Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications

Rob V. van Nieuwpoort, Thilo Kielmann, Henri E. Bal
Faculty of Sciences, Division of Mathematics and Computer Science, Vrije Universiteit
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{rob,kielmann,bal}@cs.vu.nl http://www.cs.vu.nl/manta

ABSTRACT

Divide-and-conquer programs are easily parallelized by letting the programmer annotate potential parallelism in the form of *spawn* and *sync* constructs. To achieve efficient program execution, the generated work load has to be balanced evenly among the available CPUs. For single cluster systems, *Random Stealing* (RS) is known to achieve optimal load balancing. However, RS is inefficient when applied to hierarchical wide-area systems where multiple clusters are connected via wide-area networks (WANs) with high latency and low bandwidth.

In this paper, we experimentally compare RS with existing load-balancing strategies that are believed to be efficient for multi-cluster systems, *Random Pushing* and two variants of *Hierarchical Stealing*. We demonstrate that, in practice, they gain less than optimal results. We introduce a novel load-balancing algorithm, *Cluster-aware Random Stealing* (CRS) which is highly efficient and easy to implement. CRS adapts itself to network conditions and job granularities, and does not require manually-tuned parameters. Although CRS sends more data across the WANs, it is faster than its competitors for 11 out of 12 test applications with various WAN scenarios. It has at most 4% overhead in run time compared to RS on a single, large cluster, even with high latencies and low bandwidths. These strong results suggest that divide-and-conquer parallelism is a useful model for writing distributed supercomputing applications on hierarchical wide-area systems.

Keywords

distributed supercomputing, clustered wide-area systems, Java

1. INTRODUCTION

In distributed supercomputing, platforms are often hierarchically structured. Typically, multiple supercomputers or clusters of workstations are connected via wide-area links, forming systems with a two-level communication hierarchy. When running parallel applications on such multi-cluster systems, efficient execution can only be achieved when the hierarchical structure is carefully taken into account. In previous work, we have demonstrated that various kinds of parallel applications can indeed be efficiently executed on wide-area systems [21, 24]. However, each application had to be optimized individually to reduce the utilization of the scarce wide-area bandwidth or to hide the large wide-area latency. We extracted some of these optimizations into specific runtime systems like MPI's collective communication operations (the MagPIe library [12]), and our Java-based object replication mechanism RepMI [15]. In general, however, it is hard for a programmer to manually optimize parallel applications for hierarchical systems.

The ultimate goal of our work is to create a programming environment in which parallel applications for hierarchical systems are easy to implement. The divide-and-conquer model lends itself well for hierarchically structured systems because tasks are created by recursive subdivision. This leads to a hierarchically structured task graph which can be executed with excellent communication locality, especially on hierarchical platforms. Of course, there are many kinds of applications that do not lend themselves well to a divide-and-conquer algorithm. However, we 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 [26].

In this paper we study wide-area load balancing algorithms for divide-and-conquer applications, using our Satin system [23]. Satin is designed for running divide-and-conquer applications on multi-cluster, distributed memory machines, connected by a hierarchical network. Satin's programming model has been inspired by Cilk [7] (hence its name). Satin extends the Java language with two simple primitives for divide-and-conquer programming: *spawn* and *sync*. Satin's compiler and runtime system cooperate to implement these primitives efficiently on a hierarchical wide-area system, without requiring any help or optimizations from the programmer.

We have implemented Satin by extending the Manta native Java compiler [16]. Five load-balancing algorithms were implemented in the runtime system to investigate their behavior on hierarchical wide-area systems. We will demonstrate that *Random Stealing* (RS), as used in single-cluster environments, does not perform well on wide-area systems. We found that hierarchical load balancing, as proposed for example by Atlas [3] and Dynasty [1], performs even worse for fine-grained applications. The hierarchical algorithm sends few messages over the wide area links, but suffers from bad performance inside clusters, and stalls the entire cluster when wide-area steals are issued. We show that our novel algorithm, *Cluster-aware Random Stealing* (CRS), achieves good speedups for a large range of bandwidths and latencies, although it sends more messages than the other cluster-aware load-balancing methods. We compare CRS with *Random Pushing* and *Cluster-aware Load-based Stealing*, a variant of hierarchical stealing. Both

approaches are candidates for efficient execution on multi-cluster systems. We show, however, that CRS outperforms both in almost all test cases.

We discuss the performance of all five load-balancing algorithms using twelve applications on four Myrinet-based clusters, connected by wide-area links, the DAS system.[1] We also describe the impact of different WAN bandwidth and latency parameters. We show that on a system with four clusters, even with a WAN latency of 100 milliseconds and a bandwidth of only 100 KBytes/s, 11 out of 12 applications run only 4% slower than on a single cluster with the same total number of CPUs. These strong results suggest that divide-and-conquer parallelism is a useful model for writing distributed supercomputing applications on hierarchical wide-area systems.

The remainder of the paper is structured as follows. Section 2 briefly introduces the Satin system. In Sections 3 and 4, we present the five load balancing algorithms, and their performance with twelve example applications. Section 5 discusses related approaches. Section 6 concludes our work.

2. THE SATIN SYSTEM

Satin's programming model is an extension of the single-threaded Java model. To achieve parallel execution, Satin programs do not have to use Java's threads or Remote Method Invocations (RMI). Instead, they use the much simpler divide-and-conquer primitives. Satin does allow the combination of its divide-and-conquer primitives with Java threads and RMIs. Additionally, Satin provides shared objects via RepMI [15], Manta's object replication mechanism. None of the applications described in this paper use RepMI, because this allows us to focus on the communication patterns of the load balancing algorithms.

We augmented the Java language with three keywords: `spawn`, `sync`, and `satin`. The `satin` modifier is placed in front of a method declaration. It indicates that the method may be spawned. The `spawn` keyword is placed in front of a method invocation to indicate possibly parallel execution. We call this a *spawned method invocation*. Conceptually, a new thread is started for running the method upon invocation. Satin's implementation, however, eliminates thread creation altogether [23]. A spawned method invocation is put into a local work queue (see Section 3). From the queue, the method might be transferred to a different CPU where it may run concurrently with the method that executed the `spawn`. The `sync` operation waits until all spawned calls in the current method invocation are finished. The return values of spawned method invocations are undefined until a `sync` is reached. Figure 1 illustrates how Satin's annotations can be used to implement a parallel toy program to compute the Fibonacci numbers.

Because Satin runs on distributed-memory machines, non-replicated objects that are passed as parameters in a spawned call will not be available at a remote machine. Therefore, Satin uses *call-by-value* semantics when the runtime system decides that the method will be executed remotely. In this case, the parameter objects (and all objects referenced by the parameters) will be copied to the remote machine, using Manta's highly efficient object-serialization mechanism [16]. However, the common case is that work is executed by the machine that also spawned the work. In the applications we have studied so far, typically 1 out of 1000–10000 jobs gets stolen [23]. It is thus important to minimize the overhead for work done locally. Because copying all parameter objects (i.e., using call-by-value) in the local case would be prohibitively expensive, parameters are passed by reference when the method invocation is local. Therefore, the Satin programmer can neither assume

[1] *http://www.cs.vu.nl/das/*

```
class Fibonacci {
    SATIN int fib(int n) {
        if(n < 2) return n;

        int x = SPAWN fib(n - 1);
        int y = SPAWN fib(n - 2);
        SYNC;

        return x + y;
    }
    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        int result = f.fib(10);
        System.out.println("Fib 10 = " + result);
    }
}
```

Figure 1: A Satin example: Fibonacci.

call-by-value nor call-by-reference for `satin` methods. Non-satin methods are unaffected and keep the standard Java semantics.

With these parameter semantics, Satin methods can be implemented very efficiently. Whenever a method is spawned, only its *invocation record* (containing references to the parameter objects) is inserted into the local work queue. In the case of local execution, the parameters are used call-by-reference. In the case of remote execution, the parameters are actually copied and serialized to the remote machine, resulting in call-by-value semantics. A detailed description of Satin's implementation can be found in [23].

Satin's work stealing algorithms are implemented on top of the Panda communication library [2], which has efficient implementations on a variety of networks. On the Myrinet network (which we use for the measurements in this paper), Panda is implemented on top of the LFC [4] network interface protocol. Satin uses Panda's efficient user-level locks for protecting the work queue. To avoid the overhead of operating-system calls, both LFC and Panda run in user space.

3. LOAD BALANCING IN WIDE-AREA SYSTEMS

To minimize application completion times, load-balancing algorithms try to keep all processors busy performing application-related work. The distribution of jobs causes communication among the processors that has to be performed in addition to the proper application work. Minimizing both the idle time and communication overhead are conflicting goals. Load-balancing algorithms have to carefully balance communication-related overhead and processor idle time [22]. In the case of multi-cluster wide-area systems, the differences in communication costs between the local-area (LAN) and wide-area networks (WAN) have to be taken into account as well, adding further complexity to the optimization problem.

In this section, we discuss the five different load-balancing algorithms we implemented in the Satin system. The properties of the implemented algorithms are summarized in Table 1. We start with existing algorithms for single-cluster systems that try to optimize either communication overhead (*random stealing* (RS) [8]) or idle time (*random pushing* (RP) [22]). We show that both are inefficient for multi-cluster wide-area systems. Next, we investigate *Cluster-aware Hierarchical Stealing* (CHS) [1, 3] which is believed to be efficient for multi-cluster systems. However, by optimizing wide-area communication only, this approach leads to mediocre results due to excessive local communication and causes whole clusters to stall while waiting for remote work.

Table 1: Properties of the Implemented Load Balancing Algorithms.

algorithm	optimization goal	drawbacks	heuristics	work transfer
Random Stealing (RS)	communication	high idle time due to synchronous WAN communication	no	synchronous
Random Pushing (RP)	idle time	unstable, too much WAN communication	yes	asynchronous
Cluster-aware Hierarchical Stealing (CHS)	WAN communication	too much LAN communication, cluster stalling	no	synchronous
Cluster-aware Load-based Stealing (CLS)	LAN communication WAN latency hiding	slow work distribution inside cluster, prefetching bottleneck	yes	LAN synchronous WAN prefetching
Cluster-aware Random Stealing (CRS)	LAN communication WAN latency hiding		no	LAN synchronous WAN asynchronous

To overcome the drawbacks of the three existing algorithms, we implemented two new load-balancing algorithms. The first one, *Cluster-aware Load-based Stealing* (CLS), directly improves CHS. It combines RS inside clusters with work prefetching between clusters, the latter based on monitoring the load of all nodes. This algorithm performs quite well, but still suffers from two drawbacks. It relies on manually tuned parameters and propagates work rather slowly between clusters.

The second new algorithm is called *Cluster-aware Random Stealing* (CRS). It combines RS inside clusters with controlled asynchronous work stealing from other clusters. CRS minimizes LAN communication while avoiding high idle time due to synchronous WAN stealing. CRS does not need parameter tuning and is almost trivial to implement. In Section 4 we will show that, with CRS, 11 out of 12 applications running on multiple clusters are only marginally slower than on a single large cluster, with the same number of processors using RS.

All algorithms use a double-ended work queue on each node, containing the *invocation records* of the jobs that were spawned but not yet executed. Divide-and-conquer programs progress by splitting their work into smaller pieces, and by executing the small jobs that are not worth further splitting. New jobs are inserted at the head of the queue. Each processor fetches work from the head of its own queue. All load-balancing algorithms described here use the jobs at the tail of the queue for balancing the work load. This scheme ensures that the most fine-grained jobs run locally, while the jobs at the tail of the queue are the most coarse grained ones available. They are ideal candidates for load-balancing purposes.

3.1 Random Stealing (RS)

Random stealing is a well known load-balancing algorithm, used both in shared-memory and distributed-memory systems. 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. 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. RS is provably efficient in terms of time, space, and communication for the class of fully strict computations [8, 26]; divide-and-conquer algorithms belong to this class. An advantage of RS is that the algorithm is *stable* [22]: communication is only initiated when nodes are idle. When the system load is high, no communication is needed, causing the system to behave well under high loads.

On a single-cluster system, RS is, as expected, the best performing load-balancing algorithm. On wide-area systems, however, this is not the case. With C clusters, on average $(C-1)/C$ of all steal requests will go to a node in a remote cluster. Since the steal attempts are synchronous (see Table 1), the stealing processor (the "thief") has to wait a wide-area round-trip time for a result, while there may be work available inside the local cluster. With four clusters, this already affects 75% of all steal requests. Also, when jobs contain much data, the limited wide-area bandwidth becomes a bottleneck.

3.2 Random Pushing (RP)

Random pushing is another well known load-balancing algorithm [22]. With RP, a processor checks, after insertion of a job, whether the queue length exceeds a certain threshold value. If this is the case, a job from the queue's tail (where the largest jobs are) is pushed to a randomly chosen peer processor. This approach aims at minimizing processor idle time because jobs are pushed ahead of time, before they are actually needed, but comes at the expense of additional communication overhead. One might expect RP to work well in a wide-area setting, because its communication is asynchronous (see Table 1) and thus less sensitive to high wide-area round-trip times than work stealing. A problem with RP, however, is that the algorithm is not *stable*. Under high work loads, job pushing causes useless overhead, because all nodes already have work.

Also, with C clusters, in average $(C-1)/C$ of the pushed jobs will be sent across the wide-area network, causing bandwidth problems. Unlike random stealing, random pushing does not adapt its WAN utilization to bandwidth and latency as it lacks a bound for the number of messages that may be sent, i.e. there is no inherent flow-control mechanism. Memory space is also not bounded: jobs may be pushed away as fast as they can be generated, and have to be stored at the receiver. To avoid exceeding communication buffers, Satin's implementation of RP adds an upper limit of jobs sent by each node that can be in transit simultaneously. This upper limit has to be optimized manually. Additionally, a threshold value must be found that specifies when jobs will be pushed away. A single threshold value is not likely to be optimal for all applications, or not even for one application with different wide-area bandwidths and latencies. Simply pushing away all generated jobs is found to perform well in theory [22] (without taking bandwidth and latency into account), but, in practice, has too much communication and marshalling overhead for fine-grained divide-and-conquer applications.

3.3 Cluster-aware Hierarchical Stealing (CHS)

Random stealing and random pushing both suffer from too much WAN communication. Cluster-aware Hierarchical Stealing (CHS) has been presented for load balancing divide-and-conquer applications in wide-area systems (e.g., for Atlas [3] and Dynasty [1]). The goal of CHS is to minimize wide-area communication. The idea is to arrange processors in a tree topology, and to send steal messages along the edges of the tree. When a node is idle, it first asks its child nodes for work. If the children are also idle, steal messages will recursively descend the tree. Only when the entire subtree is idle, messages will be sent upwards in the tree, asking parent nodes for work.

This scheme exhibits much locality, as work inside a subtree will always be completely finished before load-balancing messages are sent to the parent of the subtree. By arranging a cluster's nodes in a tree shape, the algorithm's locality can be used to minimize wide-area communication. Multiple cluster trees are interconnected at their root nodes via wide-area links. When the cluster root node finds its own cluster to be idle, it sends a steal message to the root node of another, randomly selected cluster. Such an arrangement is shown in Figure 2.

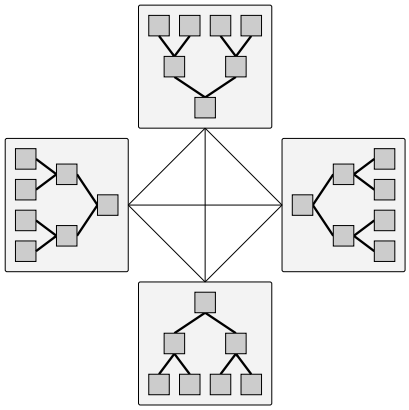

Figure 2: Cluster Hierarchical Stealing: arrange the nodes in tree shapes and connect multiple trees via wide-area links.

CHS has two drawbacks. First, the root node of a cluster waits until the entire cluster becomes idle before starting wide-area steal attempts. During the round-trip time of the steal message, the entire cluster remains idle. Second, the preference for stealing further down the tree results in jobs with finer granularity to be stolen first, leading to high LAN communication overhead, due to many job transfers.

The actual implementation of hierarchical stealing is more complex. When a job is finished that runs on a different machine than the one it was spawned on, the return value must be sent back to the node that generated the job. While the result arrives, this node may be stealing from its children or parent. However, the incoming return value may allow new jobs to be generated, so the receiver may not be idle anymore. In this case, Satin lets the receiver cancel its potentially very expensive steal request that might be forwarded recursively, even across wide-area links.

3.4 Cluster-aware Load-based Stealing (CLS)

CHS suffers from high LAN communication overhead, caused by stealing fine-grained jobs, and from the stalling of a whole cluster when stealing across a wide-area connection. To address these two problems while keeping the wide-area communication at a low level, we developed Cluster-aware Load-based Stealing (CLS). The idea behind CLS is to combine random stealing inside clusters with wide-area work prefetching performed by one coordinator node per cluster. With a single cluster, CLS is identical to RS.

Random stealing inside a cluster does not have the preference for stealing fine-grained jobs and thus reduces the LAN communication overhead, compared to the tree-based approach. The WAN communication can be controlled and minimized by letting only coordinator nodes perform wide-area work stealing. To avoid whole-cluster stalling, the coordinator nodes can prefetch jobs. Prefetching requires a careful balance between communication overhead and processor idle time. On the one hand, when jobs are prefetched too early, the communication overhead grows unnecessarily. On the other hand, when jobs are prefetched too late, processors may become idle. Pure work stealing can be seen as one extreme of this tradeoff, where jobs are never prefetched. Pure work pushing is the other extreme where jobs are always transferred ahead of time.

In CLS, prefetching is controlled by load information from the compute nodes. Each node periodically sends its load information to the cluster coordinator that monitors the overall load of its cluster. The compute nodes send their load messages asynchronously, keeping the overhead small. Satin further reduces this overhead by sending periodical load messages only when the load actually has changed. Furthermore, when a node becomes idle, it immediately sends a load message (with the value zero) to the coordinator. A good interval for sending the load messages is subject to parameter tuning. It was empirically found to be 10 milliseconds. This reflects the rather fine granularity of Satin jobs. Sending load messages with this interval does not noticeably decrease the performance. When the total cluster load drops below a specified threshold value, the coordinator initiates inter-cluster steal attempts to randomly chosen nodes in remote clusters. The coordinator can hide the possibly high wide-area round-trip time by overlapping communication and computation, because wide-area prefetch messages are asynchronous.

Another tunable parameter is the threshold value that is used to trigger wide-area prefetching. This parameter strongly depends on the application granularity and the WAN latency and bandwidth. In Satin's implementation of CLS, we use the length of the work queue as load indicator. This indicator might not be very accurate, due to the job granularity that descends with increasing recursion depth. We found that using the recursion depth as measure of a job's size does not improve performance. The DCPAR system [11] uses programmer annotations to express job granularities, but even this approach fails with irregular applications that perform pruning in the task tree. Despite its weakness, using the queue length as load indicator is our only choice, as additional, accurate information is unavailable. Our results confirm those of Kunz [13], who found that the choice of load indicator can have a considerable impact on performance, and that the most effective one is queue length.

3.5 Cluster-aware Random Stealing (CRS)

The CLS algorithm described above minimizes LAN communication overhead while simultaneously reducing WAN communication and idle time. However, CLS relies on careful parameter tuning for the volume of job prefetching. Furthermore, prefetched jobs are stored on a centralized coordinator node rather than on the idle nodes themselves. The distribution of jobs to their final destination (via random stealing) adds some overhead to this scheme. With high wide-area round-trip times, the coordinator node might even become a stealing bottleneck if the local nodes together compute jobs faster than they can be prefetched by the coordinator node.

We designed Cluster-aware Random Stealing (CRS) to overcome these problems. Like CLS, it uses random stealing inside clusters. However, it uses a different approach to wide-area stealing. The idea is to omit centralized coordinator nodes at all. Instead, we implement a decentralized control mechanism for the wide-area communication directly in the worker nodes.

Pseudocode for the new algorithm is shown in Figure 3. 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

```
void cluster_aware_random_stealing(void){
  while(NOT exiting) {
    job = queue_get_from_head();
    if(job) {
      execute(job);
    } else {
      if(nr_clusters > 1 AND NOT stealing_remotely) {
        /* no wide-area message in transit */
        stealing_remotely = true;
        send_async_steal_request(remote_victim());
      }
      /* do a synchronous steal in my cluster */
      job = send_steal_request(local_victim());
      if(job) queue_add_to_tail(job);
    }
  }
}

void handle_wide_area_reply(Job job){
  if(job) queue_add_to_tail(job);
  stealing_remotely = false;
}
```

Figure 3: Pseudo code for Cluster-aware Random Stealing.

requests to 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.

As shown in Section 4, this asynchronous wide-area stealing successfully hides the long wide-area round-trip times. The mechanism also implements an efficient way of job prefetching that delivers the new job directly on the idle node and does not need parameter tuning. The implication of this scheme is that many (or all) remote clusters will be asked for work concurrently when a large part of a cluster is idle. As soon as one remote steal attempt is successful, the work will be quickly distributed over the whole cluster, because local steal attempts are performed during the wide-area round-trip time. Thus, when jobs are found in a remote cluster, the work is propagated quickly.

Compared to RS, CRS significantly reduces the number of messages sent across the wide-area network. CRS has the advantages of random stealing, but hides the wide-area round-trip time by additional, local stealing. The first job to arrive will be executed. No extra load messages are needed, and no parameters have to be tuned. On a single cluster, CRS is identical to RS.

3.6 Other Possible Algorithms

There may be many more load-balancing algorithms that perform well in a wide-area setting. For example, Eager et al. [10] describe a form of work pushing, where the sender first polls target nodes, until one is found that is not overloaded. Only after this node is found, the work is transferred. In a wide-area setting, this would imply that multiple wide-area round-trip times may be needed to transfer work, hence giving up the benefit of asynchronous WAN communication of our RP algorithm. However, for bandwidth sensitive applications, this may perform better than the form of work pushing implemented by Satin, because jobs are not sent over wide-area links unless they are needed in the remote cluster.

Also, some of Satin's algorithms could be further improved. For example, our CLS implementation selects a random stealing victim in a random cluster. It could also ask the remote cluster coordinator what the best node in the cluster is. It is unclear what the performance of this scheme would be, because load information is outdated quickly. The hierarchical scheme, CHS, might be tuned by trying different tree shapes inside the clusters. The algorithms that are sensitive to parameter tuning (RP and CLS) might be improved by adaptively changing the threshold values at runtime.

We do not claim that we have investigated all possibilities, but we presented a number of load balancing algorithms that span the spectrum of possible design alternatives. We described a simple, cluster-aware algorithm (CRS) that performs almost as fast as random stealing in a single cluster, as will be shown in the next section. CRS does not rely on parameter tuning, but adapts itself to given WAN parameters. There may be other algorithms that perform equally well, but they are probably more complicated. Therefore, we argue that CRS is well-suited for load balancing divide-and-conquer applications on hierarchical wide-area systems.

4. PERFORMANCE EVALUATION

We evaluated Satin's performance using 12 application kernels, which are taken from [7], [11], and [14], translated to Java and modified to use Satin's parallel extensions. All measurements were performed on a cluster of the Distributed ASCI Supercomputer (DAS), containing 200 MHz Pentium Pros running RedHat Linux 6.2 (kernel 2.2.16) that are connected by 1.28 Gb/s Myrinet. Our results have been obtained on a *real parallel machine*; only the wide-area links are simulated by letting the communication subsystem insert delay loops into message delivery [21]. This allows us to investigate the performance of the load balancing algorithms with different communication performance parameters. We have verified some of the results on four geographically distributed Myrinet clusters of the DAS, which are connected by wide-area links. The results on this system are consistent with the results we provide here, with simulated wide-area links.

Most applications use a threshold value in order to improve performance: at some level in the recursion, work is not subdivided and spawned anymore, but executed with an efficient sequential algorithm, which may not use the divide-and-conquer model. For example, matrix multiplication recursively subdivides the matrix into blocks using Satin's parallel divide-and-conquer primitives, but stops at blocks of size 48×48. Then a sequential matrix multiplication code that steps 2 rows and columns at a time is used to efficiently multiply two blocks.

4.1 Low-level Benchmarks

An important indication of the performance of a divide-and-conquer system is the overhead of the parallelized application running on one machine, compared to the sequential version of the same application. The sequential version is obtained by filtering the keywords `satin`, `spawn`, and `sync` out of the parallelized program. This results in a sequential program with correct semantics, as Satin does not specify whether call-by-value or call-by-reference will be used, while sequential Java always uses call-by-reference. This approach yields efficient sequential programs, because the Satin applications use optimized sequential algorithms after the recursion threshold is reached.

The difference in run times between the sequential and parallelized programs is caused by the overhead of creation, enqueuing and dequeuing of the invocation record, and the construction of the stack frame for the Java method. Fibonacci (see Figure 1) gives an indication of the worst-case overhead, because it is very fine grained. The Satin version of Fibonacci run 7.25 times slower than the sequential version. This overhead is substantially lower than that of Atlas (61.5), and somewhat higher than Cilk's overhead (3.6). These overhead factors can be reduced at the application level by introducing threshold values for spawning large jobs only. In practice, Satin's run time overhead is below 5% for realistic ap-

plications [23].

Communication in Satin is implemented by using code from the Manta RMI system [17]. On the Myrinet network, the round-trip time for a spawned method invocation (or RMI) without parameters is 37 microseconds. The maximum throughput is 51.7 MByte/s, when the parameter to the spawned method invocation (or RMI) is a large integer array.

4.2 Wide-area Measurements

Our measurements show that for all applications and load-balancing algorithms the total send overhead, such as parameter marshalling and buffer copying, is at most 1% of the run time. Therefore, we will ignore the send overhead in the remainder of this paper. The speedups of the twelve applications, relative to the sequential program (without `spawn` and `sync`), are shown in Figure 4, for all five load-balancing algorithms we described. The first (white) bar in each graph is the speedup on a single cluster of 64 nodes (i.e., without wide-area links). The following four bars indicate runs on four clusters of 16 nodes each, with different wide-area bandwidths and latencies. The results for RP contain a number of small anomalies, where speedups increase as wide-area speed is decreased. These anomalies are due to the nondeterministic behavior of the algorithm.

Matrix multiplication is the only application that does not perform well. Due to memory constraints, we cannot make the problem large enough to obtain good performance. The problem size we use is 1536×1536, which takes only 19.8 seconds on one cluster of 64 CPUs. Even with this problem size, the program sends much data (71 MByte/second with RS in one cluster of 64 nodes). The application already has scaling problems on one cluster: the speedup on 64 nodes is only 25. On four clusters, CLS performs best for matrix multiplication, but even then, the speedup is only 11 with a latency of 10 milliseconds and a throughput of 1000 KByte/s. On one cluster of 16 nodes, the speedup is 9.02, so adding three extra clusters only slightly improves performance. Therefore, we conclude that the divide-and-conquer version of Matrix multiplication is not very well suited for wide-area systems.

4.2.1 Random Stealing (RS)

The performance of random stealing (RS) decreases considerably when wide-area latencies and bandwidths are introduced, because, on four clusters, 75% of all steal messages are now sent to remote clusters. This can be seen in Figure 5: in most cases, RS sends more wide-area messages than all other algorithms. Figure 4 shows that nine out of twelve applications are almost solely latency bound. *Matrix multiplication*, *nqueens* and *raytracer* are also sensitive to bandwidth.

We collected statistics, which show that the speedups of the nine latency-bound applications can be completely explained with the number of wide-area messages sent. We illustrate this with an example: with a latency of 100 milliseconds and a bandwidth of 100 KBytes/s, *adaptive integration* sends in total 13051 synchronous steal request messages over the wide-area links. Wide-area messages are also sent to return results of stolen jobs to the victim nodes. Return messages are asynchronous, so they do not stall the sender. In total, 1993 inter-cluster steal requests were successful, and thus 1993 return messages are required. The total number of wide-area messages is $13051 \times 2 + 1993 = 28095$ (steal attempts count twice, because there is both a request and a reply). The run time of the application is 112.3 seconds, thus, as is shown in Figure 5, $28095/112.3 = 250.2$ wide-area messages per second are sent. The run time can be explained using the wide-area steal messages: in total, the system has to wait 2610.2 seconds for steal reply messages (13051 times a round-trip latency of 0.2 seconds). Per node, this is on average 40.8 seconds (2610.2 / 64). If we subtract the waiting time from the run time we get $112.3 - 40.8 = 71.5$ seconds. The run time of *adaptive integration* on a single cluster is 71.8 seconds, so the difference between the two run times is exactly the waiting time that is caused by the wide-area steal requests.

4.2.2 Random Pushing (RP)

Random pushing (RP) shows varying performance. The results shown in Figure 4 use threshold values for the queue length that were manually optimized for each application separately. For two applications (*Fibonacci with spawn threshold* and *n choose k*), RP outperforms random stealing with high wide-area latencies, which is due to its asynchronous communication behavior. For the other applications, however, RP performs worse than RS. Moreover, our results indicate that a single threshold value (specifying which jobs are executed locally, and which ones are pushed away) is not optimal for all cluster configurations. Finding the optimal values for different applications and cluster configurations is a tedious task.

With random pushing, nodes with empty queues are idle, and wait until they receive a pushed job. We found that, for all applications, the idle times are quite large, and also have a large deviation from the mean idle time (e.g., some nodes are hardly idle, others are idle for many seconds). The idle times completely account for the high run times we measured. We conclude that the bad performance of random pushing is caused by severe load imbalance.

4.2.3 Cluster-aware Hierarchical Stealing (CHS)

The speedup graphs in Figure 4 show that cluster-aware hierarchical stealing already performs suboptimal in the single-cluster case. The statistics we gathered show that the cause is that CHS sends many more messages than RS. This is due to the high locality in the algorithm. This seems counterintuitive, but our statistics show that CHS transfers much more (small) jobs than the other algorithms. This is inherent to the algorithm, because all work in a subtree is always completed before messages are sent to the parent of the subtree. This causes many transfers of small jobs within the subtrees. For example, with the *set covering* problem on one cluster of 64 nodes, CHS transfers 40443 jobs, while RS only transfers 7734 jobs (about 19 % of CHS).

Another indication of this problem is that *nqueens* and the *traveling salesperson problem* are both slower on one large cluster than on 4 clusters with low latencies. On a single cluster, CHS organizes the nodes in one large tree, while four smaller trees are used in the multi-cluster case, decreasing the amount of locality in the algorithm. In short, the locality of the algorithm causes very fine grained load balancing. RS does not suffer from this problem, because it has no concept of locality.

With multiple clusters, CHS sends fewer wide-area messages per second than all other algorithms, as can be seen in Figure 5. However, this comes at the cost of idle time, which is visible in the speedup graphs (see Figure 4): for all applications, RS outperforms CHS. The problem is that CHS waits until the entire cluster is idle before wide-area steal attempts are done. This minimizes wide-area traffic, but it also means that all nodes in the cluster must wait for the wide-area round-trip time, as only the cluster root initiates wide-area steal attempts. With RS, any node can initiate a wide-area steal attempt, and only the thief node has to wait for the round-trip time.

4.2.4 Cluster-aware Load-based Stealing (CLS)

As can be seen in Figure 5, CLS sends more messages over the wide-area network than CHS, but performs much better (see Fig-

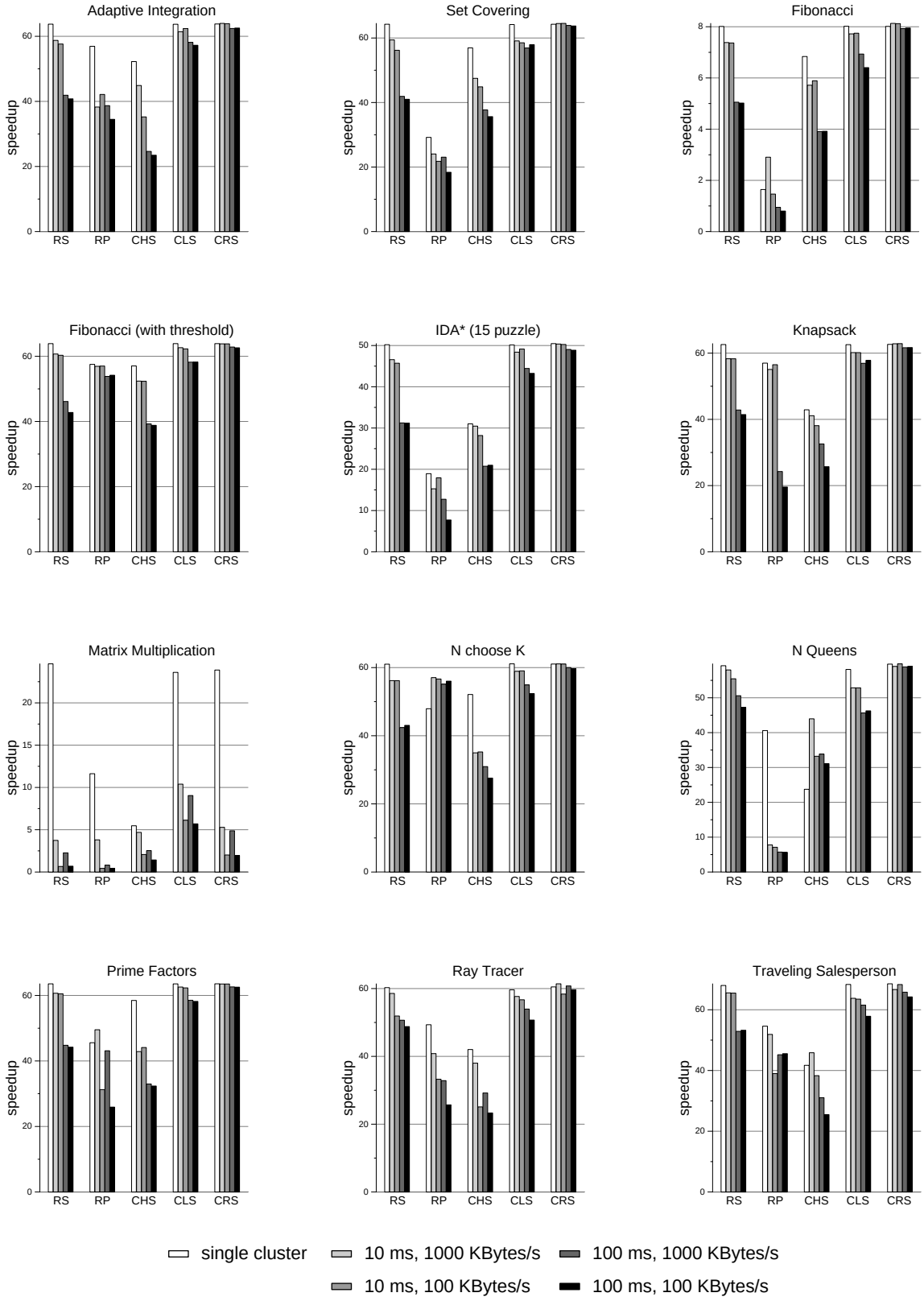

Figure 4: Speedups of 12 applications on 64 CPUs with 5 load balancing algorithms and different wide-area latencies and bandwidths.

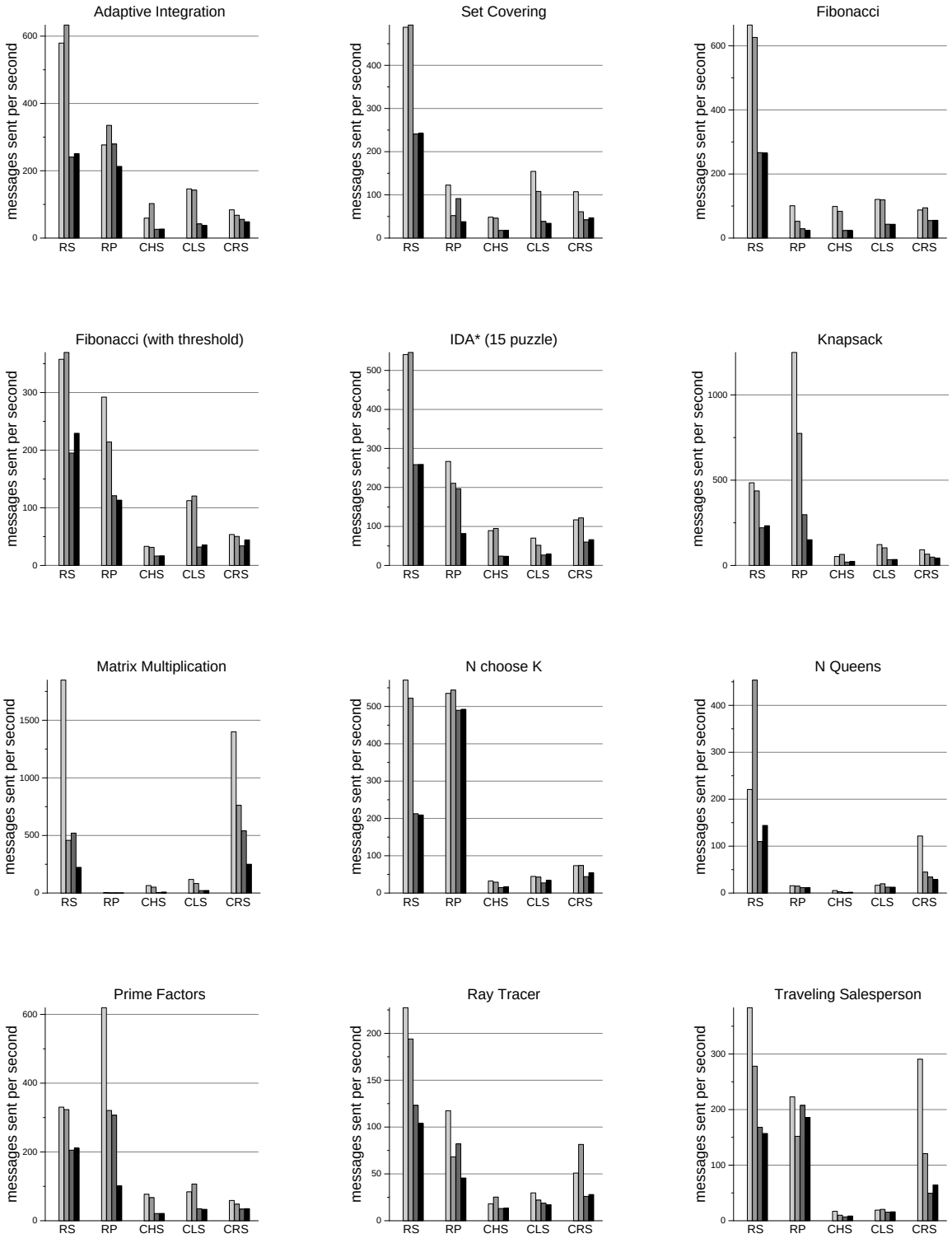

Figure 5: Total number of inter-cluster messages sent per second for 12 applications on 64 CPUs with 5 load balancing algorithms and different wide-area latencies and bandwidths.

ure 4): CLS asynchronously prefetches work from remote clusters when the cluster load drops below the threshold. Also, it does not suffer from the locality problem as does CHS, because within a cluster random stealing is used. We ran all applications with load threshold values ranging from 1 to 64, but found that no single threshold value works best for all combinations of application, bandwidth and latency. Figures 4 and 5 show the results with the best cluster-load threshold for each application.

With high round-trip times, CLS sends fewer wide-area messages than CRS (see Figure 5), but it performs worse. However, with both algorithms, all inter-cluster communication is related to prefetching, and the load balancing within the clusters is the same. Also, both algorithms use the same location policy: they choose a random victim in a random remote cluster to prefetch work from. We can conclude that the CLS coordinator can not prefetch sufficient work for its worker CPUs; due to the high wide-area round-trip times, the stealing frequency gets too low to acquire enough jobs.

4.2.5 Cluster-aware Random Stealing (CRS)

Our novel load balancing algorithm, cluster-aware random stealing (CRS), performs best with all bandwidth/latency combinations, for all applications except *matrix multiplication*. Figure 4 shows that CRS is almost completely insensitive to bandwidth and latency, although it sends more wide-area messages than CHS, and also more than CLS with high WAN latencies. CRS achieves superior performance because it neither has the locality problem of CHS, nor the prefetch limitation of CLS. With CRS, all nodes prefetch work over wide-area links, albeit only one job at a time per node. Statistics show that CRS prefetches up to 2.8 times the number of jobs prefetched by CLS. Moreover, CRS does not require any parameter tuning (such as the load threshold in CLS).

The results show that for these eleven applications, with four clusters, CRS has only 4 % run-time overhead compared to a single, large Myrinet cluster with the same number of CPUs, even with a wide-area bandwidth of 100 KBytes/s and latency of 100 millisecond.

5. RELATED WORK

We discussed load balancing in Satin, a divide-and-conquer extension of Java. Satin is designed for wide-area systems, without shared memory. Many divide-and-conquer systems are based on the C language. Among them, Cilk [7] only supports shared-memory machines, CilkNOW [6] and DCPAR [11] run on local-area, distributed-memory systems, but do not support shared data. SilkRoad [20] is a version of Cilk for distributed memory that uses a software DSM to provide shared memory to the programmer, targeting at small-scale, local-area systems. Alice [9] and Flagship [25] offer specific hardware solutions for parallel divide-and-conquer programs. Satin is purely software based, and neither requires nor provides a single address space. Instead, it uses Manta's object-replication mechanism to provide shared objects [15].

Mohr et al. [18] describe the importance of avoiding thread creation in the common, local case (lazy task creation). Targeting distributed memory adds the problem of copying the parameters of parallel invocations (marshalling). Satin builds on the ideas of lazy task creation, and avoids both the starting of threads and the copying of parameter data by choosing a suitable parameter passing mechanism [23].

A compiler-based approach is also taken by Javar [5]. In this system, the programmer uses annotations to indicate divide-and-conquer and other forms of parallelism. The compiler then generates multi-threaded Java code, that runs on any JVM. Therefore, Javar programs run only on shared memory machines and DSM systems, whereas Satin programs run on wide-area systems with distributed memory. Java threads impose a large overhead, which is why Satin does not use threads at all, but uses light-weight invocation records instead. The Java classes presented in [14] can also be used for divide-and-conquer algorithms. However, they are restricted to shared-memory 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 2.0 [19] 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 2.0 are designed for wide-area systems. While Satin is targeted at efficiency, Atlas and Javelin are designed with heterogeneity and fault tolerance in mind, and both use a hierarchical scheduling algorithm. We found that this is inefficient for fine-grained applications and that *Cluster-aware Random Stealing* performs better. Based on the information given in [19], we were not able to evaluate the performance of Javelin 2.0. Unfortunately, performance numbers for Atlas are not available at all.

Backschat et al. [1] describe a form of hierarchical scheduling for a system called Dynasty, targeting at large workstation networks. The algorithm is based on economic principles, and requires intensive parameter tuning. However, the best-performing wide-area load-balancing method in Satin is not hierarchical within clusters, and requires no parameter tuning. Hence, it is much simpler to implement and it achieves better performance.

Eager et al. [10] show that, for lightly loaded systems, sender-initiated load sharing (work pushing) performs better than receiver-initiated load sharing (work stealing). However, they assume zero latency and infinite bandwidth, which is inappropriate for wide-area systems. Furthermore, they describe a form of work pushing, where the sender first polls the target nodes, until a node is found that is not overloaded. Only after this node is found, the work is transferred. In a wide-area setting, this would imply that multiple wide-area round-trip times may be needed before work may be transferred. We use a different form of work pushing, where the work is pushed to a randomly-selected remote node, without further negotiations. If the receiving node is overloaded as well, it will immediately push the work elsewhere. The advantage of this scheme is that it is completely asynchronous.

6. CONCLUSIONS

We have described our experience with five load balancing algorithms for parallel divide-and-conquer applications on hierarchical wide-area systems. Our experimentation platform is Satin, which builds on the Manta high-performance Java system. Using 12 applications, we have shown that the traditional load balancing algorithm used with shared-memory systems and workstation networks, random stealing, achieves suboptimal results in a wide-area setting.

We have demonstrated that hierarchical stealing, proposed in the literature for load balancing in wide-area systems, performs even worse than random stealing for our applications, because its load balancing is too fine grained, and because it forces all nodes of a cluster to wait while work is transfered across the wide-area network. Two other alternatives, random pushing and load-based stealing, only work well after careful, application-specific tuning of their respective threshold values.

We introduced a novel load-balancing algorithm, called *Cluster-aware Random Stealing*. With this algorithm, 11 out of 12 applications are only 4% slower on four small clusters, compared to a large single-cluster system, even with high WAN latencies and low WAN bandwidths. These strong results suggest that divide-and-conquer

parallelism is a useful model for writing distributed supercomputing applications on hierarchical wide-area systems.

Acknowledgments

This work is supported in part by a USF grant from the Vrije Universiteit. The wide-area DAS system is an initiative of the Advanced School for Computing and Imaging (ASCI). We thank John Romein, Grégory Mounié, and Martijn Bot for their helpful comments on a previous version of this manuscript. We thank Ronald Veldema, Jason Maassen, Ceriel Jacobs, and Rutger Hofman for their work on the Manta system. Kees Verstoep and John Romein are keeping the DAS in good shape.

7. REFERENCES

[1] M. Backschat, A. Pfaffinger, and C. Zenger. Economic Based Dynamic Load Distribution in Large Workstation Networks. In *Euro-Par'96*, number 1124 in Lecture Notes in Computer Science, pages 631–634. Springer, 1996.

[2] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

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

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

[5] A. Bik, J. Villacis, and D. Gannon. javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, November 1997.

[6] R. 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, California, 1997.

[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*, pages 207–216, Santa Barbara, California, July 1995.

[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)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[9] J. Darlington. Alice: a multi-processor reduction machine for the parallel evaluation of applicative languages. In *1st Conference on Functional Programming Languages and Computer Architecture*, pages 65–76, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6(1):53–68, Mar 1986.

[11] B. Freisleben and T. Kielmann. Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.

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

[13] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Trans. Software Eng.*, 17(7):725–730, July 1991.

[14] D. Lea. A java fork/join framework. In *ACM Java Grande 2000 Conference*, San Francisco, California, June 2000.

[15] J. Maassen, T. Kielmann, and H. E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 2001.

[16] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[17] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.

[18] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[19] M. O. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-based parallel computing on the internet. In *Proc. Euro-Par 2000*, pages 1231–1238, Munich, Germany, August 2000.

[20] 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)*, pages 243–249, Chemnitz, Saxony, Germany, Nov. 2000.

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

[22] N. G. Shivaratri, P. Krueger, and M. Ginghal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 92.

[23] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Euro-PAR 2000*, number 1900 in Lecture Notes in Computer Science, pages 690–699, Munich, Germany, Aug. 2000. Springer.

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

[25] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: A parallel architecture for declarative programming. In *15th IEEE/ACM Symp. on Computer Architecture*, pages 124–130, Honolulu, Hawaii, 1988.

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