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 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

*This work has been supported in part by the Czech Academy of Sciences grant No. 1ET408050503 and GACR grant No. 201/06/1338.

Dagstuhl Seminar Proceedings 08332
Distributed Verification and Grid Computing
http://drops.dagstuhl.de/opus/volltexte/2008/1630

they can be distributed effectively using a large distributed memory. A prominent example in this category is the DIVINE [3] 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 very small instances to more realistic proportions, gradually more trust can be gained in the verification results. However, seemingly simple, restricted specifications can still quickly 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 [26] resulted in verification of various scenarios, several of which could not be verified using the efficient (sequential) SPIN [17] model checker, due to their very large state space. As DIVINE also supports SPIN specifications, additional scenarios can now be verified using a cluster [25].

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 five 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 five 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, state space reduction, symbolic state representation, etc., are used to reduce the size of the model, thus allowing a single computer to still process large systems. Howeve, 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 compute 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 [23]. 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) [22]. 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 exploring principle, namely the *postorder* as computed by the DFS. It is a well-known fact that the postorder 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 *reachability* and *elimination* 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 [13]. Similar results are communicated in [19] 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. Studies of temporal properties [8, 12] reveal that verification of up to 90% of LTL properties leads to weak automaton graphs.

MAP: Maximal Accepting Predecessors Algorithm

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

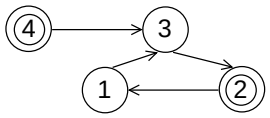

Figure 1. Undiscovered cycle

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

Figure 2. Distributed graph traversal skeleton

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, 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. For an automaton graph without accepting cycles, the repetitive application of the deleting transformation results in an automaton graph with an empty set of 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 works 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 [3] tool 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.

Distributed Graph Traversal

As shown in Figure 2, the core of each graph traversal algorithm is a while loop over a queue of vertices waiting to be processed. Each time a vertex is dequeued, edges 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's distributed termination detection algorithm [11] is used.

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 asynchronous 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 algorithm. Unlike the serial case, where the space complexity of a graph traversal algorithm is asymptotically linear in the number of vertices, the distributed algorithm exhibits space complexity that is asymptotically linear in the number of vertices and edges, hence, up to asymptotically quadratic in the number of vertices. Our experimental experience showed that even for graphs with a relatively small number of transitions (an average outdegree of 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 [10] (discussed below), allowing performance evaluation at a much larger scale. However, it was soon determined that performance of several DiVinE algorithms did not scale well with the number of nodes. It was unclear whether this was caused by inherent scalability limitations of the underlying (distributed) algorithms, or whether it was due to other causes.

An important reason for the 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, 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 [24].

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 were applied:

Auto-tune receive 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 – 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 transfers too long. Furthermore, distributed termination detection is a re-

quirement 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 distributed termination detection – Another aspect that was optimized, is the flushing of messages during the distributed termination phases (as discussed in Section 2, each DIVINE algorithm has several of these phases). The original implementation simply flushed *all* outstanding (non-full) messages in a fixed sequential order, which caused much network congestion due to hotspots in the network and at the receivers. Also, as every message was flushed indiscriminately, the average message size during these phases dropped substantially, causing a relatively high overhead. The distributed termination detection phase was optimized quite similarly to the Awari application [24]. 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 upper-bound in the outgoing traffic rate (thus avoiding the syndrome of frantically sending extremely tiny messages).

Pre-establish network connections – 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 [16]) establishes TCP connections on-demand. This 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 a-typical 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 Myricom 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. In this paper we will first mostly focus on the DAS-3 cluster at VU University, which 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 [20] (this specification concerns the correctness of a

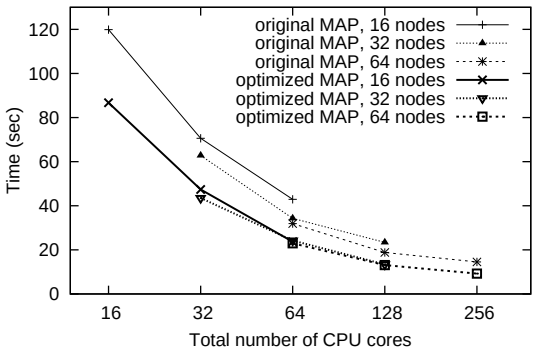

Figure 3. Optimization effects for MAP

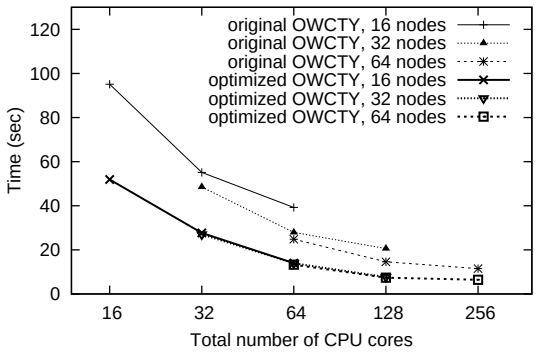

Figure 4. Optimizations effects for OWCTY

Table 1. Efficiency of MAP

Cores	Runtime (sec)	Efficiency
1	1052.5	100%
64	23.0	72%
128	13.1	63%
256	8.9	46%

Table 2. Efficiency of OWCTY

Cores	Runtime (sec)	Efficiency
1	631.7	100%
64	13.3	74%
128	7.4	67%
256	5.0	49%

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), MAP becomes 9.4 times faster, and OWCTY 8.1 times;
- Due to reduced overheads, the performance of the optimized version is insensitive to the placement of multiple processes on the same node, unlike the original version.

To get an estimate on 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 Tables 1 and 2. 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.

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, but the same effects are present on the instances discussed above.

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

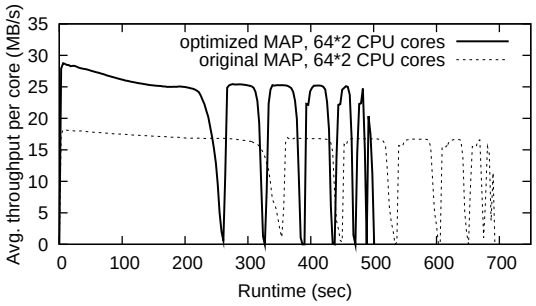

Figure 5. Per-core throughput of MAP

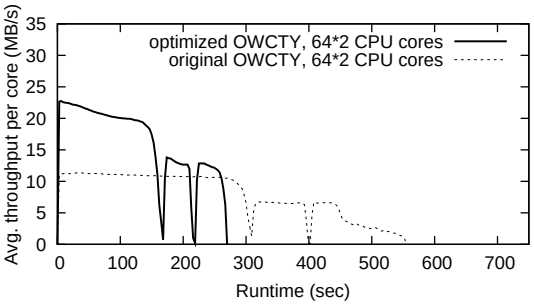

Figure 6. Per-core throughput of OWCTY

Table 3. Large-scale models used

Model	Description	BEEM database acronym	State space
Anderson	Anderson's mutual exclusion	anderson.8.prop2	141.4 GB
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
Lann	Token ring leader election	lann.8.prop1	over 320 GB

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.

4 Scalability of the Optimized DIVINE System

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 3. All models are taken from the on-line BEEM database [20]. In the first three cases (Anderson, Elevator and Publish-subscribe) the LTL formula provided is valid in the model, as a result of which the entire state space has to be built and analyzed for the presence of accepting cycles. In the last two models (AT and Lann), the LTL formula being verified is false, i.e., there exists a counterexample that has to be found by the DIVINE tool. 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%.

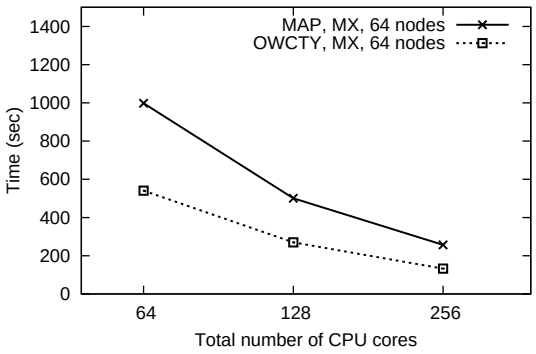

Figure 7. Anderson on 64 nodes

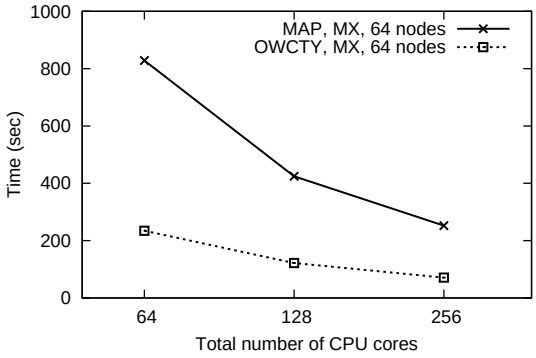

Figure 8. Elevator on 64 nodes

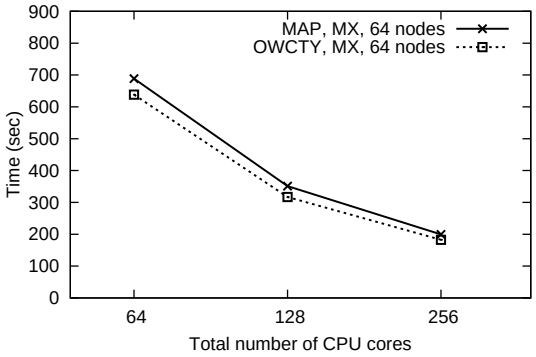

Figure 9. Publish-subscribe on 64 nodes

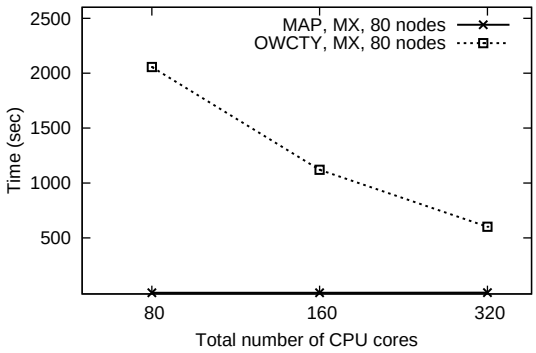

Figure 10. AT on 80 nodes

4.1 DAS-3 Cluster Performance

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

The first three specifications requiring a full space search are shown in Figures 7, 8, and 9. Interestingly, the first two specifications (Anderson and Elevator) show largely the same pattern: OWCTY is much faster than MAP, but both show good scalability when increasing the number of cores. For Publish-subscribe, OWCTY and MAP are about equally fast, because MAP for Publish-subscribe only requires 9 very short cycle-searching phases after the first on-the-fly one (Anderson and Elevator require resp. 15 and 21 much longer ones). Overall, MAP thus only takes slightly longer than the (minimal) three phases of OWCTY in this case.

The two inconsistent specifications show a rather different picture. As seen in Figure 10, 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 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.

MPI primitive	MX 64 cores	MX 128 cores	MX 256 cores	TCP 64 cores	TCP 128 cores	TCP 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_Iprobe (failed)	1.9	2.6	4.6	38.7	51.2	87.7
MPI_Iprobe (success)	4.2	4.5	5.1	3.2	3.7	4.9

Table 4. Average MPI host overhead in μs on DAS-3/VU

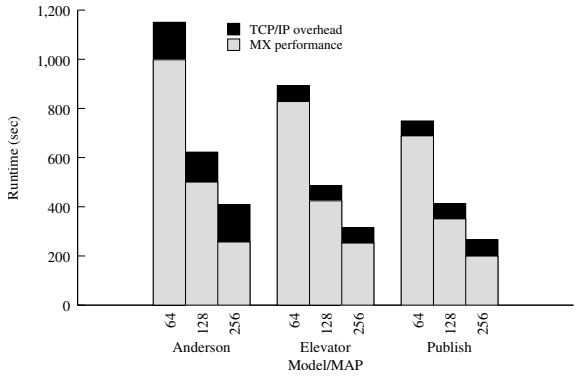

Figure 11. TCP/IP overhead for MAP

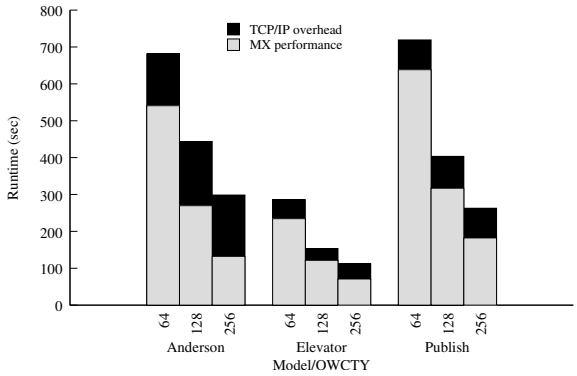

Figure 12. TCP/IP overhead for OWCTY

4.2 Network Impact

In this section we will compare performance using Myri-10G's highly optimized native MX interface with TCP/IP over Myri-10G. We examine TCP/IP to provide insight into DIVINE's performance on a higher-overhead network, much as would be the case on a general purpose 1Gb/s or 10Gb/s Ethernet. In either case, exactly the same version of Open MPI is used, as Open MPI conveniently allows selection of a specific network backend at runtime.

Table 4 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 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 11 and 12 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 counterbalances the gain due to the reduced total number of transfers per core.

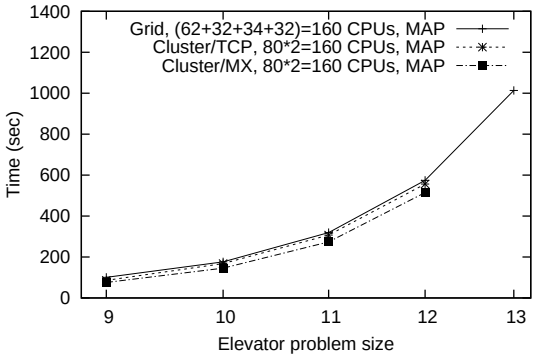

Figure 13. Elevator/MAP on a grid

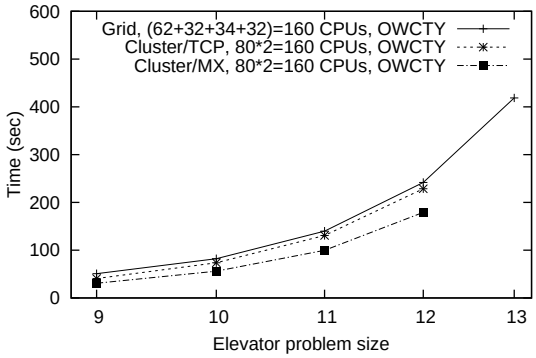

Figure 14. Elevator/OWCTY on a grid

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 StarPlane [21], 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 [18]; 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 13 and 14 show the results for running an increasingly large instance of the Elevator specification on 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 [4], equivalence checking [5], state space construction [15], and μ-calculus model checking [6]. 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 [14], 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.

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 five representative large models, having quite different characteristics. 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 a useful technique in several cases where applications have to employ non-blocking polling due to the irregularity of the communication patterns [24].

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 optical 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 can be downloaded from *http://anna.fi.muni.cz/divine/tool*. The current version is 0.7.2. An update including the optimizations discussed in this paper will be available shortly.

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] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification. In *CAV'06*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006.

[4] 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.

[5] S. Blom and S. Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. *STTT*, 7(1):74–86, 2005.

[6] B. Bollig, M. Leucker, and M. Weber. Parallel Model Checking for the Alternation Free mu-Calculus. In *TACAS'01*, volume 2031 of *LNCS*, pages 543–558. Springer, 2001.

[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] The Distributed ASCI Supercomputer 3 (DAS-3). http://www.cs.vu.nl/das3.

[11] E.W. Dijkstra. Shmuel Safra's version of termination detection. EWD Manuscripts, no. 998, January 1987.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[13] 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.

[14] 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.

[15] 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.

[16] R.L. Graham, G.M. Shipman, B.W. Barrett, R.H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: a High-Performance, Heterogeneous MPI. In *Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, 2006.

[17] G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[18] J. Maassen, K. Verstoep, H.E. Bal, and C. de Laat. Assessing the Impact of Future Reconfigurable Optical Networks on Application Performance. Submitted to IPDPS'09, 2009.

[19] R. Pelánek. Typical Structural Properties of State Spaces. In *SPIN'04*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[20] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'07*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[21] StarPlane project. http://www.starplane.org.

[22] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, pages 146–160, January 1972.

[23] M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 322–331. Computer Society Press, 1986.

[24] 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.

[25] M. Weber. An Embeddable Virtual Machine for State Space Generation. In *SPIN'07*, pages 168–186, 2007.

[26] O. Wibling, J. Parrow, and A. Neville Pears. Automatized Verification of Ad Hoc Routing Protocols. In *FORTE'04*, volume 3235 of *LNCS*, pages 343–358. Springer, 2004.