Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 220 (2008) 3–17

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Large-Scale Parallel Computing on Grids

Henri Bal and Kees Verstoep[1]

VU University, Faculty of Sciences, Dept. of Computer Science, Amsterdam, The Netherlands

Abstract

This paper argues that computational grids can be used for far more types of applications than just trivially parallel ones. Algorithmic optimizations like latency-hiding and exploiting locality can be used effectively to obtain high performance on grids, despite the relatively slow wide-area networks that connect the grid resources. Moreover, the bandwidth of wide-area networks increases rapidly, allowing even some applications that are extremely communication intensive to run on a grid, provided the underlying algorithms are latency-tolerant. We illustrate large-scale parallel computing on grids with three example applications that search large state spaces: transposition-driven search, retrograde analysis, and model checking. We present several performance results on a state-of-the-art computer science grid (DAS-3) with a dedicated optical network.

Keywords: Model checking, retrograde analysis, search algorithms, grids, distributed supercomputing, optical networks

1 Introduction

Computational grids are interesting platforms for solving large-scale computational problems, because they consist of many (geographically distributed) resources. Thus far, grids have mainly been used for high-throughput computing on independent (or trivially parallel) jobs. However, advances in grid software (programming environments [21], schedulers [13]) and optical networking technology [7] make it more and more feasible to use grids for solving challenging large-scale problems. The goal of this paper is to discuss our experiences in implementing and optimizing several challenging applications on a state-of-the-art grid, thus showing that more applications are suitable for grid computing than just trivially parallel ones.

The paper will first describe a state-of-the-art grid infrastructure, the Dutch DAS-3 Computer Science grid. DAS-3 contains a flexible (and ultimately reconfigurable) 20–40 Gbit/s optical network called StarPlane between its five clusters. From a parallel programming point of view, grids like DAS-3 are characterized by a high-latency/high-bandwidth network and a hierarchical structure.

[1] {bal,versto}@cs.vu.nl

1571-0661 © 2008 Elsevier B.V. Open access under CC BY-NC-ND license.

doi:10.1016/j.entcs.2008.11.010

Next, the paper will discuss how algorithms and applications can be optimized to run in such an environment. We focus on applications that search large state spaces. As an introductory example, we will summarize earlier work and describe how a communication-intensive heuristic search algorithm can be optimized to run on a grid. As a more recent case study, we have implemented a retrograde analysis application that solves the game of Awari, which has 900 billion different states. Several optimizations were needed to obtain high performance on the DAS-3 grid. Finally, the paper discusses some preliminary results in using the DiVinE model checking toolkit on DAS-3. As we will show, DiVinE has many characteristics in common with the Awari solver, making it an interesting application for further research.

2 DAS-3: an optical Computer Science Grid

The DAS (Distributed ASCI Supercomputer) project started over 10 years ago (1997) and was an initiative of the Dutch ASCI (Advance School in Computing and Imaging) research school. The idea behind DAS is to create a joint infrastructure for experimental research on distributed systems (like grids). Each DAS system consists of multiple (4–5) clusters located at different Dutch universities, connected by a wide-area network. The DAS systems were set up specifically to allow distributed experiments, so users always get an account on the entire system. To allow easy experimentation, the systems are largely homogeneous: all clusters have the same operating system (Linux), the same processor architecture and (largely) the same local network (Myrinet). The systems are not used for production jobs, but only for relatively short-running computer science experiments, so the load of the clusters is deliberately kept low.

Three generations of DAS systems have been built so far:

- DAS-1 (1997) consisted of 4 PentiumPro clusters connected by a 6 Mbit/s dedicated ATM network;
- DAS-2 (2002) consisted of 5 Pentium-III clusters connected by the university internet backbone (1 Gbit/s);
- DAS-3 (2006-now) consists of 5 AMD Opteron clusters connected by multiple dedicated 10 Gbit/s light paths (see Figure 1).

The current system, DAS-3, consists of 272 nodes (792 cores) and 1 TB of memory. Unlike DAS-1 and DAS-2, the current system is slightly heterogeneous: the sites differ in having single- or multi-core CPUs and the CPU speeds also differ (between 2.2 and 2.6 GHz). All clusters except the one at TU Delft use the Myri-10G network to connect their local nodes and to connect to the optical wide-area network. In addition, all clusters have a 1 Gbit/s Ethernet network. The entire setup allows experiments with heterogeneous systems, which many projects requested. DAS-3 also has a 10 Gbit/s dedicated light path to Grid'5000 in France, allowing even larger-scale heterogeneous experiments [5].

The DAS systems have been used for many research projects on programming

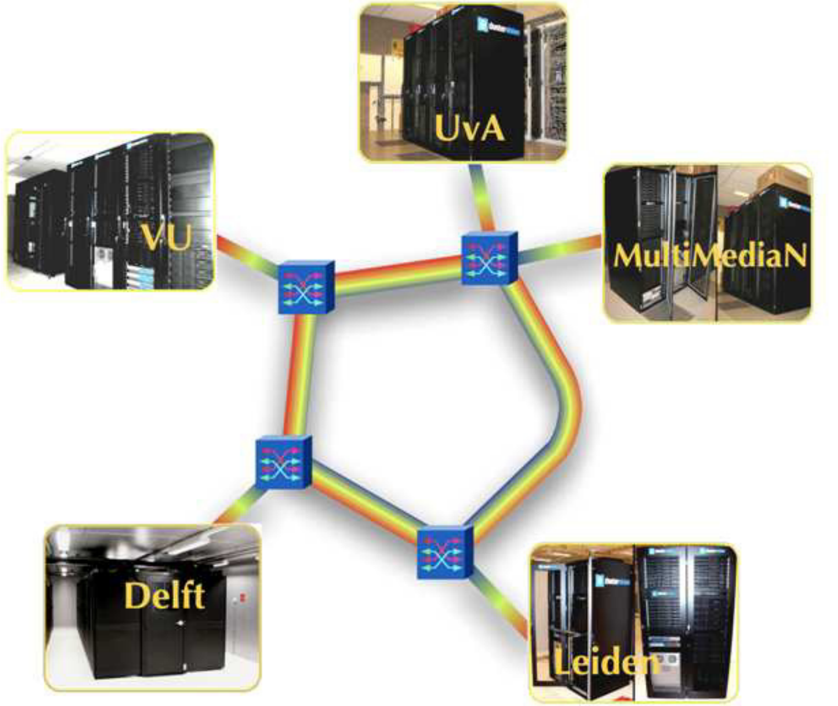

Fig. 1. DAS-3 clusters interconnected by StarPlane.

systems, operating systems, network protocols, grid computing, virtual laboratories, computational science, etc. Over 40 Ph.D. theses have used the systems. User experiences with the systems are described in [1,5].

We think DAS-3 is representative for future grid systems. It is hierarchical and consists of multiple clusters with a fast local interconnect and a higher-latency wide-area interconnect. The wide-area network has a high bandwidth. In summary, the system can be characterized as hierarchical, high wide-area latency, and high bandwidth. In the rest of the paper, we discuss how to design parallel applications for such an environment, which are called distributed supercomputing applications.

3 Wide-area parallel computing

The DAS systems are excellent testbeds for studying wide-area parallel algorithms, because they can be used for controlled experiments. Unlike on very heterogeneous production systems, it is feasible to do clean speedup experiments on DAS. We have produced many papers on this topic (e.g., [11,15]).

What we have learned is that far more applications are suitable for distributed supercomputing than one might expect. Of course, the wide-area network between the clusters has many orders of magnitude higher latency than the local-area network. In DAS, the wide-area latency typically is several milliseconds, while the local-area latency (over Myrinet) is several microseconds. One thus might expect that only trivially parallel algorithms could be run on such a wide-area system.

However, we have learned that many applications can be optimized to deal with the high latency of the wide-area network. The key idea is to be latency-tolerant. The bandwidth of wide-area networks is increasing very fast (much faster than CPU speed increases), as can easily be seen by comparing the bandwidths of the three DAS systems.

Several optimizations are feasible to obtain good speedups for wide-area algorithms:

- Many algorithms can exploit the hierarchical structure of grids. Grids typically consist of several different clusters with fast internal communication and relatively slow wide-area communication. By doing locality optimizations, algorithms can often reduce the amount of wide-area communication.
- Likewise, several algorithms can be made latency-tolerant, by using asynchronous communication. In this case, useful computations can be done while the wide-area communication takes place.

These optimizations are well-known, but we have discovered that they often are even more effective for wide-area systems than for clusters. As a good example, we summarize our earlier work on the TDS (Transposition Driven Search) algorithm (adapted from [17,18]).

4 Transposition Driven Search

Heuristic search algorithms recursively expand a state into successor states. If the successor states are independent of each other, different processors can analyze different portions of the search space. During the searches, each processor maintains a list of work yet to be completed (the *work queue*). When a processor completes all its assigned work, it tries to acquire more work from busy processors, which is called *work stealing*.

However, many search applications use so-called transposition tables, which enhance the search but also introduce interdependencies between states, making efficient parallelization more difficult. A transposition table is a large store (accessed through a hash function) in which newly expanded states are placed. The table prevents the expansion of previously seen states, which saves much computation time for applications where a state can have multiple predecessors (i.e., when the search space is a graph rather than a tree). Depending on the characteristics of the search algorithm, the table is implemented either as permanent store or as a cache, for efficiency reasons.

Unfortunately, transposition tables are difficult to implement efficiently in parallel search programs that run on distributed-memory machines. Usually, the transposition table is partitioned among the local memories of the processors. Before a processor expands a state, it first does a remote lookup, by sending a message to the processor that manages the entry and then waiting for the reply. This can result in sending many thousands of messages per second, introducing a large communica-

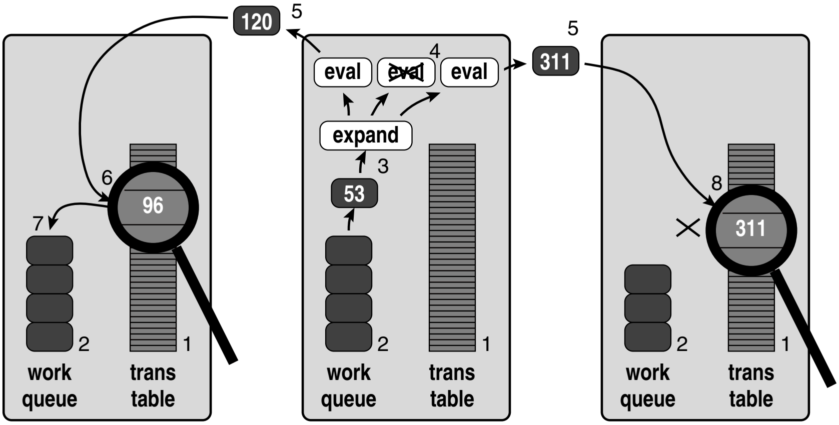

Fig. 2. Transposition Driven Scheduling.

tion overhead. Moreover, each processor wastes much time waiting for the results of remote lookups.

The *transposition table driven work scheduling* (TDS) algorithm integrates the parallel search algorithm and the transposition table mechanism: drive the work scheduling by the transposition table accesses. The state to be expanded is migrated to the processor on which the transposition for the state may be stored (see Figure 2), as determined by the hash function. This processor performs the local table lookup to check if the state has already been solved (as in the right part of the figure). If not, the processor stores the state in its work queue (as in the left part of the figure). Although this approach may seem counterintuitive, it has several important advantages:

(i) All communication is asynchronous (nonblocking). Expanding a state amounts to sending its children to their destination processors, where they are entered in the work queue. After sending the messages the processor continues with the next piece of work. Processors never have to wait for the results of remote lookups.

(ii) Since all messages are asynchronous, they need not be sent immediately, but they can be delayed and batched up into fewer large messages. This optimization, called message combining, results in bulk transfers that are much more efficient on most networks.

(iii) The network latency is hidden by overlapping communication and computation. This latency hiding is effective as long as there is enough bandwidth in the network to cope with all the asynchronous messages. With modern high-speed networks (like light paths) such bandwidth usually is amply available.

(iv) The algorithm also can have good load balancing, because it distributes the work randomly. However, the advantages of good load balancing should be balanced against loss of locality and additional network processing. Also, the highly asynchonous algorithm described does assume a homogeneous execution environment.

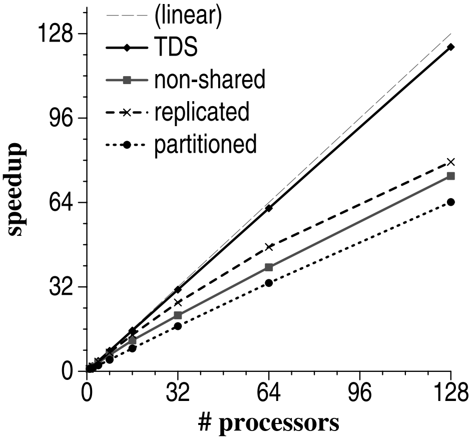

Fig. 3. Performance of the parallel Rubik's cube solver.

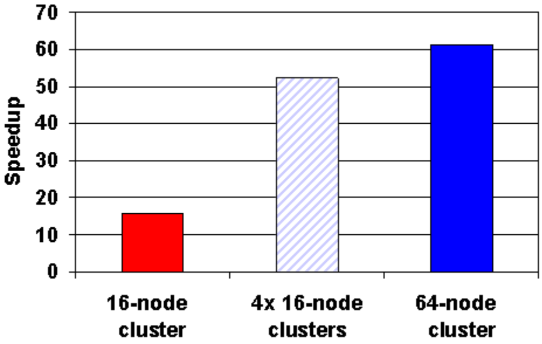

Fig. 4. Parallel Rubik's cube solver on a grid.

Overall, the TDS algorithm resulted in huge performance improvements over traditional search algorithms (like IDA* [12]) even on a local cluster. Figure 3 gives an example for a program that solves Rubik's cube, measured on DAS-2. The figure shows the relative speedups (up to 128 processors) of TDS, and compares it against algorithms that partition the transposition table, replicate it, or use local (nonshared) tables. As can be seen, TDS almost doubles the efficiency.

The latency-hiding optimization of TDS also allows it to run efficiently on a wide-area system. In [17], we discussed some further optimizations to make TDS suitable for DAS-2, which used a relatively slow (1 Gb/s, shared) wide-area network. Figure 4 shows the speedups of TDS on the wide-area DAS-2 system. On four clusters of 16 CPUs it is much faster than on one cluster of 16 CPUs. In fact, the algorithm runs only slightly slower on four clusters than on a single large (64-CPU) cluster.

This work thus started with a highly fine-grained algorithm that obtained mediocre performance (at most 50% efficient) even on a fast Myrinet cluster. The algorithm was optimized into a latency-insensitive one that could run efficiently even on a wide-area system where the wide-area latency was three orders of magnitude higher than the local-area latency. This experiment clearly illustrates the power of algorithmic optimizations for wide-area systems.

5 An Awari solver on DAS-3

In this section we will summarize our more recent work on using distributed supercomputing for large search-applications [22]. We will discuss how we implemented a retrograde analysis program that solves the game of Awari on the wide-area DAS-3 system. The original program [16] ran on a single cluster and was extremely communication intensive: it sent one Petabit of data over the local area network in 51 hours. We will discuss how such a communication-intensive application can be run efficiently using light path technology.

Awari is a 3500 year old game that uses a board with 12 pits (6 for each player), each initially containing 4 stones. The two players in turn select one of their own pits and "sow" its stones counterclockwise, capturing stones if the last pit is owned by the other player and now contains 2 or 3 stones. So, the number of stones initially is 48, but decreases during the play.

Our Awari solver uses retrograde analysis (RA). Unlike more familiar 2-person search techniques like MiniMax and Alpha-Beta, RA searches backwards, starting with the final (solved) positions in the game, such as those containing only 1 stone. Through backwards reasoning (i.e., doing "unmoves"), RA computes the game-theoretical outcome of every possible position in the game, up to the opening position, thus solving the entire game. The program builds a sequence of databases for board positions with different numbers of stones, ending with the 48-stone database. Since Awari contains 889,063,398,406 possible game positions, this entire process is extremely memory and CPU intensive.

The RA program can be parallelized by partitioning the entire state space over the processors, much like the transposition tables discussed above. Each processor is assigned a random set of positions (based on a hash function), resulting in good load balancing. As with TDS, each processor can repeatedly send information (about solved states) to other processors in an asynchronous way, using message combining to obtain efficient bulk transfers. The parallel RA program for Awari is extremely communication intensive, but processors hardly ever have to wait for the communication to finish. In other words, the program effectively overlaps communication and computation. Therefore, the initial parallel program ran efficiently on the DAS-2 Myrinet cluster [16].

With the advance of optical networking technology, it becomes feasible to even run this type of application on a grid. We therefore studied the performance of the Awari RA program on the wide-area DAS-3 system [22]. We initially just ported the original C/MPI program and tested it on DAS-3. The (small) differences in CPU speeds in DAS-3, however, caused flow control problems, resulting in the faster CPUs to overwhelm the slower ones with work. This effect resulted in unrestricted job queue growths and thus memory problems. In Awari we solved this problem using global synchronizations (an alternative, proposed in a memory-limited, se-

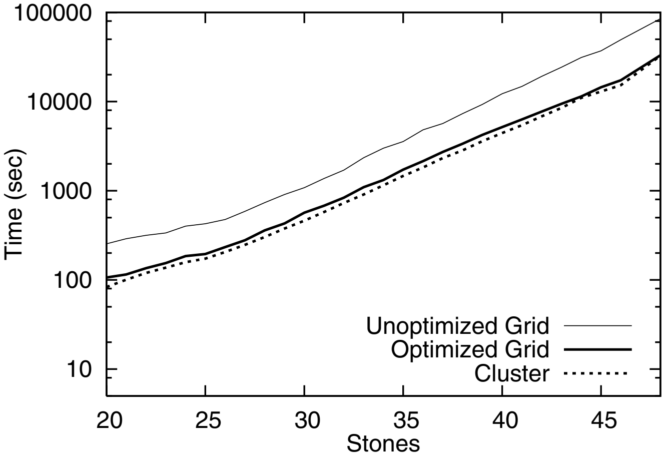

Fig. 5. Impact of grid optimizations on Awari.

quential model checking context, is to temporarily store new jobs in a local log file [9]). The resulting version did work correctly on DAS-3, but it was twice as slow as the same program on a single big cluster with the same number of CPUs.

Subsequently, we applied several optimizations to the wide-area program. The optimizations that turned out to have the largest impact were related to communication patterns during the synchronization phases. The original synchronization algorithms performed well on a single cluster, but were much less scalable in a high-latency grid environment. Another important optimization was to ensure that completion of asynchronous communication to remote grid sites would not stall initiation of other communication. Increasing the amount of message combining also improved the performance due to a higher overall throughput, but to a much lesser extent.

The result of the optimizations was a 50% performance improvement compared to the original program. Figure 5 (taken from [22]) shows the execution times of the original (unoptimized) version, the optimized one, and the single cluster version, for the different databases. Moreover, the optimized grid version was only 15% slower than a single cluster version, despite the huge amount of wide-area communication, showing that even communication-intensive algorithms like retrograde analysis are suitable for distributed supercomputing on grids.

Figure 6 illustrates the impact of the grid optimizations for Awari by means of the single-CPU wide-area throughput measured during the computation of the 40-stone database; the patterns shown here are quite similar for the other large databases. Clearly, it is not so much improvements in sustained or peak throughput that caused runtime to improve significantly. Rather, optimizations related to efficient processing during synchronization phases (the distributed termination detection phases and the barriers resolving grid performance differences) are shown to be much more important.

The usage of fast light paths is essential for this application, as regular 1 Gbit/s Internet links would be completely flooded by the cumulative traffic for even a

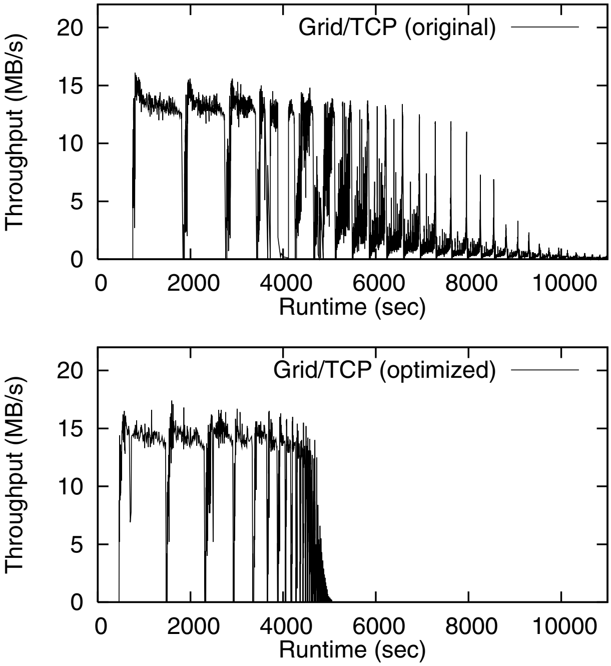

Fig. 6. Per-CPU wide-area throughput of Awari.

modest number of CPUs. As the bandwidth requested by the application scales linearly in the number of compute nodes, and every CPU tries to send data over the wide area network at about 15 MByte/s, even a 10 Gbit/s light path will at some point limit further speed improvement. In the context of the StarPlane [19] project we are currently examining the *on-demand* scheduling of *multiple* 10 Gbit/s light paths to increase application scalability for these scenarios.

6 DiVinE on DAS-3

The DiVinE Tool [3] is a parallel, distributed memory, enumerative model-checking toolkit for the verification of concurrent systems. It contains a collection of state-of-the-art distributed verification algorithms suitable for running on clusters. The DiVinE toolkit is part of the DiVinE Distributed Verification Environment which is being developed at Masaryk University, Brno, Czech Republic.

DiVinE uses a Linear Temporal Logic (LTL) approach to model checking, where a verification problem is reduced to cycle detection in a graph representing the state space. The toolkit contains several parallel cycle detection algorithms designed to run efficiently on a cluster [2]. In addition, the toolkit implements on-the-fly distributed state-space generation for error detection and deadlock discovery. The DiVinE toolkit supports specifications in both its native modeling language DVE and in Promela [4], the modeling language of the popular model checker SPIN [10].

6.1 Parallel implementation

The DiVinE toolkit is implemented in C++ and uses MPI for communication. Most graph traversal algorithms in the DiVinE toolkit are based on breadth-first search (BFS), which unlike depth-first search (DFS) can be parallelized in a straightforward fashion [6,20]. Every compute node is given a portion of the state space, based on a hash function that randomizes the state distribution. The resulting communication pattern can be characterized as irregular all-to-all: every compute node repeatedly sends asynchronous messages with batched state updates to other compute nodes, in an apparently unpredictable order. Note that this shows a remarkable resemblance to the Awari solver.

The communication rate for a single compute node is in itself not very demanding on the network: we found it typically to be in the order of 10–20 MByte/s per CPU, depending on the problem. However, the *accumulated* network load can be very substantial, since, assuming ideal speedup, this scales linearly with the number of compute nodes used. With DiVinE, we were therefore almost at the same situation as initially with Awari: DiVinE's performance had only been evaluated on a single cluster, and its high networking demands appeared to make use on traditional large scale grids infeasible. However, as with Awari, the introduction of high bandwidth light paths was found to make all the difference.

To run the DiVinE toolkit on DAS-3, we used Open MPI [8], which is able to access the Myri-10G network in multiple ways. In single-cluster runs, the most efficient protocol in general is Myri-10G's native MX layer, and indeed this can improve performance for DiVinE. As our focus in this paper is on grid performance, we used Open MPI in TCP/IP mode for both grid- and cluster-based runs. This way, our results also match other common grid environments, where low-cost 1 Gbit/s Ethernet NICs are used as cluster interconnect, and the switch has a 1 (or 10) Gbit/s link to the local backbone for external connectivity over the Internet.

The specific network to be used is selected by means of a runtime Open MPI parameter. Although the DiVinE toolkit was specifically designed for use on clusters, our experiments show that it also runs efficiently on modern computational grids, due to the use of scalable, latency-tolerant algorithms implemented using asynchronous communication.

6.2 Grid performance

For our initial performance analysis we took two representative benchmark problems from the BEEM model checking database [14]: Anderson and Elevator. The Anderson specification concerns the correctness of a mutual exclusion algorithm for a varying number of processes. The Elevator specification determines the correctness of an elevator controller, given a certain number of floors and persons waiting for the elevator. For each problem we let DiVinE check the LTL specification of a correctness property using the Maximal Accepting Predecessors tool (`distr map`) [2]. In both cases the entire state space must be searched, since there is no violation of the given property.

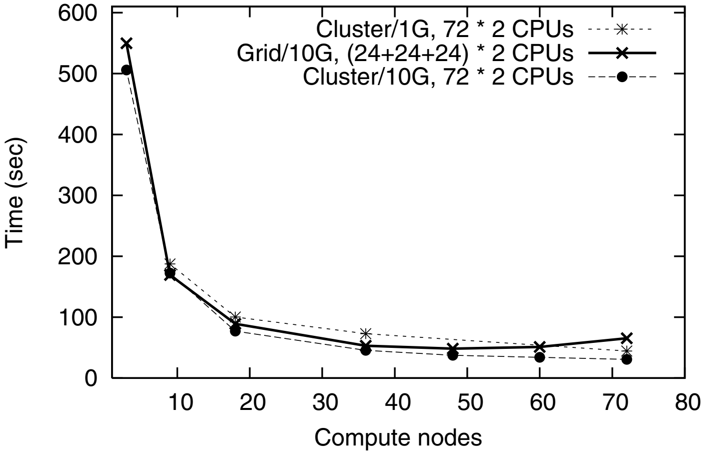

Fig. 7. Model checking Anderson with DiVinE on 144 CPUs

Figure 7 shows the total runtime for verifying the Anderson specification for 6 synchronizing processes on 72 compute nodes (using 144 CPUs) using the DAS-3 VU University, University of Amsterdam, and Leiden clusters, interconnected by 10 Gbit/s light paths. For comparison, the figure also shows the performance on a single DAS-3/VU cluster, using the same number of CPUs. In all cases only 2 CPU cores per compute node were used, as explicit-state model checking is very memory intensive and using more cores does not significantly improve performance in the current MPI version of DiVinE.

As shown in Figure 7, up to 64 CPUs, the 10G grid obtains a performance close to that of a single cluster with the same number of CPUs. On average, every compute node sends about 11.4 MByte/s to other nodes, which does not cause wide-area capacity problems. DiVinE's use of asynchronous communication also effectively hides the latency difference in local versus remote TCP/IP communication. As a result, a 64-CPU grid configuration using 10 Gbit/s links is in fact more efficient than a 64-CPU cluster configuration using 1 Gbit/s Ethernet. The capacity of a 1 Gbit/s Ethernet is in itself sufficient, but its overhead for sending messages is higher than on Myri-10G.

However, on configurations larger than 64 CPUs, grid performance deteriorates for this model checking instance. On the other hand, scalability of the *cluster* version is still good, both with local 1 Gbit/s Ethernet and with Myri-10G interconnect, so the state graph of this model in principle allows sufficient parallelism. As the communication patterns and data volumes in the DiVinE tool used are very similar to those in Awari, we expect that some of the Awari grid optimizations can also be employed here. As with Awari, a 1 Gbit/s Internet link between the grid sites is no match for this distributed model checking tool. Even for the smaller problem sizes, the run time on a grid is already up to 10 times higher than on a local cluster.

It is important to note that the parallel efficiency of DiVinE can vary depending on the particular models being checked. In explicit-state model checkers, the

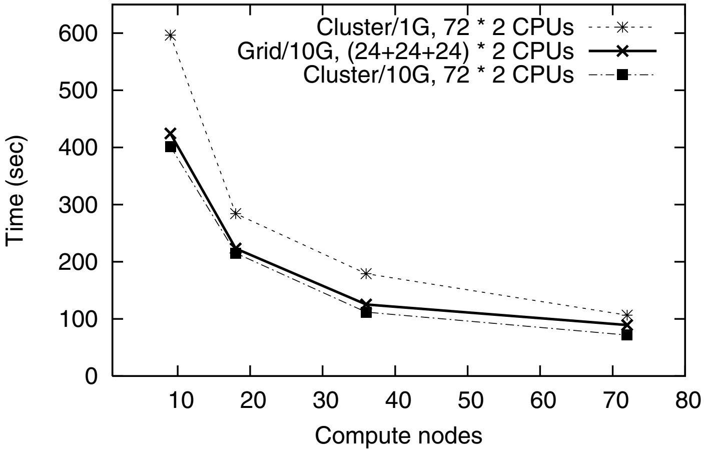

Fig. 8. Model Checking Elevator with DiVinE on 144 CPUs

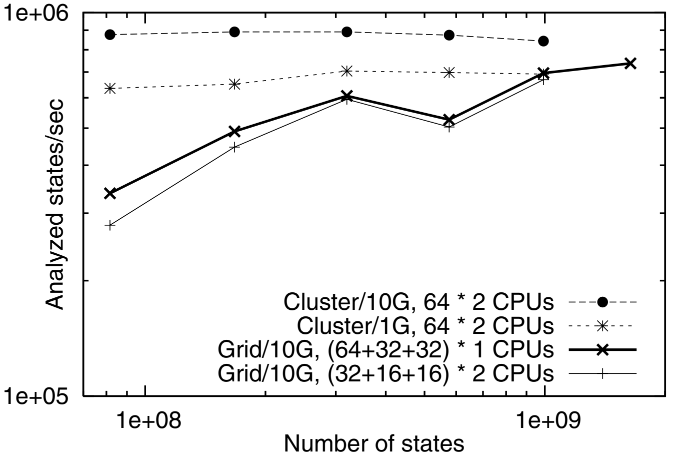

Fig. 9. Scaling the Elevator problem on 128 CPUs

actual communication/computation ratio will generally depend on several factors, for example, state graph topology, individual state size, and the complexity for computing subsequent states [14]. Also, the number of synchronizations and the load imbalance can be very different, depending on the specification. In addition to Anderson, we therefore also looked at another specification, called Elevator.

Figure 8 shows the scalability results for Elevator. It is interesting to see that Elevator has indeed much better scaling behavior than Anderson on the grid. In contrast to Anderson, up to 144 CPUs it still scales reasonably well, even though the data rates and data volumes are similar to Anderson. There are several potential causes for this difference in parallel performance; we only mentioned some of the more important ones above. A thorough investigation of the various factors involved – specifically with an eye towards further improving large-scale grid performance – would be very interesting future work.

An attractive feature of computational grids is that they can in principle also be used to solve problems that are *infeasible* on a single compute node or cluster. Explicit-state model checking is one of the important application areas where this can be very useful. Due to state space explosion, seemingly simple models are often practically impossible to be checked sequentially, since they simply do not fit into main memory. State space reduction techniques such as partial order reduction and state compression are very useful, but they help only up to a point. Cluster-based model checking, as supported by DiVinE, can be employed to shift the barrier further by employing efficient distributed memory algorithms.

As a third example we will therefore look what happens when we scale up the size of the Elevator model specification by means of a model parameter. Figure 9 shows the performance results for increasingly large instances of the Elevator specification. Horizontally is the number of elevator floors, representing the model instance; the largest instance of 13 floors requires 382 GByte of main memory for the state space alone (this instance was about 1.6 billion states big, and took 40 minutes to be checked). Vertically we indicate the performance obtained by means of the number of states processed per second. We measured performance on four configurations with 128 CPUs: a cluster with 1 and 10 Gbit/s, both with two CPUs per node, and a three-cluster grid using either one or two CPUs per node.

There are several interesting trends to be observed from this figure:

- The efficiency of the grid configurations increases with the model size, approaching that of an equally sized single cluster;
- Increasing the number of compute nodes on a grid allows checking larger problem instances, with very good performance;
- The platform's memory system allows two CPUs per node to be employed efficiently, despite the high memory load;
- On a single cluster, using a switch with a fast backplane, 1G Ethernet has sufficient capacity to achieve reasonably good performance with DiVinE. Myri-10G is faster, but for the larger problems the difference is less than 25%.

Provided that computational grids will increasingly be interconnected by high-bandwidth optical links, these platforms therefore indeed appear to offer attractive additional opportunities to efficiently search huge state spaces resulting from realistic specifications.

7 Conclusions

We have shown through several examples that it becomes more and more feasible to run challenging parallel applications on large-scale grids. Both algorithmic optimizations and advances in optical network technology make grid computing an interesting alternative. For applications like model checking, grids have the great advantage that the total amount of memory of all clusters together can be used effectively for solving a single problem. For example, we have run a model that requires almost 400 GB memory, which is hard to do on a more centralized system. We have also pointed out that there are many resemblances between solving games with retrograde analysis and model checking. Even the performance characteristics of these applications are similar. We therefore think it is interesting to study further optimizations of distributed model checkers, similar to the ones we describe for our Awari solver. Also, it is important to develop programming environments that simplify programming and deployment of grid applications, which is the topic of our ongoing research on Ibis [21].

Acknowledgement

We would like to thank Jiří Barnat, Luboš Brim, and Michael Weber for their help with DiVinE and the applications. We also thank Michael Weber and Wan Fokkink for their suggestions improving the paper. The initial work on TDS and Awari was performed largely by John Romein. The various DAS systems have been co-funded by the Netherlands Organization for Scientific Research (N.W.O.), the Netherlands National Computing Facilities foundation (N.C.F.), the Virtual Laboratory for e-Science project, the MultimediaN project, and the Gigaport project (which are supported by BSIK grants from the Dutch Ministry of Education, Culture and Science).

References

[1] Bal, H. E. et al., *The Distributed ASCI Supercomputer Project*, ACM Special Interest Group Operating Systems Review **34** (2000), pp. 76–96.

[2] Barnat, J., L. Brim and I. Černá, *Cluster-Based LTL Model Checking of Large Systems*, in: *Proc. of Formal Methods for Components and Objects*, LNCS **4111**, 2006, pp. 259–279.

[3] Barnat, J., L. Brim, I. Černá, P. Moravec, P. Ročkai and P. Šimeček, *DiVinE – A Tool for Distributed Verification*, in: *Computer Aided Verification*, LNCS **4144** (2006), pp. 278–281.

[4] Barnat, J., V. Forejt, M. Leucker and M. Weber, *DivSPIN - a SPIN compatible distributed model checker*, in: M. Leucker and J. van de Pol, editors, *4th Int. Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, Lisbon, Portugal, 2005, pp. 95–100.

[5] Cappello, F. and H. Bal, *Towards an International "Computer Science Grid" (keynote paper)*, in: *7th IEEE Int. Symp. on Cluster Computing and the Grid (CCGRID'07)*, Rio de Janeiro, Brazil, 2007, pp. 230–237.

[6] Ciardo, G., J. Gluckman and D. Nicol, *Distributed state-space generation of discrete-state stochastic models*, INFORMS Journal on Comp. **10** (1998), pp. 82–93.

[7] DeFanti, T., C. de Laat, J. Mambretti, K. Neggers and B. S. Arnaud, *TransLight: A Global-Scale LambdaGrid for E-Science*, Commun. ACM **46** (2003), pp. 34–41.

[8] Graham, R. L., T. S. Woodall and J. M. Squyres, *Open MPI: A flexible high performance MPI*, in: *Proc. 6th Ann. Int. Conf. on Parallel Processing and Applied Mathematics*, Poznan, Poland, 2005, pp. 228–239.

[9] Hammer, M. and M. Weber, *"To Store or Not To Store" reloaded: Reclaiming memory on demand*, in: L. Brim, B. Haverkort, M. Leucker and J. van de Pol, editors, *Formal Methods: Applications and Technology*, Lecture Notes in Computer Science **4346** (2006), pp. 51–66.

[10] Holzmann, G., *The Model Checker Spin*, IEEE Trans. on Software Engineering **23** (1997), pp. 279–295.

[11] Kielmann, T., R. F. H. Hofman, H. E. Bal, A. Plaat and R. A. F. Bhoedjang, MagPIe*: MPI's Collective Communication Operations for Clustered Wide Area Systems*, in: *Seventh ACM SIGPLAN Symp. on Princ. and Pract. of Parallel Programming (PPoPP'99)*, Atlanta, GA, 1999, pp. 131–140.

[12] Korf, R. E., *Depth-first Iterative Deepening: an Optimal Admissible Tree Search*, Artif. Intell. **27** (1985), pp. 97–109.

[13] Mohamed, H. H. and D. H. Epema, *The Design and Implementation of the KOALA Co-Allocating Grid Scheduler*, in: *Advances in Grid Computing - European Grid Conf. 2005*, LNCS **3470** (2005), pp. 640–650.

[14] Pelánek, R., *BEEM: Benchmarks for Explicit Model Checkers*, in: *Proc. of SPIN Workshop*, LNCS **4595** (2007), pp. 263–267.

[15] Plaat, A., H. E. Bal and R. F. Hofman, *Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects*, in: *Proc. of High Performance Computer Architecture (HPCA-5)*, Orlando, FL, 1999, pp. 244–253.

[16] Romein, J. W. and H. E. Bal, *Solving the Game of Awari using Parallel Retrograde Analysis*, IEEE Computer **38** (2003), pp. 26–33.

[17] Romein, J. W., H. E. Bal, J. Schaeffer and A. Plaat, *A Performance Analysis of Transposition-Table-Driven Work Scheduling in Distributed Search*, IEEE Trans. on Parallel and Distributed Systems **13** (2002), pp. 447–459.

[18] Romein, J. W., A. Plaat, H. E. Bal and J. Schaeffer, *Transposition Driven Work Scheduling in Distributed Search*, in: *Proc. 16th AAAI National Conference*, Orlando, FL, 1999, pp. 725–731.

[19] *StarPlane project*, http://www.starplane.org.

[20] Stern, U. and D. L. Dill, *Parallelizing the Murφ verifier*, in: O. Grumberg, editor, *9th Int. Conf. Computer-Aided Verification*, LNCS **1254** (1997), pp. 256–267.

[21] van Nieuwpoort, R. V., J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann and H. E. Bal, *Ibis: a Flexible and Efficient Java-based Grid Programming Environment*, Concurrency & Computation: Practice & Experience **17** (2005), pp. 1079–1107.

[22] Verstoep, K., J. Maassen, H. E. Bal and J. W. Romein, *Experiences with Fine-grained Distributed Supercomputing on a 10G Testbed*, in: *Proc. 8th IEEE Int. Symp. on Cluster Computing and the Grid (CCGRID'08)*, Lyon, France, 2008, pp. 376–383.