Article submission to Kluwer Journal of Supercomputing

Adaptive Load Balancing for Divide-and-Conquer Grid Applications

Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska,
Thilo Kielmann, Henri E. Bal

Vrije Universiteit, Amsterdam, The Netherlands

Contact author:

```
Thilo Kielmann
Vrije Universiteit
Dept. of Computer Science
De Boelelaan 1081a
1081HV Amsterdam
The Netherlands

kielmann@cs.vu.nl

Phone: +31 20 444 7789
Fax:   +31 20 444 7653
```

Adaptive Load Balancing for Divide-and-Conquer Grid Applications

Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska,
Thilo Kielmann, Henri E. Bal
Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{rob,jason,gosia,kielmann,bal}@cs.vu.nl
http://www.cs.vu.nl/ibis/

Abstract. Divide-and-conquer has been demonstrated as a simple and efficient programming model for grid applications. In previous work, we have presented the divide-and-conquer based Satin system and its load balancing algorithm, *cluster-aware work stealing* (CRS). In this paper, we provide a detailed analysis of CRS with respect to important properties of grid systems, namely scalability, heterogeneous compute clusters, and dynamically changing network interconnects. Our results show that CRS automatically adapts both to heterogeneous processor speeds and varying network performance, resulting in efficient utilization of the computing resources.

Keywords: Grid computing, Java, Ibis, Satin, load balancing, work stealing, CRS

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.

The Global Grid Forum has investigated possible grid programming models [11]. Recently, *GridRPC* has been proposed as a grid programming model [16]. 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 codes 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 [19].

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 [15], 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. In [17], we presented Ibis, our new Java-based grid programming platform that combines Java's "run anywhere" paradigm with highly efficient yet flexible communication mechanisms. There, we also presented a case study in which we have run a Satin application on the testbed of the EU-funded GridLab project [1], consisting of various heterogeneous systems across Europe, connected by the Internet.

In this work, we present a detailed evaluation of Satin's CRS algorithm with respect to important properties of grid systems, namely scalability, heterogeneous compute clusters, and dynamically changing network interconnects. In Section 2, we briefly present Satin's divide-and-conquer programming model. Section 3 covers a detailed analysis of Satin's load-balancing algorithm, CRS. Section 4 discusses related work, and in Section 5 we draw conclusions and outline future work.

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 [12]. In this paper, however, we focus on pure divide-and-conquer programs.

```
interface FibInter extends satin.Spawnable {
    public long fib(long n);
}

class Fib extends satin.SatinObject
    implements FibInter {
    public long fib(long n) {
        if(n < 2) return n;

        long x = fib(n-1); // spawned
        long y = fib(n-2); // spawned
        sync();

        return x + y;
    }

    public static void main(String[] args) {
        Fib f = new Fib();
        long res = f.fib(10); // spawned
        f.sync();
        System.out.println("Fib 10 = " + res);
    }
}
```

Figure 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 *marker interfaces* to indicate that certain method invocations need to be considered for parallel (so called spawned) execution, rather than being executed synchronously like normal methods. Furthermore, a mechanism is needed to synchronize with spawned method invocations, namely to wait for their results. With Satin, this can be expressed using a special interface, satin.Spawnable, and the class satin.SatinObject. This is shown in Fig. 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 [15], 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 [6]. 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 [6, 19]. 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, 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. A detailed description of Satin's wide-area work stealing algorithm can be found in [15].

3. Analysis of Cluster-aware Random Stealing (CRS)

In this section, we analyze the behavior of CRS. First, we compare CRS with RS, the traditional random stealing algorithm. Next, we analyze the sensitivity of CRS to varying WAN latency and bandwidth, followed

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

	single cluster		20 ms 1 MByte/s		20 ms 100 KByte/s		200 ms 1 MByte/s		200 ms 100 KByte/s	
app.	time	%eff.	time	%eff.	time	%eff.	time	%eff.	time	%eff.
integrate										
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

by a scalability analysis for larger numbers of clusters. We investigate the impact of heterogeneous and dynamically changing WAN connections on the behavior of CRS. We conclude our analysis by discussing two variations of CRS: cluster-aware multiple random stealing (CMRS) and adaptive cluster-aware random stealing (ACRS).

For our analysis, we use multiple, virtual clusters within a single, large cluster computer. The wide-area network in between virtual clusters has been simulated with our Panda WAN simulator [9]. This setting allows us to evaluate the behavior of our load balancing algorithms in an undisturbed and reproducible environment. We would like to emphasize that our performance results have been obtained by running Satin applications in parallel on a real cluster system; only the wide-area network is simulated by inserting additional delay for those messages that cross the borders of the virtual clusters. The results presented in this section have been obtained on 200 MHz Pentium Pro's with a Myrinet network, running the Manta parallel Java system [13].

3.1. Comparison of RS and CRS

We compare the behavior of RS and CRS by running four different Satin applications as described in the following. A summary of our comparison can be found in Table I. 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. We simulated the combinations of 20 ms and 200 ms roundtrip latency with bandwidth capacities of 100 KByte/s and 1000 KByte/s.

In Table I, 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 efficiency. 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 in Equation (1).

$$\text{efficiency} = \frac{t_s}{t_p \cdot N} * 100\% \tag{1}$$

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 are able to identify the individual effects of wide-area latency and bandwidth, these results are limited to homogeneous In-

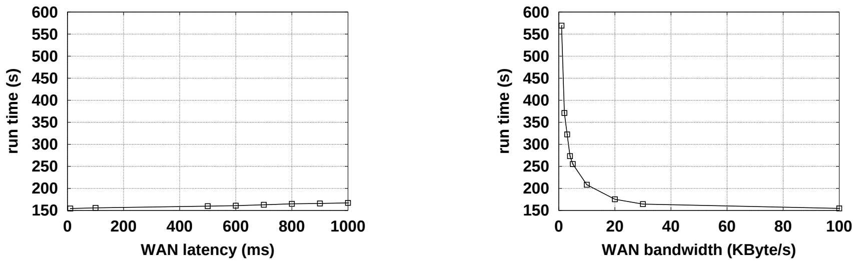

Figure 2. Left: the effect of latency (one-way) on the raytracer, bandwidth is fixed to 100 KByte/s. Right: the effect of bandwidth on the raytracer, one-way latency is fixed to 100 ms.

tel/Linux clusters (due to the Manta compiler). Furthermore, we only tested clusters of identical size. An evaluation on a real grid testbed, with heterogeneous CPUs, JVMs, and networks has been presented in [17]. In the following, we analyse the behaviour of Satin using emulated wide-area network connections. This excludes disturbing third-party traffic, yielding reproducible results and fair comparisons of the investigated approaches.

3.2. Sensitivity to WAN bandwidth and latency

We now investigate the behavior of Satin's CRS algorithm under different WAN bandwidth and latency conditions. We focus on the raytracer application. From the group of test applications presented above, the raytracer is the one which is the most sensitive to WAN bandwidth. With other applications, effects will be similar but less pronounced. Again we analyze a system of four virtual clusters of 16 CPUs each.

Figure 2 (left side) shows the effect of latency on the raytracer application. It is clear that the CRS algorithm is insensitive to the WAN latency, as it is asynchronous. Even with a one-way WAN latency of one second, the run time of the raytracer with the CRS algorithm is only 167.1 seconds. For comparison, on one cluster of 64 nodes (i.e., without WAN links), the application takes 147.8 seconds. This means that even with WAN links with a one-way latency of one second, the overhead compared to a single cluster is only 13%.

The effect of WAN bandwidth on CRS is more significant, as is shown in Figure 2 (right side). While reducing the bandwidth from 100 KByte/s, the run times increase almost linearly, down to about 20 KByte/s. At this point, the speedup relative to the sequential version still is 50.8. When the bandwidth is further reduced, the run time increases exponentially. With a WAN bandwidth of only 1 KByte/s, the raytracer runs for 569.2 seconds, the speedup relative to the sequential version then is 15.6. Thus, with this low bandwidth, running on multiple clusters does not make sense anymore, as the same run time can be achieved on a single cluster of 16 machines. The other applications show similar behavior, but the bandwidth at which the run times begin to increase exponentially is even lower.

These results are promising, as CRS can tolerate very high WAN latencies. CRS also works well with low WAN bandwidths. However, there is a certain amount of WAN bandwidth that an application requires. If the bandwidth drops below that point, the run times increase exponentially. This bandwidth is very low for the applications we tried, 20 KByte/s for the raytracer which is the most demanding of our test applications. Furthermore, we believe that WAN bandwidth keeps improving, so this will become less of a problem. Latency however, is ultimately bound by the speed of light. It is therefore more important that CRS can tolerate high WAN latencies.

3.3. Scalability analysis

We now investigate the scalability of CRS to more than four clusters. It is interesting to know how large the clusters should minimally be to achieve good performance with CRS on hierarchical wide-area systems. Therefore, we measured the performance of the raytracer (because this application uses the most WAN bandwidth) with differently sized clusters. We keep the total number of machines in the runs the same, so we can compare the run times. With clusters of size one (all links are WAN links), the system is no longer hierarchical, so we can quantatively investigate how much the load balancing of the applications benefit from our assumption that systems are hierarchical in nature. We present measurements on 32 nodes, because of a limitation of the Panda cluster simulator.

The run times of the raytracer with differently sized clusters are shown in Figure 3. The numbers show that CRS scales quite well to many small clusters. Even when using 16 clusters of only two nodes, the run time is only increased with 5% relatively to one single cluster (from 288.9 seconds to 303.1 seconds). When the system is no longer hierarchical (i.e., 32 clusters of one node), performance decreases considerably. In that case, CRS (like RS) uses only synchronous stealing, and is always idle during the WAN round-trip time.

We also investigated the performance of the raytracer on systems with differently sized clusters. Because of the way CRS selects steal targets in remote clusters, all nodes in the system have the same chance of being stolen from. This way, machines in smaller clusters are not favored over machines in larger clusters, and a good load balance is

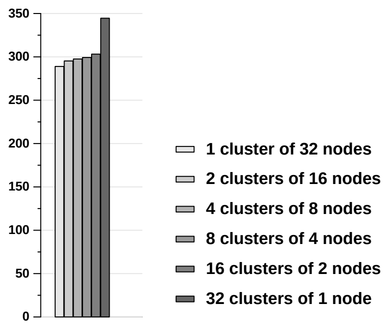

Figure 3. Run times of the raytracer on 32 nodes, with differently sized clusters. One-way latency used is 100 ms, bandwidth is 100 KByte/s.

achieved. Compared to RS, CRS performs much better. When a system with two clusters, one of size 16, and one of size 48 is used (for a total of 64 nodes), with a one-way WAN latency of 100 ms and a bandwidth of 100 KByte/s, the speedup of RS is only 42.6, while CRS achieves a speedup of 54.3. The speedup is not as good as the speedup of CRS on four clusters with the same WAN link speed (59.7), because the total WAN bandwidth in the system is much lower (one link with 100 KByte/s, instead of six links with 100 KByte/s each). If the WAN bandwidth of the asymmetrical system is increased to 600 KByte/s, the speedup of CRS increases to 58.4, only slightly less than the symmetrical system, while RS achieves a speedup of 53.6.

When the system is made even more asymmetrical, for instance a system with three clusters of 8 nodes and one cluster of 40 nodes (i.e., 64 in total), again with 100 ms one-way WAN latency and 100 KByte/s bandwidth, CRS still performs good with a speedup of 54.7, while RS achieves only 45.8.

3.4. Heterogeneous and dynamic wide-area networks

We now present a case study in which Satin runs across various emulated WAN scenarios. Again we use the Panda system to emulate a grid on a single large cluster, with various user-defined performance scenarios for the wide-area links of the emulated grid. We give a detailed performance evaluation of several load-balancing algorithms in Satin using this system.

We evaluate Satin's work-stealing algorithms by running our four test applications (as introduced in Section 3.1) across four emulated clusters. We use the following nine different WAN scenarios of increasing complexity.

1. The WAN is fully connected. The latency of all links is 100 ms (one-way); 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.

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 one-way 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 [18] measurements of the real DAS system.

Figure 4 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 cluster of 64 nodes. The work stealing algorithms RS and CRS are compared.

The most important scenario is Scenario 9, because that is a replay of NWS data, and thus is an indication of the performance of CRS in a real production system. We use a replay of measured data instead of running on the real wide-area systems to ensure deterministic results, allowing a fair comparison between the load-balancing algorithms.

RS sends by far the most WAN messages. The speedups it achieves are significantly worse, 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.

CRS always performs better than RS. 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

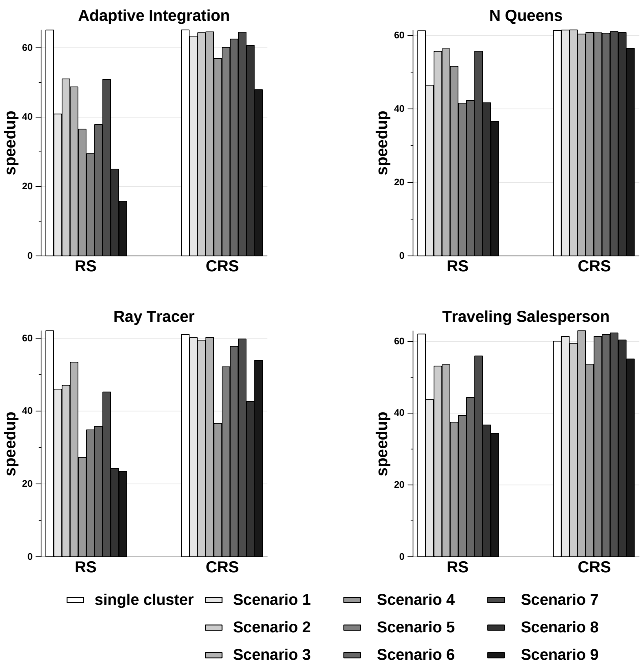

Figure 4. Speedups of 4 Satin applications with 2 load-balancing algorithms and 9 different, emulated WAN scenarios.

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 outperforms RS.

The measurements show that CRS performs better in scenarios with missing links (i.e., Scenario 6 and 7) than in scenarios with slow links (i.e., Scenario 4). This is caused by the routing of messages in the underlying system. When the simulator is configured with a non-fully connected network, the messages are routed around the missing links. When a link is just slow, this does not happen, and the slow link between two clusters is always used, instead of routing messages around the slow link.

A careful analysis of the scenarios where CRS performs suboptimally shows that CRS does not work well on systems that are highly asymmetrical. We will explain this using two additional *extreme* Scenarios 10 and 11, which are shown in Figure 5. In these scenarios, there are two orders of magnitude between the slowest and the fastest WAN links, and the slow links have a bandwidth of only 1 KByte/s. The

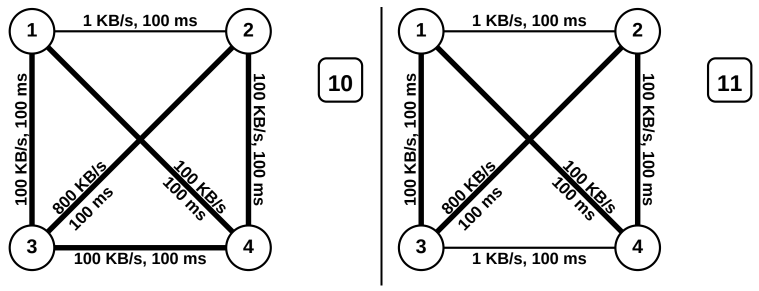

Figure 5. Emulated WAN Scenarios 10 and 11.

scenarios are not realistic, but were deliberately chosen to emphasize the shortcomings of CRS in asymmetrical systems. Scenario 10 is highly asymmetrical, because there is only one extremely slow link, between Clusters 1 and 2. Scenario 11 is similar to Scenario 10, but it is made symmetrical by also using a slow link between Clusters 3 and 4. Thus, the total bandwidth of Scenario 11 is lower than that of Scenario 10. Therefore, one would expect that applications perform better on Scenario 10.

We investigate these scenarios using the raytracer, because this application uses the most WAN bandwidth. As shown above, the raytracer needs about 20 KByte/s of WAN bandwidth per link, therefore the slow WAN links are likely to become a bottleneck. The speedup of CRS with Scenario 10 is 19.9, while the speedup with Scenario 11 is 26.4, with 64 nodes each. CRS achieves a lower speedup on Scenario 10, even though this scenario has more total bandwidth than Scenario 11!

This behavior can be explained as follows: when a node in Cluster 1 issues a wide-area steal attempt, it has a 33% chance of stealing from Cluster 2, as CRS uses random stealing. However, due to the low bandwidth between Cluster 1 and 2, the wide-area steals from a node in Cluster 1 to a node in Cluster 2 will take longer than a steal attempt from a node in Cluster 1 to a node in Cluster 3 or 4. Thus, nodes in Cluster 1 will spend a larger amount of time stealing from nodes in Cluster 2 than they will spend stealing from nodes in Clusters 3 and 4, even though the WAN links to Clusters 3 and 4 are faster. This leads to the underutilization of the fast WAN link between Clusters 1 and 3 and between 1 and 4, while the slow link between Cluster 1 and 2 is overloaded. To summarize: even though the chance that a node in Cluster 1 issues a wide-area steal to a node in Cluster 2 is 33 %, the amount of time spent stealing from Cluster 2 is larger than 33 % of the total wide-area stealing time, because steal attempts to nodes in Cluster 2 take longer than steal attempts to nodes in the other clusters.

This behavior results in load imbalance, as the nodes in Cluster 1 spend a large fraction of the time stealing over the slow WAN link, while nodes in Cluster 3 and 4 both spend 33% of their time stealing from nodes in Cluster 1 (both Clusters 3 and 4 have three WAN links, and all their links have the same speed). Thus, the nodes in Clusters 3 and 4 drain the work out of Cluster 1.

The same thing happens with Cluster 2: the work is drained by the nodes in the Clusters 3 and 4 while the nodes in Cluster 2 are stealing from nodes in Cluster 1, using the slow WAN link. The fact that Cluster 1 and 2 are lightly loaded leads to a downward spiral: it takes longer to steal over the slow WAN links, and the chance of getting work back is small, as the target cluster is being drained via the fast WAN links.

For highly dynamic or asymmetrical grids, more sophisticated algorithms could achieve even better performance than CRS does. Some improvements on CRS are investigated in the remainder of this section.

3.5. Cluster-aware multiple random stealing (CMRS)

To improve the performance of CRS on the aforementioned Scenarios 4, 8 and 9, we experimented with letting nodes send multiple wide-area steal requests to different clusters in parallel. The rationale behind this is that more prefetching of work, and prefetching from different locations, might alleviate the asymmetry problem described above.

We implemented this as an extension of CRS. Each node gets a number of *WAN credits* to spend on wide-area steal requests. In fact, CRS is a special case of CMRS: it is CMRS with only one WAN credit. CMRS works best when the available credits are used for different clusters. This way, the algorithm does not spend multiple WAN credits to attempt to steal work from the same cluster.

Measurements of the applications with CMRS have shown that it always performs worse than CRS. We believe this is the result of inefficient use of the available WAN bandwidth. Precious bandwidth is wasted on parallel wide-area steal request, while the first request may already result in enough work to keep the cluster busy. Moreover, CMRS does not solve the problem of asymmetry. In fact, it only worsens the situation, because, in Scenarios 10 and 11, the nodes in Cluster 3 and 4 also steal in parallel over the fast links. More prefetching drains the amount of work in Clusters 1 and 2 even faster.

3.6. Adaptive cluster-aware random stealing (ACRS)

Another approach we investigated to improve the performance of CRS on the aforementioned Scenarios 4, 8 and 9, is actively detecting and

Table II. An example of modifying the wide-area steal chances in ACRS.

Cluster	t (ms)	1 / t	% chance of stealing from cluster
2	110	0.009	42.86
3	200	0.005	23.81
4	130	0.007	33.33
total	440	0.021	100.00

circumventing slow links. Key is that the algorithm should balance *the amount of time* that is spent on steal requests to all remote clusters. The algorithm should also adapt to changes in WAN link speeds over time. We call the resulting algorithm Adaptive Cluster-aware Random Stealing (ACRS). Adaptivity can be implemented by changing the stochastic properties of the algorithm.

ACRS is almost identical to CRS. However, ACRS measures the time each wide-area steal request takes. The chances of sending a steal request to a remote cluster depends on the performance of the WAN link to that cluster (in this case, we use the time that the last steal request to that cluster took). Because we want the algorithm to prefer fast links over slow links, we do not calculate the chance distribution using the steal time, but use the inverse ($1/t$) instead.

$$\text{chance}_c = \frac{\frac{1}{t_c}}{\sum_i \frac{1}{t_i}} * 100\% \tag{2}$$

In Equation (2), the probability for stealing from cluster c, called chance$_c$, is calculated as the inverse of the time t_c needed for the most recent steal request from c, divided by the sum of the inverse times t_i for all clusters, and multiplying the result with 100%. An example is shown in Table II. In this example, the distribution of probabilities of Cluster 1 in a system with four clusters is shown. The second column shows the time the last steal request to the destination cluster took. The numbers show that ACRS indeed has a preference for the fast links. Other performance data than the last steal request completion time could also be used, such as WAN-link performance data from the NWS.

The speedups of the four applications from Section 3.1 with ACRS on the nine different emulated WAN scenarios are shown in Figure 6. The slow WAN links are still a bottleneck (the chance of stealing over them is not zero), but the speedups of the raytracer with ACRS are significantly better than the speedups with CRS. Adaptive integra-

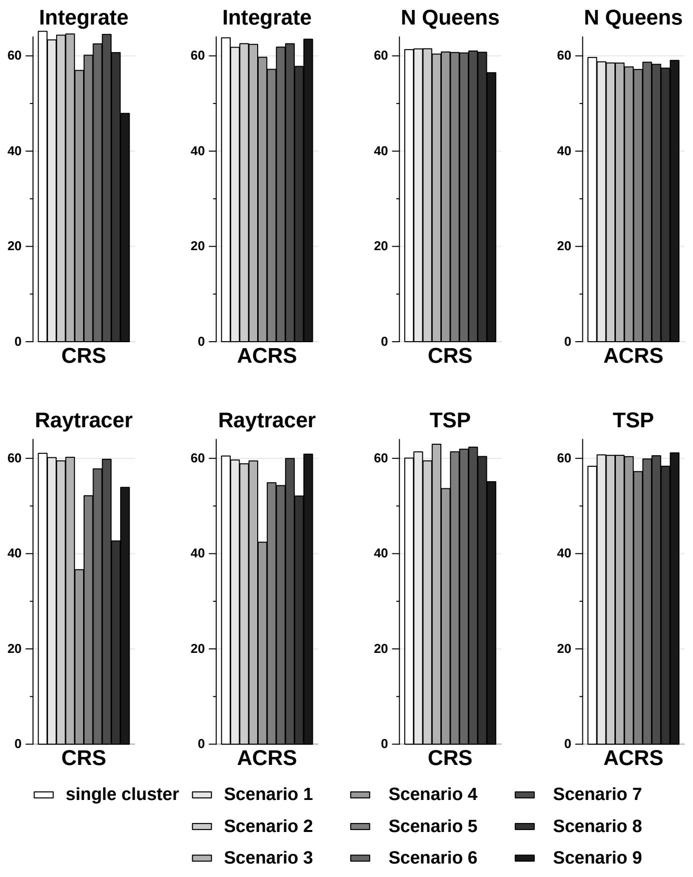

Figure 6. Speedups of 4 Satin applications with ACRS and 9 different, emulated WAN scenarios.

tion, N-queens and the traveling salesperson problem perform nearly optimal in all scenarios with ACRS. For example, the speedup of adaptive integration with Scenario 9 (the WAN links are configured using NWS data) improves from 47.9 to 63.5. The raytracer, which is more bandwidth sensitive than the others, also improves with ACRS. The scenarios that performed worst with CRS, 4, 8 and 9, perform significantly better with ACRS: the speedup of Scenario 4 improves from 36.6 to 42.4, Scenario 8 improves from 42.7 to 52.1, and Scenario 9 goes from a speedup of 53.9 to 60.1. The performance of the other scenarios is also slightly better. The performance of ACRS with the real-life Scenario 9, the playback of NWS data, is nearly perfect for all four applications.

ACRS also improves Satin's performance in the two extreme Scenarios 10 and 11, shown in Table III. The speedup of the raytracer

Table III. Speedups for the raytracer with CRS and ACRS, with different scenarios.

Scenario	description	speedup CRS	speedup ACRS
9	NWS	53.9	60.1
10	asymmetrical	19.9	30.2
11	symmetrical	26.4	29.5

in Scenario 10 goes from 19.9 with CRS to 30.2 with ACRS, while the performance with Scenario 11 goes from 26.4 to 29.5. These results show that ACRS does not suffer from the asymmetrical systems as CRS does, as the speedup in Scenario 10 now is better than the speedup in Scenario 11.

There is a solution that might even perform better than ACRS. It is possible to use source routing for the WAN links at the Satin runtime system level. This way, the runtime system could route WAN steal messages around the slow links. With Scenario 10, for instance, messages from cluster one to cluster two could be routed via Cluster 3. However, this scheme also has several disadvantages. Besides WAN link performance data, global topology information is also needed for the routing of the messages. This information should be gathered during the run, because Internet routing tables may be updated at any time. This leads to a large run time overhead. Moreover, the system is much more complicated than CRS and ACRS, which perform sufficiently, except for very extreme, artificial scenarios.

4. Related work

The AppLeS (short for application-level scheduling) project provides a framework for adaptively scheduling applications on the grid [4]. 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 [5] only supports shared-memory machines, Cilk-NOW [7] and DCPAR [8] run on local-area, distributed-memory systems. The Java classes presented by Lea [10] 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.

Alt et al. [2] 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 [2], 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 in [6] 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 [15], 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 [3]. Atlas is a set of Java classes that can be used to write divide-and-conquer programs. Javelin 3 [14] 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 [15].

5. Conclusions and future work

Satin makes it possible to write divide-and-conquer applications in Java, and is targeted at clustered wide-area systems. To achieve high performance, Satin uses CRS, a special grid-aware load-balancing algorithm. In this paper, we have thoroughly analyzed the behavior of our CRS algorithm. We have demonstrated its superiority compared to the traditional RS in hierarchically organized, cluster-based grid environments. We have shown that CRS is almost insensitive even to very high WAN latencies, while each Satin application requires a specific, but rather low, minimal WAN bandwidth. We have also demonstrated the scalability of CRS to large numbers of possibly small clusters and found only minor performance degradation. We found CRS to be fairly robust to heterogeneous and dynamically changing WAN connections, except for cases of highly asymmetrical connections. Asymmetry leads to load imbalance and in consequence to application slowdown. Based on our findings, we have discussed two possible improvements to CRS, namely cluster-aware multiple random stealing (CMRS) and adaptive cluster-aware random stealing (ACRS). We have shown that ACRS indeed helps improving application speedups in cases of asymmetry. Our measurements presented in [17] show that Satin's CRS algorithm indeed outperforms the widely used RS algorithm by a wide margin on a *real* heterogeneous grid.

Our next step in building grid programming environments is to add fault tolerance to Satin. The divide-and-conquer paradigm presents several advantages when implementing fault tolerance. Function execution will always produce the same outputs if given the same inputs, a property also known as *referential transparency*. The outcome of a computation does not depend on the computation order. Therefore, the work lost in a crash of a process can be redone at any time during execution of the application. Exploiting this feature of the divide-and-conquer paradigm makes it possible to create a fault tolerance low-overhead mechanism based on redoing the work lost by crashed processors. We strongly believe that, with these additions, Satin will become even better suited as a grid programming environment.

Acknowledgments

Part of this work has been supported by the European Commission, grant IST-2001-32133 (GridLab) and by the Netherlands Organisation for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment). We would also like to thank Rutger Hofman and Ceriel Jacobs for their contributions to the Satin code. Kees Verstoep is doing a marvelous job maintaining the DAS clusters. John Romein provided valuable feedback on this manuscript.

References

1. Allen, G., 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: 2003, 'Enabling Applications on the Grid - A GridLab Overview'. *International Journal of High Performance Computing Applications* **17**(4), 449–466.
2. Alt, M., H. Bischof, and S. Gorlatch: 2002, 'Program Development for Computational Grids using Skeletons and Performance Prediction'. *Parallel Processing Letters* **12**(2), 157–174. World Scientific Publishing Company.
3. Baldeschwieler, E. J., R. Blumofe, and E. Brewer: 1996, 'ATLAS: An Infrastructure for Global Computing'. In: *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications.* Connemara, Ireland, pp. 165–172.
4. Berman, F., R. Wolski, S. Figueira, J. Schopf, and G. Shao: 1996, 'Application-level Scheduling on Distributed Heterogeneous Networks'. In: *Proceedings of the*

ACM/IEEE Conference on Supercomputing (SC'96). Pittsburgh, PA. Online at http://www.supercomp.org.

5. Blumofe, R. D., C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou.: 1995, 'Cilk: An Efficient Multithreaded Runtime System'. In: *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*. Santa Barbara, CA, pp. 207–216.
6. Blumofe, R. D. and C. E. Leiserson: 1994, 'Scheduling Multithreaded Computations by Work Stealing'. In: *35th Annual Symposium on Foundations of Computer Science (FOCS '94)*. Santa Fe, New Mexico, pp. 356–368.
7. Blumofe, R. D. and P. Lisiecki: 1997, 'Adaptive and Reliable Parallel Computing on Networks of Workstations'. In: *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*. Anaheim, CA, pp. 133–147.
8. Freisleben, B. and T. Kielmann: 1995, 'Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs'. *Computers and Artificial Intelligence* **14**(6), 579–596.
9. Kielmann, T., H. E. Bal, J. Maassen, R. van Nieuwpoort, L. Eyraud, R. Hofman, and K. Verstoep: 2002, 'Programming Environments for High-Performance Grid Computing: the Albatross Project'. *Future Generation Computer Systems* **18**(8), 1113–1125.
10. Lea, D.: 2000, 'A Java Fork/Join Framework'. In: *Proceedings of the ACM 2000 Java Grande Conference*. San Francisco, CA, pp. 36–43.
11. Lee, C., S. Matsuoka, D. Talia, A. Sussmann, M. Müller, G. Allen, and J. Saltz: 2001, 'A Grid Programming Primer'. Global Grid Forum.
12. Maassen, J., T. Kielmann, and H. E. Bal: 2001a, 'Parallel Application Experience with Replicated Method Invocation'. *Concurrency and Computation: Practice and Experience* **13**(8-9), 681–712.
13. Maassen, J., R. van Nieuwpoort, R. Veldema, H. E. Bal, T. Kielmann, C. Jacobs, and R. Hofman: 2001b, 'Efficient Java RMI for Parallel Programming'. *ACM Transactions on Programming Languages and Systems* **23**(6), 747–775.
14. Neary, M. O. and P. Cappello: 2002, '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, pp. 56–65.
15. Nieuwpoort, R. V. v., T. Kielmann, and H. E. Bal: 2001, '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, pp. 34–43.
16. Tanaka, Y., H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka: 2003, 'Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing'. *Journal of Grid Computing* **1**(1), 41–51.
17. van Nieuwpoort, R. V., J. Maassen, T. Kielmann, and H. E. Bal: 2004, 'Satin: Simple and Efficient Java-based Grid Programming'. *Parallel and Distributed Computing Practices*. Special Issue on Adaptive Grid Middleware.
18. Wolski, R., N. Spring, and J. Hayes: 1999, 'The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing'. *Future Generation Computing Systems* **15**(5–6), 757–768.
19. Wu, I.-C. and H. Kung: 1991, 'Communication Complexity for Parallel Divide-and-Conquer'. In: *32nd Annual Symposium on Foundations of Computer Science (FOCS '91)*. San Juan, Puerto Rico, pp. 151–162.