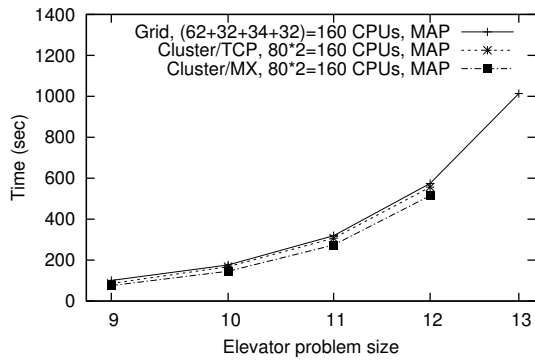


Figure 16. Elevator/MAP on a grid

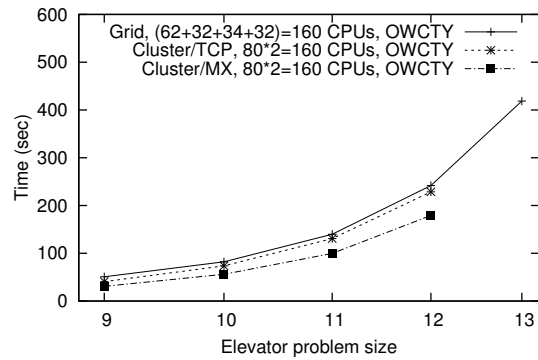


Figure 17. Elevator/OWCTY on a grid

both the grid and a DAS-3/VU cluster. Note that problem size 13 is too big to be run on the single DAS-3/VU cluster alone. The figures show that despite the additional wide-area latencies incurred, for both MAP and OWCTY grid performance is actually very close to single-cluster performance using the same TCP/IP protocol stack.

5 Discussion and Conclusions

The work on distributed-memory verification is quite extensive, and growing in recent years. In this paper we discussed a distributed-memory tool for enumerative model checking. Distributed-memory techniques have also been applied in other verification areas, e.g., verification of timed systems [3], equivalence checking [4], state space construction [13], and μ -calculus model checking [5]. However, these systems have thus far not been evaluated on and optimized for large-scale clusters (or grids). A possible exception is the *symbolic* model checker in [12], but no scalability results are reported.

Our paper is novel in bringing distributed model checking closer to industrial applications. Both the scale and efficiency with which we are now able to verify very large systems is to the best of our knowledge without precedent. For very large-scale models, state space reduction may still remain necessary, but this can often be orthogonal to the techniques discussed in this paper. For example, for partial order reduction, a *static* transformation approach is known [18], but distributed algorithms also exist [6]. State compression is another popular technique that directly applies in a distributed context. Keeping parts of the state space on secondary storage, while still maintaining good performance, is also a possibility [15].

We have discussed two main distributed algorithms in DIVINE, and we have shown how several optimizations together improved their performance by 30 to 50%. We compared the performance of these two algorithms on seven representative large models, having quite different charac-

teristics. We have shown that the optimizations allow the algorithms to scale well, up to at least 256 cores, efficiently exploiting current multi-core architectures. However, as *many-core* is an inevitable trend in computer architecture, it appears likely that at some point a single-address-space multithreaded implementation should be integrated with the current version for best performance [2].

Some of the optimizations discussed are not unique to DIVINE, but will also be applicable in other distributed applications. For example, the auto-tuning polling rate optimization described will be useful in several cases where applications have to employ non-blocking polling due to the irregularity of the communication patterns [27].

The performance differences shown can be used to plan an efficient model checking workflow, during the development of an abstract specification. If a property of a model is expected to hold, and the state space fits completely into (distributed) memory, the OWCTY algorithm will typically be preferable as it can give up to three times faster results than MAP. However, if the status of a property is uncertain, MAP will generally be preferable instead, as it works on-the-fly, and may thus find counterexamples quickly (even when the entire state space would not fit into memory). Also, if a property holds after all, MAP will still perform quite well due to its good scalability.

In this paper, we also analyzed the sensitivity of the model checking algorithms to network protocol overhead, and we have shown how the consistent use of asynchronous communication even allows efficiently running the model checker on a large-scale computational grid. This thus enables further scaling up the model checker for realistic use cases, where the state space to be examined quickly grows even beyond the capacity of a single large compute cluster.

Note – DIVINE is available from <http://divine.fi.muni.cz>. The cluster-based tools, containing the optimizations discussed, are now part of DIVINE-CLUSTER, distinguishing them from other instances of DIVINE.

References

- [1] J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *FMCO'05*, volume 4590 of *LNCS*, pages 281–293. Springer, 2006.
- [2] J. Barnat, L. Brim, and P. Rockai. Scalable Multi-core LTL Model-Checking. In *SPIN'07*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.
- [3] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed Timed Model Checking — How the Search Order Matters. In *CAV'00*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.
- [4] S. Blom and S. Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. *STTT*, 7(1):74–86, 2005.
- [5] B. Bollig, M. Leucker, and M. Weber. Parallel Model Checking for the Alternation Free μ -Calculus. In *TACAS'01*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.
- [6] L. Brim, I. Černá, P. Moravec, and J. Šimša. Distributed Partial Order Reduction. *Electronic Notes in Theoretical Computer Science*, 128:63–74, 2005.
- [7] F. Cappello and H.E. Bal. Toward an International “Computer Science Grid” (keynote). In *CCGrid'07*, pages 3–12, 2007.
- [8] I. Černá and R. Pelánek. Relating Hierarchy of Temporal Properties to Model Checking. In *MFCS'03*, volume 2747 of *LNCS*, pages 318–327. Springer, 2003.
- [9] C. Courcoubetics, M. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *CAV'91*, pages 233–242. Springer, 1991.
- [10] E.W. Dijkstra. Shmuel Safra’s version of termination detection. EWD Manuscripts, no. 998, January 1987.
- [11] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is There a Best Symbolic Cycle-detection Algorithm? In *TACAS'01*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.
- [12] L. Fix, O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. Verifying Very Large Industrial Circuits Using 100 Processes and Beyond. *Int. J. Found. Comput. Sci.*, 18(1), 2007.
- [13] H. Garavel, R. Mateescu, and I.M. Smarandache. Parallel State Space Construction for Model-Checking. In *SPIN'01*, volume 2057 of *LNCS*, pages 216–234. Springer, 2001.
- [14] R.L. Graham, T.S. Woodall, and J.M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proc. 6th Int. Conf. on Par. Proc. and Appl. Math.*, pages 228–239, Poznan, Poland, September 2005.
- [15] M. Hammer and M. Weber. “To Store or Not To Store” Reloaded: Reclaiming Memory on Demand. In *Formal Methods: Application and Technology (FMICS'2006)*, volume 4346 of *LNCS*, pages 51–66. Springer, 2007.
- [16] G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
- [17] M. Kamel and S. Leue. Formalization and Validation of the General Inter-Orb Protocol (GIOP) using Promela and SPIN. In *In: Software Tools for Technology Transfer*, pages 394–409. Springer, 2000.
- [18] R.P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Combining Software and Hardware Verification Techniques. *Form. Methods Syst. Des.*, 21(3):251–280, 2002.
- [19] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1):5–20, March 2007.
- [20] J. Maassen, K. Verstoep, H.E. Bal, P. Grosso, and C. de Laat. Assessing the Impact of Future Reconfigurable Optical Networks on Application Performance. In *IPDPS'09: 6th High-Performance Grid Computing Workshop (HPGC 2009)*, 2009.
- [21] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3):161–175, 1987.
- [22] R. Pelánek. Typical Structural Properties of State Spaces. In *SPIN'04*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.
- [23] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'07*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
- [24] StarPlane project. <http://www.starplane.org>.
- [25] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, pages 146–160, January 1972.
- [26] M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 322–331. Computer Society Press, 1986.
- [27] K. Verstoep, J. Maassen, H.E. Bal, and J.W. Romein. Experiences with Fine-grained Distributed Supercomputing on a 10G Testbed. In *CCGrid'08*, 2008.
- [28] M. Weber. An Embeddable Virtual Machine for State Space Generation. In *SPIN'07*, pages 168–186, 2007.
- [29] O. Wibling, J. Parrow, and A. Neville Pears. Automated Verification of Ad Hoc Routing Protocols. In *FORTE'04*, volume 3235 of *LNCS*, pages 343–358. Springer, 2004.