Experiences with Fine-grained Distributed Supercomputing on a 10G Testbed

Kees Verstoep, Jason Maassen and Henri E. Bal
Dept. of Computer Science, Faculty of Sciences
Vrije Universiteit, Amsterdam, The Netherlands
{versto,jason,bal}@cs.vu.nl

John W. Romein
Stichting ASTRON
Dwingeloo, The Netherlands
romein@astron.nl

Abstract

This paper shows how lightpath-based networks can allow challenging, fine-grained parallel supercomputing applications to be run on a grid, using parallel retrograde analysis on DAS-3 as a case study. Detailed performance analysis shows that several problems arise that are not present on tightly-coupled systems like clusters. In particular, flow control, asynchronous communication, and host-level communication overheads become new obstacles. By optimizing these aspects, however, a 10G grid can obtain high performance for this type of communication-intensive application. The class of large-scale distributed applications suitable for running on a grid is therefore larger than previously thought realistic.

1. Introduction

Most computational grids are currently used to provide a combined, shared resource to run workloads of single-cluster or even single-node jobs, also known as high throughput computing. Running *challenging*, fine-grained parallel applications has thus far largely remained the domain of traditional high-end, tightly-coupled supercomputers. Up until recently, wide area networks, certainly as provided by the general purpose Internet, had insufficient capacity to fulfill the high demands posed by "grand challenge"-style parallel applications. However, recently lightpath-based Optical Private Networks (OPN) have been introduced, both nationally and internationally, offering wide area bandwidths of 10 Gbit/s and more [3].

Intuitively, computational grids with 10G wide area connectivity should be suitable to run parallel applications that are computation- and communication-intensive but latency-insensitive. As a concrete example of this class, we will study a retrograde analysis application (Awari) that can send up to 10 Gbit/s but that is largely asynchronous.

As we will show, however, there still are a number of important problems to be addressed to obtain good performance on a 10G grid. First, the execution on a heterogeneous grid causes additional flow control issues. Second, asynchronous communication must be implemented carefully, as the communication characteristics are no longer identical between each host pair. Third, host-level communication overheads become much more significant compared to a highly tuned local interconnect. All these aspects are relatively unimportant on a single, tightly-connected cluster, but they prove to be essential for good performance on a grid. The contributions of the paper are that we analyze the impact of these performance problems, and describe optimizations that resolve these obstacles for an important class of fine-grained, distributed applications. For Awari, we are able to achieve an efficiency improvement of 50%, with performance on a 10G grid close to that on a single larger cluster using the same communication protocols.

The remainder of this paper is structured as follows. We first discuss related work and introduce DAS-3, the hardware platform used for our experiments. We then give an overview of the parallel implementation of Awari and discuss initial cluster and grid performance of the application. Next, we discuss the optimizations that were vital for improving grid performance significantly. We then discuss our findings and conclude.

2. Background and Related Work

Only occasionally researchers report about ambitious attempts to use a grid as a distributed system to run truly fine-grained, parallel applications efficiently [5]. Typically, this means modifying the application itself, altering the communication pattern to avoid wide-area communication at all cost, often trading communication for additional computation [10].

Many parallel algorithms remain inherently difficult to be run on a grid. These applications have a high communication/computation ratio, are largely immune to all known optimizations in that respect, yet no alternative, more efficient algorithms are known [9]. In this paper we focus on one such application: retrograde analysis, which is a commonly used technique for analyzing games. Recently, the game of Checkers has been solved by a combination of forward search and retrograde analysis [12]. Other application domains employ communication patterns that can benefit from optimizations similar to the ones we describe. An important example is distributed model checking, which, like retrograde analysis, searches huge state spaces [1].

Prominent projects in e-Science have recently begun processing ever larger data sets produced by remote experiments (e.g., CERN's LHC in High-Energy Physics and the Dutch LOFAR project in Astronomy). If all this data would be routed across the national backbones, other Internet traffic would be completely swamped. For this reason, several NRENs (National Research and Educational Networks) worldwide have started to provide research institutes with so-called *lambdas*, i.e., dedicated wavelengths of light, each typically providing 10 Gbit/s of bandwidth on an optical DWDM (Dense Wavelength Division Multiplexing) network [3, 16]. How these lambdas can be employed most effectively in areas like distributed computing, data mining and visualization, is a major topic of research [8,13]. In this paper, we focus on a concrete, communication-intensive example application on an advanced optical grid, discussed next.

3. DAS-3 and StarPlane

DAS-3 (the Distributed ASCI Supercomputer 3) is a heterogeneous five-cluster, wide-area distributed system for Computer Science research in the Netherlands. As one of its distinguishing features, DAS-3 employs a *dedicated* wide-area interconnect based on lightpaths. Besides using the ubiquitous 1 and 10 Gbit/s Ethernet, DAS-3 is exceptional in that it employs the high speed Myri-10G networking technology from Myricom both as an internal high-speed interconnect as well as an interface to remote DAS-3 computing resources.

Wide-area communication is supported by the fully optical DWDM backbone of SURFnet-6 (the Dutch NREN). The goal of the *StarPlane* [14] project, in which the Vrije Universiteit (VU) and the University of Amsterdam (UvA) collaborate, is to allow part of the photonic network infrastructure of SURFnet-6 to be manipulated by grid applications to optimize the performance of specific e-Science applications. Figure 1 shows the DAS-3 testbed and the StarPlane network as used in this paper. DAS-3 is heterogeneous: every cluster has AMD Opteron CPUs with different clock speeds and/or number of CPU cores. All nodes are dual-CPU SMPs, each CPU having either 1 or 2 cores, giving 2 or 4 cores per compute node.

For this paper we used OpenMPI [4], which is able to access the Myri-10G network in multiple ways. In single-cluster runs, the most efficient protocol is Myri-10G's na-

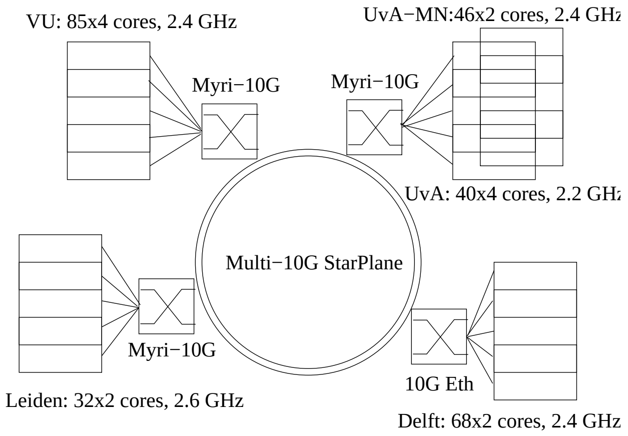

Figure 1. DAS-3 clusters interconnected by StarPlane.

tive MX layer, which is directly supported by OpenMPI. For grid-based runs, we cannot use MX but run the same application with OpenMPI in TCP/IP mode, which uses the kernel's sockets interface to the same Myri-10G device. OpenMPI is both efficient and versatile in that a specific network can be selected at runtime; we used this feature frequently during the optimization of our application.

The latencies between the DAS-3 clusters are determined by both layer 1 layout (the physical topology of the DWDM network) and layer 2 aspects: Ethernet's standard Spanning Tree Protocol running between the 10G switches over StarPlane reduces the logical ring back into to a tree. Due to the proximity of the DAS-3 clusters in the Netherlands, this results in low round-trip latencies of resp. 1.40 msec (VU–Leiden), 1.26 msec (Leiden–UvA) and 2.64 msec (UvA–VU). Still, these WAN latencies are orders of magnitude higher than both the local MX latency (below 10 μsec) and the local IP latency (below 30 μsec). The compute nodes of DAS-3/Delft have no Myri-10G NICs. Therefore, this cluster is not used in this paper.

4. Implementing Awari on DAS-3

Awari is an ancient two-player board game, originating in Africa. The game starts with a configuration of 48 stones evenly divided over 12 "pits", 6 pits for each player. Players can move and capture stones according to certain rules [11]. The Awari application used for this paper computes the best possible move for *any* given position using a technique known as retrograde analysis [6]. The idea is to gradually build up databases solving the complete game. We start with the smallest databases for simple end-game positions (i.e., the empty board, the ones with one stone, two stones, etc.) and gradually reason backward, building up more and more knowledge until for every position the best possible move is known. During a phase with N stones on the board, we need access to the previously computed databases with up to N-2 stones due to the capturing moves leading to positions already known.

An important reason to use a parallel algorithm is the very large state space of Awari: keeping a large fraction of the almost 900 *billion* board positions in main memory is required to search efficiently, yet practically impossible on a single computer. On the other hand, the distribution of states over multiple compute nodes causes computing the optimal moves in the game to result in an extremely high communication volume. To still achieve good performance, *asynchronous* communication is used. We will first discuss the parallel implementation of Awari and its performance on a single DAS-3 cluster and focus on the aspects that are relevant to understand the parallel performance.

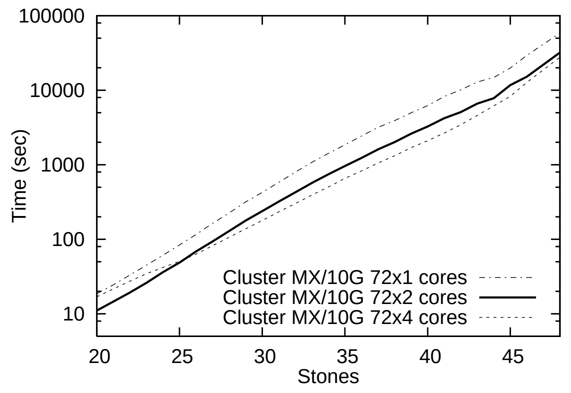

Figure 2. Performance of Awari on the DAS-3/VU cluster with high speed Myri-10G interconnect using MX.

Implementing Awari on a cluster

Board positions are represented as nodes of a directed graph, linked through edges indicating the moves between those positions. When building the Awari database for N stones, all possible board positions with N stones are given an index, which are evenly spread over the compute nodes. Each compute node is thus responsible for an equally-sized portion of the entire state space. A part of the state space can be directly evaluated using only local information (e.g., if a board position only depends on previously determined evaluations for capturing moves which are already computed), but in general a large portion of the states depends on state information residing elsewhere.

The communication pattern induced by the state distribution is highly random, but also evenly spread. The parallel performance is limited by the slowest participating compute node, unless additional load balancing would be implemented, e.g., by altering the state distribution function. As mentioned above, an essential aspect of the efficient parallel implementation is that the resolution of interrelated remote states is done completely asynchronously, thus making it latency tolerant. Rather than directly fetching remote state information using round-trip communication, job descriptors regarding the evaluation of board positions are sent over to other CPUs. Upon arrival, these jobs are resolved directly or pushed onward to other CPUs, should additionally required information reside elsewhere. Another important aspect is that these asynchronous jobs can be accumulated into larger messages, thus allowing the grain size to be increased, reducing communication overhead and improving parallel efficiency significantly.

One complication is that for the larger databases, the state space containing the evaluated scores of all positions becomes so big that it can no longer be kept in core. Yet, for fast computation of the dependent states, all states need to be present in main memory (as state accesses are random, disk accesses would ruin performance). This issue is resolved by computing the game-theoretical scores for all states in several *phases*, using an increasing bound on the states' score during each round [11]. Each phase ends with a distributed termination detection algorithm to ensure that all states are resolved. The updated state information is then sequentially written to the local disks.

The parallel efficiency of Awari increases with problem size: for problems up to 30 stones, it is already over 50% in a 72-node dual-core configuration. For larger problems, efficiency is even higher, but sequential runs then quickly become infeasible due to the limited amount of memory. Figure 2 shows the performance of the base implementation on a single cluster using the Myri-10G interconnect with MPI over MX. The 72-node dual-core configuration is a factor 1.9 faster than the 72-node, single-core configuration. The 72 node, 4-core configuration is about 30% faster still for medium to large problem databases, but this advantage diminishes to only 14% for the larger database instances due to local disk I/O contention, as the 4 cores share the same disk (for small problems, higher synchronization costs due to the larger total number of cores cause performance to be lower still). Also, not all DAS-3 clusters have 4-core compute nodes. We therefore use 72 nodes with dual CPU cores (i.e., 144 CPU cores in total) as the base configuration in the remainder of the paper. In all cases where the total number of CPU cores on a compute node is higher than the number of Awari processes, the NUMA-aware scheduler of the Linux kernel will allocate the processes on different CPUs, and memory is allocated from memory banks most closely attached to the CPU. The remaining cores on these CPUs are left unused, so the performance comparison is fair.

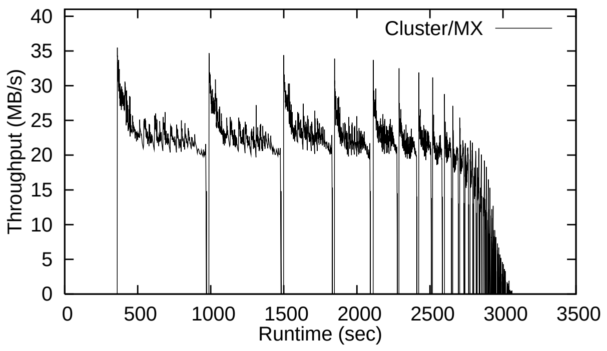

Figure 3. MX send/receive throughput per core on DAS-3/VU for the 40 stones DB.

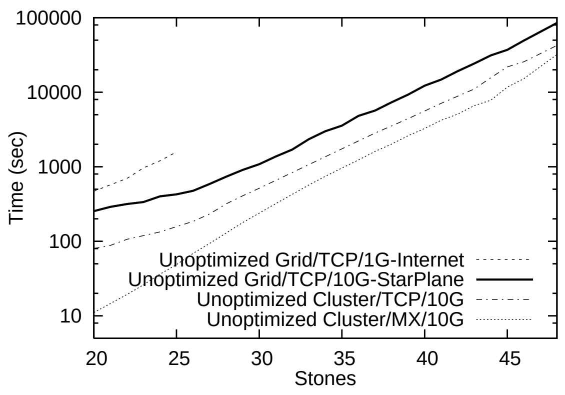

Figure 4. Performance of the unoptimized version of Awari on DAS-3/VU and a grid of three DAS-3 clusters.

Figure 3 shows the send/receive throughputs during the computation of the 40-stone database of Awari on the DAS-3/VU cluster with 72 dual-core nodes. The presence of iterative computing phases with increasing bound, which are also the most throughput-intensive ones, can be clearly identified. Note that the per-core throughput is not itself limited by the 10 Gbit/s peak bandwidth (as with almost any realistic fine-grained application) but the cumulative *sustained* network throughput during a large portion of the total runtime is 144 times 25 MByte/s, i.e., 28.8 Gbit/s!

Implementing Awari on a grid

Given the demanding communication pattern, it is an interesting question how feasible it would be to run Awari on a grid with high speed network links. The required wide area bandwidth is certainly very high: on a grid with three equally-sized clusters, two thirds of the data transfers will go across wide area links.

As the base implementation of Awari was available with an MPI communication module, it would seem the application should almost immediately be able to run on the grid using a TCP implementation of MPI. However, given the heterogeneous nature of our DAS-3 grid – as is the case for most grids – we need to make sure that the faster-running CPUs will not overwhelm the slower ones with an almost infinite stream of asynchronous work to be performed. We found this quickly leads to unrestricted job queue growth, eventually causing paging (thus reducing performance), until compute nodes finally run out of memory.

Implementing an additional *point-to-point* flow control mechanism is possible, but this would add complexity and overhead to the parallel implementation. There are also several subtleties: if a node simply tells others to back down sending, these other nodes do need to keep on processing incoming messages (still generating more work that has to be held) to assure progress. But the congested node can only get out of this state by completing its current work queue, again often generating more work for others (and potentially for itself at a later time). Unless carefully designed with these considerations in mind, point-to-point flow control could therefore easily lead to deadlock.

However, on DAS-3 the network latencies are sufficiently moderate that *global* synchronizations can be employed as a simple alternative. Adding these global synchronizations to the various phases of the application (a barrier each time after having processed a certain amount of local states) turned out to be sufficient to avoid extreme local congestion. An important reason why this approach works well for Awari, is that the amount of processing is well-balanced over the CPUs due to the random spread of states and dependencies among them.

Figure 4 shows the performance of the base implementation of Awari on a DAS-3 grid consisting of three clusters (24 nodes with 2 cores on each of DAS-3/VU, DAS-3/Leiden and DAS-3/UvA) compared to a single-cluster configuration (using MX on 72 nodes with 2 cores at DAS-3/VU). For comparison, the figure also includes Cluster/TCP performance (using TCP over Myri-10G on DAS-3/VU), as that can be considered a baseline for DAS-3 Grid/TCP performance. Furthermore, the figure shows the performance of Awari when using traditional 1 Gbit/s interfaces on the compute nodes and coupling the sites over the public Internet via the universities' backbones. Several conclusions can already be drawn from this:

- Traditional 1 Gbit/s Ethernet with regular 1 Gbit/s uplinks to the public Internet is clearly no match for this application. We limited ourselves to smaller database instances to avoid congesting links shared with regular university traffic, but we already see over 300% slowdown compared with faster private networks.

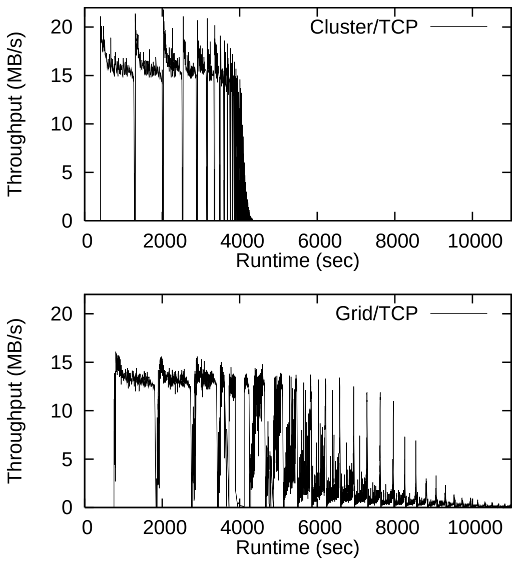

Figure 5. Unoptimized Cluster- and Grid/TCP throughput per core for the 40 stones DB.

- TCP protocol overhead compared with highly optimized MX over Myri-10G appears to be significant, considering the roughly 70% percent increase in runtime when using MPI over TCP on a single cluster, even though the same Myri-10G NICs are used in both cases.
- On the larger problem sizes, for which Awari spends a significant fraction of its time in the compute- and communication intensive stages, the performance over StarPlane follows the single-cluster performance over TCP with 10 Gbit/s at a stable distance, but there still is an additional factor 2.1 performance loss to be accounted for.

Figure 5 shows the send and receive throughput per CPU core, monitored over the runtime of Awari for the 40-stone database on the DAS-3 grid. For comparison, we also include the single-cluster DAS-3/VU throughput when using TCP. Clearly, average throughput is much reduced compared with the performance over Myri-10G using MX. Of the average throughput of 13.5 MByte/s per CPU core, two thirds is sent to other clusters, resulting in an accumulated WAN throughput of $9.0 * 48 = 432$ MB/s. This is almost 70% of the 5 Gbit/s bandwidth available between each cluster pair (the WAN links are shared due to the spanning tree topology, as explained in Section 3). Yet, reduced throughput is only a partial explanation of the overall increase in runtime. In fact, on the grid most time is spent on the less communication intensive phases. This is surprising, as the network is mostly uncongested during these phases, and the application is also designed to be latency tolerant.

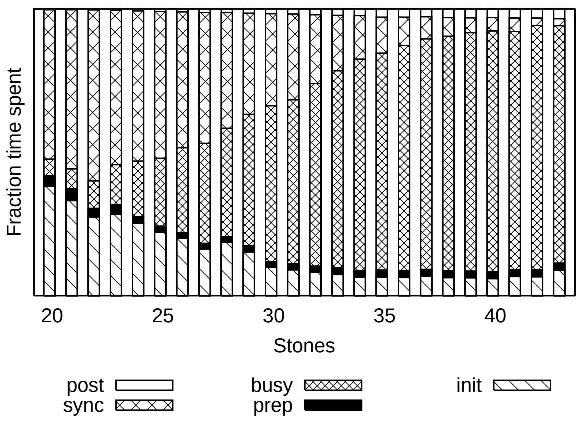

Figure 6. Split up of Awari's work phases in the Grid/TCP version.

5. Optimizing Awari for the Grid

To further investigate these performance issues, we instrumented Awari with timers around MPI calls. After analyzing the results we applied several optimizations to the communication sublayer which we will discuss next. Most of these optimizations are quite general and will apply to many other parallel applications on the grid.

Do not just look at peak performance

Considering the huge slowdown for the smaller problems, we plotted accumulated times for the various phases of the application; see Figure 6. The most important phases are "init" (reading previous databases), "busy" (processing the current database), and "sync" (processing during the distributed termination detection phase). Clearly, for the smaller problems the issue with reduced performance is not so much inability to transfer large volumes of data efficiently while processing states: the busy phase takes only a fraction of the time. The overheads induced by synchronization and sending and receiving smaller messages were much more important here. Similar effects were seen for the larger databases, during the later phases.

It is important to note the relevance of the LogP model for parallel applications here [7]. Often, an implicit assumption is made that a parallel application will perform well on a "low latency" and/or "high throughput" network, while in reality the host-level *overhead* of sending the data across the network is frequently a much more determining factor [15].

MPI primitive	Cluster MX	Cluster TCP	Grid TCP
MPI_Isend	6.8	18.3	17.8
MPI_Recv	5.3	7.3	7.2
MPI_Iprobe (failed)	2.3	60.6	59.3
MPI_Iprobe (success)	4.7	3.8	3.9

Table 1. Average MPI overhead in μs on the DAS-3/VU cluster and a DAS-3 grid.

As shown in Table 1, the send, receive and polling overhead (i.e., the number of compute cycles spent in the host-level network protocol layers) differ significantly for TCP/IP and Myri-10G's native MX (for Cluster/TCP and Grid/TCP they are almost identical). The "failed" MPI probes are in fact places where the OpenMPI implementation performs work related to asynchronously posted send or receive operations – a simple reduction of the amount of MPI probes causes this overhead to be shifted to other invocations of the single-threaded MPI layer. Also, in the case of TCP, much of the work for a succesful Iprobe may actually have been done in a previously unsuccessful Iprobe. On the grid, each CPU core sends and receives more than 1200 messages a second, averaged over the entire runtime of Awari; the polling rate is a multiple of that, depending on the data intensity of the application phase. In total, 14.8 *billion* messages with an average size of 2.0 KB are transferred for the largest database. Host-level overhead is thus indeed a significant factor explaining the overall difference in performance.

Implement scalable global synchronization

The original implementation used a ring to synchronize all CPUs. Upon arrival of a synchronization message, a compute node would also flush its message buffers to other nodes in a straightforward fashion. This simple approach worked very well on a single cluster with low-latency interconnect and native flow control.

However, this turns out to cause several problems on the Grid/TCP and on the Cluster/TCP implementation. First, the increased latency over a TCP/IP network causes synchronization rounds to last much longer, during which time only some of the compute nodes are truly active processing and sending messages. Furthermore, flushing messages to other nodes using a fixed order triggered by synchronization causes slowdown due to flow control issues and receiver overrun. Also, as explained above, for the heterogeneous grid implementation, more synchronizations are needed to prevent overrunning (slightly) slower CPUs, so the impact of synchronization costs is even higher.

We improved the performance of global synchronizations in two ways. First, in the communication module we implemented a scalable, tree-based synchronization primitive that is used for both barriers and distributed termination detection. Second, the termination detection phase itself was modified. The optimized version no longer flushes messages upon receiving synchronization messages. Instead, while the application moves to a synchronization phase, the flushing of (incomplete) message buffers is done *autonomously* by the compute nodes themselves. After a certain number of MPI polls (which are done continuously while the application is synchronizing), the communication module now automatically flushes a certain fraction of the largest pending messages. As a result, compute nodes are no longer dependent on the timely arrival of synchronization messages. This also conveniently spreads the traffic over the network due to the random distribution of pending outgoing message sizes.

Make communication really asynchronous

The original version of Awari was already written using asynchronous communication primitives: message transfers were initiated using MPI_Isend, and the resulting descriptors were put into a circular buffer so that they could be recycled later. This indeed performs well, provided communication is evenly spread. Regarding data volume this is the case for Awari, but on the grid not all communication has identical performance: part goes to nodes on the local cluster (via a switch with a very high bandwidth backplane), but the remaining data also needs to be transferred over wide area links. These links do have high bandwidth, but still lower than the switch backplane. Also, the limited amount of buffering at the 10 Gbit/s WAN ports of the Myri-10G switches will on average induce more TCP-level retransmissions with steeply increasing timeouts (due to standard TCP congestion protocols).

The effect is that asynchronous remote transfers are sometimes pending so long that they cause new transfers to *other* destinations to stall, because an old MPI_Isend call must first be completed at that point. It is hard to predict how large the MPI descriptor ring should be made to avoid the described effect. Instead we resolved this by using a descriptor ring *per destination* rather than a shared global one. Pending remote transfers no longer block other ones, unless the network congestion is so severe that a ring becomes completely filled with uncompleted send requests.

Polling messages is not free

As the Awari application involves a huge amount of random communication that is initiated asynchronously, the question is when and how this data should be received. Overall, every node sends about as much data as it receives, so a reasonable first-order strategy is to poll for incoming messages each time a compute node sends a message of its own. However, this is not enough: due to CPU speed differences, network congestion, etc., occasionally data will be available while the node is not sending for a while or vice versa. For that reason, the main computation loops of the application also frequently poll themselves, with good performance for the Cluster/MX configuration.

However, analysis of the MPI statistics showed that for the TCP-based configurations, performance was *highly* dependent on the polling rates during the various phases. As shown in Table 1, the polling overhead is noticeably higher for TCP than for MX (e.g., due to system calls involved with TCP sockets). Polling too frequently therefore results in low throughput due to polling overhead (since many polls will be unsuccessful), while not polling frequently enough will result in reduced throughput due to MPI-level flow control. Note from Figure 5 that data volume in the first phases of the application is much higher than in the later ones. This also influences the required polling rate. In the optimized version the polling rates are tuned accordingly, with notable reduction in overhead.

The optimal grain size is network dependent

The grain size of the Awari application is determined by the amount of message combining applied before sending. The original version used 4 KByte buffers for this, but on 10 Gbit/s Ethernet the use of 8 KByte buffers results in better performance. The reason is two-fold: since the 10G network supports "jumbo" packets (with an MTU of 9000 rather than 1500 bytes), the packet rate, and hence interrupt rate goes down, and also the per-packet MPI send and receive overheads are reduced. Often, increasing the grain size comes with the risk of increasing load imbalance. In this particular application this does not happen due to the random spread of states and their dependencies. Also, an 8 KByte grain size is still extremely small compared to the overall state space for the larger databases.

Optimized Awari grid performance

Figure 7 shows the performance of the optimized implementation. Clearly, the application now runs much more efficiently on all problem sizes: on the grid it is on average over 50% faster than the original version; some of the optimizations in fact also help single-cluster performance, especially the TCP version, albeit to a smaller extent. The grid performance is now also much closer to the performance on a single large cluster using TCP. The remaining performance difference (mostly below 15%) is largely explained by the heterogeneous nature of our computational grid: each CPU core currently has the same amount of work

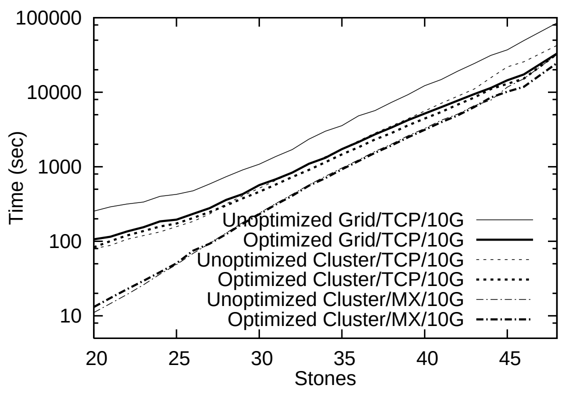

Figure 7. Performance of the optimized Awari on DAS-3/VU and a DAS-3 grid.

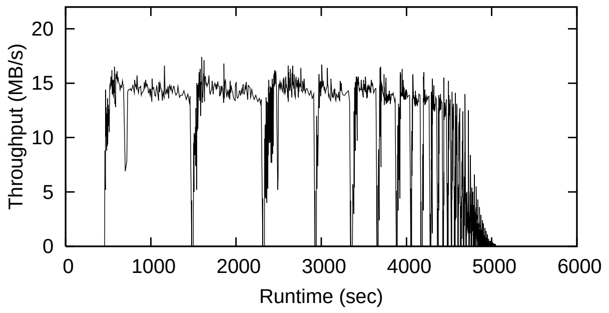

Figure 8. Optimized send/receive throughput per core for the 40 stones DB on a DAS-3 grid.

to do, irrespective of its speed. Additional load balancing while assigning states to nodes would diminish this effect.

To further assess the impact of the optimizations, we recomputed a medium-sized problem on the grid with only subsets of the five optimizations enabled. The addition of the two scalable global synchronization optimizations turned out to have the largest impact, accounting for respectively 30 and 45% of the performance improvement. The asynchronous communication optimization had a relative impact of another 15%, while the polling and grain size optimizations gave smaller improvements of about 5% each. Figure 8 shows the throughputs of the final implementation with all five optimizations enabled. Note that, compared to Figure 5, the overall throughput has indeed increased somewhat, but the largest reduction in runtime is due to the significant shrinking of the application's later work phases, which are much less data intensive.

6. Conclusions and Future Work

In this paper we discussed our experiences with running a very demanding, fine-grained parallel application on a computational grid interconnected with 10G lightpaths. With the important trend of using Optical Private Networks specifically to support grids, important new application domains now become feasible that were previously committed to traditional supercomputers or large, tightly-connected clusters. However, simply providing a huge amount of reliable, sustainable bandwidth to a parallel application developed for a controlled homogeneous platform will often turn out to be insufficient. As clearly illustrated by Awari, application- or runtime-system-specific optimizations, using truly asynchronous and latency-tolerant communication patterns with low overhead, will still be required to obtain good parallel efficiency on the grid.

For future work, we are planning to investigate several further topics in distributed application performance using lightpaths; Awari can then be used as a benchmark. Awari itself has ample possibilities for further optimization that can be investigated. For example, making use of MPI-2's one-sided operations may be beneficial to further reduce the receive and polling overhead. Furthermore, auto-tuning the most effective polling rate is conceivable by taking the timing and success of recent polls into account.

Awari's 10G wide-area link utilization is already quite high when using a distributed 144 CPU core configuration. If we further increase the amount of CPU cores, 10G WAN link capacity would become a limiting factor. However, with DAS-3 we have the option to employ the LACP Link Aggregation protocol (supported by the Myri-10G switches), allowing *multiple* 10G links between two DAS-3 sites. Application-specific dynamic link management like this is a major topic of the StarPlane project [14].

Finally, although DAS-3 is heterogeneous, in this paper we have only explored that aspect to a limited extent. Very recently, a 10G lightpath has become operational between DAS-3 and the French Grid'5000 system, allowing for experiments with compute clusters co-allocated in both the Netherlands and France. Given the additional heterogeneity aspects and higher WAN latency, efficiently running highly demanding parallel applications like Awari on this European-scale grid will be a challenging follow-up project [2].

Acknowledgments

This work is part of the StarPlane project, which is funded by the Netherlands Organization for Scientific Research (NWO). We thank SURFnet and Nortel for their efforts to provide DAS-3 with a state-of-the-art Optical Private Network. We also thank Cees de Laat, JP Velders, Paola Grosso and Li Xu at the UvA for their help in setting up the StarPlane network. DAS-3 is funded by NWO/NCF, the VL-e project (the Virtual Laboratory for e-Science), and the participating organizations.

References

[1] J. Barnat, L. Brim, and I. Černá. Cluster-Based LTL Model Checking of Large Systems. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 259–279, Nov. 2005.

[2] F. Cappello and H. E. Bal. Toward an International "Computer Science Grid" (keynote). In *7th IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid)*, pages 3–12, Rio de Janeiro, Brazil, May 2007.

[3] T. DeFanti, C. de Laat, J. Mambretti, K. Neggers, and B. S. Arnaud. TransLight: A Global-Scale LambdaGrid for E-Science. *Commun. ACM*, 46(11):34–41, Nov. 2003.

[4] R. L. Graham, 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*, pages 228–239, Poznan, Poland, Sept. 2005.

[5] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, R. Veldema, R. Hofman, C. Jacobs, and K. Verstoep. The Albatross Project: Parallel Application Support for Computational Grids. In *Proc. 1st European GRID Forum Workshop*, Poznan, Poland, Apr. 2000.

[6] T. Lincke and A. Marzetta. Large Endgame Databases with Limited Memory Space. *ICGA Journal*, 23(3):131–138, 2000.

[7] R. P. Martin, A. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proc. 24th Int. Symp. on Comp. Arch. (ISCA)*, pages 85–97, 1997.

[8] Phosphorus project. http://www.ist-phosphorus.eu.

[9] A. Plaat, H. E. Bal, and R. F. H. Hofman. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. In *Proc. 5th Int. Symp. on High Performance Comp. Arch. (HPCA)*, pages 244–253, Orlando, FL, Jan. 1999.

[10] J. W. Romein and H. E. Bal. Wide-area transposition-driven scheduling. In *Proc. 10th IEEE Int. Symp. on High Performance Distributed Computing (HPDC)*, pages 347–355, San Francisco, CA, 2001.

[11] J. W. Romein and H. E. Bal. Solving Awari with Parallel Retrograde Analysis. *IEEE Computer*, 36(10):26–33, Oct. 2003.

[12] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is Solved. *Science*, 317(5844), Sept. 2007.

[13] L. L. Smarr, A. A. Chien, T. DeFanti, J. Leigh, and P. M. Papadopoulos. The OptIPuter. *Commun. ACM*, 46(11):58–67, 2003.

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

[15] K. Verstoep, R. A. F. Bhoedjang, T. Rühl, H. E. Bal, and R. F. H. Hofman. Cluster Communication Protocols for Parallel-Programming Systems. *ACM Trans. on Computer Systems (TOCS)*, 22(3), Aug. 2004.

[16] VIOLA testbed. http://www.viola-testbed.de.