SATIN: SIMPLE AND EFFICIENT JAVA-BASED GRID PROGRAMMING

ROB V. VAN NIEUWPOORT, JASON MAASSEN, THILO KIELMANN, HENRI E. BAL
DEPT. OF COMPUTER SCIENCE, VRIJE UNIVERSITEIT, AMSTERDAM, THE NETHERLANDS

{ROB,JASON,KIELMANN,BAL}@CS.VU.NL HTTP://WWW.CS.VU.NL/IBIS

Abstract. Grid programming environments need to be both *portable* and *efficient* to exploit the computational power of dynamically available resources. In previous work, we have presented the divide-and-conquer based Satin model for parallel computing on clustered wide-area systems. In this paper, we present the Satin implementation on top of our new Ibis platform which combines Java's *write once, run everywhere* with efficient communication between JVMs. We evaluate Satin/Ibis on the testbed of the EU-funded GridLab project, showing that Satin's load-balancing algorithm automatically adapts both to heterogeneous processor speeds and varying network performance, resulting in efficient utilization of the computing resources. Our results show that when the wide-area links suffer from congestion, Satin's load-balancing algorithm can still achieve around 80% efficiency, while an algorithm that is not grid aware drops to 26% or less.

Key words. Satin, Ibis, divide-and-conquer, load balancing, distributed supercomputing.

1. Introduction. In computational grids, applications need to simultaneously tap the computational power of multiple, dynamically available sites. The crux of designing grid programming environments stems exactly from the dynamic availability of compute cycles: grid programming environments need to be both *portable* to run on as many sites as possible, and they need to be *flexible* to cope with different network protocols and dynamically changing groups of heterogeneous compute nodes.

Existing programming environments are either portable and flexible (Jini, Java RMI), or they are highly efficient (MPI). The Global Grid Forum also has investigated possible grid programming models [19]. Recently, *GridRPC* has been proposed as a grid programming model [30]. GridRPC allows writing grid applications based on the manager/worker paradigm.

Unlike manager/worker programs, divide-and-conquer algorithms operate by recursively dividing a problem into smaller subproblems. This recursive subdivision goes on until the remaining subproblem becomes trivial to solve. After solving subproblems, their results are recursively recombined until the final solution is assembled. By allowing subproblems to be divided recursively, the class of divide-and-conquer algorithms subsumes the manager/worker algorithms, thus enlarging the set of possible grid applications.

Of course, there are many kinds of applications that do not lend themselves well to a divide-and-conquer algorithm. However, we (and others) believe the class of divide-and-conquer algorithms to be sufficiently large to justify its deployment for hierarchical wide-area systems. Computations that use the divide-and-conquer model include geometry procedures, sorting methods, search algorithms, data classification codes, n-body simulations and data-parallel numerical programs [33].

Divide-and-conquer applications may be parallelized by letting different processors solve different subproblems. These subproblems are often called *jobs* in this context. Generated jobs are transferred between processors to balance the load in the computation. The divide-and-conquer model lends itself well to hierarchically-structured systems because tasks are created by recursive subdivision. This leads to a task graph that is hierarchically structured, and which can be executed with excellent communication locality, especially on hierarchical platforms.

In previous work [26], we presented our *Satin* system for divide-and-conquer programming on grid platforms. Satin implements a very efficient load balancing algorithm for clustered, wide-area platforms. So far, we could only evaluate Satin based on simulations in which all jobs have been executed on one single, homogeneous cluster. In this work, we evaluate Satin on a real grid testbed [2], consisting of various heterogeneous systems, connected by the Internet.

In Section 2, we briefly present Satin's programming model and some simulator-based results that indicate the suitability of Satin as a grid programming environment. In Section 3, we present Ibis, our new Java-based grid programming platform that combines Java's "run everywhere" paradigm with highly efficient yet flexible communication mechanisms. In Section 4, we evaluate the performance of Satin on top of Ibis in the GridLab testbed, spanning several sites in Europe. Section 5 discusses related work, and in Section 6 we draw conclusions.

2. Divide-and Conquer in 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 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. In this paper, however, we focus on pure divide-and-conquer programs.

```
1  interface FibInter extends satin.Spawnable {
2      public long fib(long n);
3  }
4
5  class Fib extends satin.SatinObject
6      implements FibInter {
7      public long fib(long n) {
8          if(n < 2) return n;
9
10         long x = fib(n-1); // spawned
11         long y = fib(n-2); // spawned
12         sync();
13
14         return x + y;
15     }
16
17     public static void main(String[] args) {
18         Fib f = new Fib();
19         long res = f.fib(10);
20         f.sync();
21         System.out.println("Fib 10 = " + res);
22     }
23 }
```

FIG. 2.1. Fib*: an example divide-and-conquer program in Satin.*

Satin expresses divide-and-conquer parallelism entirely in the Java language itself, without requiring any new language constructs. Satin uses so-called *marker interfaces* to indicate that certain method invocations need to be considered for potentially parallel (so called spawned) execution, rather than being executed synchronously like normal methods. Furthermore, a mechanism is needed to synchronize with (wait for the results of) spawned method invocations. With Satin, this can be expressed using a special interface, satin.Spawnable, and the class satin.SatinObject. This is shown in Fig. 2.1, using the example of a class Fib for computing the Fibonacci numbers. First, an interface FibInter is implemented which extends satin.Spawnable. All methods defined in this interface (here fib) are marked to be spawned rather than executed normally. Second, the class Fib extends satin.SatinObject and implements FibInter. From satin.SatinObject it inherits the sync method, from FibInter the spawned fib method. Finally, the invoking method (in this case main) simply calls Fib and uses sync to wait for the result of the parallel computation.

Satin's byte code rewriter generates the necessary code. Conceptually, a new thread is started for running a spawned 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 spawned method. The sync method 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.

Spawned method invocations are distributed across the processors of a parallel Satin program by work stealing from the work queues mentioned above. In [26], we presented a new work stealing algorithm, *Cluster-aware Random Stealing* (CRS), specifically designed for cluster-based, wide-area (grid computing) systems. CRS is based on the traditional Random Stealing (RS) algorithm that has been proven to be optimal for homogeneous (single cluster) systems [8]. We briefly describe both algorithms in turn.

2.1. 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 [8, 33]. 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.

2.2. 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 cluster, 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. Below, we will show that CRS performs almost as good as with a single, large cluster, even in extreme wide-area network settings.

2.3. Simulator-based comparison of RS and CRS. A detailed description of Satin's wide-area work stealing algorithm can be found in [26]. We have extracted the comparison of RS and CRS from that work into Table 2.1. The run times shown in this table are for parallel runs with 64 CPUs each, either with a single cluster of 64 CPUS, or with 4 clusters of 16 CPUs each.

The wide-area network between the virtual clusters has been simulated with our Panda WAN simulator [17]. We simulated all combinations of 20 ms and 200 ms roundtrip latency with bandwidth capacities of 100 KByte/s and 1000 KByte/s. The tests had been performed on the predecessor hardware to our current DAS-2 cluster. DAS consists of 200 MHz Pentium Pro's with a Myrinet network, running the Manta parallel Java system [23].

	single cluster		20 ms 1000 KByte/s		20 ms 100 KByte/s		200 ms 1000 KByte/s		200 ms 100 KByte/s	
application	time	eff.	time	eff.	time	eff.	time	eff.	time	eff.
adaptive integration										
RS	71.8	99.6%	78.0	91.8%	79.5	90.1%	109.3	65.5%	112.3	63.7%
CRS	71.8	99.7%	71.6	99.9%	71.7	99.8%	73.4	97.5%	73.2	97.7%
N-queens										
RS	157.6	92.5%	160.9	90.6%	168.2	86.6%	184.3	79.1%	197.4	73.8%
CRS	156.3	93.2%	158.1	92.2%	156.1	93.3%	158.4	92.0%	158.1	92.2%
TSP										
RS	101.6	90.4%	105.3	87.2%	105.4	87.1%	130.6	70.3%	129.7	70.8%
CRS	100.7	91.2%	103.6	88.7%	101.1	90.8%	105.0	87.5%	107.5	85.4%
ray tracer										
RS	147.8	94.2%	152.1	91.5%	171.6	81.1%	175.8	79.2%	182.6	76.2%
CRS	147.2	94.5%	145.0	95.9%	152.6	91.2%	146.5	95.0%	149.3	93.2%

TABLE 2.1

Performance of RS and CRS with different simulated wide-area links (times in seconds).

In Table 2.1 we compare RS and CRS using four parallel applications, with network conditions degrading from the left (single cluster) to the right (high latency, low bandwidth). For each case, we present the parallel run time and the corresponding efficiency (labeled "eff." in the table). With t_s being the sequential run time for the application, with the Satin operations excluded, (not shown) and t_p the parallel run time as shown in the table, and $N = 64$ being the number of CPUs, we compute the efficiency as follows:

$$efficiency = \frac{t_s}{t_p \cdot N} * 100\%$$

Adaptive integration numerically integrates a function over a given interval. It sends very short messages and has also very fine grained jobs. This combination makes RS sensitive to high latency,

in which case efficiency drops to about 65 %. CRS, however, successfully hides the high round trip times and achieves efficiencies of more than 97 % in all cases.

N Queens solves the problem of placing n queens on a $n \times n$ chess board. It sends medium-size messages and has a very irregular task tree. With efficiency of only 74 %, RS again suffers from high round trip times as it can not quickly compensate load imbalance due to the irregular task tree. CRS, however, sustains efficiencies of 92 %.

TSP solves the problem of finding the shortest path between n cities. By passing the distance table as parameter, is has a somewhat higher parallelization overhead, resulting in slightly lower efficiencies, even with a single cluster. In the wide-area cases, these longer parameter messages contribute to higher round trip times when stealing jobs from remote clusters. Consequently, RS suffers more from slower networks (efficiency > 70 %) than CRS which sustains efficiencies of 85 %.

Ray Tracer renders a modeled scene to a raster image. It divides a screen down to jobs of single pixels. Due to the nature of ray tracing, individual pixels have very irregular rendering times. The application sends long result messages containing image fractions, making it sensitive to the available bandwidth. This sensitivity is reflected in the efficiency of RS, going down to 76 %, whereas CRS hides most WAN communication overhead and sustains efficiencies of 91 %.

To summarize, our simulator-based experiments show the superiority of CRS to RS in case of multiple clusters, connected by wide-area networks. This superiority is independent of the properties of the applications, as we have shown with both regular and irregular task graphs as well as short and long parameter and result message sizes. In all investigated cases, the efficiency of CRS never dropped below 85 %.

Although we were able to identify the individual effects of wide-area latency and bandwidth, these results are limited to homogeneous Intel/Linux clusters (due to the Manta compiler). Furthermore, we only tested clusters of identical size. Finally, the wide area network has been simulated and thus been without possibly disturbing third-party traffic.

An evaluation on a real grid testbed, with heterogeneous CPUs, JVMs, and networks, becomes necessary to prove the suitability of Satin as a grid programming platform. In the following, we first present Ibis, our new *run everywhere* Java environment for grid computing. Then we evaluate Satin on top of Ibis on the testbed of the EU GridLab project.

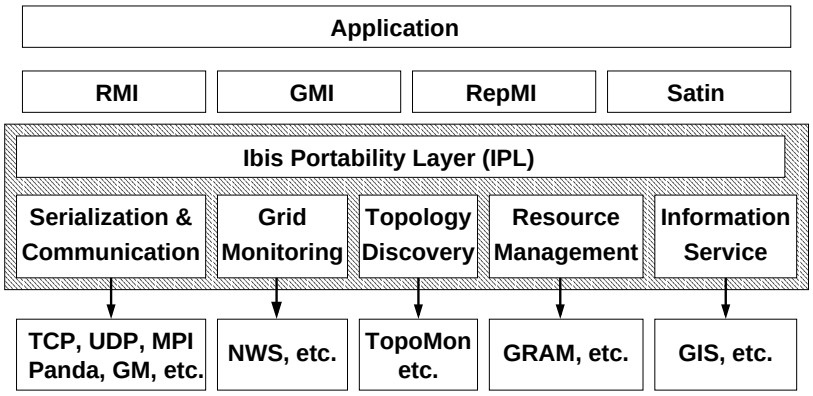

FIG. 3.1. *Design of Ibis. The various modules can be loaded dynamically, using run time class loading.*

3. Ibis, flexible and efficient Java-based Grid programming. The Satin runtime system used for this paper is implemented on top of Ibis [31]. In this section we will briefly explain the Ibis philosophy and design. The global structure of the Ibis system is shown in Figure 3.1. A central part of the system is the Ibis Portability Layer (IPL) which consists of a number of well-defined interfaces. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. The IPL defines serialization and communication, but also typical grid services such as topology discovery and monitoring. Although it is possible to use the IPL directly from an application, Ibis also provides more high-level programming models. Currently, we have implemented four. Ibis RMI [31] provides Remote Method Invocation, using the same interface as

Sun RMI, but with a more efficient wire protocol. GMI [21] provides MPI-like collective operations, cleanly integrated into Java's object model. RepMI [22] extends Java with replicated objects. In this paper, we focus on the fourth programming model that Ibis implements, Satin.

3.1. Ibis Goals. A key problem in making Java suitable for grid programming is how to design a system that obtains high communication performance while still adhering to Java's "write once, run everywhere" model. Current Java implementations are heavily biased to either portability or performance, and fail in the other aspect. (The recently added *java.nio* package will hopefully at leas partially address this problem). The Ibis strategy to achieve both goals simultaneously is to develop reasonably efficient solutions using standard techniques that work "everywhere", supplemented with highly optimized but non-standard solutions for increased performance in special cases. We apply this strategy to both computation and communication. Ibis is designed to use any standard JVM, but if a native, optimizing compiler (e.g., Manta [23]) is available for a target machine, Ibis can use it instead. Likewise, Ibis can use standard communication protocols, e.g., TCP/IP or UDP, as provided by the JVM, but it can also plug in an optimized low-level protocol for a high-speed interconnect, like GM or MPI, if available. The challenges for Ibis are:

1. how to make the system flexible enough to run *seamlessly* on a variety of different communication hardware and protocols;
2. how to make the standard, 100% pure Java case efficient enough to be useful for grid computing;
3. study which additional optimizations can be done to improve performance further in special (high-performance) cases.

With Ibis, grid applications can run simultaneously on a variety of different machines, using optimized software where possible (e.g., a native compiler, the GM Myrinet protocol, or MPI), and using standard software (e.g., TCP) when necessary. Interoperability is achieved by using the TCP protocol between multiple Ibis implementations that use different protocols (like GM or MPI) locally. This way, all machines can be used in one single computation. Below, we discuss the three aforementioned issues in more detail.

3.2. Flexibility. The key characteristic of Ibis is its extreme flexibility, which is required to support grid applications. A major design goal is the ability to seamlessly plug in different communication substrates without changing the user code. For this purpose, the Ibis design uses the IPL. A software layer on top of the IPL can negotiate with Ibis instantiations through the well-defined IPL interface, to select and load the modules it needs. This flexibility is implemented using Java's dynamic class-loading mechanism.

Many message passing libraries such as MPI and GM guarantee reliable message delivery and FIFO message ordering. When applications do not require these properties, a different message passing library might be used to avoid the overhead that comes with reliability and message ordering. The IPL supports both reliable and unreliable communication, ordered and unordered messages, implicit and explicit receipt, using a single, simple interface. Using user-definable properties (key-value pairs), applications can create exactly the communication channels they need, without unnecessary overhead.

3.3. Optimizing the Common Case. To obtain acceptable communication performance, Ibis implements several optimizations. Most importantly, the overhead of serialization and reflection is avoided by compile-time generation of special methods (in byte code) for each object type. These methods can be used to convert objects to bytes (and vice versa), and to create new objects on the receiving side, without using expensive reflection mechanisms. This way, the overhead of serialization is reduced dramatically.

Furthermore, our communication implementations use an optimized wire protocol. The Sun RMI protocol, for example, resends type information for each RMI. Our implementation caches this type information per connection. Using this optimization, our protocol sends less data over the wire, but more importantly, saves processing time for encoding and decoding the type information.

3.4. Optimizing Special Cases. In many cases, the target machine may have additional facilities that allow faster computation or communication, which are difficult to achieve with standard

Java techniques. One example we investigated in previous work [23] is using a native, optimizing compiler instead of a JVM. This compiler (Manta), or any other high performance Java implementation, can simply be used by Ibis. The most important special case for communication is the presence of a high-speed local interconnect. Usually, specialized user-level network software is required for such interconnects, instead of standard protocols (TCP, UDP) that use the OS kernel. Ibis therefore was designed to allow other protocols to be plugged in. So, lower-level communication may be based, for example, on a locally-optimized MPI library. The IPL is designed in such a way that it is possible to exploit efficient hardware multicast, when available.

Another important feature of the IPL is that it allows a zero-copy implementation. Implementing zero-copy (or single-copy) communication in Java is a non-trivial task, but it is essential to make Java competitive with systems like MPI for which zero-copy implementations already exist. The zero-copy Ibis implementation is described in more detail in [31]. On fast networks like Myrinet, the throughput of Ibis RMI can be as much as 9 times higher than previous, already optimized RMI implementations such as KaRMI [28].

4. Satin on the GridLab testbed. In this section, we will present a case study to analyze the performance that Satin/Ibis achieves in a *real* grid environment. We ran the ray tracer application introduced in Section 2.3 on the European GridLab [2] testbed. More precisely, we were using a characteristic subset of the machines on this testbed that was available for our measurements at the time the study was performed. Because simultaneously starting and running a parallel application on multiple clusters still is a tedious and time-consuming task, we had to restrict ourselves to a single test application. We have chosen the ray tracer for our tests as it is sending the most data of all our applications, making it very sensitive to network issues. The ray tracer is written in pure Java and generates a high resolution image (4096×4096, with 24-bit color). It takes approximately 10 minutes to solve this problem on our testbed.

This is an interesting experiment for several reasons. Firstly, we use the Ibis implementation on top of TCP for the measurements in this section. This means that the numbers shown below were measured using a 100% Java implementation. Therefore, they are interesting, giving a clear indication of the performance level that can be achieved in Java with a "run everywhere" implementation, without using any native code.

Secondly, the testbed contains machines with several different architectures; Intel, SPARC, MIPS, and Alpha processors are used. Some machines are 32 bit, while others are 64 bit. Also, different operating systems and JVMs are in use. Therefore, this experiment is a good method to investigate whether Java's "write once, run everywhere" feature really works in practice. The assumption that this feature successfully hides the complexity of the different underlying architectures and operating systems, was the most important reason for investigating the Java-centric solutions presented in this paper. It is thus important to verify the validity of this claim.

Thirdly, the machines are connected by the Internet. The links show typical wide-area behavior, as the physical distance between the sites is large. For instance, the distance from Amsterdam to Lecce is roughly 2000 kilometers (about 1250 miles). Figure 4.1 shows a map of Europe, annotated with the machine locations. This gives an idea of the distances between the sites. We use this experiment to verify Satin's load-balancing algorithms in practice, with *real* non-dedicated wide-area links. We have run the ray tracer both with the standard random stealing algorithm (RS) and with the new cluster-aware algorithm (CRS) as introduced above. For practical reasons, we had to use relatively small clusters for the measurements in this section. The simulation results in Section 2.3 show that the performance of CRS increases when larger clusters are used, because there is more opportunity to balance the load inside a cluster during wide-area communication.

Some information about the machines we used is shown in Table 4.1. To run the application, we used whichever Java JIT (Just-In-Time compiler) that was pre-installed on each particular system whenever possible, because this is what most users would probably do in practice.

Because the sites are connected via the Internet, we have no influence on the amount of traffic that flows over the links. To reduce the influence of Internet traffic on the measurements, we also performed measurements after midnight (CET). However, in practice there still is some variability in the link speeds. We measured the latency of the wide-area links by running *ping* 50 times, while

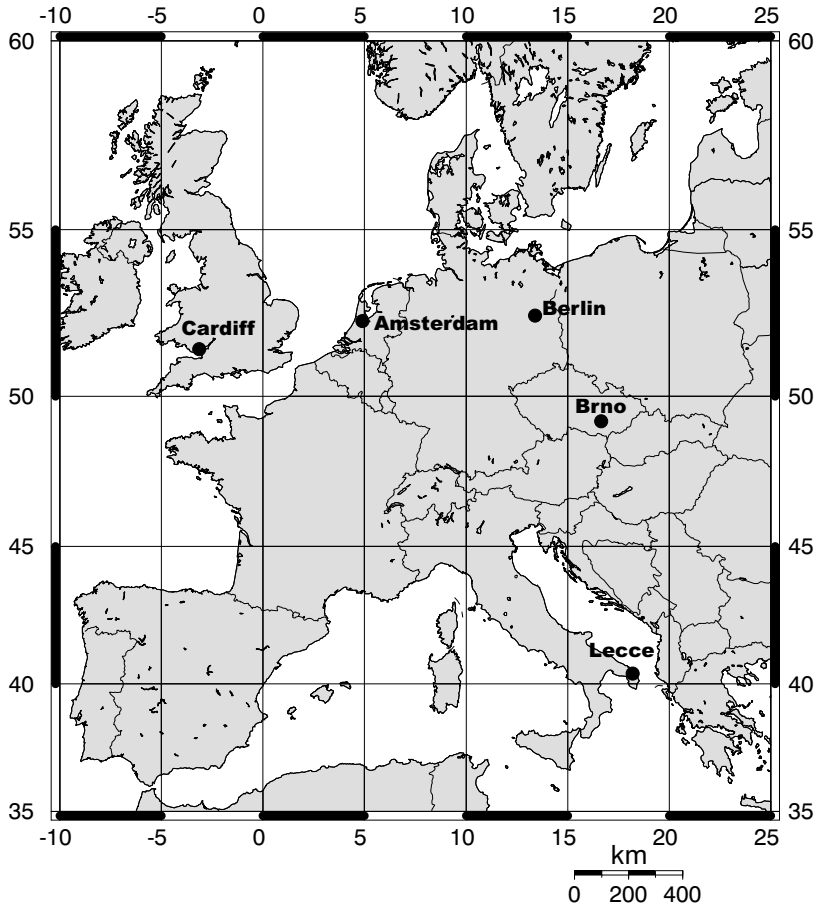

FIG. 4.1. *Locations of the GridLab testbed sites used for the experiments.*

the achievable bandwidth is measured with *netperf* [25], using 32 KByte packets. The measured latencies and bandwidths are shown in Table 4.2. All sites had difficulties from time to time while sending traffic to Lecce, Italy. For instance, from Amsterdam to Lecce, we measured latencies from 44 milliseconds up to 3.5 seconds. Also, we experienced packet loss with this link: up to 23% of the packets were dropped along the way. We also performed the same measurement during daytime, to investigate how regular Internet traffic influences the application performance. The measurements show that there can be more than a factor of two difference in link speeds during daytime and nighttime, especially the links from and to Lecce show a large variability. It is also interesting to see that the link performance from Lecce to the two sites in Amsterdam is different. We verified this with *traceroute*, and found that the traffic is indeed routed differently as the two machines use different network numbers despite being located within the same building.

Ibis, Satin and the ray tracer application were all compiled with the standard Java compiler *javac* on the DAS-2 machine in Amsterdam, and then just copied to the other GridLab sites, without recompiling or reconfiguring anything. On most sites, this works flawlessly. However, we did run into several practical problems. A summary is given in Table 4.3. Some of the GridLab sites have firewalls installed, which block Satin's traffic when no special measures are taken. Most sites in our testbed have some open port range, which means that traffic to ports within this range can pass through. The solution we use to avoid being blocked by firewalls is straightforward: all sockets used for communication in Ibis are bound to a port within the (site-specific) open port range. We are working on a more general solution that multiplexes all traffic over a single port. Another solution is to multiplex all traffic over a (Globus) ssh connection, as is done by Kaneda et al. [16], or using a mechanism like SOCKS [20].

location	architecture	Operating System	JIT	nodes	CPUs / node	total CPUs
Vrije Universiteit *Amsterdam The Netherlands*	Intel Pentium-III 1 GHz	Red Hat Linux kernel 2.4.18	IBM 1.4.0	8	1	8
Vrije Universiteit *Amsterdam The Netherlands*	Sun Fire 280R UltraSPARC-III 750 MHz 64 bit	Sun Solaris 8	SUN HotSpot 1.4.2	1	2	2
ISUFI/High Perf. Computing Center *Lecce, Italy*	Compaq Alpha 667 MHz 64 bit	Compaq Tru64 UNIX V5.1A	HP 1.4.0 based on HotSpot	1	4	4
Cardiff University *Cardiff, Wales, UK*	Intel Pentium-III 1 GHz	Red Hat Linux 7.1 kernel 2.4.2	SUN HotSpot 1.4.1	1	2	2
Masaryk University, *Brno, Czech Republic*	Intel Xeon 2.4 GHz	Debian Linux kernel 2.4.20	IBM 1.4.0	4	2	8
Konrad-Zuse-Zentrum für Informationstechnik *Berlin, Germany*	SGI Origin 3000 MIPS R14000 500 MHz	IRIX 6.5	SGI 1.4.1-EA based on HotSpot	1	16	16

TABLE 4.1
Machines on the GridLab testbed.

	daytime						nighttime					
source	to A'dam DAS-2	to A'dam Sun	to Lecce	to Cardiff	to Brno	to Berlin	to A'dam DAS-2	to A'dam Sun	to Lecce	to Cardiff	to Brno	to Berlin
latency from												
A'dam DAS-2	—	1	204	16	20	42	—	1	65	15	20	18
A'dam Sun	1	—	204	15	19	43	1	—	62	14	19	17
Lecce	198	195	—	210	204	178	63	66	—	60	66	64
Cardiff	9	9	198	—	28	26	9	9	51	—	27	21
Brno	20	20	188	33	—	22	20	19	64	33	—	22
Berlin	18	17	185	31	22	—	18	17	59	30	22	—
bandwidth from												
A'dam DAS-2	—	11338	42	750	3923	2578	—	11442	40	747	4115	2578
A'dam Sun	11511	—	22	696	2745	2611	11548	—	46	701	3040	2626
Lecce	73	425	—	44	43	75	77	803	—	94	110	82
Cardiff	842	791	29	—	767	825	861	818	37	—	817	851
Brno	3186	2709	26	588	—	2023	3167	2705	37	612	—	2025
Berlin	2555	2633	9	533	2097	—	2611	2659	9	562	2111	—

TABLE 4.2
Round-trip wide-area latency (in milliseconds) and achievable bandwidth (in KByte/s) between the GridLab sites.

Another problem we encountered was that the JITs installed on some sites contained bugs. Especially the combination of threads and sockets presented some difficulties. There seems to be a bug in Sun's 1.3 JIT (HotSpot) related to threads and socket communication. In some circumstances, a blocking operation on a socket would block the whole application instead of just the thread that does the operation. The solution for this problem was to upgrade to a Java 1.4 JIT, where the problem is solved.

Finally, some machines in the testbed are multi-homed: they have multiple IP addresses. The original Ibis implementation on TCP got confused by this, because the *InetAddress.getLocalHost* method can return an IP address in a private range, or an address for an interface that is not accessible from the outside. Our current solution is to manually specify which IP address has to be used when multiple choices are available. All machines in the testbed have a Globus [10] installation, so we used GSI-SSH (Globus Security Infrastructure Secure Shell) [11] to login to the GridLab sites. We had to start the application by hand, as not all sites have a job manager installed. When a job manager is present, Globus can be used to start the application automatically.

As shown in Table 4.1, we used 40 processors in total, using 6 machines located at 5 sites all over Europe, with 4 different processor architectures. After solving the aforementioned practical problems, Satin on the TCP Ibis implementation ran on all sites, in pure Java, without having to recompile anything.

As a benchmark, we first ran the parallel version of the ray tracer with a smaller problem size (512×512, with 24 bit color) on a single machine on all clusters. This way, we can compute the relative speeds of the different machines and JVMs. The results are presented in Table 4.4. To calculate the relative speed of each machine/JVM combination, we normalized the run times relative

problem	solution
firewalls	bind all sockets to ports in the open range
buggy JITs	upgrade to Java 1.4 JITs
multi-homes machines	use a single, externally valid IP address

TABLE 4.3
Problems encountered in a real grid environment, and their solutions.

site	architecture	run time (s)	relative node speed	relative total speed of cluster	% of total system
A'dam DAS-2	1 GHz Intel Pentium-III	233.1	1.000	8.000	32.4
A'dam Sun	750 MHz UltraSPARC-III	445.2	0.523	1.046	4.2
Lecce	667 MHZ Compaq Alpha	512.7	0.454	1.816	7.4
Cardiff	1 GHz Intel Pentium-III	758.9	0.307	0.614	2.5
Brno	2.4 GHz Intel Xeon	152.8	1.525	12.200	49.5
Berlin	500 MHz MIPS R14000	3701.4	0.062	0.992	4.0
total				*24.668*	100.0

TABLE 4.4
Relative speeds of the machine and JVM combinations in the testbed.

to the run time of the ray tracer on a node of the DAS-2 cluster in Amsterdam. It is interesting to note that the quality of the JIT compiler can have a large impact on the performance at the application level. A node in the DAS-2 cluster and the machine in Cardiff are both 1 GHz Intel Pentium-IIIs, but there is more than a factor of three difference in application performance. This is caused by the different JIT compilers that were used. On the DAS-2, we used the more efficient IBM 1.4 JIT, while the SUN 1.4 JIT (HotSpot) was installed on the machine in Cardiff.

Furthermore, the results show that, although the clock frequency of the machine at Brno is 2.4 times as high as the frequency of a DAS-2 node, the speed improvement is only 53%. Both machines use Intel processors, but the Xeon machine in Brno is based on Pentium-4 processors, which do less work per cycle than the Pentium-III CPUs that are used by the DAS-2. We have to conclude that it is in general not possible to simply use the clock frequencies to compare processor speeds.

Finally, it is obvious that the Origin machine in Berlin is slow compared to the other machines. This is partly caused by the inefficient JIT, which is based on the SUN HotSpot JVM. Because of the combination of slow processors and the inefficient JIT, the 16 nodes of the Origin we used are about as fast as a single 1 GHz Pentium-III with the IBM JIT. The Origin thus hardly contributes anything to the computation. The table shows that, although we used 40 CPUs in total for the grid run, the relative speed of these processors together adds up to 24.668 DAS-2 nodes (1 GHz Pentium-IIIs). The percentage of the total compute power that each individual cluster delivers is shown in the rightmost column of Table 4.4.

algorithm	run time (s)	communication time (s)	communication overhead	parallelization time (s)	parallelization overhead	efficiency
nighttime						
RS	877.6	198.5	36.1%	121.9	23.5%	62.6%
CRS	676.5	35.4	6.4%	83.9	16.6%	81.3%
daytime						
RS	2083.5	1414.5	257.3%	111.8	21.7%	26.4%
CRS	693.0	40.1	7.3%	95.7	18.8%	79.3%
single cluster 25						
RS	579.6	11.3	2.0%	11.0	1.9%	96.1%

TABLE 4.5
Performance of the ray tracer application on the GridLab testbed.

We also ran the ray tracer on a single DAS-2 machine, with the large problem size that we will use for the grid runs. This took 13746 seconds (almost four hours). The sequential program without the Satin constructs takes 13564 seconds, the overhead of the parallel version thus is about 1%. With perfect speedup, the run time of the parallel program on the GridLab testbed would be 13564 divided by 24.668, which is 549.8 seconds (about nine minutes). We consider this run time

the upper bound on the performance that can be achieved on the testbed, $t_{perfect}$. We can use this number to calculate the efficiency that is achieved by the real parallel runs. We call the actual run time of the application on the testbed t_{grid}. In analogy to Section 2.3, efficiency can be defined as follows:

$$efficiency = \frac{t_{perfect}}{t_{grid}} * 100\%$$

We have also measured the time that is spent in communication (t_{comm}). This includes idle time, because all idle time in the system is caused by waiting for communication to finish. We calculate the relative communication overhead with this formula:

$$communication\ overhead = \frac{t_{comm}}{t_{perfect}} * 100\%$$

Finally, the time that is lost due to parallelization overhead (t_{par}) is calculated as shown below:

$$t_{par} = t_{grid} - t_{comm} - t_{perfect}$$

$$parallelization\ overhead = \frac{t_{par}}{t_{perfect}} * 100\%$$

	intra cluster		inter cluster	
alg.	messages	MByte	messages	MByte
nighttime				
RS	3218	41.8	11473	137.3
CRS	1353295	131.7	12153	86.0
daytime				
RS	56686	18.9	149634	154.1
CRS	2148348	130.7	10115	82.1
single cluster 25				
RS	45458	155.6	n.a.	n.a.

TABLE 4.6
Communication statistics for the ray tracer application on the GridLab testbed.

The results of the grid runs are shown in Table 4.5. For reference, we also provide measurements on a single cluster, using 25 nodes of the DAS-2 system. The results presented here are the fastest runs out of three experiments. During daytime, the performance of the ray tracer with RS showed a large variability, some runs took longer than an hour to complete, while the fastest run took about half an hour. Therefore, in this particular case, we took the best result of six runs. This approach thus is in favor of RS. With CRS, this effect does not occur: the difference between the fastest and the slowest run during daytime was less than 20 seconds. During night, when there is little Internet traffic, the application with CRS is already more than 200 seconds faster (about 23%) than with the RS algorithm. During daytime, when the Internet links are heavily used, CRS outperforms RS by a factor of three. Regardless of the time of the day, the efficiency of a parallel run with CRS is about 80%.

The numbers in Table 4.5 show that the parallelization overhead on the testbed is significantly higher compared to a single cluster. Sources of this overhead are thread creation and switching caused by incoming steal requests, and the locking of the work queues. The overhead is higher on the testbed, because five of the six machines we use are SMPs (i.e. they have a shared memory architecture). In general, this means that the CPUs in such a system have to share resources, making memory access and especially synchronization potentially more expensive. The latter has a negative effect on the performance of the work queues. Also, multiple CPUs share a single network interface,

making access to the communication device more expensive. The current implementation of Satin treats SMPs as clusters (i.e., on a N-way SMP, we start N JVMs). Therefore, Satin pays the price of the SMP overhead, but does not exploit the benefits of SMP systems, such as the available shared memory. An implementation that does utilize shared memory when available is planned for the future.

Communication statistics of the grid runs are shown in Table 4.6. The numbers in the table totals for the whole run, summed over all CPUs. Again, statistics for a single cluster run are included for reference. The numbers show that almost all of the overhead of RS is in excessive wide-area communication. During daytime, for instance, it tries to send 154 MByte over the busy Internet links. During the time-consuming wide-area transfers, the sending machine is idle, because the algorithm is synchronous. CRS sends only about 82 MBytes over the wide-area links (about half the amount of RS), but more importantly, the transfers are asynchronous. With CRS, the machine that initiates the wide-area traffic concurrently tries to steal work in the local cluster, and also concurrently executes the work that is found.

CRS effectively trades less wide-area traffic for more local communication. As shown in Table 4.6, the run during the night sends about 1.4 million local-area messages. During daytime, the CRS algorithm has to do more effort to keep the load balanced: during the wide-area steals, about 2.1 million local messages are sent while trying to find work within the local clusters. This is about 60% more than during the night. Still, only 40.1 seconds are spent communicating. With CRS, the run during daytime only takes 16.5 seconds (about 2.4%) longer than the run at night. The total communication overhead of CRS is at most 7.3%, while with RS, this can be as much as two thirds of the run time (i.e. the algorithm spends more time on communicating than on calculating useful work).

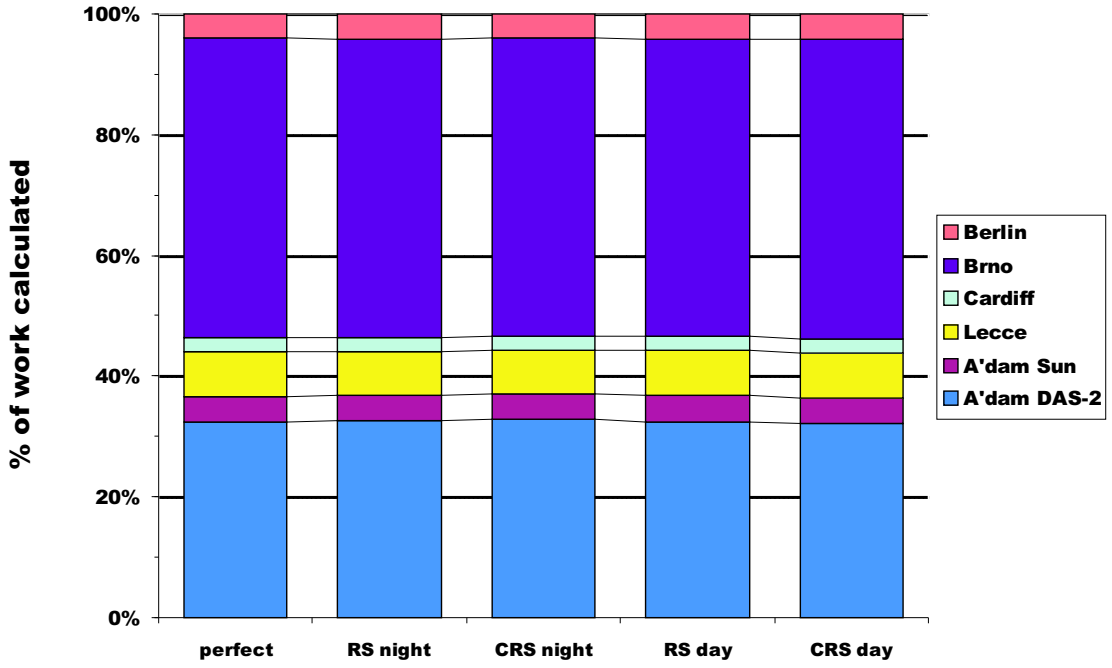

FIG. 4.2. *Distribution of work over the different sites.*

Because all idle time is caused by communication, the time that is spent on the actual computation can be calculated by subtracting the communication time from the actual run time (t_{grid}). Because we have gathered the communication statistics per machine (not shown), we can calculate the total time a whole *cluster* spends computing the actual problem. Given the amount of time a cluster performs useful work and the relative speed of the cluster, we can calculate what fraction of

the total work is calculated by each individual cluster. We can compare this workload distribution with the ideal distribution which is represented by the rightmost column of Table 4.4. The ideal distribution and the results for the four grid runs are shown in Figure 4.2. The difference between the perfect distribution and the actual distributions of the four grid runs is hardly visible. From the figure, we can conclude that, although the workload distribution of both RS and CRS is virtually perfect, the RS algorithm itself spends a large amount of time on *achieving* this distribution. CRS does not suffer from this problem, because wide-area traffic is asynchronous and is overlapped with useful work that was found locally. Still, it achieves an almost optimal distribution.

To summarize, the experiment described in this section shows that the Java-centric approach to grid computing, and the Satin/Ibis system in particular, works extremely well in practice in a real grid environment. It took hardly any effort to run Ibis and Satin on a heterogeneous system. Furthermore, the performance results clearly show that CRS outperforms RS in a real grid environment, especially when the wide-area links are also used for other (Internet) traffic. With CRS, the system is idle (waiting for communication) during only a small fraction of the total run time. We expect even better performance when larger clusters are used, as indicated by our simulator results from Section 2.3.

5. Related work. We have discussed a Java-centric approach to writing wide-area parallel (grid computing) applications. Most other grid computing systems (e.g., Globus [10] and Legion [13]) support a variety of languages. GridLab [2] is building a toolkit of grid services that can be accessed from various programming languages. Converse [15] is a framework for multi-lingual interoperability. The SuperWeb [1], and Bayanihan [29] are examples of global computing infrastructures that support Java. A language-centric approach makes it easier to deal with heterogeneous systems, since the data types that are transferred over the networks are limited to the ones supported in the language (thus obviating the need for a separate interface definition language) [32].

The AppLeS (short for application-level scheduling) project provides a framework for adaptively scheduling applications on the grid [5]. AppLeS focuses on selecting the best set of resources for the application out of the resource pool of the grid. Satin addresses the more low-level problem of load balancing the parallel computation itself, given some set of grid resources. AppLeS provides (amongst others) a template for master-worker applications, whereas Satin provides load balancing for the more general class of divide-and-conquer algorithms.

Many divide-and-conquer systems are based on the C language. Among them, Cilk [7] only supports shared-memory machines, CilkNOW [9] and DCPAR [12] run on local-area, distributed-memory systems. SilkRoad [27] is a version of Cilk for distributed memory systems that uses a software DSM to provide shared memory to the programmer, targeting at small-scale, local-area systems.

The Java classes presented by Lea [18] can be used to write divide-and-conquer programs for shared-memory systems. Satin is a divide-and-conquer extension of Java that was designed for wide-area systems, without shared memory. Like Satin, Javar [6] is compiler-based. With Javar, the programmer uses annotations to indicate divide-and-conquer and other forms of parallelism. The compiler then generates multithreaded Java code, that runs on any JVM. Therefore, Javar programs run only on shared-memory machines and DSM systems.

Herrmann et al. [14] describe a compiler-based approach to divide-and-conquer programming that uses skeletons. Their DHC compiler supports a purely functional subset of Haskell, and translates source programs into C and MPI. Alt et al. [3] developed a Java-based system, in which skeletons are used to express parallel programs, one of which for expressing divide-and-conquer parallelism. Although the programming system targets grid platforms, it is not clear how scalable the approach is: in [3], measurements are provided only for a local cluster of 8 machines.

Most systems described above use some form of random stealing (RS). It has been proven [8] that RS is optimal in space, time and communication, at least for relatively tightly coupled systems like SMPs and clusters that have homogeneous communication performance. In previous work [26], we have shown that this property cannot be extended to wide-area systems. We extended RS to perform asynchronous wide-area communication interleaved with synchronous local communication. The resulting randomized algorithm, called CRS, does perform well in loosely-coupled systems.

Another Java-based divide-and-conquer system is Atlas [4]. Atlas is a set of Java classes that can be used to write divide-and-conquer programs. Javelin 3 [24] provides a set of Java classes that allow programmers to express branch-and-bound computations, such as the traveling salesperson problem. Like Satin, Atlas and Javelin 3 are designed for wide-area systems. Both Atlas and Javelin 3 use tree-based hierarchical scheduling algorithms. We found that such algorithms are inefficient for fine-grained applications and that CRS performs better [26].

6. Conclusions. Grid programming environments need to be both *portable* and *efficient* to exploit the computational power of dynamically available resources. Satin makes it possible to write divide-and-conquer applications in Java, and is targeted at clustered wide-area systems. The Satin implementation on top of our new Ibis platform combines Java's *run everywhere* with efficient communication between JVMs. The resulting system is easy to use in a grid environment. To achieve high performance, Satin uses a special grid-aware load-balancing algorithm. Previous simulation results suggested that this algorithm is more efficient than traditional algorithms that are used on tightly-coupled systems. In this paper, we verified these simulation results in a real grid environment.

We evaluated Satin/Ibis on the highly heterogeneous testbed of the EU-funded GridLab project, showing that Satin's load-balancing algorithm automatically adapts both to heterogeneous processor speeds and varying network performance, resulting in efficient utilization of the computing resources. Measurements show that Satin's CRS algorithm indeed outperforms the widely used RS algorithm by a wide margin. With CRS, Satin achieves around 80% efficiency, even during daytime when the links between the sites are heavily loaded. In contrast, with the traditional RS algorithm, the efficiency drops to about 26% when the wide-area links are congested.

Acknowledgments. Part of this work has been supported by the European Commission, grant IST-2001-32133 (GridLab). We would also like to thank Olivier Aumage, Rutger Hofman, Ceriel Jacobs, Maik Nijhuis and Gosia Wrzesińska for their contributions to the Ibis code. Kees Verstoep is doing a marvelous job maintaining the DAS clusters. Aske Plaat suggested performing an evaluation of Satin on a real grid testbed. John Romein, Matthew Shields and Massimo Cafaro gave valuable feedback on this manuscript.

REFERENCES

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

[2] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, *Enabling Applications on the Grid - A GridLab Overview*, nternational Journal of High Performance Computing Applications, (2003). accepted for publication.

[3] M. Alt, H. Bischof, and S. Gorlatch, *Program Development for Computational Grids using Skeletons and Performance Prediction*, Parallel Processing Letters, 12 (2002), pp. 157–174. World Scientific Publishing Company.

[4] E. J. Baldeschwieler, R. Blumofe, and E. Brewer, *ATLAS: An Infrastructure for Global Computing*, in Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, Connemara, Ireland, September 1996, pp. 165–172.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, *Application-level Scheduling on Distributed Heterogeneous Networks*, in Proceedings of the ACM/IEEE Conference on Supercomputing (SC'96), Pittsburgh, PA, November 1996. Online at http://www.supercomp.org.

[6] A. Bik, J. Villacis, and D. Gannon, *Javar: A Prototype Java Restructuring Compiler*, Concurrency: Practice and Experience, 9 (1997), pp. 1181–1191.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou., *Cilk: An Efficient Multithreaded Runtime System*, in 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95), Santa Barbara, CA, July 1995, pp. 207–216.

[8] R. D. Blumofe and C. E. Leiserson, *Scheduling Multithreaded Computations by Work Stealing*, in 35th Annual Symposium on Foundations of Computer Science (FOCS '94), Santa Fe, New Mexico, November 1994, pp. 356–368.

[9] R. D. Blumofe and P. Lisiecki, *Adaptive and Reliable Parallel Computing on Networks of Workstations*, in USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems, Anaheim, CA, 1997, pp. 133–147.

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

[11] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, *A security architecture for computational grids*, in 5th ACM Conference on Computer and Communication Security, San Francisco, CA, November 1998, pp. 83–92.
[12] B. Freisleben and T. Kielmann, *Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs*, Computers and Artificial Intelligence, 14 (1995), pp. 579–596.
[13] A. Grimshaw and W. A. Wulf, *The Legion Vision of a Worldwide Virtual Computer*, Comm. ACM, 40 (1997), pp. 39–45.
[14] C. A. Herrmann and C. Lengauer, *HDC: A Higher-Order Language for Divide-and-Conquer*, Parallel Processing Letters, 10 (2000), pp. 239–250.
[15] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, *Converse: An interoperable framework for parallel programming*, in Intl. Parallel Processing Symposium, 1996.
[16] K. Kaneda, K. Taura, and A. Yonezawa, *Virtual private grid: A command shell for utilizing hundreds of machines efficiently*, in 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), Berlin, Germany, May 2002, pp. 212–219.
[17] T. Kielmann, H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman, and K. Verstoep, *Programming Environments for High-Performance Grid Computing: the Albatross Project*, Future Generation Computer Systems, 18 (2002), pp. 1113–1125.
[18] D. Lea, *A Java Fork/Join Framework*, in Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, June 2000, pp. 36–43.
[19] C. Lee, S. Matsuoka, D. Talia, A. Sussmann, M. Müller, G. Allen, and J. Saltz, *A Grid programming primer*. Global Grid Forum, August 2001.
[20] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, *RFC 1928: SOCKS protocol version 5*, April 1996.
[21] J. Maassen, T. Kielmann, and H. Bal, *GMI: Flexible and Efficient Group Method Invocation for Parallel Programming*, in In proceedings of LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, Washington DC, March 2002, pp. 1–6.
[22] J. Maassen, T. Kielmann, and H. E. Bal, *Parallel Application Experience with Replicated Method Invocation*, Concurrency and Computation: Practice and Experience, 13 (2001), pp. 681–712.
[23] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, *Efficient Java RMI for Parallel Programming*, ACM Transactions on Programming Languages and Systems, 23 (2001), pp. 747–775.
[24] M. O. Neary and P. Cappello, *Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing*, in Proceedings of the Joint ACM 2002 Java Grande - ISCOPE (International Symposium on Computing in Object-Oriented Parallel Environments) Conference, Seattle, November 2002, pp. 56–65.
[25] *Public netperf homepage*. www.netperf.org.
[26] R. V. v. Nieuwpoort, T. Kielmann, and H. E. Bal, *Efficient Load Balancing for Wide-area Divide-and-Conquer Applications*, in Proceedings Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01), Snowbird, UT, June 2001, pp. 34–43.
[27] L. Peng, W. Wong, M. Feng, and C. Yuen, *SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters*, in IEEE International Conference on Cluster Computing (Cluster2000), Chemnitz, Saxony, Germany, November 2000, pp. 243–249.
[28] M. Philippsen, B. Haumacher, and C. Nester, *More efficient serialization and RMI for Java*, Concurrency: Practice and Experience, 12 (2000), pp. 495–518.
[29] L. F. G. Sarmenta, *Volunteer Computing*, PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2001.
[30] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, *Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing*, Journal of Grid Computing, 1 (2003), pp. 41–51.
[31] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal, *Ibis: an Efficient Java-based Grid Programming Environment*, in Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, USA, November 2002, pp. 18–27.
[32] A. Wollrath, J. Waldo, and R. Riggs, *Java-Centric Distributed Computing*, IEEE Micro, 17 (1997), pp. 44–53.
[33] I.-C. Wu and H. Kung, *Communication Complexity for Parallel Divide-and-Conquer*, in 32nd Annual Symposium on Foundations of Computer Science (FOCS '91), San Juan, Puerto Rico, Oct. 1991, pp. 151–162.