Programming Environments for High-Performance Grid Computing: the Albatross Project

Thilo Kielmann [a,*] Henri E. Bal [a], Jason Maassen [a], Rob van Nieuwpoort [a], Lionel Eyraud [b], Rutger Hofman [a], Kees Verstoep [a]

[a] *Division of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands*

[b] *École Normale Supérieure de Lyon, France*

Abstract

The aim of the Albatross project is to study applications and programming environments for computational Grids. We focus on high performance applications, running in parallel on multiple clusters or MPPs that are connected by wide-area networks (WANs). We briefly present three Grid programming environments developed in the context of the Albatross project: the MagPIe library for collective communication with MPI, the Replicated Method Invocation mechanism for Java (RepMI), and the Java-based Satin system for running divide-and-conquer programs on Grid platforms.

A major challenge in investigating the performance of such applications is the actual WAN behavior. Typical wide-area links are just part of the Internet and thus shared among many applications, making runtime measurements irreproducible and thus scientifically hardly valuable. To overcome this problem, we developed a WAN emulator as part of Panda, our general-purpose communication substrate. The WAN emulator allows us to run parallel applications on a single (large) parallel machine with only the wide-area links being emulated. The Panda emulator is highly accurate and configurable at runtime. We present a case study in which Satin runs across various emulated WAN scenarios.

Key words: Grid computing, wide-area network emulation, Albatross, MagPIe, Panda, RepMI, Satin

* Corresponding author.

Email addresses: kielmann@cs.vu.nl (Thilo Kielmann), bal@cs.vu.nl (Henri E. Bal), jason@cs.vu.nl (Jason Maassen), rob@cs.vu.nl (Rob van Nieuwpoort),

1 Introduction

The development of computational Grids opens up possibilities for completely new types of applications, ranging from access to remote data and instruments to distributed supercomputing on geographically distributed resources. Experience with several distributed supercomputing applications shows that this technique can effectively solve challenging problems that cannot be done with more tradional approaches. Examples include RSA-155 [1], SETI@home [2], and Entropia [3]. Unfortunately, these case studies are limited to parallel applications that are extremely coarse-grained.

In our research, called the Albatross project, we study whether this approach can be made more general by running medium-grained high-performance applications on a Grid. The key problem of course is the low communication performance of the wide-area networks (WANs) in a Grid, which typically are orders of magnitude slower than local interconnects. We believe, however, that in practice many parallel Grid applications will run on collections of clusters, NOWs, or supercomputers, rather than on individual workstations on the Internet. A collection of, say, clusters can be seen as a *hierarchical* system with fast local communication (over the LAN) and slow wide-area communication (over the WAN). We therefore study how parallel applications can be optimized to run efficiently on hierachical systems. To do useful performance experiments, we also have built a geographically distributed cluster system, called DAS, which consists of four Myrinet-based clusters located at different universities in The Netherlands.

In the first phase of the Albatross project, we have succesfully optimized many medium-grained applications to run efficiently on a DAS-like system, showing that there is far more opportunity for distributed supercomputing than may be expected. Next, we have developed several programming environments that ease the development parallel Grid applications. Each environment takes the hierarchical structure of the Grid into account and optimizes certain aspects: MagPIe (an MPI library) optimizes collective communication, RepMI (a Java extension) supports object replication on Grids, and Satin is a Java-centric divide-and-conquer system that optimizes load balancing. In the paper, we summarize these three programming environments briefly. Finally, we describe new research that aims at a methological performance evaluation of parallel applications and programming systems on a Grid. The key idea is the development of a testbed that emulates a Grid on a single large cluster and supports various user-defined performance scenarios for the wide-area links of the emulated Grid. We give a detailed performance evaluation of several load

`leyraud@ens-lyon.fr` (Lionel Eyraud), `rutger@cs.vu.nl` (Rutger Hofman), `versto@cs.vu.nl` (Kees Verstoep).

balancing algorithms in Satin using this testbed.

The outline of the paper is as follows. In Section 2 we describe the DAS system and the three programming environments MagPIe, RepMI, and Satin. In Section 3 we describe the Panda wide-area emulator. In Section 4 we present the case study for the Satin load balancing algorithms. Finally, Section 5 discusses related work and Section 6 concludes.

2 The Albatross Grid Programming Environments

The Albatross project started by investigating the behavior of medium-grained parallel applications, running on multiple cluster computers that are connected by wide-area links [4–6]. Our experimentation platform is the Distributed ASCI Supercomputer (DAS), as shown in Fig. 1. It consists of Myrinet-based cluster computers located at four Dutch universities that participate in the ASCI research school. [1] Each DAS compute node is a 200 MHz Pentium-Pro, running RedHat Linux. By the end of 2001, a follow-up system, called DAS-2, will be operational. DAS-2 will consist of five Myrinet-based clusters with dual Pentium-III nodes, enabling us to investigate the behavior of parallel applications on multiple clusters of SMPs.

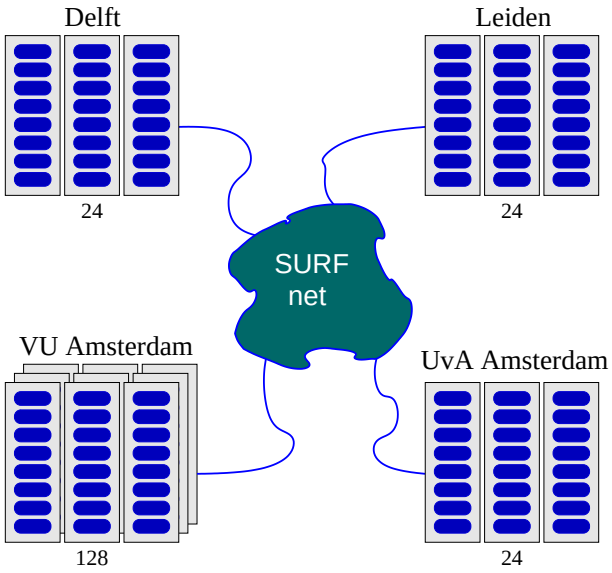

Fig. 1. The wide-area DAS system

The findings from [4–6] indicate that parallel applications that have been written for homogeneous systems (like a single cluster computer) do not run efficiently on multi-cluster systems with hierachical network interconnects.

[1] The ASCI research school is unrelated to, and came into existence before, the Accelerated Strategic Computing Initiative.

However, most applications can be *rewritten* in order to tolerate the high latency and the low bandwidth of the WAN links. High WAN latency can be tolerated by overlapping computation with asynchronous communication. Low WAN bandwidth can be tolerated by reducing communication overhead, both by avoiding redundant communication between clusters and by combining several short messages into longer ones that can be processed more efficiently. However, such applications can not run efficiently on multi-cluster systems that either inherently require high inter-cluster bandwidth or that rely on frequent synchronization between processes. In the latter case, the high WAN latency causes the performance problems.

Our manual modifications to the application source code were effective but also increased code complexity. In an ideal case, the multi-cluster aspects of communication should be separated from the application-specific parts of the source code. For this purpose, we developed the Grid programming environments presented in the following subsections.

2.1 MagPIe

The collective communication operations as defined by the MPI standard [7] describe an important set of communication patterns occuring between groups of processes. Frequently used examples are the broadcast, barrier, and reduce operations. Our MagPIe library [8,9] implements MPI's collective operations with optimizations for wide area systems (Grids). Existing parallel MPI applications can be run on Grid platforms using MagPIe by relinking the programs with our library. No change in application code is necessary. MagPIe is independent of the underlying MPI platform. MagPIe has a simple API through which the underlying Grid computing platform (Panda, in our case) provides the information about the number of clusters in use, and which process is located in which cluster.

MagPIe's basic idea is to adapt MPI's collective algorithms to the hierarchical shape of Grid-based systems. Our hierarchical collective algorithms speed up collective completion time by reducing the utilization of the slow wide-area links to the necessary minimum. For this purpose, MagPIe ensures that each sender-receiver path contains at most one wide-area link and that each data item is sent at most once to each receiving cluster. We have shown in [8,9] that MagPIe significantly reduces the completion times of individual collective operations as well as that of parallel applications, compared to Grid-unaware collective algorithms. Actual performance improvements depend on the number of clusters and on WAN latency/bandwidth. With long messages, wide-area bandwidth needs to be utilized carefully. MagPIe achieves this by splitting long messages into small segments which can be sent in parallel over multiple

wide-area links.

2.2 RepMI

Our work in [4] investigated the use of Java RMI for running parallel applications on Grid platforms. We found that manually optimized Java applications can indeed run efficiently on a Grid platform, at the price of using RMI in a style resembling "message passing." Sharing objects using RMI, however, leads to prohibitive performance penalties.

An important observation is that many shared objects have a very high ratio of read to write operations. Using object replication can help solving the performance problems for such objects. For this purpose, we have delevoped the Replicated Method Invocation mechanism (RepMI) [10]. RepMI is a compiler-based approach for object replication in Java that is designed to resemble a Remote Method Invocation. Our model does not allow arbitrarily complex object graphs to be replicated, but deliberately imposes restrictions to obtain a clear programming model and high performance. Briefly, our model allows the programmer to define closed groups of objects, called clouds, that are replicated as a whole. A cloud has a single entry point, called the root object, on which its methods are invoked. The compiler and runtime system together determine which methods will only read (but not modify) the object cloud; such read-only methods are executed locally, without any communication. Methods that modify any data in the cloud are broadcast and applied to all replicas. RepMI implements a MagPIe-like broadcast operation for Grid environments. The semantics of such replicated method invocations are similar to those of RMI. We have implemented RepMI in the Manta high-performance Java system [11].

2.3 Satin

Satin's programming model is an extension of the single-threaded Java model. To achieve parallel execution, Satin programs do not have to use Java's threads or Remote Method Invocations (RMI). Instead, they use the much simpler divide-and-conquer primitives. Satin does allow the combination of its divide-and-conquer primitives with Java threads and RMIs. Additionally, Satin provides shared objects via RepMI.

We augmented the Java language with three keywords, much as in the Cilk [12] system: `spawn`, `sync`, and `satin`. The `satin` modifier is placed in front of a method declaration. It indicates that the method may be spawned. The `spawn` keyword is placed in front of a method invocation to indicate possibly paral-

lel execution. We call this a *spawned method invocation*. Conceptually, a new thread is started for running the method upon invocation. Satin's implementation, however, eliminates thread creation altogether. A spawned method invocation is put into a local work queue. From the queue, the method might be transferred to a different CPU where it may run concurrently with the method that executed the `spawn`. The `sync` operation waits until all spawned calls in the current method invocation are finished; the return values of spawned method invocations are undefined until a `sync` is reached. A detailed description of Satin's implementation can be found in [14].

Spawned method invocations are distributed across the processors of a parallel Satin program by work stealing from the work queues mentioned above. In [15], we presented a new work stealing algorithm, *Cluster-aware Random Stealing* (CRS), specifically designed for cluster-based, wide-area (Grid computing) systems. In Section 4, we will present a case study with Satin running across a variety of emulated wide-area network scenarios. We run four parallel applications, for each comparing the following three work stealing algorithms. A detailed description of Satin's wide-area work stealing can be found in [15].

Random Stealing (RS) RS attempts to steal a job from a randomly selected peer when a processor finds its own work queue empty, repeating steal attempts until it succeeds [12,13]. This approach minimizes communication overhead at the expense of idle time. No communication is performed until a node becomes idle, but then it has to wait for a new job to arrive. On a single-cluster system, RS is the best performing load-balancing algorithm. On wide-area systems, however, this is not the case. With C clusters, on average $(C-1)/C \times 100\%$ of all steal requests will go to nodes in remote clusters, causing significant wide-area communication overheads.

Cluster-Hierarchical Stealing (CHS) CHS has been proposed for load balancing divide-and-conquer applications in wide-area systems [16,17]. CHS minimizes wide-area communication. The idea is to arrange processors in a tree topology, and to send steal messages along the edges of the tree. When a node is idle, it first asks its child nodes for work. If the children are also idle, steal messages will recursively descend the tree. Only when the entire subtree is idle, messages will be sent upwards in the tree (e.g., across WAN links), asking parent nodes for work. CHS has the drawback that all nodes of a cluster have to become idle before wide-area steal attempts are started. During the round-trip time of the steal message, the entire cluster remains idle.

Cluster-aware Random Stealing (CRS) In CRS, each node can directly steal jobs from nodes in remote clusters, but at most one job at a time. Whenever a node becomes idle, it first attempts to steal from a node in a remote cluster. This wide-area steal request is sent asynchronously: Instead of waiting for the result, the thief simply sets a flag and performs additional, synchronous steal requests to randomly selected nodes within its own clus-

ter, until it finds a new job. As long as the flag is set, only local stealing will be performed. The handler routine for the wide-area reply simply resets the flag and, if the request was successful, puts the new job into the work queue. CRS combines the advantages of RS inside a cluster with a very limited amount of asynchronous wide-area communication. In Section 4 we will show that CRS performs almost as good as with a single, large cluster, even in extreme wide-area network settings.

3 The Panda Wide-area Network Emulator

On the Distributed ASCI Supercomputer (DAS) system, parallel programming environments run on top of our Panda communication library [18]. For MPI-style message passing, we ported the MPICH library [19] to run on top of Panda. Both RepMI and Satin use our Manta high-performance Java system [11], which also communicates via Panda.

Panda provides an efficient portability layer for parallel applications and runtime systems. Its lower, system-level modules provide threads and communication primitives. Panda's interface modules provide higher-level communication like message passing, remote procedure call (RPC), and group communication. Panda adapts itself to the underlying communication system; e.g. it implements reliable communication if the underlying network does not guarantee packet delivery. The nodes within a DAS cluster communicate via Myrinet [20], to which Panda has access via the LFC communication substrate [21].

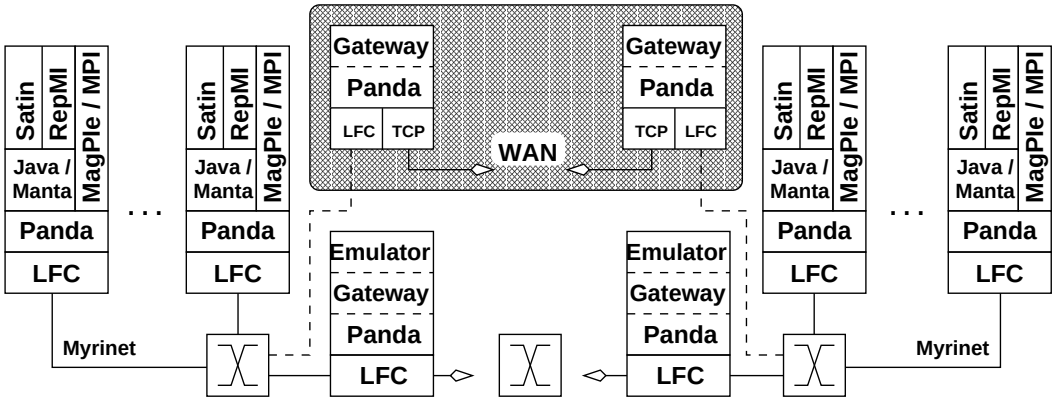

Fig. 2. Local and wide-area communication with Panda and the WAN emulator

Panda also allows to run parallel applications across multiple DAS clusters. For this purpose, one dedicated node in each cluster acts as a *gateway*. Whenever an application node wants to send a message to a node in a different cluster, it sends the message to its local gateway node, which in turn forwards it to the gateway node of the remote cluster, where the message gets forwarded to the receiver node. Between cluster gateways, Panda communicates using the

standard TCP protocol. This communication path is shown in Fig. 2, using the upper, shaded path between the two clusters (on the left and on the right sides). The Panda gateway nodes run binaries of the actual application program. During program startup, a Panda gateway enters the code for message forwarding rather than the application's main() function.

A major challenge in investigating the performance of parallel Grid applications is the actual WAN behavior. Typical wide-area (Internet) links are shared among many applications, making runtime measurements irreproducible and thus scientifically hardly valuable. To overcome this problem, we developed a WAN emulator for Panda. The WAN emulator allows us to run parallel applications on a single (large) cluster with only the wide-area links being emulated. For this purpose, Panda provides an emulator version of its gateway functionality. Here, communication between gateway nodes physically occurs inside a single cluster, in our case using Myrinet. This communication path is shown in Fig. 2, using the lower path between the two clusters.

The actual emulation of WAN behavior occurs in the receiving cluster gateways which delay incoming messages before forwarding them to the respective receivers. On arrival of a message from a remote cluster, the gateway computes the emulated arrival time, taking into account the emulated latency and bandwidth from sending to receiving cluster, and the message length. The message is then put into a queue and gets delivered as soon as the delay expires. With this setup, the WAN emulation is completely transparent to the application processes, allowing realistic and simultaneously reproducible wide-area experimentation.

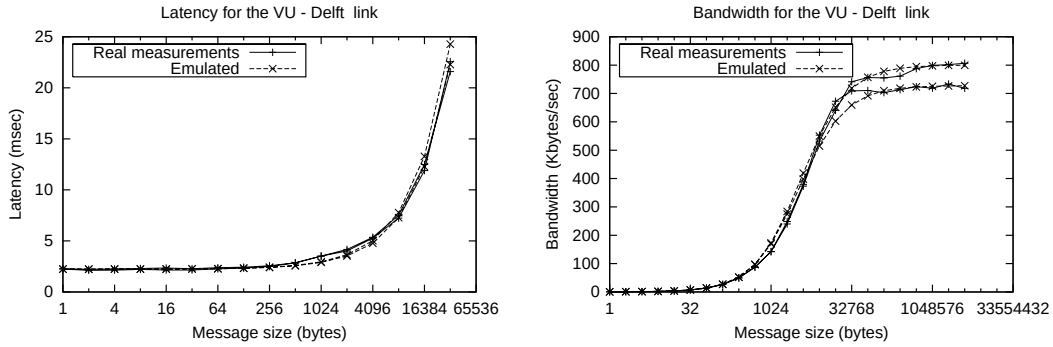

Fig. 3. Measured vs. emulated latency and bandwidth between 2 DAS clusters (in both directions)

We also investigated the precision of our emulator. Therefore, we measured bandwidth and latency between the DAS clusters using ping-pong tests with messages of varying sizes. We then fed the measured parameters into the emulator and re-ran our tests. Fig. 3 compares real and emulated latency and bandwidth between the DAS clusters at VU (Amsterdam) and Delft University of Technology (in both directions). In the graphs, the respective pairs of lines

are hardly distinguishable, giving evidence for the close match between the real system and its emulation. The measurements for the other wide-area DAS links show similar behavior.

Whenever a message arrives at a gateway node, its delay time is computed. For making the emulation dynamically configurable, the delay computation is encapsulated in an upcall routine, that is called upon message arrival. The gateways can be configured at any time of an application run by activating one of the following kinds of upcall routine. Furthermore, our emulator allows additional, user-defined upcalls to be used.

The *constant* upcall emulates a WAN in which each link has constant latency and bandwidth. However, each sender-receiver pair of gateways may have different link speeds.

The *script* upcall is a variation of the constant upcall. Here, the gateways interpret a configuration script in order to change the setting of the link parameters throughout the application run. Fig. 4 shows a sample script used for the case study presented in Section 4.

The *TCP* upcall is another variation of the constant upcall. Here, the gateways accept commands from a remote process on a given TCP port. We have developed a Java-based GUI process that allows a human user to dynamically change the emulated links while an application is running.

The *measure* upcall lets each gateway read latency and bandwidth values from prerecorded files containing time series, e.g. from measurements of real wide-area links. One of the scenarios in Section 4 uses the measure upcall to emulate the behavior of the real DAS system, as measured by the Network Weather Service (NWS) [22].

The emulation upcalls can be activated in two different ways. First, Panda can interpret command line options to select and parameterize an upcall. This way, the emulation is completely transparent to the application program. More flexible, although not transparent to the application, is Panda's emulation API that allows a running application program to directly influence the gateway behavior. The emulation API allows, for example, the activation of a user-defined upcall or the controlled experimentation from inside the application itself.

4 A Case Study: Evaluation of Satin using various WAN Scenarios

We will now present a case study in which we evaluate Satin's work stealing algorithms by running four different applications across four emulated clusters. We use the following nine different WAN scenarios of increasing complexity, demonstrating the flexibility of Panda's WAN emulator. Fig. 5 illustrates sce-

narios 1–8 in detail.

(1) The WAN is fully connected. The latency of all links is 100 ms; but the bandwidth differs between the links.
(2) The WAN is fully connected. The bandwidth of all links is 100 KB/s; but the latency differs between the links.
(3) The WAN is fully connected. Both latency and bandwidth differ between the links.
(4) Like scenario 3, but the link between clusters 1 and 4 drops every third second from 100 KB/s and 100 ms to 1 KB/s and 300 ms, emulating being busy due to unrelated, bursty network traffic. Fig. 4 shows the emulator script used for this scenario.
(5) Like scenario 3, but every second all links change bandwidth and latency to random values between 10% and 100% of their nominal bandwidth, and between 1 and 10 times their nominal latency.
(6) All links have 100 ms latency and 100 KB/s bandwidth. Unlike the previous scenarios, two WAN links are missing, causing congestion among the different clusters.
(7) Like scenario 3, but two WAN links are missing.
(8) Like scenario 5, but two WAN links are missing.
(9) Bandwidth and latency are taken from pre-recorded NWS measurements of the real DAS system.

```
read scenario3
sleep 2000
forever
set_sym one 1 4 1000 0.3
sleep 1000
set_sym one 1 4 1000000 0.001
sleep 2000
```

Fig. 4. The emulator script for scenario 4

We used the following Satin applications, taken from the set presented in [15].

Adaptive Integration numerically integrates a function over a given interval by recursive interval division. This application is mostly sensitive to latency because the job descriptions and results can be sent in very short messages.

N Queens solves the problem of placing n queens on a $n \times n$ chess board. This application sends medium-size messages and has a very irregular task tree.

Ray Tracer is a simple ray tracing program. It divides a screen down to jobs of single pixels. The individually calculated pixel colors are composed into larger image segments. This application sends long result mesages, making it sensitive to the available bandwidth.

Traveling Salesperson solves the famous problem of finding the shortest

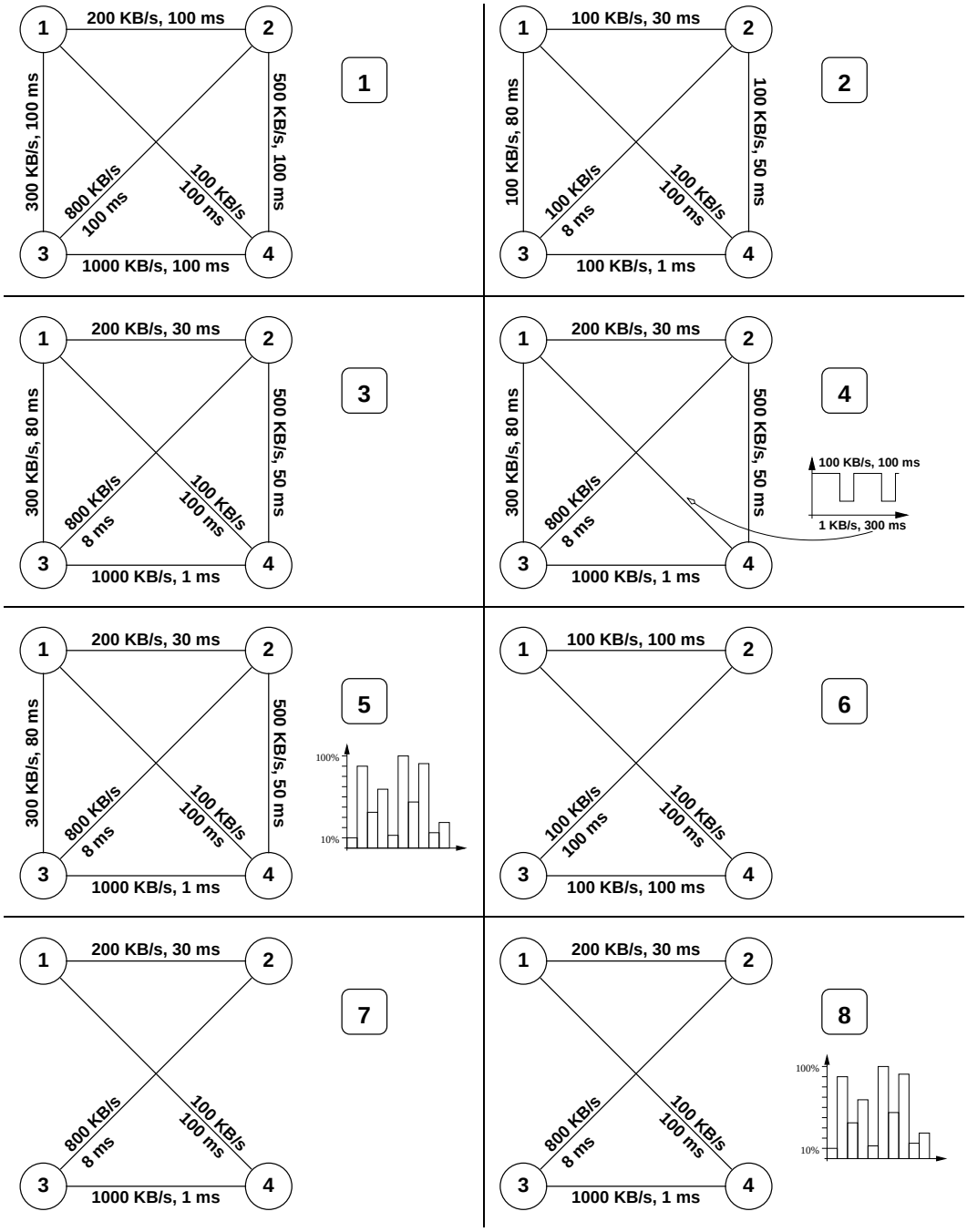

Fig. 5. Emulated WAN scenarios 1–8

path between n cities. By passing the distance table as a parameter, medium-sized messages are exchanged.

Fig. 6 shows the speedups achieved by the four applications on four clusters of 16 nodes each, with the WAN links between them being emulated according to the nine scenarios described above. For comparison, we also show the speedups for a single, large cluster of 64 nodes. The three work stealing algorithms described in Section 2 are compared with each other. RS sends by far the most

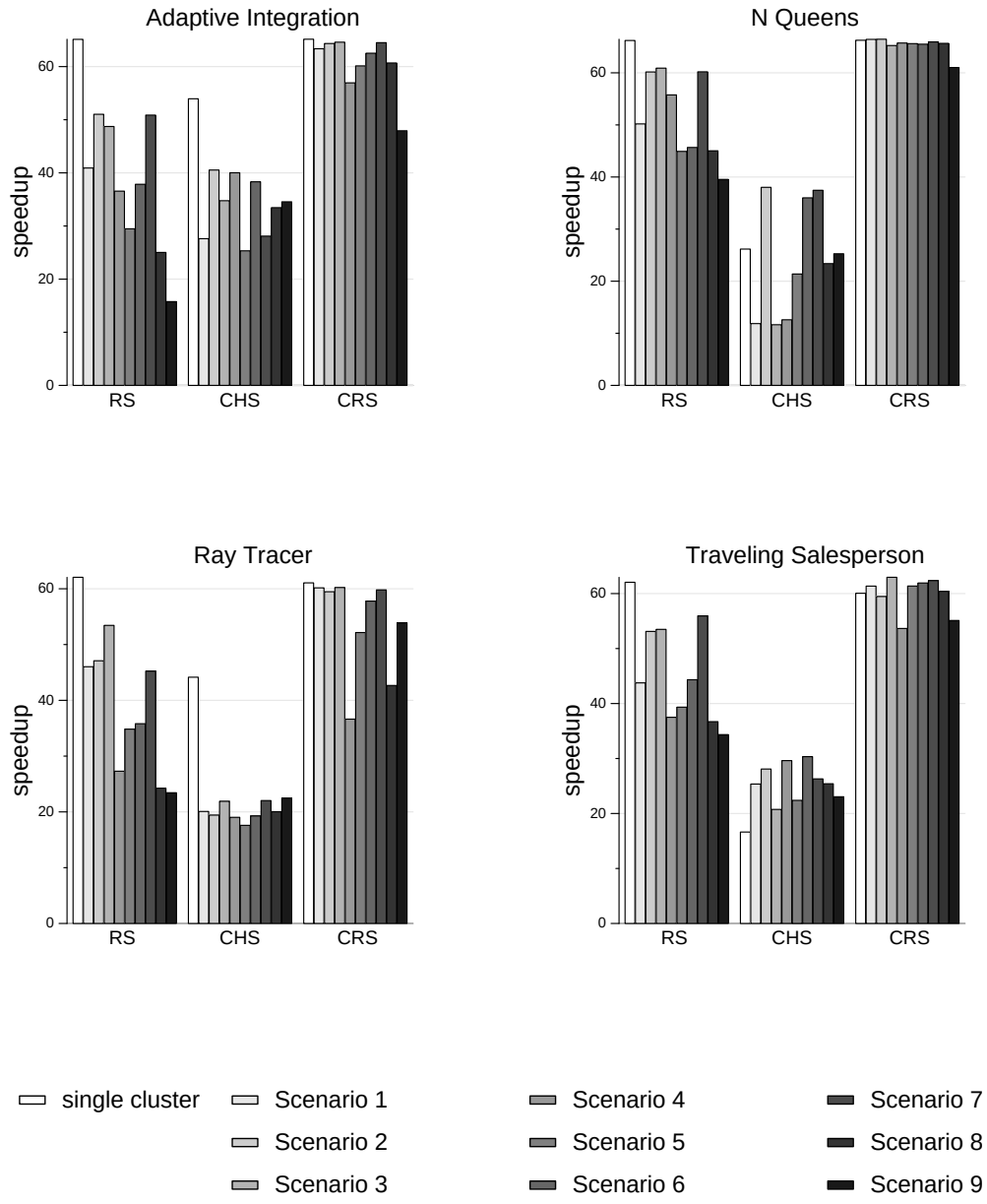

Fig. 6. Speedups of 4 Satin applications with 3 load balancing algorithms and 9 different, emulated WAN scenarios

messages across the WAN links. The speedups it achieves are significantly smaller, compared to a single, large cluster. This is especially the case in scenarios in which high WAN latency causes long idle times or in which low bandwidth causes network congestion. CHS is always the worst-performing algorithm, even within a single cluster, due to complete clusters being idle during a work-stealing message roundtrip time.

CRS always is the best performing algorithm. Due to its limited and asynchronous wide-area communication, it can tolerate even very irregular WAN scenarios, resulting in speedups close to a single, large cluster. However, there are a few exceptions to the very high speedups achieved by CRS which occur whenever the WAN bandwidth becomes too low for the application's requirements. This happens with scenarios 4, 8, and 9. But even in those cases, CRS still is Satin's best performing work-stealing algorithm.

5 Related Work

Many Grid computing projects focus on building software infrastructures that enable application execution in Grid evironments [23–25]. Our Panda library can provide the communication-related runtime system for parallel applications that are running on top of such Grid infrastructures.

Network simulators like NSE [26] or DaSSF [27] focus on packet delivery and network protocols, rather than the network behavior as it is observed by an application. LAPSE [28] simulates parallel applications on configurations with more than the available number of CPUs; the network behavior simulates the Intel Paragon machines. The MicroGrid software [29] virtualizes the Grid resources like memory, CPU, and networks. For the simulation, all relevant system calls are trapped and mediated through the MicroGrid scheduler and the NSE network simulator. This approach goes further than Panda's network emulation, but also impacts the sequential execution of the application binaries. Panda's wide-area emulator, however, allows to run unmodified binaries of a parallel application, connecting them via physical LANs and emulated WANs. This network emulation provides a unique environment for experimentation with parallel applications on Grid platforms, which has led to the development of our Grid programming environments.

Some projects provide Grid-enabled implementations of the message passing interface, MPI. MPI_Connect [30] focuses on interoperability between heterogeneous platforms. The works in [31,32] provide Grid-optimized implementations for some of MPI's collective operations. Our MagPIe library, however, provides the most complete and advanced set of collective operations for Grid platforms.

Systems like the SuperWeb [33], Gateway [34], and Bayanihan [35] provide Java-centric environments for Grid computing platforms. However, efficient Grid-aware communication mechanisms for parallel applications are not an issue for these systems. Atlas [16] and Javelin 2.0 [36] are other Java-based divide-and-conquer systems for Grid computing. Their main focus is on heterogeneity and fault tolerance. Satin's main objective is application speed.

Both MPJ [37] and CCJ [38] provide MPI-style message passing and collective communication for Java. Although their set of communication mechanisms is richer than RMI and RepMI, they do not come with implementations that are optimized for wide-area Grid platforms. Finally, Hyperion [39], Jackal [40], and Javanaise [41] implement shared Java objects based on object caching. These systems do not aim at Grid computing either. In contrast, our RepMI mechanism is based on message shipping for which we also provide a Grid-aware implementation.

6 Conclusions

In the Albatross project, we study the efficiency of high-performance applications with medium-grained communication patterns on Grid computing platforms. The key problem is the low communication performance of the wide-area networks (WANs) in a Grid, which typically are orders of magnitude slower than local interconnects. In the initial phase of the project, we developed several strategies for modifying a parallel application to improve its runtime efficiency on a Grid. We use these modification strategies to build programming environments for writing high-performance Grid applications.

In this paper, we have described three such environments, MagPIe, RepMI, and Satin. A major challenge in investigating the performance of Grid applications is the actual WAN behavior. Typical wide-area links are shared among many applications, making runtime measurements irreproducible and thus scientifically hardly valuable. To allow a realistic performance evaluation of Grid programming systems and their applications, we have developed the Panda WAN emulator, a testbed that emulates a Grid on a single, large cluster. The testbed runs the applications in parallel but emulates wide-area links by adding artificial delays. The latency and bandwidth of the WAN links can be specified by the user in a highly flexible way. This network emulation provides a unique environment for experimentation with high-performance applications on Grid platforms.

We have used the Panda WAN emulator to evaluate the performance of one of the three programming environments (Satin) under many different WAN scenarios. The emulator allowed us to compare several load balancing algorithms used by Satin under conditions that are realistic for an actual Grid, but that are hard to reproduce on such a Grid. Our experiments showed that Satin's CHS algorithm can actually tolerate a large variety of WAN link performance settings, and schedule parallel divide-and-conquer applications such that they run almost as fast on multiple clusters as they do on a single, large cluster. In the near future, we will also investigate our other Grid programming environments on a variety of different wide-area network scenarios.

Acknowledgements

This work is supported in part by a USF grant from the Vrije Universiteit. The wide-area DAS system is an initiative of the Advanced School for Computing and Imaging (ASCI). We thank Ceriel Jacobs, Grégory Mounié, John Romein, and Ronald Veldema for their contributions to this paper.

References

[1] RSA Laboratories, Factorization of RSA-155, http://www.rsasecurity.com/rsalabs/challenges/factoring/rsa155.html.

[2] SETI@home, The Search for Extraterrestrical Intelligence, http://setiathome.ssl.berkeley.edu/.

[3] Entropia Inc., Distributed Computing, http://www.entropia.com/.

[4] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, R. Veldema, Wide-Area Parallel Programming using the Remote Method Invocation Model, Concurrency: Practice and Experience 12 (8) (2000) 643–666.

[5] A. Plaat, H. E. Bal, R. F. H. Hofman, T. Kielmann, Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects, Future Generation Computer Systems 13 (8–9) (2001) 769–782.

[6] J. W. Romein, H. E. Bal, Wide-Area Transposition-Driven Scheduling, in: IEEE International Symposium on High Performance Distributed Computing (HPDC-10), San Francisco, CA, 2001.

[7] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, International Journal of Supercomputing Applications 8 (3/4).

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

[9] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, R. F. Hofman, Network Performance-aware Collective Communication for Clustered Wide Area Systems, Parallel Computing 27 (11) (2001) 1431–1456.

[10] J. Maassen, T. Kielmann, H. E. Bal, Parallel Application Experience with Replicated Method Invocation, Concurrency & Computation: Practice & Experience 13 (8–9) (2001) 681–712.

[11] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for Parallel Programming, Accepted for

publication in ACM Transactions on Programming Languages and Systems (TOPLAS) (2001) .

[12] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, in: 35th Annual Symposium on Foundations of Computer Science (FOCS'94), Santa Fe, New Mexico, 1994, pp. 356–368.

[13] I.-C. Wu, H. Kung, Communication Complexity for Parallel Divide-and-Conquer, in: 32nd Annual Symposium on Foundations of Computer Science (FOCS '91, San Juan, Puerto Rico, 1991, pp. 151–162.

[14] R. van Nieuwpoort, T. Kielmann, H. E. Bal, Satin: Efficient Parallel Divide-and-Conquer in Java, in: Euro-PAR 2000, no. 1900 in Lecture Notes in Computer Science, Springer, Munich, Germany, 2000, pp. 690–699.

[15] R. van Nieuwpoort, T. Kielmann, H. E. Bal, Efficient Load Balancing for Wide-area Divide-and-Conquer Applications, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01), Snowbird, Utah, 2001, pp. 34–43.

[16] J. Baldeschwieler, R. Blumofe, E. Brewer, ATLAS: An Infrastructure for Global Computing, in: Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, 1996.

[17] M. Backschat, A. Pfaffinger, C. Zenger, Economic Based Dynamic Load Distribution in Large Workstation Networks, in: Euro-Par'96, no. 1124 in Lecture Notes in Computer Science, Springer, 1996, pp. 631–634.

[18] T. Rühl, H. E. Bal, G. Benson, R. A. F. Bhoedjang, K. Langendoen, Experience with a Portability Layer for Implementing Parallel Programming Systems, in: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), Sunnyvale, CA, 1996, pp. 1477–1488.

[19] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A High-performance, Portable Implementation of the MPI Message Passing Interface Standard, Parallel Computing 22 (6) (1996) 789–828.

[20] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, W. Su, Myrinet: A Gigabit-per-second Local Area Network, IEEE Micro 15 (1) (1995) 29–36.

[21] R. A. F. Bhoedjang, T. Rühl, H. E. Bal, User-Level Network Interface Protocols, IEEE Computer 31 (11) (1998) 53–60.

[22] R. Wolski, Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service, in: Proc. High-Performance Distributed Computing (HPDC-6), Portland, OR, 1997, pp. 316–325, the network weather service is at http://nws.npaci.edu/.

[23] I. Foster, C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, International Journal of Supercomputer Applications 11 (2) (1997) 115–128.

[24] A. Grimshaw, W. A. Wulf, The Legion Vision of a Worldwide Virtual Computer, Communications of the ACM 40 (1) (1997) 39–45.

[25] G. E. Fagg, K. Moore, J. J. Dongarra, A. Geist, Scalable Network Information Processing Environment (SNIPE), in: SC'97, 1997, http://www.supercomp.org/sc97/proceedings/.

[26] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldara, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, D. Zappala, Improving Simulation for Network Research, Technical Report 99–702, University of Southern California (1999).

[27] J. Liu, D. M. Nicol, DaSSF 3.1 User's Manual, http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/ (2001).

[28] P. M. Dickens, P. Heidelberger, D. M. Nicol, A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs, in: Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94), 1994.

[29] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, A. Chien, The MicroGrid: a Scientific Tool for Modeling Computational Grids, in: Supercomputing 2000, Dallas, TX, 2000.

[30] G. E. Fagg, K. S. London, J. J. Dongarra, MPI_Connect: Managing Heterogeneous MPI Applications Interoperation and Process Control, in: Proc. 5th European PVM/MPI Users' Group Meeting, no. 1497 in LNCS, Liverpool, UK, 1998, pp. 93–96.

[31] E. Gabriel, M. Resch, T. Beisel, R. Keller, Distributed Computing in a Heterogeneous Computing Environment, in: Proc. 5th European PVM/MPI Users' Group Meeting, no. 1497 in LNCS, Liverpool, UK, 1998, pp. 180–187.

[32] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, J. Bresnahan, Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance, in: Proc. International Parallel and Distributed Processing Symposium (IPDPS 2000), IEEE, Cancun, Mexico, 2000, pp. 377–384.

[33] A. D. Alexandrov, M. Ibel, K. E. Schauser, C. J. Scheiman, SuperWeb: Research Issues in Java-Based Global Computing, Concurrency: Practice and Experience 9 (6) (1997) 535–553.

[34] T. Haupt, E. Akarsu, G. Fox, A. Kalinichenko, K.-S. Kim, P. Sheethalnath, C.-H. Youn, The Gateway System: Uniform Web Based Access to Remote Resources, in: ACM 1999 Java Grande Conference, San Francisco, CA, 1999, pp. 1–7.

[35] L. F. G. Sarmenta, S. Hirano, Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java, Future Generation Computer Systems 15 (5/6).

[36] M. O. Neary, A. Phipps, S. Richman, P. Cappello, Javelin 2.0: Java-based parallel computing on the internet, in: Proc. Euro-Par 2000, Munich, Germany, 2000, pp. 1231–1238.

[37] B. Carpenter, V. Getov, G. Judd, A. Skjellum, G. Fox, MPJ: MPI-like Message Passing for Java, Concurrency: Practice and Experience 12 (11) (2000) 1019–1038.

[38] A. Nelisse, T. Kielmann, H. E. Bal, J. Maassen, Object-based Collective Communication in Java, in: Joint ACM JavaGrande-ISCOPE 2001 Conference, Stanford University, 2001, pp. 11–20.

[39] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, R. Namyst, The Hyperion system: Compiling multithreaded Java bytecode for distributed execution, Parallel Computing .

[40] R. Veldema, R. Hofman, C. Jacobs, R. Bhoedjang, H. Bal, Source-Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01), Snowbird, Utah, 2001, pp. 83–92.

[41] D. Hagimont, D. Louvegnies, Javanaise: Distributed Shared Objects for Internet Cooperative Applications, in: Proc. Middleware'98, The Lake District, England, 1998.