Satin: a High-Level and Efficient Grid Programming Model

ROB V. VAN NIEUWPOORT, GOSIA WRZESIŃSKA, CERIEL J.H. JACOBS and HENRI E. BAL. Vrije Universiteit Amsterdam

Computational grids have an enormous potential to provide compute power. However, this power remains largely unexploited today for most applications, except trivially parallel programs. Developing parallel grid applications simply is too difficult. Grids introduce several problems not encountered before, mainly due to the highly heterogeneous and dynamic computing and networking environment. Furthermore, failures occur frequently, and resources may be claimed by higher priority jobs at any time.

In this paper, we solve these problems for an important class of applications: divide-and-conquer. We introduce a system called Satin that simplifies the development of parallel grid applications by providing a rich high-level programming model that completely hides communication. All grid issues are transparently handled in the run time system, not by the programmer. Satin's programming model is based on Java, features spawn-sync primitives and shared objects, and uses asynchronous exceptions and an abort mechanism to support speculative parallelism.

To allow an efficient implementation, Satin consistently exploits the idea that grids are hierarchically structured. Dynamic load-balancing is done with a novel cluster-aware scheduling algorithm that hides the long wide-area latencies by overlapping them with useful local work. Satin's shared object model lets the application define the consistency model it needs. If an application needs only loose consistency, it does not have to pay high performance penalties for wide-area communication and synchronization.

We demonstrate how grid problems such as resource changes and failures can be handled transparently and efficiently. Finally, we show that adaptivity is important in grids. Satin can increase performance considerably by adding and removing compute resources automatically, based on the application's requirements and the utilization of the machines and networks in the grid.

Using an extensive evaluation on real grids with up to 960 cores, we demonstrate that it is possible to provide a simple high-level programming model for divide-and-conquer applications, while achieving excellent performance on grids. At the same time, we show that the divide-and-conquer model scales better on large systems than the master-worker approach, since it has no single central bottleneck.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Languages**]: Concurrent, distributed, and parallel languages

General Terms: Languages, Performance

Additional Key Words and Phrases: grid computing, divide-and-conquer. programming model

Author's address: Rob V. van Nieuwpoort, Henri E. Bal and Ceriel J.H. Jacobs, Faculteit Exacte Wetenschappen, Vrije Universiteit, De Boelelaan 1081, 1081 HV, Amsterdam, The Netherlands, rob@cs.vu.nl, bal@cs.vu.nl, ceriel@cs.vu.nl; Gosia Wrzesińska, Pointlogic Development, P.O. Box 29147, 3001 GC Rotterdam, The Netherlands, wrzesinska@pointlogic.com.

1. INTRODUCTION

Grids offer the potential for unprecedented large-scale computing, but unfortunately writing efficient parallel programs for grids is notoriously difficult. To begin with, finding and allocating grid resources, transferring the program and its data, and launching the application is complicated and only partially standardized. Even if all these deployment problems are taken for granted, however, grids differ in several more fundamental ways from traditional parallel machines. Grids are far more heterogeneous, they have relatively slow wide-area interconnection networks, their performance characteristics vary between resources and change over time, and resource failures are more likely [Foster and Kesselman 2003]. Therefore, most grid applications use only very simple forms of parallelism. Grids typically are used for running independent jobs (high-throughput computing), for trivially parallel master/worker programs or they are used as a shared batch queue that runs each application on a single site at a time.

Based on several successful earlier projects with Grid algorithms [Plaat et al. 1999; Romein et al. 2002], programming systems [Kielmann et al. 1999; Maassen et al. 2001], and hardware infrastructures [Cappello and Bal 2007; Verstoep et al. 2008], we have arrived at the conclusion that large-scale grid computing has a much broader applicability. Many (although certainly not all) parallel applications can exploit the hierarchical structure that grids typically have to do locality optimizations, making the application less sensitive to high latencies and variations in resource performance. Most existing parallel programming models, however, are a poor match to grids, and easily result in programs that are sensitive to these problems and that are therefore complex to write.

In this paper, we describe a programming model that is specifically designed for grids. The model addresses the fundamental problems with grids in such a way that (1) it hides much of the complexity of a grid, (2) it can be implemented efficiently on a grid, and (3) it can be used for a reasonably broad range of applications. Finding the right balance between these three goals is difficult. We will advocate that an *extended divide-and-conquer model* provides such a balance. Since divide-and-conquer is a hierarchical computational model, it maps well onto a hierarchical grid. Also, the programming model is much easier to use than message passing and other models. We will show that the divide-and-conquer model can be extended with primitives for weakly-coherent shared data and speculative parallelism, making it more generally usable. Also, we will show that the model can be implemented efficiently on a heterogeneous and dynamic grid and that it can adapt itself transparently to resource changes and failures.

We have implemented this model in a system called *Satin*, which has been used for applications such as N-body simulations, SAT-solvers, VLSI-routing, grammar induction, and many others, and has been run efficiently on heterogeneous grids with up to 1000 processors. In recent experiments we were also able to successfully execute applications on clouds (e.g., the Amazon Elastic Compute Cloud).

The result is a system that can be used for a reasonably broad range of parallel applications, although we of course certainly do not claim that all applications are suitable for this model. For most applications that do fit the model, we obtain high speedups on real grids, despite the high (and varying) wide-area latencies.

Moreover, programming Satin applications merely requires the code to be written or restructured into a divide-and-conquer form. The programmer is not concerned with locking, multithreading, communication mechanics, load balancing, fault tolerance, and adaptivity. Satin thus is vastly easier to use than message passing languages.

In Section 2 we first give an overview of Satin, focussing on its rationale. In Sections 3 and section 4, we discuss the design and implementation in more detail. Section 5 presents several parallel applications and performance results. Section 6 discusses related work and Section 7 provides our conclusions.

2. DESIGN OF SATIN

There are numerous reasons why grids are complicated to program [Foster and Kesselman 2003]. We focus on programming models for *parallel* applications on grids and only address problems related to this topic. Many other issues exist related to deployment (e.g., resource allocation, security), network configuration (firewalls), data-intensive applications (I/O, streaming), and so on, which are outside the scope of this paper. From a parallel programming perspective, grids have the following fundamental differences with traditional supercomputers or clusters:

—Grids are strongly *hierarchical* and consist of resources (clusters or supercomputers) with fast local interconnects but relatively slow wide-area links. In contrast to the Local Area Networks (LANs), the Wide Area Networks (WANs) necessarily have a *high latency*, due to the well-known speed-of-light argument, easily causing three or more orders of magnitude difference between LAN and WAN latencies. The WAN bandwidth may or may not be less than the LAN bandwidth, depending on the technologies used [Verstoep et al. 2008].

—Grids are *heterogeneous* and use different types of CPUs, networks, operating systems, etc.

—Grids are *dynamic*: the network and CPU performance vary over time due to changes in loads. The number of resources available may increase or decrease during the application. The sheer size and complexity of grids make crashes more likely [Foster and Kesselman 2003].

The design of Satin is driven by these properties. Some design decisions are visible directly in the programming model, while others are more subtle and can only be understood by studying the implementation (see Table I). Here, we therefore give the overall design, focusing on the rationale of Satin. More details are given in Section 3 (programming model) and Section 4 (implementation).

The hierarchical structure of grids and the high WAN latencies are not only a problem but also an opportunity, as hierarchical systems are excellent candidates for *locality optimizations*. The programming model should therefore be designed to take locality into account. As a first step in the design of Satin, we therefore chose a *divide-and-conquer* (D&C) programming model. Divide-and-conquer itself is a hierarchical computational model, as it recursively splits up computations into subcomputations. This computational model thus maps well onto hierarchical grids.

Heterogeneity poses two problems. First, there is a major software engineering problem in running an application at the same time on various types of operating systems, CPUs, and networks. We address this problem using a *Java-centric* approach [Wollrath et al. 1997], relying on Java's "write-once run everywhere" portability. All our software thus is written in Java, including not only the applications but also the runtime systems and communication protocols. Our entire software stack can thus run out of the box on any machine providing a standard Java Virtual Machine (JVM). The challenge here is how to obtain high performance for such a Java-centric approach. In a nutshell, we use bytecode rewriting techniques to optimize programs in a completely portable way. We will discuss the details extensively in later sections. The second aspect of heterogeneity is performance related: how to deal with differences in CPU and network speeds. This problem is

Aspect	Programming model	Sec.	Implementation	Sec.
Hierarchical, heterogeneous	Java-centric D&C	3.1	RTS + bytecode rewriter	4.1
Data sharing	Shared objects	3.2	Replication	4.2
Speculative parallelism	Inlet/abort	3.3	Exception handling	4.3
Dynamic	No CPU info	3.4	Malleability/fault tolerance	4.4
			Self-adaptivity	4.5

Table I. Overview of the design and implementation of Satin.

a special (static) case of the third grid property, discussed next.

The dynamic nature of grids requires programs to deal with performance differences, even over time. Again, divide-and-conquer is an excellent match here, although the reasons are subtle. Briefly, divide-and-conquer parallelism results in a large number of small subcomputations that can be scheduled dynamically over a grid, giving more tasks to processors or clusters that (currently) are faster. It is possible to do this scheduling efficiently even over slow WANs. Satin can even add and delete processors (or entire clusters) dynamically, so Satin applications are malleable [Kale et al. 2002], which is useful when reservation systems dynamically withdraw or add resources. Satin applications also can survive resource crashes, and thus also are fault-tolerant. Finally, Satin can use its malleability mechanism to decide for itself that it is more efficient to delete certain resources (CPUs or entire clusters) that are overloaded or have overloaded links; Satin can also actively request new resources if it finds there currently is enough parallelism for it. So, Satin is self-adaptive.

Unfortunately, pure divide-and-conquer parallelism has limited applicability and is unsuitable for many applications. Foremost, it allows no data sharing between tasks, except by passing input and result parameters between parent and child tasks. We have therefore extended the model with a well-designed form of *shared objects* that can be implemented efficiently on a grid. Again, the main obstacle here is the high latency of grids, which makes efficient strongly-consistent shared memory fundamentally impossible. Instead, we have developed a weak consistency model under control of the programmer, enabling a broader range of applications.

Another shortcoming of pure divide-and-conquer parallelism is its lack of support for expressing speculative parallelism, which is needed in many search applications. We have therefore designed an inlet/abort mechanism for Satin that can prune speculative computations. Again, this mechanism was designed to work efficiently on high-latency grids, by using asynchronous events.

An overview of Satin is given in Table I. The programming model is discussed in Section 3, including the basic divide-and-conquer primitives, shared objects, and speculative parallelism. We also explain that transparent malleability, fault tolerance, and adaptivity can be made possible by hiding any machine information from the programmer. The implementation of Satin is discussed in Section 4. We first look at the basic implementation, including the runtime system (RTS) and bytecode rewriter. Next, we discuss how shared objects and inlet/abort are implemented. Subsequently, we explain how Satin transparently deals with malleability and fault-tolerance and how Satin applications can be made self-adaptive.

3. THE SATIN PROGRAMMING MODEL

In this section, we describe the Satin programming model in more detail. Satin exploits Java's "write once, run everywhere" property to deal with the heterogeneous nature of the grid. Satin integrates cleanly into Java without any language extensions. The Satin model can express master-worker and divide-and-conquer applications, and supports shared data and speculative parallelism. The model is designed in such a way that load-balancing, adaptivity to changing conditions on the grid, malleability and fault-tolerance can be implemented transparently. The actual implementation of these features are the topic of Section 4.

3.1 Divide-and-Conquer on the Grid

Our vision is that grids are, and will be hierarchical in nature. A grid does not consist of a collection of independent PCs, but of entire clusters. Often, several clusters are combined into a virtual organization (e.g, DAS-3, Grid'5000 [Cappello and Bal 2007], InTrigger [Kitsuregawa 2007]). Clusters and/or virtual organizations are then connected to form a grid. In addition, PCs are also becoming more and more hierarchical, as they often have multiple processors with multiple cores per processor, resulting in even more levels of hierarchy.

Satin's programming model was designed to exploit this hierarchical structure. The divide-and-conquer programming model is inherently hierarchical, as work is repeatedly split up into smaller subproblems. We believe that a divide-and-conquer model can be implemented more efficiently on a grid than non-hierarchical programming models, such as master/worker or GridRPC [Seymour et al. 2002]. Moreover, divide-and-conquer has no single central point that limits scalability, which the master/worker paradigm does have. This is important, because grids are typically very large. It is possible to write master/worker applications in Satin and we regard this paradigm as a subclass of divide-and-conquer that uses only one level of division of work.

Satin's programming model is an extension of the single-threaded Java model. Satin programmers thus need not use Java's multithreading and synchronization constructs or Java's Remote Method Invocation mechanism, but can use the much simpler divide-and-conquer primitives described below.[1]

Parallel divide-and-conquer systems have at least two primitives: one to spawn work, and one to wait until the spawned work is finished. Cilk [Blumofe et al. 1995] introduces new keywords into the C language to implement these primitives. Satin exploits Java's standard mechanisms of inheritance and marker interfaces (e.g., java.io.Serializable) to extend Java with divide-and-conquer primitives. A *spawn* operation is a special form of a method invocation. Methods that can be spawned are defined by tagging them with a special marker interface. We will call such methods Satin methods. The marker interfaces are recognized by the Satin bytecode rewriter, which generates special code for them (as explained in Section 4).

An invocation of a Satin method is called a *spawned method invocation*. With a spawn operation, conceptually a new thread is started to run the method; the imple-

[1]While the use of Java threads is not necessary with Satin, it is possible to combine Satin programs with Java threads. This can be useful for user interfaces, for instance. Furthermore, it is possible to use RMI [Waldo 1998], or any other communication mechanism in combination with Satin.

mentation of Satin, however, eliminates thread creation altogether. The spawned method will run concurrently with the method that executed the spawn. The *sync* operation waits until all spawned calls in this method invocation are finished. The return values of spawned method invocations are undefined until a sync is reached. The assignment of the return values is done between the spawn and sync.

The programmer can create an interface which extends the marker interface called *satin.Spawnable*, and define the signatures of methods that must be spawned. A class that spawns work must extend the special class *satin.SatinObject* to inherit the *sync* method. This mechanism closely resembles standard Java RMI [Waldo 1998].

We will illustrate the spawn and sync operations as well as most other Satin primitives (discussed later) using a single realistic example. Barnes-Hut is a well-known parallel N-body simulation that uses an oct-tree to store information about the simulated bodies. During every iteration of the simulation, the algorithm repeatedly traverses the tree to compute the new positions, velocities, etc. of the bodies. Barnes-Hut is known to be a challenging application, as it suffers both from load-balancing and communication overhead.

The use of spawn and sync is illustrated on Lines 34–38 of Figure 1. The method *computeForces* computes all forces of a subspace *s* on a collection of bodies. Initially, the subspace is the entire space, as shown by the invocation on Line 48 of *main*. A call of *computeForces* recursively invokes this method for each of the subspaces (eight in 3D space), as shown on Line 36. Since the space is represented by a tree, a subspace is accessed by traversing the tree (i.e., following the *child* nodes). The method *computeForces* is tagged with a special marker interface *satin.Spawnable* (Line 20) so this method will be spawned. After all spawns have been done, the method calls *sync*, causing it to wait until all subtasks have been completed.

As explained in Section 4.1.1, the programmer can assume neither call-by-value nor call-by-reference semantics for spawnable methods. The reason is that most spawnable methods (typically 99%) are executed locally and are implemented using call-by-reference; only methods that are actually executed remotely use call-by-value: their parameters are copied using Java's serialization mechanism, just as with standard RMI. Copying the parameters of local invocations would be prohibitively expensive.

3.2 Shared Objects

In a pure divide-and-conquer model, the only way of sharing data between tasks is by passing parameters and returning results. Therefore, a task can share data with its subtasks and the other way around, but the subtasks cannot share data with each other. For several applications, this lack of shared data is too restrictive, which limits the applicability of the divide-and-conquer model. Several applications can be expressed in a hierarchical computational model, but do require data sharing. Branch-and-bound algorithms, for example, typically search a tree space but need a global variable for storing the best solution found so far (the "bound"). Game tree search algorithms likewise need to share data like transposition tables. Also, some divide-and-conquer applications need to pass large constant data structures as parameters. With pure divide-and-conquer, these data structures would have to be sent over the network each time a new task is created remotely, which is

```
1  // Marker interface that defines updateBodies as a global method.
2  interface BodiesInterface extends satin.GlobalMethods {
3    void updateBodies(BodyUpdates b, int iter);
4  }
5
6  // A shared object containing the tree of bodies.
7  class Bodies extends satin.SharedObject implements BodiesInterface {
8    BodyTreeNode root;
9
10   public void updateBodies(BodyUpdates b, int iter) { // Global method.
11     root.applyUpdates(b, iter); // Update bodies in our tree.
12   }
13
14   BodyTreeNode getRoot() { // Local method.
15     return root;
16   }
17 }
18
19 // Mark the computeForces method as a spawn operation.
20 interface BHSpawns extends satin.Spawnable {
21   BodyUpdates computeForces(Subtree s, int iter, Bodies bodies);
22 }
23
24 class BarnesHut extends satin.SatinObject implements BHSpawns {
25   public boolean guard_computeForces(Subtree s, int iter, Bodies bodies) {
26     return bodies.iter + 1 == iter;
27   }
28
29   // Spawnable method. The "bodies" parameter is a shared object.
30   public BodyUpdates computeForces(Subtree s, int iter, Bodies bodies) {
31     BodyUpdates[] res = new BodyUpdates[s.nrChildren];
32     if(s.hasNoChildren) {
33       computeSequentially(s, iter, bodies.getRoot());
34     } else { // Divide the work and spawn tasks (recursion step).
35       for(int i=0; i<s.nrChildren; i++) {
36         res[i] = computeForces(s.child[i], iter, bodies); // Spawn.
37       }
38       sync(); // Wait for the spawn operation to finish.
39
40       return mergeSubresults(res); // Merge results and return.
41     }
42   }
43
44   public static void main(String[] args) {
45     BarnesHut bh = new BarnesHut();
46     Bodies bodies = new Bodies(); // Create shared object.
47     for (int iter = 0; iter < N; iter++) {
48       results = bh.computeForces(root, iter, bodies); // Spawn.
49       bh.sync(); // Wait for the spawn operation to finish.
50       bodies.update(results, iter); // Shared method invocation.
51     }
52   }
53 }
```

Fig. 1. Pseudo-code for Barnes-Hut in Satin.

highly inefficient. Satin therefore provides a form of data sharing, making the programming model more powerful by allowing tasks to share data. The model has been designed to allow an efficient implementation on a grid. In particular, Satin uses a new user-controlled, relaxed consistency model called guard consistency that allows efficient *replication* of shared data [Wrzesińska et al. 07 B].

Shared data in Satin are encapsulated in *shared objects*. Much as in the Orca language, Satin uses an update-based replication protocol, because that has been shown to be efficient for object-based languages [Bal et al. 1998]. The Satin runtime system will thus automatically replicate shared objects on processors that can access the object, so local operations can use the replica, without requiring communication. Global operations will be applied to all copies. Unlike Orca, Satin uses a relaxed consistency model. The details of object sharing are explained next.

Shared objects are passed by reference to the spawned tasks. As an example, consider again the Barnes Hut example of Figure 1. The Satin program stores the positions, velocities, etc. of the bodies in a shared object that is created on Line 46 and passed as a parameter in the invocation of *computeForces* on Line 48. The class *Bodies* is defined on Lines 6–17 and extends the special class *satin.SharedObject* to indicate its objects will be shared.

Satin requires the programmer to distinguish between *global* methods that will be applied to all replicas of a shared object and *local* methods that will only be applied to the local copy. The special interface *satin.GlobalMethods* is used to mark global methods, as shown in Lines 2–4. The method *updateBodies* appears in this marker interface, so it is a global method. On the other hand, the method *getRoot* (Lines 14–16) does not appear in the marker interface, so it is a local method. In this way, the programmer can control when replicas are updated.

Satin provides a relaxed consistency model for shared objects called *guard consistency*. The programmer can define the application consistency requirements using *guard functions*, which are associated with divide-and-conquer tasks. Conceptually, a guard function is executed before each divide-and-conquer task. A guard checks the state of the shared objects accessed by the task and has to return *true* if the objects are in a correct state, or *false* otherwise. The name of the guard function is "guard_<spawnable_function>". A guard has exactly the same parameter list as the spawned task. Therefore, it can access the shared objects used by this task, and the task parameters that depend on the state of the parent that has spawned the task. Satin iteratively evaluates the guards. As long as a guard evaluates to *false*, Satin has to make the local replica more consistent. This is done by waiting for pushed updates to arrive, or by fetching a more recent replica. With a guard, the programmer can thus ensure that the state seen by a task is consistent with the state seen by its parent.

In the Barnes Hut example of Figure 1, the method *computeForces* uses a simple guard function (called *guard_computeForces*) to make sure it received the updates belonging to the previous iteration of the algorithm (see Lines 25–27).

Satin allows replicas to become inconsistent as long as guards are satisfied: the updates are propagated to remote replicas on a best-effort basis. Satin does not guarantee that all updates will be applied on all replicas, and updates could be duplicated or be applied in different order on different replicas. This makes a

scalable and efficient implementation on a grid possible (using techniques such as unreliable multicast or gossiping). Also, processors dynamically joining and leaving the computation are supported. When a guard is not satisfied, Satin invalidates the local replica of a shared object and fetches a consistent replica from another processor.

The guard consistency model allows combinations of function shipping and data shipping. In fact, as we will explain in Section 4.2, our implementation combines these two approaches to achieve good performance on wide-area systems. We use *function shipping* to push updates to replicas, since this typically requires less data. Satin uses deep copies for the parameters to shared method calls. Thus, if the first node of a linked list is passed as a parameter, the entire list is automatically serialized and shipped to all replicas. For newly joined machines, and when a guard fails and a non-reliable multicast is used, the Satin runtime system also pulls updates in, using *data shipping*. In this case, the entire shared object including all objects that are referenced are transferred.

3.3 Speculative Parallelism and Exception Handling

Many parallel applications need a mechanism to abort useless computations. For example, many heuristic tree search algorithms speculatively execute work. When, during the computation, it becomes clear that the speculative work can never give a better solution than the results found so far, the speculative work should be aborted. Spawn and sync primitives in combination with shared objects are not sufficient to *efficiently* express algorithms with speculative parallelism, since the sync primitive waits until all spawned tasks have completed.

Satin therefore provides a new mechanism called inlets that can be used to execute code (e.g., an abort) as soon as some task has finished. An inlet must run in the context of the spawner, because it should be able to access its local variables. We therefore use Java's exception mechanism to implement inlets. Inlets are needed to express speculative parallelism, because the abort operation is only useful if it can be executed as soon as the result of speculative work becomes available. Abort operations are thus typically executed from an inlet.

Figure 2 shows an example of the use of inlets and aborts in Satin. In the example, a number of *depthFirstSearch* tasks are spawned speculatively (Line 25); as soon as one of the tasks generates a result that is better than the current pivot value, the other tasks are aborted (Line 29). The *abort* method is inherited from *satin.SatinObject*. The result of *depthFirstSearch* is returned via an exception of the type *Result*. This makes it possible to use the catch block around the spawn of *depthFirstSearch* (Lines 26–32) as an inlet. The *Result* class extends the *satin.Inlet* class provided by Satin.

Because the control flow in the inlet is not restricted (it might leave the catch block), we chose to conceptually spawn a new thread for each inlet. This thread is different from normal Satin threads (and normal Java threads), as it *shares* its local variables and parameters with the spawner thread (e.g., *score* on Lines 27–29). Satin guarantees that no two inlets of the same spawner will run at the same time, and that an inlet does not run concurrently with its spawner. This way, no locks are needed to protect access to local variables.

Our approach circumvents many of the well-known problems [Marlow et al. 2001;

```
1  class Result extends satin.Inlet {
2    int score;
3
4    Result(int score) {
5      this.score = score;
6    }
7  }
8
9  interface DepthFirstSearchInterface extends satin.Spawnable {
10   void depthFirstSearch(Node node, int pivot, int depth)
11     throws Result;
12 }
13
14 public void depthFirstSearch(Node node, int pivot, int depth)
15     throws Result {
16   if(depth == 0) { // Stop if the depth is 0.
17     throw new Result(node.evaluate());
18   }
19
20   Node[] children = node.generateChildren();
21   int score = node.score;
22
23   // Speculatively search children in parallel.
24   for(int i = 0; i<children.length; i++) {
25     try {
26       depthFirstSearch(children[i], 1-pivot, depth-1); // Spawn.
27     } catch (Result result) { // The inlet.
28       if(result.score > score) {
29         score = result.score;
30         if(score >= pivot) abort(); // Abort useless work.
31       }
32       return; // Exit the inlet, do not fall through.
33     }
34   }
35   sync();
36
37   throw new Result(score);
38 }
```

Fig. 2. *Depth first search*: an example of an inlet and abort in Satin.

Butenhof 1997] that arise when asynchronous exceptions are used. Systems that support asynchronous exceptions have to interrupt the spawner, and must support critical regions [Marlow et al. 2001; Butenhof 1997] to avoid the interruption of a thread at an inconvenient time. In Satin, inlets run in a new thread, and the spawner of the work does not have to be interrupted. By guaranteeing that inlets do not run concurrently with the spawner, critical sections are not needed. Still, our mechanism is as expressive, because the new thread can read and write the local variables and parameters of the spawner.

The inlet/abort model matches well with grid environments. The model is asynchronous and point-to-point, and it follows the hierarchical structure of the spawned task tree. Also, the abort mechanism is a *best effort* operation. Satin will do its

best to abort as much work as possible, but it might not abort all work that has become useless. Furthermore, it means that a sequentially compiled Satin program, where the abort has no effect, also has correct semantics. Moreover, the abort operation is asynchronous: the application continues while Satin tries to retract the aborted work. The abort mechanism itself also follows the hierarchical task structure, aborting parents and child tasks recursively, like they were spawned. This again matches well with the hierarchical structure of the grid. For a more detailed description of Satin's support for speculative parallelism, see [Nieuwpoort 2003].

3.4 Malleability, Fault Tolerance, and Self-Adaptivity

The Satin programming model was designed with transparent fault tolerance and malleability in mind. The most important insight is that, to be transparent, the model cannot provide a concept of machines. Machines cannot be used as a unique identifier, because machines can crash or leave. Also, it is possible to run multiple Satin instances on a single machine, for instance on SMPs or multi-core machines. Therefore, exporting the underlying machine infrastructure to the programmer means that the programmer also has to deal with faults, or at least is aware of them. Likewise, Satin does not use any form of *ranks*, as present in MPI, since they also become invalid after changes.

Instead, Satin takes care of the work distribution itself, also when machines join, leave or crash. Satin just allows a programmer to spawn work, but makes it impossible to specify *where* the work has to be spawned to. The programmer cannot influence the load balancing of the parallel computation. This makes it impossible to implement the small class of algorithms that depend on locations, such as Transposition Driven Scheduling (TDS) [Romein et al. 2002]. However, because there is no concept of location, fault-tolerance, malleability, and self-adaptivity can be implemented *completely transparently* for the programmer. This is an enormous benefit, because dealing with faults is extremely difficult and error prone in general. Satin is able to hide all this complexity in the runtime system (see Section 4.4).

3.5 Integrating Native or Legacy Code

Although our entire software stack is written in Java, applications can integrate legacy code (e.g., C++, Fortran, etc.), using the Java Native Interface (JNI) to call native library functions. This approach does reduce portability, since the native code has to be compiled for each platform. Alternatively, a separate executable program can be executed from Java. We have used both approaches with our Ibis communication platform that we used to implement Satin (see Section 4.1).

In 2008, we participated in the 1st IEEE International Scalable Computing Challenge, with a multimedia application that performs object recognition. The application itself is written in Java, using Ibis for communication, while the image analysis code is written in C++. The Java code uses the JNI to call the C++ library. Using this hybrid application, we won the first prize in the contest [Seinstra et al. 2008].

We also participated in the Data Challenge, held in conjunction with the Cluster2008 conference. Here, we used a C application that searches an astronomy image for super novae. The main application is written in Java, using Ibis for communication. We simply execute the separate legacy application from Java to do the analysis. Again, we won the first prize in this challenge [Maassen et al. 2008].

4. THE SATIN IMPLEMENTATION

We first describe the basic implementation of Satin. Next, we discuss the implementation of shared objects and the inlet/abort mechanism. Finally, we explain how we deal with the dynamic aspects of grids, including malleability and fault tolerance and adaptivity.

4.1 Basic Implementation of Satin

Figure 3 gives an overview of how Satin programs are compiled and deployed. Because Satin uses no language extensions but only marker interfaces, a Satin program can be compiled with any Java compiler (e.g., *javac*). The output is a collection of class files which contain bytecode that can be executed sequentially on any JVM. This is useful for testing and benchmarking purposes. To run the code in parallel, an additional compilation step is done on the generated bytecode. The Satin compiler, *satinc*, rewrites the sequential bytecode, inserting code to implement the spawn and sync operations. The rewritten bytecode can be deployed on the grid with any grid middleware system, such as Globus [Foster 2006] or our Java Grid Application Toolkit [Nieuwpoort et al. 2007]. For communication, Satin uses the Ibis communication layer. Below, we will first explain the code generation and the Satin runtime system, including the load balancing mechanism. Next, we discuss the Ibis communication layer.

4.1.1 *Spawn and Sync.* When a program executes a spawned method invocation, Satin redirects the method call to a stub. This stub creates an *invocation record*, which describes the method to be invoked, the parameters that are passed to the method, and a handle to where the method's return value has to be stored.

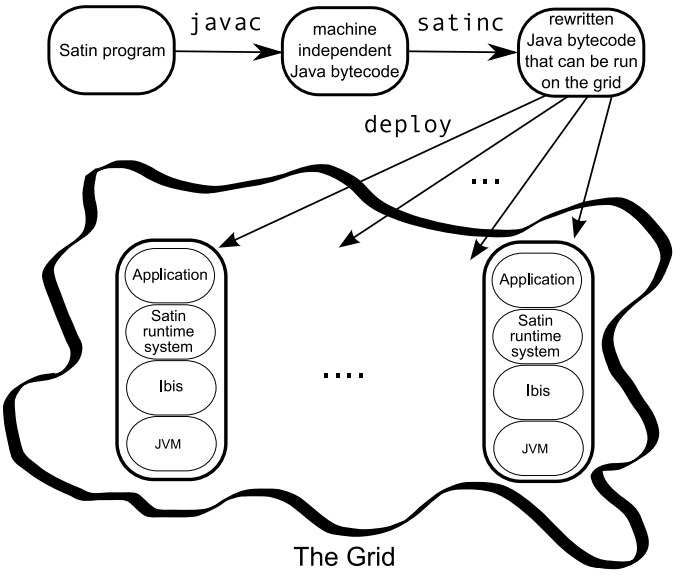

Fig. 3. Compiling and running Satin programs.

The invocation records are method specific, and are generated by the Satin bytecode rewriter. This way, no runtime type inspection is required. Given an invocation record, the original call can be executed by pushing the values of the parameters (which were stored in the record) onto the stack, and by calling the Java method, whether it concerns a local or a stolen (obtained from a remote processor) job. The runtime system also puts a unique stamp on each job, which is later used to identify it. The stamp of the parent is stored in the invocation record as well. The latter is used to implement fault tolerance and the abort primitive.

The large majority of jobs that are spawned will not be transfered, but will just run on the machine that spawned the work. For example, in almost all applications we have studied so far, at most 1 out of 150 jobs is transfered to a remote machine. Therefore, it is important to reduce the overhead that the Satin runtime system generates for such jobs as much as possible. The key problem here is that the decision whether to copy the parameters must be made at the moment the work is either executed or stolen, not when the work is generated. Therefore, Satin's runtime system implements *serialization on demand*. For primitive types, the value of the parameter is copied into the invocation record. For reference types (objects, arrays, interfaces), only a reference is stored in the record. The parameters are serialized (i.e., converted to bytes) only when the work is actually stolen. In the local case, no serialization is used, which is of critical importance for the overall performance. Moreover, the Satin implementation avoids thread creation altogether which has a large positive impact on performance. The large overhead for creating threads or building task descriptors (copying parameters) was also recognized in the lazy task creation work by Mohr et al. [Mohr et al. 1990] and by Cilk [Blumofe et al. 1995].

Methods that spawn work are rewritten by the Satin bytecode rewriter to keep a list of invocation records that represents the outstanding work that was spawned but is not yet finished. When an invocation record is created, it is added to this *outstandingSpawnsList*. Next, the generated code calls the Satin runtime system to put the invocation record in the *work queue*. Satin maintains one work queue per JVM. The Satin runtime system implements a load balancing algorithm that transfers jobs between the work queues, as discussed below.

The sync operation is rewritten to a call to the Satin runtime system which executes work from the work queue until the *outstandingSpawnsList* of the method that executed the sync becomes empty. After the sync operation, code is inserted that traverses the *outstandingSpawnsList* and assigns the results of the spawned methods out of the invocation records to their destinations.

4.1.2 *Load Balancing.* In spawn operations, the invocation records are stored at the *head* of a double-ended job queue. During the sync operation, Satin starts to execute work, also from the *head* of the queue. When a node runs out of work, it will start stealing work from other nodes. Idle nodes will poll remote queues for jobs, at the *tail* of the queue. The tail of the queue contains the jobs that are spawned first. These are jobs that are more likely to be large-grain, because they are higher up the divide-and-conquer tree. In this way, large-grain jobs are stolen, reducing communication overhead [Frigo et al. 1998].

Random Stealing (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 [Blumofe and Leiserson 1994]. 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. For homogeneous (single cluster) systems, RS is known to achieve optimal load balancing. It is proven to be optimal in space, time and communication [Blumofe and Leiserson 1994]. On wide-area systems, however, this is not the case. With C clusters, on average $(C-1)/C \times 100\%$ of all steal requests (e.g., already 75% on 4 clusters) will go to nodes in remote clusters, causing significant wide-area communication overheads.

Satin uses a novel load balancing algorithm, called *Cluster-aware Random Stealing* (CRS), where 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. When a reply for a wide-area steal arrives, CRS 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.

We also implemented other load-balancing algorithms in Satin, such as a grid-aware load-based mechanism, and a grid-aware hierarchical algorithm, that minimizes inter-cluster communication. Like RS and CRS, these algorithms are initiated by the (idle) receiver. We also evaluated randomized pushing, a sender-initiated scheme. In [Nieuwpoort et al. 2001], we explain CRS in more detail, and compare its performance with RS, work pushing, grid-aware load-based and grid-aware hierarchical alternatives. We demonstrated that hierarchical algorithms, often proposed in the literature for load balancing in wide-area systems, perform even worse than random stealing, even though they do reduce the wide-area communication to an absolute minimum. The reason for this is that their load balancing is too fine grained, and that they force all nodes of a cluster to wait while work is transfered across the wide-area network. The other alternatives, random pushing and load-based stealing, only work well after careful, application-specific tuning.

4.1.3 *Grid-enabled Communication: Ibis.* Satin needs efficient communication with the flexibility to run on dynamically changing sets of heterogeneous processors and networks. Ibis [Nieuwpoort et al. 05 B] is a platform that tries to meet these requirements. Like Satin, Ibis is written in Java. Ibis provides efficient communication in combination with any JVM. It is a flexible system that can provide communication support for any grid application, from the broadcasting of video to massively parallel computations. It provides a unified framework for reliable and unreliable communication, unicasting and multicasting of data. Because Ibis is Java-based, it has the advantages that come with Java, such as portability, support for heterogeneity and security. Ibis has been designed to combine high performance and flexibility. It can combine standard techniques that work "everywhere" (e.g., using TCP) with highly-optimized solutions for special cases, like a local high speed Myrinet or gigabit Ethernet network without changing the user code.

To obtain acceptable communication performance, Ibis implements several optimizations. Most importantly, the overhead of serialization and reflection is avoided by compile-time generation of special methods (in bytecode) for each object type. These methods can be used to convert objects to bytes (and vice versa), and to create new objects on the receiving side, without using expensive reflection mechanisms. This way, the overhead of serialization is reduced dramatically. Satin makes extensive use of serialization. After rewriting the application bytecode, the Satin bytecode rewriter (see Figure 3) invokes the Ibis compiler on the rewritten bytecode to generate the serialization code.

4.2 Shared Objects

As explained in Section 3.2, shared objects are implemented using object replication. Replicas of shared objects are created in the following way. If a processor receives a task with a shared object as a parameter, and it does not have a replica yet, it copies the object from the machine it received the task from. A processor can thus join the computation at any moment and receive up-to-date replicas of all shared objects it needs.

Updates to shared objects are forwarded to remote replicas *asynchronously*. We do not try to prevent updates from getting lost or being duplicated. We do use reliable communication, but since processors can join or leave the computation at any moment, a processor can miss an update or receive it twice. The updates may also arrive in different orders at different machines.

Satin provides a *message combining* facility for shared object updates. If message combining is enabled, updates are not forwarded immediately, but delayed for a short period of time, or until a substantial amount of data is gathered. All updates accumulated during this period are forwarded at the end of it in one big transfer. Satin's message combining thus is transparent for the programmer. In addition, the programmer can use application-level message combining.

Guard consistency is enforced by conceptually evaluating a guard function for each task containing a guard. The implementation, however, makes an important optimization by only evaluating guards for *remote* tasks which were obtained from other machines. When a parent and a child task are executed on the same machine, if a shared object was in a consistent state when the parent was executed, it will also be consistent when the child is executed. If a guard evaluates to false, the system first waits a certain amount of time for late updates to arrive. If the guard still evaluates to false, the runtime system contacts the processor from which the task was received and requests the replicas of the shared objects used by this task. The machine from which a task was received is the machine on which the parent of this task was executed. So, this machine certainly contains replicas of shared objects that are consistent for this task. Due to the hierarchical structure of the task tree in divide-and-conquer programs, the replicas are also transfered over the network in a hierarchical way. There thus needs to be no central point (often called "home node" in this context). The lack of a central point makes sure that the shared objects scale well in a grid environment [Wrzesińska et al. 07 B].

4.3 Speculative Parallelism: Inlets and Aborts

As explained in Section 3.3, Satin provides two mechanisms to support speculative parallelism: inlets and aborts. Inlets are triggered if a spawned job throws an exception. When running in parallel, exceptions thrown by a remote job are intercepted and sent back to the CPU that spawned the method. The inlet is executed there. Thus, the communication pattern follows the normal hierarchical structure of the spawn tree.

When an abort operation is executed, all spawn operations performed by the current method have to be aborted, *including all their child jobs*. The jobs that have to be aborted can be in four different states: they can be already finished, they can be on the stack, in the local work queue, or they may have been stolen by a remote machine. If a job is already finished, the abort is a no-op. When a job is on the stack, the Satin runtime system sets a flag for that job, indicating that it should be aborted. The code that is generated by the Satin bytecode rewriter checks this flag, and stops the job if it has been set. If a job is in the local work queue, it can be aborted by simple removing it from the queue. The difficult case is killing a job that was stolen. In that case, an asynchronous abort message containing an identifier for the job to be killed is sent to the thief, which in turn uses the mechanism described above to abort the job and all its children.

A nice property of our implementation is that all network messages that are sent to do an abort are *asynchronous*. This means that the machine that executed the abort can immediately continue working. This is especially advantageous in a wide-area setting with high latencies. The cost of an abort operation is virtually independent of network latency. Furthermore, all messages sent by the abort mechanism are point-to-point, and follow the hierarchical divide-and-conquer model. Because this maps well on the hierarchical structure of the grid, the implementation is efficient. For more information on Satin's support for speculative parallelism we refer to [Nieuwpoort 2003].

4.4 Malleability and Fault Tolerance

Satin supports both malleability and fault tolerance. The difference between them is that with malleability, processors leave gracefully, so it is possible to take actions (e.g., save results) before a processor leaves. Also, malleability implies that processors can be added dynamically.

The basic idea to implement malleability and fault tolerance is to recompute work done by a leaving or crashing processor, which is possible since jobs do not have any side effects. The main problem with recomputing divide-and-conquer tasks is the possibility of *orphans*: if work is stolen from a machine that subsequently leaves or crashes, the stolen job becomes an orphan, because it is no longer clear what should be done with the result. Also, this approach fails if *all* processors temporarily leave, which typically happens if a program is suspended. Finally, recomputing all results of a processor that leaves voluntarily is inefficient, since the process could have saved its results before leaving. To solve all these problems, Satin uses a combination of two techniques: orphan work saving and checkpointing.

Orphan jobs are saved in the following way: when a processor discovers that a job it was working on is an orphan, it stores its result locally and broadcasts a

(jobID, processorID)-tuple to all other processors. If a processor P wants to leave gracefully, it first transfers its finished jobs to another randomly selected processor. Next, the new owner broadcasts a (jobID, processorID)-tuple for each job. Before a processor recomputes a job after a machine has crashed or left, it first checks its local tuples to see if the job has already been computed elsewhere, thus avoiding redundant computations. When processors leave unexpectedly (crash), the work they have done is recomputed, but the results of orphans caused by this crash are reused. The tuple messages are small since they do not contain the job itself. Moreover, they can be broadcast asynchronously and can be subject to message combining, resulting in an efficient, low-overhead implementation.

The novelty of our approach is the restructuring of the computation tree to reuse as many already computed partial results as possible. When processors are leaving gracefully, our mechanism can save nearly all the work done by the leaving processors. Therefore, we use our technique for efficient *migration* of the computation: to migrate the computation from one cluster to another, we first add the new cluster to the computation and then (gracefully) remove the old one. More details are provided in [Wrzesińska et al. 2005].

Orphan result saving uses only the state inside the main memory of the surviving (not leaving) processors to recover from faults. This mechanism is sufficient to guarantee the successful completion of an application, as long as at least one processor remains alive. (Even if the root processor crashes, the runtime system elects a new one.) To be able to also survive a total loss of all processors, Satin periodically checkpoints the orphan-result data structures [Wrzesińska et al. 07 C]. It uses a light-weight mechanism and avoids synchronization between the processors. The checkpoint files are implemented using the Java GAT [Nieuwpoort et al. 2007], which automatically selects an appropriate lower-level protocol (FTP, GridFTP, etc.) and optimizes any adjustable parameters. This checkpointing extension also allows Satin applications to be suspended and resumed later, possibly at another set of resources.

4.5 Adaptivity

Malleability is needed when external entities like grid schedulers change the resources on which an application runs. Satin can also use its malleability mechanism to decide for itself to change the resources. In this way, Satin is able to adapt automatically to changing resource conditions such as overloaded CPUs or networks. Some grid systems try to adapt based on performance prediction models, but constructing such models is difficult. Satin therefore is *self-adaptive* and does not use predictions.

During the application run, a separate *coordinator* process periodically collects information about the communication times and idle times of the processors. The coordinator uses these statistics to automatically estimate the resource requirements of the application. Next, it adjusts the resource set the application is running on by adding or removing compute nodes or even entire clusters, using the malleability mechanisms described above.

The efficiency of the application is estimated by the following formula, adapted from the definition in [Eager et al. 1989], which we call *weighted average efficiency*:

$$wa\_efficiency = \frac{1}{n} * \sum_{i=1}^{n} speed_i * (1 - overhead_i)$$

where n is the number of processors and $overhead_i$ is the fraction of time the i^{th} processor spends being idle or communicating. Each processor periodically computes this overhead over the previous period. The useful work done by a processor $(1 - overhead_i)$ is weighted by multiplying it by the speed of this processor relative to the fastest processor. The speed of a processor is estimated by running a benchmark, for example the actual application with a smaller input problem. This benchmark is executed periodically, because the speed of a processor might change if it becomes overloaded by another application (for time-shared machines).

The coordinator tries to keep the weighted average efficiency between E_{min} and E_{max}. When it exceeds E_{max}, the coordinator requests new processors from the scheduler. The higher the weighted average efficiency, the more processors are requested. The coordinator starts removing processors when the weighted average efficiency drops below E_{min}. The lower the efficiency, the more nodes are removed. The thresholds we use are $E_{max} = 50\%$, because we know that adding processors when the efficiency is lower does not make sense, and $E_{min} = 30\%$. An efficiency of 30% or lower might indicate performance problems such as low bandwidth or overloaded processors. In that case, removing bad processors will be beneficial for the application. Such a low efficiency might also indicate that we simply have too many processors. In that case, removing some processors may not be beneficial but it will not harm the application. The coordinator always tries to remove the "worst" processors, taking their overheads and speeds into account. Additionally, the coordinator computes inter-cluster overhead and removes entire clusters if their overhead exceeds a certain threshold, since bandwidth on the link between this cluster and the Internet backbone may be insufficient for the application. Our experiences with self-adaptation in Satin are described in detail in [Wrzesińska et al. 07 A].

cluster	location	cores per machine	machines	total cores	speed (GHz)	network
VU Univ.	Amsterdam	4	64	256	2.4	Myrinet 10G
Leiden Univ.	Leiden	2	24	48	2.6	Myrinet 10G
Univ. of A'dam	Amsterdam	4	32	128	2.2	Myrinet 10G
TU Delft	Delft	2	48	96	2.4	Ethernet 1G
MultimediaN	Amsterdam	2	24	48	2.4	Myrinet 10G

Table II. Characteristics of the DAS-3 clusters.

5. EXPERIMENTAL RESULTS

In this section, we will analyze the performance that Satin achieves. We present measurements on the DAS-3 system, which consists of five different clusters in the Netherlands. The clusters are largely homogeneous, but there are differences in the number of cores per machine, the clock frequency, and the network. The details of the clusters are shown in Table II. All machines have 4 GB of memory. We use the Sun JVM version 1.6.0, an off-the-shelf JIT.

We use the Ibis implementation on top of TCP for the measurements in this section. This means that the results shown below were measured using a 100% Java implementation of Ibis, e.g. there is no native code in our communication software stack. This is especially interesting, because the results give a clear indication of the performance level that can be achieved in Java with a "run everywhere" implementation, without using any native code. The code thus works on any platform and on any stock JVM that implements the JVM specification [Lindholm and Yellin 1999], without recompilation. The SmartSockets library we use automatically selects the fastest available network for each connection [Maassen and Bal 2007]. Depending on the DAS-3 cluster, this is either 10 Gigabit Myrinet or 1 Gigabit Ethernet.

We will start by analyzing the performance of Satin's spawn and sync primitives using micro benchmarks. Next, we evaluate the performance of real Satin applications, first on a singe cluster and then on the wide-area DAS-3 system. We evaluate the performance of Satin's fault-tolerance mechanism. Furthermore, we show that Satin's adaptivity features improve performance in realistic grid scenarios. For practical reasons, these experiments are done on the DAS-2 system[2], a completely homogeneous system containing 200 Dual Pentium-III nodes. We also present an experiment on 960 cores of the Grid 5000 system in France, winning the first prize in the Grids@work Plugtest contest, for the largest number of machines deployed in a single parallel application. Finally, we show the results of a European-scale run where we use an extremely heterogeneous environment.

5.1 Micro Benchmarks

Table III shows the performance of the Satin spawn and sync primitives and the inlet mechanism (on the VU cluster). The numbers are the average of one million calls, and show that a single spawn operation, followed by a sync costs 314 *ns* on our hardware. Adding extra parameters costs only 1 nanosecond extra, regardless of the type. Although the cost is about 39 times higher than a normal method

[2]See http://www.cs.vu.nl/das2.

benchmark	time (nanoseconds)
normal method invocation, no parameter, no return	8
normal method invocation, 1 parameter + return value	8
normal spawn/sync, no parameter, no return	314
normal spawn/sync, 1 parameter + return value	315
inlet spawn/sync, no parameter, no return	378
inlet spawn/sync, 1 parameter + return value	384

Table III. Low-level benchmarks.

invocation, the absolute time is very small. A spawn operation that throws an exception containing one field, followed by the execution of an empty inlet costs 384 ns, independent of the type of the parameter to the spawn and the result type stored in the inlet. This is only marginally more that a regular spawn operation. We can conclude from these numbers that Satin's basic operations can indeed be implemented efficiently in pure Java, even on off-the-shelf JITs. In the following section we will investigate whether Satin is also efficient at the application level.

5.2 Applications

To evaluate Satin's performance we implemented several real-world applications with Satin. When possible, we wrote both divide-and-conquer and master-worker versions, allowing comparisons of performance and scalability of both programming models. Also, we use different versions of the applications, with and without shared objects and aborts, demonstrating the usefulness and performance of Satin's divide-and-conquer extensions. In all cases, speedups and efficiencies are computed relative to sequential versions of the application that do not use Satin. It is important to note that the applications were not initially designed for grid computing, and also were not originally developed by us. We describe the applications below.

5.2.1 *Barnes-Hut N-Body Simulation.* Barnes-Hut is an $O(N \log N)$ N-body simulation. It can be applied to various domains, including astrophysics, fluid dynamics, electrostatics and computer graphics. We wrote a Satin program based on the code by Blackston and Suel [Blackston and Suel 1997]. We have used the Barnes-Hut code as example in Section 3, see Figure 1. We implemented two different divide-and-conquer versions: one that distributes all data as parameters to spawned invocations, and a version that uses Satin's shared objects to replicate the simulated bodies on all machines.

5.2.2 *Gene Sequence Alignment.* Pairwise sequence alignment is a bioinformatics application where DNA sequences are compared with each other to identify similarities and differences. We have parallelized an existing gene sequence alignment application that uses the well-known Smith-Waterman [Smith and Watherman 1981] algorithm using Satin's divide-and-conquer programming style. The application uses the NeoBio standard library that was developed elsewhere [de Carvalho Jr. and Crochemore]. For comparison reasons, we also implemented a master-worker version.

5.2.3 *Grammar induction.* The grammar induction application learns a Deterministic Finite State Automaton (DFA) from labeled sentences, i.e. sentences that

must be accepted or rejected by the target DFA. It starts out by building a (tree-shaped) DFA that exactly accepts the sentences in the learning sample, and nothing else. Then, it uses heuristics to generalize, by merging states and making the resulting DFA deterministic again by applying further merges, under the condition that the resulting DFA still rejects the sentences from the sample that should be rejected. The algorithm used to determine the merge candidates is blue-fringe [Lang et al. 1998], while Evidence Driven State Merging (EDSM) [Lang et al. 1998] is used to do the merging. The application uses an iterative deepening search strategy up to a maximum depth. We implemented both a divide-and-conquer version and a master-worker version. Both use a shared object to store the learning sample. For more details on the Satin implementation of this application we refer to [Adriaans and Jacobs 2006].

5.2.4 *Othello.* Othello [Iwata and Kasai 1994], also known as Reversi, is a strategy board game on an 8x8 board. We use the evaluation function of the powerful Marvin implementation by Voges[3]. Our implementation uses the MTD(f) algorithm [Plaat et al. 1996] to do a forward search through the possible game states. MTD(f) and other parallel search algorithms use a large amount of speculative parallelism. A transposition table [Slate and Atkin 1977] is used to avoid searching identical positions multiple times. The parallel version speculatively searches multiple states in parallel, and replicates (a part of) the transposition table to avoid search overhead. For performance reasons, application-level message combining is used to aggregate multiple transposition table updates into a singe network message, in addition to Satin's built in transparent message combining. Since the updates are very small and extremely frequent, this avoids an excessive number of calls to the satin runtime system, and reduces communication. To achieve good performance, search overhead (the result of speculatively searching subtrees) should be minimized. If some processor finds a promising solution, it should be forwarded to the others, allowing them to prune work. Therefore, the transposition table is replicated using Satin's shared objects mechanism. We used a transposition table with 16.8 million entries, occupying about 368 MBytes memory. Othello can further optimize the speculative parallelism by using Satin's inlet and abort mechanisms. If a spawned child job generates a cutoff (i.e., it represents a good solution), other child jobs are aborted, reducing search overhead. For comparison reasons we use two versions, with and without aborts.

5.3 Single cluster measurements

In this section, we evaluate the applications on a single cluster (at the VU in Amsterdam) of the DAS-3 system. This way, we use a controlled environment, eliminating the complexity of the grid, resulting in reproducible results. Speedup graphs are shown in Figure 4, while we show important application and communication statistics in Table IV.

Barnes-Hut is a challenging application that needs high communication throughputs. Satin's shared objects model is able to effectively reduce the application's communication needs: the pure divide-and-conquer version sends more than 84

[3]See http://www.voges.info/marvin for more information.

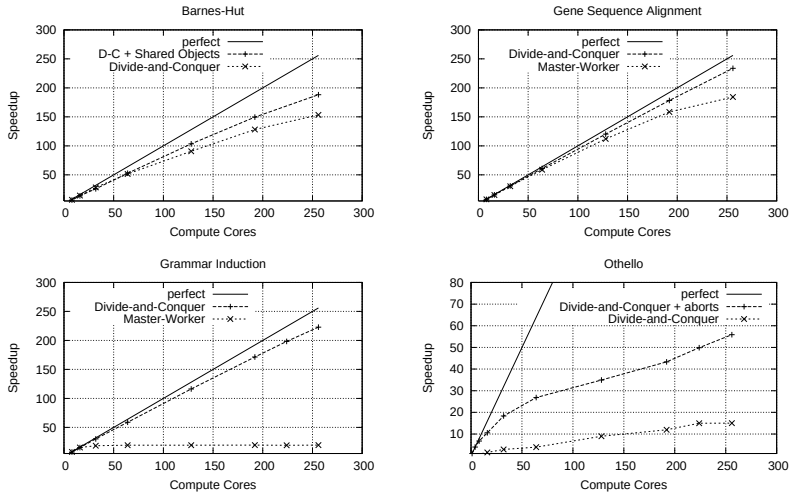

Fig. 4. Speedups of Satin applications.

application	spawns	syncs	jobs stolen	messages	total data	256 core efficiency
Barnes-Hut SO	118,420	14,816	16,425	3,514,010	428.0 MB	73.5 %
Barnes-Hut DC	118,420	14,816	14,787	2,740,089	84.1 GB	60.0 %
Gene Seq. DC	1,049,087	524,800	5,462	653,554	2.4 GB	91.3 %
Gene Seq. MW	8,978	1	8,978	30,676	667.4 MB	72.0 %
Grammar DC	224,026	96,008	21,209	5,947,901	128.5 MB	87.1 %
Grammar MW	128,025	7	128,021	386,032	79.6 MB	7.6 %
Othello+aborts	3,313,234	418,833	1,022,000	70,390,763	2.1 GB	21.8 %
Othello	6,973,693	849,450	4,144,009	394,054,014	10.5 GB	5.9 %

Table IV. Application statistics.

Gigabytes of data, while the divide-and-conquer version with shared objects sends only 428 Megabytes. This also improves scalability, increasing the parallel efficiency from 60 to 73.5 percent on 256 cores.

Gene sequencing and grammar induction are examples that demonstrate that the divide-and-conquer model scales better than the master-worker model. This is especially true for grammar induction, where a significant amount of work is needed to generate a new job. With the divide-and-conquer model, generating new jobs is done in parallel on all nodes, while with the master-worker model only the master generates new work. For gene sequencing, the amount of communication in the master-worker version is a bottleneck: all workers need to retrieve jobs from a central point, the master. With the divide-and-conquer model, there is no such central point. In this case, the divide-and-conquer version sends more data and more messages, but the communication is distributed over all machines.

Othello starts from the initial game position, and uses a search depth of 21. On a single core, Othello visits 2.9 *billion* different positions. The parallel runs can visit more nodes, because of the speculative nature of the search. Figure 4 shows the

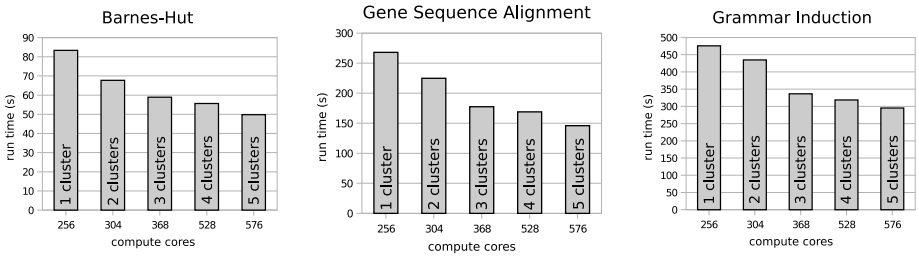

Fig. 5. Speedups of Satin applications on up to 576 compute cores on up to 5 clusters.

speedups of Othello both using Satin's abort mechanism, and without aborts. The efficiency with aborts on 256 cores is about 21.8%. This may seem disappointing, but game-tree search in general is hard to parallelize efficiently, due to search overhead. Other researchers report similar speedups [Kishimoto and Schaeffer 2002]. When no abort mechanism is used, Othello achieves an efficiency of less than six percent. The statistics in Table IV show that Satin's abort mechanism successfully aborts speculatively spawned work. The version without aborts suffers from search overhead, spawning about two times more parallel jobs, sending five times more messages and data. The version with aborts performs 310,482 abort operations, causing Satin to send 178,080 abort messages, cancelling 1,136,942 speculatively spawned jobs.

The measurements presented in this section demonstrate that the divide-and-conquer paradigm is not only more general than the master-worker paradigm, but can also outperform it, even on a single cluster. Furthermore, we demonstrated that the use of a shared objects model can increase both the expressiveness and the performance of parallel divide-and-conquer applications. Finally, for applications using speculative parallelism, Satin's abort mechanism can reduce search overhead considerably.

5.4 Wide-area Measurements

In the previous section, we demonstrated that Satin and the divide-and-conquer model work effectively on a cluster. Now, we will evaluate the performance of Satin on a grid. We use the wide-area DAS-3 system, since this is a relatively homogeneous grid designed for controlled computer science experiments. This way, we can present reproducible results, while avoiding influences from outside. We use the best performing versions of the applications of the previous section. In all cases, this is the divide-and-conquer version using shared objects. Figure 5 shows the run times. We start with a single cluster and incrementally add an additional cluster, until we use all five clusters of our system simultaneously. On the x-axis, we show the total number of cores used. See Table II for more details on the cluster configurations. Note that the x-axis does not use a linear scale, because the clusters do not have identical sizes. It is clear that the applications scale well to multiple clusters. Adding additional clusters improves performance significantly in all cases.

Table V shows application statistics, while Table VI shows communication statistics, using five clusters with 576 cores in total in both cases. The results show that

application	spawns	syncs	efficiency
Barnes-Hut	118,420	14,816	74 %
Gene Seq.	6,117,236	3,062,437	82 %
Grammar	224,026	96,008	72 %

Table V. Application statistics on 5 clusters with 576 cores.

	local			wide-area		
application	jobs stolen	messages	data	jobs stolen	messages	data
Barnes-Hut	25,270	17,511,522	588 MB	11,858	227,901	140 MB
Gene Seq.	8,489	22,792,842	1.2 GB	5,515	313,706	624 MB
Grammar	30,020	135,095,129	2.3 GB	17,236	798,586	36 MB

Table VI. Application statistics on 5 clusters with 576 cores.

the applications achieve excellent performance, with between 72 and 82 percent efficiency. To compute the efficiencies, we ran the sequential application on all clusters, and used these run times to normalize the run time of the parallel version (see [Nieuwpoort et al. 05 A] for details). We have to do this since the processors speeds differ slightly. The applications use fine-grained parallelism, spawning between one hundred thousand and six million parallel jobs. Furthermore, the applications communicate intensively, sending millions of messages.

Table VI shows that Satin's grid-aware CRS scheduling algorithm performs well. For all applications, the number of jobs that is transfered within a cluster is much larger than between clusters. The same is true for the number of messages and the amount of data sent. With the standard RS algorithm, the reverse is true. If a victim is selected at random, it is likely to be remote. For example, in a system with five clusters of equal size, 80% of all nodes would be in another cluster. We refer to [Nieuwpoort et al. 2001] for more details and a comparison between CRS and RS and other algorithms.

The high efficiencies also demonstrate that Satin copes well with the differences in CPU and network speeds. Finally, the results show that the divide-and-conquer model in general, and more specifically Satin, can scale to large numbers of machines and clusters, while providing a powerful and easy to use programming model that features shared objects and an abort mechanism. The master-worker model, in contrast, already has scalability problems on a single large cluster.

5.5 Fault tolerance and migration measurements

In [Wrzesińska et al. 2005], we demonstrated that the overhead of the basic fault-tolerance mechanism is negligible if no crashes occur. Furthermore, Satin's checkpointing mechanism has a small overhead, depending on the parameter size of the spawned methods in the application. Since Satin uses concurrent checkpointing, the impact of writing the checkpoint file is minimized. In this section, we will show that Satin's fault tolerance mechanism is effectively able to handle crashes, while saving orphaned jobs. First, we compare the performance of our *basic* fault-tolerance algorithm (with orphan saving but no checkpointing) with the traditional

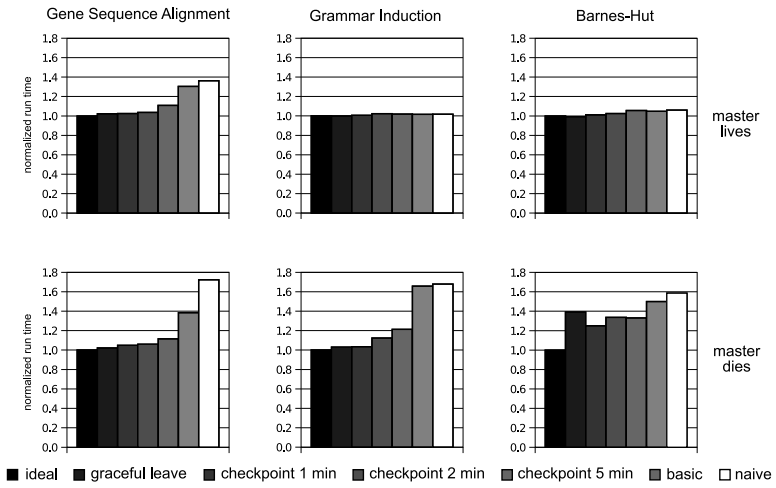

Fig. 6. Performance of fault tolerance, crashing 1 cluster.

'*naive*' algorithm that recomputes lost work. Next, we compare the performance of the *checkpointing* extension with the performance of the basic algorithm. We will look at different checkpointing intervals. Finally, we will evaluate the performance of our algorithm when the nodes are leaving *gracefully*, after a prior notification.

In these experiments, we run the three applications on 32 nodes in 2 clusters. We remove one of the clusters in the middle of the computation, that is, after half of the time the computation would take on 2 clusters without processors leaving. This case is the most demanding, because the largest number of orphan jobs is created in this case. Typically, the number of orphans does not depend on the moment when processors leave, except for the initial and final phase in the computation. To allow a fair comparison between various checkpointing intervals, we made sure that the crash occurs exactly in the middle of a checkpointing interval. Finally, we distinguish two cases: the scenario where the cluster that contains the master crashes, and the scenario where the other cluster crashes.

Figure 6 shows the normalized run times of the applications, including the checkpointing overhead if applicable. On average, our basic fault-tolerance algorithm outperforms the traditional, 'naive' approach (that recomputes lost work) by 9%. Checkpointing improves the performance by an additional 26%. The performance improvement is largest with small checkpointing intervals, which demonstrates that Satin's concurrent checkpointing mechanism introduces only a small overhead. If nodes are leaving gracefully, the orphan saving algorithm provides up to 69% performance improvement over the 'naive' algorithm. The Barnes-Hut application is an interesting case, since it is an iterative application. If the cluster with the master crashes or leaves gracefully, the application has to be restarted, recomputing all iterations. Checkpointing allows more work to be reused than the other methods, since work from previous iterations was also saved.

Table VII lists the number of jobs stored in orphan tables, and the percentage of jobs that is reused. With the basic fault-tolerance algorithm, almost all jobs are

	graceful	ckpt 1 min	ckpt 2 min	ckpt 5 min	basic
Gene Seq., master lives					
jobs in orphan tables	253	1542	845	392	138
% jobs reused	99.6%	23.6%	35.0%	70.7%	97.8%
broadcast messages	29	42	19	23	26
Gene Seq., master dies					
jobs in orphan tables	526	1723	1127	608	302
% jobs reused	100%	29.3%	42.6%	82.1%	100%
broadcast messages	118	101	106	88	69
Grammar, master lives					
jobs in orphan tables	61	1019	585	168	0
% jobs reused	100%	5.5%	0.0%	0.0%	0.0%
broadcast messages	14	2	2	2	0
Grammar, master dies					
jobs in orphan tables	165	1057	699	261	79
% jobs reused	100%	15.1%	34.2%	63.2%	100%
broadcast messages	54	47	49	40	34
Barnes, master lives					
jobs in orphan tables	270	1216	971	442	66
% jobs reused	88.5%	22.0%	36.9%	20.4%	100%
broadcast messages	10	11	13	11	8
Barnes, master dies					
jobs in orphan tables	601	1683	1069	670	297
% jobs reused	99.3%	68.0%	85.5%	100%	98.0%
broadcast messages	31	25	28	34	31

Table VII. Orphan saving statistics

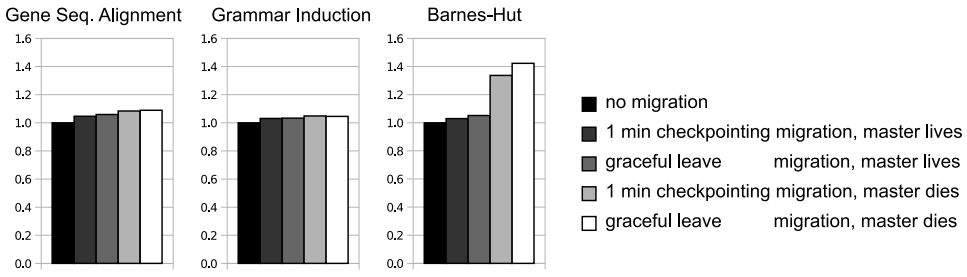

Fig. 7. Performance of migration, crashing 1 cluster.

reused, while with checkpointing, only 15% to 86% of the jobs is reused. This is caused by the fact that some jobs in the checkpoint file are redundant; their parent or other ancestor was checkpointed as well. In such cases, only the ancestor is used. Checkpoint compression can reduce the number of redundant jobs. Table VII also lists the number of broadcast messages sent in order to keep orphan tables up to date. Because message combining is used, this number is small and independent of the number of jobs in the orphan tables.

We also evaluate the overhead of malleability-based migration using the applications on 32 nodes in 2 clusters, both with and without checkpointing. In the middle of the computation, we gracefully removed one of the clusters and replaced it with another cluster with the same number of processors (16). For compari-

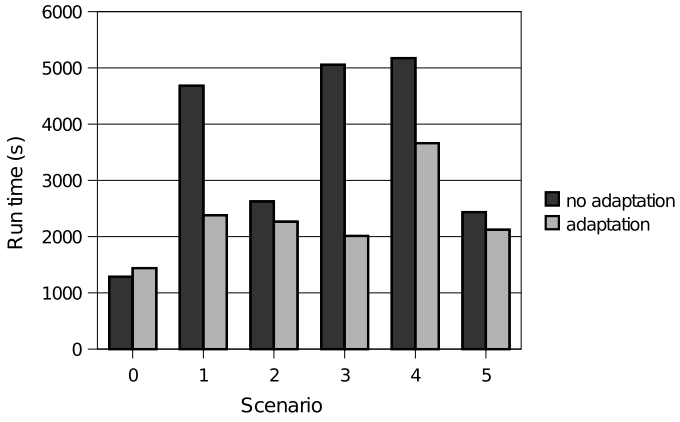

Fig. 8. Performance of adaptivity in different scenarios.

son, we include a run without migration. The normalized run times are shown in Figure 7. The difference in the run times shows the overhead of migration. With our approach, the overhead is less than 9%, except in one special case, with the Barnes-Hut application, and when the master dies. Since Barnes-Hut is an iterative application, all iterations are recomputed after the master is killed. We reuse all stored jobs in the orphan job table, but this still results in some overhead. Checkpointing-based migration is slightly more efficient in all cases. There are two sources of overhead in the normal case. First, the results from the leaving processors need to be sent over the network. Depending on the application, the amount of data to be sent can be significant. Second, part of the jobs need to be recomputed after migration, as only jobs that are finished at the moment the migration is requested are saved and transferred to other processors. The overhead is small, however, which shows that our mechanism can be used for efficient migration.

5.6 Adaptivity measurements

In this section, we evaluate Satin's adaptivity mechanism using several scenarios that are typical for grid environments. We perform the measurements with the Barnes-Hut application described above, using real hardware, i.e., we are not using a simulator. The scenarios allow us to demonstrate that Satin's adaptation support can *automatically* avoid serious performance bottlenecks such as overloaded processors or network links. For each scenario, we compare the performance with adaptation support to a non-adaptive run (See Figure 8). In the non-adaptive run, the coordinator does not collect statistics or perform benchmarking (for measuring processor speeds).

Scenario 0: adaptivity overhead. Barnes-Hut is started on 36 nodes in 3 equally large clusters. For the adaptive version, we measure the performance with collecting statistics and benchmarking turned on, but without allowing Satin to change the number of nodes, thus measuring the adaptivity overhead. In this experiment it is around 15%. Almost all overhead comes from benchmarking; it can be reduced by decreasing the benchmarking frequency. We used a relatively high frequency

due to the relatively short application run time, since this is more challenging, and using longer running applications would not allow us to finish the experimentation in a reasonable time. However, For longer running applications, the adaptation overhead can be kept much lower. For example, with the Barnes-Hut application, if the monitoring period is extended to 10 minutes, the overhead drops to 6%.

Scenario 1: expanding to more nodes. We start on less nodes (8 nodes in 1 cluster) than the application can efficiently use. When adaptation is turned on, the application gradually expanded to 36-40 nodes located in 4 clusters. This reduced the run time by 50%.

Scenario 2: overloaded processors. Barnes-Hut was started on 36 nodes in 3 clusters. After 200 seconds, we introduced a heavy, artificial load on all processors in one of the clusters. The adaptive version observed a very low weighted average efficiency and reacted by removing the overloaded nodes. The efficiency rose to around 65%, which triggered Satin to add new nodes, reaching 38 in total. The total run time was reduced by 14%.

Scenario 3: overloaded network link. We start the application on 36 nodes in 3 clusters. We simulated that the uplink to one of the clusters was overloaded and we reduced the bandwidth on this uplink to approximately 100 KB/s. To simulate low bandwidth we use the traffic-shaping techniques described in [Chiu et al. 2000]. The adaptive version observed a weighted average efficiency of 25% and a high WAN communication overhead in one of the clusters. Therefore it removed the badly connected cluster after the first monitoring period. As a result, the weighted average efficiency rose to around 65% and new nodes were gradually added until their number reached 38. The total runtime was reduced by 60%.

Scenario 4: overloaded processors and an overloaded network link. We ran Barnes-Hut on 36 nodes in 3 clusters. Again, we simulated an overloaded uplink to one of the clusters. Additionally, we simulated processors with heterogeneous speeds by inserting a relatively light artificial load on the processors in one of the remaining clusters. The adaptive version removed the badly connected cluster after the first monitoring period. This already improved performance considerably. Next, Satin removed several of the nodes that suffered from the background load. Over the whole run, Satin's adaptation reduced the total runtime by 30%.

Scenario 5: crashing nodes. Again, we start Barnes-Hut on 36 nodes in 3 clusters. After 500 seconds, we crashed 2 out of 3 clusters. After the crash, the weighted average efficiency rose to around 70%, which triggered adding new nodes in the adaptive version. The number of nodes gradually went back to 36, which reduced the total runtime by 13%.

The results show that Satin's adaptation strategy deals with problems that are typical for grids: expand to a larger number of nodes or shrink to a smaller number of nodes if the application was started on an inappropriate number of processors, remove inadequate nodes and replace them with better ones, replace crashed processors, avoid slow networks, etc. The application adapts *fully automatically* to changing conditions. We demonstrated that our approach can yield significant per-

site	architecture	speed (MHz)	operating system	CPUs
Amsterdam, Netherlands	Intel P3	1000	32-bit Linux	8
Amsterdam, Netherlands	Sun SPARC	750	64-bit Solaris	2
Lecce, Italy	Compaq Alpha	667	64-bit Tru64 Unix	4
Cardiff, UK	Intel P3	1000	32-bit Linux	2
Brno, Czech Republic	Intel Xeon	2400	32-bit Linux	8
Berlin, Germany	SGI MIPS	500	32-bit Irix	16

Table VIII. Machines used in a European-scale experiment.

formance improvements (up to 60% in our experiments), while the overhead is small when no adaptation is needed. For a more detailed evaluation of Satin's adaptation performance we refer to [Wrzesińska et al. 07 A].

5.7 Large-scale Grid experiments

We further evaluated Satin's scalability and performance on a real grid system by participating in the Grids@work event in October 2005 in Sophia Antipolis, France. A part of the event was the 2nd Grid Plugtest, which consisted of an N-Queens contest. The aim was to find the number of solutions to the N-Queens problem. We wrote a parallel N-Queens solver with Satin. We compared our N-Queens program with the fastest known C version by Takahashi. We found that our parallel version on a single machine was only 0.5% slower than the C version. The portability we gained by writing the program in Java and the parallelization with Satin thus had almost no measurable impact on performance.

During the contest we ran N-Queens on Grid'5000, a heterogeneous wide-area grid system distributed over France[4]. We used five clusters with 960 nodes in total, with the clusters being up to 1000 kilometers apart. We ran N-Queens with N=22 in 1516 seconds (25 minutes). The Satin application spawned 4.7 million jobs in 25 minutes, and the Satin runtime system sent about 800,000 messages to balance the load in the parallel computation. We measured the overhead of deploying and starting the jobs, as well as the communication cost in the Satin runtime system. In total, these overheads add up to 15% of the total run time. Therefore, 85% of the total run time is spent in the application itself. During the contest, we found and solved several scalability problems in Ibis and Satin. For instance, we setup connections for Satin's load-balancing algorithm on-demand, and limit the total number of connections we keep. In addition, we removed all central bottlenecks from both the Ibis and Satin implementations. There now is no single central point in Satin. Using Satin, we won the prize for the largest number of parallel nodes deployed during the contest. For more details on the scalability issues and this experiment we refer to [Nieuwpoort et al. 2006].

In [Nieuwpoort et al. 05 A] we present measurements of a Satin parallel raytracer application on an even more distributed and heterogeneous European scale grid, see Table VIII. In this case, the differences in CPU speeds are more than a factor of four, latencies differ by a factor of 200, and bandwidths even by a factor of 1200. Still, Satin achieves an efficiency of 80%. Finally, in recent experiments we successfully ran applications on the Amazon Elastic Compute Cloud.

[4]See http://www.grid5000.fr for more information.

System	Performance	Ease of Use	Applicability	fault-tolerance malleable migration	Adaptivity	Portability
Cilk	+	+	+/-	-	-	-
Javelin 3	-	+	-	-	-	+
Atlas	-	+	-	+	-	+
MW	-	+	-	+	+/-	-
Grid SuperSc.	-	+	+/-	-	-	-
HPJava	+	-	+/-	-	-	-
MPI	+	-	+	+/-	-	-
MPJ	+	-	+	+/-	-	+
ProActive	+	+/-	+	+/-	-	+
GridRPC	+	-	+	-	+/-	-
RMI	+	-	+	-	+/-	+
Satin	+	+	+/-	+	+	+

Table IX. Characteristics of Satin and related systems.

6. RELATED WORK

We discussed Satin, a Java-based divide-and-conquer system that provides shared objects, asynchronous exceptions and an abort mechanism. Satin is designed for heterogeneous and dynamic wide-area systems, without shared memory. Satin targets fine-grained distributed supercomputing applications. Satin does not explicitly target task farming or SETI@home-like trivially parallel applications, but it is easy to also write such applications in Satin. Finally, Satin aims to provide a high-level powerful programming model. Satin is implemented in 100% pure Java, so the only requirement to run Satin is a JVM. This facilitates the deployment of Satin on the grid. In this section, we compare Satin's features with related work. Table IX shows a summary of features provided by the systems we discuss in this section. For fault-tolerance, malleability, migration and adaptivity, a "+/-" indicates that a feature is present but not transparent, and has to be implemented by the application programmer.

6.1 Divide and Conquer Models

Many divide-and-conquer systems have been designed in the past, most are based on the C language. Most systems target shared memory systems or small-scale distributed systems (e.g., clusters). Cilk [Blumofe et al. 1995] only supports shared-memory machines, CilkNOW [Blumofe and Lisiecki 1997] and DCPAR [Freisleben and Kielmann 1995] run on local-area, distributed-memory systems, but do not support shared data. SilkRoad [Peng et al. 2000] is a version of Cilk for distributed memory systems that uses a software DSM to provide shared memory to the programmer, targeting at small-scale, local-area systems.

The Java classes presented by Lea et al. ([Lea 2000]) can also be used for divide-and-conquer algorithms on shared-memory systems.

JCilk [Danaher et al. 2005] is a Java-based multithreaded programming language that provides asynchronous exceptions, similar to the mechanism provided by Satin.

Javelin 3 [Neary and Cappello 2002] provides a set of Java classes for branch-and-bound computations, such as the traveling salesperson problem. Like Satin,

Javelin 3 is designed for wide-area systems. However, Javelin 3 uses a tree-based hierarchical scheduling algorithm. We found that such algorithms are inefficient for fine-grained applications and that CRS performs better [Nieuwpoort et al. 2001]. Moreover, none of the systems mentioned above provide fault-tolerance, malleability, adaptivity or distributed speculative parallelism.

Atlas [Baldeschwieler et al. 1996] is a wide-area divide-and-conquer system that has been designed with heterogeneity and fault tolerance in mind. Its fault-tolerance mechanism is also based on redoing the work. The problem of orphan jobs is not addressed in Atlas. Moreover, Atlas uses a hierarchical scheduling algorithm that has suboptimal performance [Nieuwpoort et al. 2001], and it does not provide adaptivity.

6.2 Grid Models

MW [Goux et al. 2000] is a framework for writing grid-enabled master-worker applications. The master-worker paradigm is very popular in grid computing. Since the tasks are independent, little communication is needed and high performance can be achieved even on wide-area networks. The MW API is extremely simple: the programmer needs only to provide a small number of functions: a function to split up work, worker initialization routine, a function performing the actual task etc. The runtime system takes care of load balancing, inter-processor communication and fault-tolerance. MW supports only embarrassingly parallel applications, and thus is less expressive than Satin. Nevertheless, many useful applications exhibit this structure. MW transparently handles worker crashes. If a worker fails, the task executed by this worker is re-assigned to another worker by the runtime system. A failure of the master is treated in a special non-transparent way. MW also supports malleability. Finally, since MW uses dynamic load-balancing, it adapts to varying processor speeds, a simple form of adaptivity. Since MW is written in C++, it is inherently less portable than Satin, applications may need to be recompiled for different grid sites.

Grid superscalar [Badia et al. 2003] provides a high-level C++-based programming model which hides most of the grid complexity and parallel-programming issues. The programmer structures the application as a set of possibly repetitive, sequential tasks. Such tasks can be executed in parallel on the grid. Each task operates on a set of files. Tasks that operate on the same file can have a data dependency. The grid superscalar compiler analyzes the data dependencies automatically. The runtime system can use the Globus Toolkit [Foster 2006] to execute tasks on a set of servers. Since an expensive grid middleware operation is performed to spawn each task, fine-grained applications will not perform well. Currently grid superscalar does not yet support fault tolerance, malleability, migration or adaptivity.

HPJava [Lee et al. 2004] is a Java-based framework supporting an HPF-like *data-parallel* programming style. It extends sequential Java with support for *distributed arrays*: arrays that are physically distributed over the memories of the participating processors. The programmer manipulates those arrays using high-level constructs such as *overall* construct which denotes a distributed, parallel loop. If a process needs to access an element held by another processor, explicit communication must take place. Since HPJava is an explicit communication programming model, it

is more low-level than Satin (or HPF). The application programmer will have to take the responsibility for grid-specific optimizations, such as dynamic load balancing and latency hiding. HPJava currently does not support fault tolerance, malleability, migration or adaptivity. Although HPJava uses Java technology, the distributed-memory implementation of HPJava relies on native communication interfaces, limiting portability.

Explicit *message passing* is a popular parallel programming paradigm. Several grid-aware implementations of the MPI standard exist. *MPICH-G2* [Karonis et al. 2003] allows running MPI applications across multiple clusters. MPICH-G2 is an integration of the popular MPICH [Gropp et al. 1996] implementation with the Globus Toolkit [Foster 2006]. Other implementations of MPI which address some grid issues are PACX-MPI [Gabriel et al. 1998] which provide grid-aware collective communications or OpenMPI [Graham et al. 2006] and MetaMPI [Eickermann et al. 1999] which support multiple communication protocols. MPI applications typically achieve high performance on cluster and supercomputers. On grids, the programmer must explicitly manage heterogeneity. Compared to higher-level models such as grid superscalar, master-worker or divide-and-conquer, message passing is relatively cumbersome and error-prone. The programmer has to explicitly deal with load-balancing and inter-processor communication and differences in processor speeds. In MPI applications, there are two approaches to providing fault tolerance, malleability and migration. One approach is system-level checkpointing and/or message logging. In MPI-TM [Robinson et al. 1996] and AMPI [Huang et al. 2006] for example, little or no effort is required from the application programmer, but the complexity, the large amount of data that needs to be saved and the lack of portability make it unsuitable for grid environments. The second approach is to let the programmer provide fault tolerance, malleability and migration. The MPI-2 standard [MPIF 1996] supports dynamic process management. Adding adaptivity to an MPI application is the responsibility of the application programmer. Adaptive MPI applications have for instance been developed in the context of the GrADS project [Vadhiyar and Dongarra 2005]. Grid-enabled MPI implementations hide many platform-specific details, which enhances portability. However, MPI is typically used in combination with C, C++ or Fortran. Applications written in those languages cannot be ported to another architecture without recompilation. Java bindings of the MPI interface exist, for example MPJ [Carpenter et al. 2000]. MPJ/Ibis [Bornemann et al. 2005] is an implementation of MPJ in pure Java. Phoenix [Taura et al. 2003] is a message passing programming model that is explicitly designed for malleable applications.

ProActive [Baduel et al. 2006] is a Java middleware which supports the so-called Object-Oriented SPMD programming model. A ProActive application is structured as a set of active objects that have their own thread of control. Method calls to active objects are asynchronous with transparent *future objects*. ProActive provides various group communication primitives based on method invocations. Since ProActive is an explicit communication model, the programmer is responsible for applying grid-specific optimizations. ProActive supports migration of active objects between JVMs. This facility can be used to implement application malleability and migration. ProActive also provides transparent fault tolerance through *Commu-*

nication Induced Checkpointing. Providing adaptivity is the responsibility of the application programmer. Portability of ProActive applications is ensured through the use of the Java technology.

RPCs (Remote Procedure Calls) [Birrel and Nielson 1984] have been widely used in programming parallel and distributed applications. Java's Remote Method Invocation (RMI) [Sun Microsystems 2008] is an object-oriented variant of RPC. GridRPC [Seymour et al. 2002] extends RPC with a number of important primitives. GridRPC defines *asynchronous* calls and primitives to operate on those calls. Using these primitives, GridRPC also supports the fork-join parallelism. GridRPC is suitable for medium-to-coarse-grained parallel applications but not for fine-grained parallelism. Example implementations of GridRPC are Netsolve [Arnold et al. 2002] and Ninf [Tanaka et al. 2003]. Applications based on RPCs can achieve high performance in grid environments. For example, in [Nieuwpoort et al. 2000] several grid applications programmed with RMI are demonstrated. However, it is the responsibility of the programmer to apply grid specific optimizations. GridRPC supports this by providing for example asynchronous procedure calls. Providing fault-tolerance, malleability and migration in applications using RPCs is the responsibility of the programmer. Some implementations of GridRPC API provide some form of transparent adaptivity. For example, Ninf-G uses dynamic information from Network Weather Service [Wolski et al. 1999] to dynamically select the best resource to execute an RPC call.

7. CONCLUSIONS

In this paper, we have discussed the design and implementation of a high-level programming model for distributed supercomputing. Our goal was to create a programming model that is programmer friendly, but allows the efficient execution of parallel applications on heterogeneous wide-area systems, or grids. To achieve this, we needed to deal with typical grid problems such as the dynamic nature of the system, different architectures and processor speeds, crashes and low bandwidths and long network delays between grid sites. We observed that grids are typically hierarchically structured, and that this property can be exploited to achieve high performance. Therefore, we did not attempt to provide a generic model, instead we focussed on two important application classes: master-worker and divide-and-conquer. The divide-and-conquer model is inherently hierarchical, and in this paper we have demonstrated that this model maps well onto modern grid systems.

We designed and implemented a divide-and-conquer and master-worker system called Satin. We have shown that Satin is easy to use, because it provides an extremely high-level programming model, and all problems caused by targeting grid systems are completely hidden from the programmer. In fact, there even is no concept of machines or communication in Satin, and work distribution is completely automatic and transparent. We have demonstrated that, although we target the difficult grid environment, it is possible to provide a rich programming model, while still achieving high performance. Satin's programming model supports master-worker and divide-and-conquer applications by using simple "spawn" and "sync" primitives. To support shared data, Satin provides a powerful shared object model. Finally, Satin has support for speculative parallelism, in the form of asynchronous exceptions and an abort mechanism.

Satin deals with the difficult problems introduced by the harsh grid environment in several ways. First, we use a Java-centric approach. Satin applications are compiled to Java bytecode, and Satin's runtime system is implemented in Java as well. Therefore, we can exploit Java's "Write once, run anywhere" property and deploy our grid applications to any system that has a JVM, without recompilation or reconfiguration. Second, we use a special grid-aware load-balancing algorithm, called CRS, that exploits the hierarchical structure of the grid when distributing work across machines. Third, we introduce a novel shared object model, that has application-defined consistency. This allows efficient implementations, using point-to-point communication, grid-aware broadcasting mechanisms or gossiping techniques, depending on the required consistency.

Furthermore, we have demonstrated how to deal with joining and leaving machines and crashes, problems that frequently occur in grids. Because of Satin's well-designed high-level programming model, support for malleability, migration and fault-tolerance can be implemented in the runtime system, and is completely transparent to the application. Using a novel orphan saving technique, Satin makes sure that as little work as possible is lost when machines or entire clusters leave or crash.

We also have described several techniques to deal with the changing performance characteristics of the grid. The CRS load-balancing algorithm tolerates large changes in network performance. The Satin runtime system provides adap-

tivity in a transparent way, by automatically adding and removing machines from the parallel application when needed.

We have used several realistic applications to demonstrate that the divide-and-conquer model can be efficiently executed on hierarchical systems (e.g., the grid), without any wide-area optimizations by the application programmer. Finally, we have shown that divide-and-conquer applications generally scale better than master-worker applications, since there is no single central bottleneck.

REFERENCES

Adriaans, P. and Jacobs, C. 2006. Using MDL for grammar induction. In *8th International Colloquium on Grammatical Inference (ICGI '06)*. Tokyo, Japan.

Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Seymour, K., Sagi, K., Shi, Z., and Vadhiyar, S. 2002. Users' guide to NetSolve V1.4.1. Tech. Rep. ICL-UL-02-05, University of Tennessee, Knoxville, TN, USA. June.

Badia, R. M., Labarta, J., Sirvent, R., Perez, J. M., Cela, J. M., and Grima, R. 2003. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing 1,* 2.

Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., and Quilici, R. 2006. *Grid Computing: Software Environments and Tools*. Springer Verlag.

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

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

Birrel, A. D. and Nielson, B. J. 1984. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems 2,* 1 (February), 39–59.

Blackston, D. and Suel, T. 1997. Highly Portable and Efficient Implementations of Parallel Adaptive N-Body Methods. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'97)*. Online at `http://www.supercomp.org`.

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

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

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

Bornemann, M., Nieuwpoort, R. V. v., and Kielmann, T. 2005. Mpj/ibis: a flexible and efficient message passing platform for java. In *EuroPVM/MPI 2005*. Lecture Notes in Computer Science (LNCS), vol. 3666. Springer Verlag Berlin Heidelberg, 217–224.

Butenhof, D. 1997. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley Pub. Co. ISBN: 0201633922.

Cappello, F. and Bal, H. 2007. Towards an international computer science grid (keynote paper). In *7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*. Rio de Janeiro, Brazil, 230–237.

Carpenter, B., Getov, V., Judd, G., Skjellum, A., and Fox, G. 2000. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience 12,* 11, 1019–1038.

Chiu, D.-M., Kadansky, M., Provino, J., and Wesley, J. 2000. Experiences in programming a traffic shaper. In *5th IEEE Symposium on Computers and Communications (ISCC 2000)*. 470–476.

Danaher, J. S., Lee, I.-T. A., and Leiserson, C. E. 2005. The JCilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. San Diego, California.

de Carvalho Jr., S. A. and Crochemore, M. The neobio library. See `neobio.sourceforge.net`.

Eager, D. L., Zahorjan, J., and Lazowska, E. D. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers 38,* 3 (March), 408–423.

Eickermann, T., Grund, H., and Henrichs, J. 1999. Performance issues of distributed MPI applicatins in a German gigabit testbed. In *6th European PVM/MPI Users' Group Meeting on*

Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer-Verlag, LNCS 1697, 3–10.

Foster, I. 2006. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*. LNCS 3779. Springer-Verlag, 2–13.

Foster, I. and Kesselman, C., Eds. 2003. *The Grid 2: Blueprint for a New Computing Infrastructure*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann; 2 edition (November 18, 2003).

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

Frigo, M., Leiserson, C. E., and Randall, K. H. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. Montreal, Canada, 212–223.

Gabriel, E., Resch, M., Beisel, T., and Keller, R. 1998. Distributed computing in a heterogeneous computing environment. In *5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, LNCS 1497, 180–187.

Goux, J.-P., Kulkarni, S., Yoder, M., and Linderoth, J. 2000. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. Pittsburgh, PA, 43–50.

Graham, R., Shipman, G., Barrett, B., Castain, R., Bosilca, G., and Lumsdaine, A. 2006. Open MPI: A high-performance, heterogeneous MPI. In *Cluster Computing, 2006 IEEE International Conference on*. 1–9. ISSN: 1552-5244, ISBN: 1-4244-0327-8, DOI: 10.1109/CLUSTR.2006.311904.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. A high-performance, portable implementation of the MPI message passing interface. *Parallel Computing 22,* 6 (September), 789–828.

Huang, C., Zheng, G., Kumar, S., and Kale, L. V. 2006. Performance evaluation of Adaptive MPI. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*.

Iwata, S. and Kasai, T. 1994. The Othello game on an n*n board is PSPACE-complete. *Theor. Comp. Sci. 123,* 123, 329–340. doi:10.1016/0304-3975(94)90131-7.

Kale, L. V., Kumar, S., and DeSouza, J. 2002. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*. Berlin, Germany, 230–237.

Karonis, N. T., Toonen, B., and Foster, I. 2003. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing 63,* 5 (May), 551–563.

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

Kishimoto, A. and Schaeffer, J. 2002. Transposition Table Driven Work Scheduling in Distributed Game-Tree Search. In *Proceedings of Fifteenth Canadian Conference on Artificial Intelligence (AI'2002)*. LNAI, vol. 2338. Springer-Verlag, 56–68.

Kitsuregawa, M. 2007. The info-plosion project. In *NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*. IEEE Computer Society, Washington, DC, USA, 3–6.

Lang, K. J., Pearlmutter, B. A., and Price, R. A. 1998. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. V. Honavar and G. Slutzki, Eds. LNAI, vol. 1433. Springer, 1–12.

Lea, D. 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Java Grande Conference*. San Francisco, CA, 36–43.

Lee, H.-K., Carpenter, B., Fox, G., and Lim, S. B. 2004. HPJava: Programming Support for High-Performance Grid-Enabled Applications. *International Journal of Parallel Algorithms and Applications 19,* 2-3, 175–193.

Lindholm, T. and Yellin, F. 1999. *Java(TM) Virtual Machine Specification, The 2nd Edition.* Prentice Hall PTR. ISBN: 978-0201432947.

Maassen, J. and Bal, H. E. 2007. Smartsockets: Solving the connectivity problems in grid computing. In *16th International Symposium on High-Performance Distributed Computing (HPDC'07).* Monteray, CA, USA, 1–10.

Maassen, J., Nieuwpoort, R. V. v., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., and Hofman, R. 2001. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems 23,* 6, 747–775.

Maassen, J., Seinstra, F., Kemp, R., Drost, N., and Nieuwpoort, R. V. v. 2008. Going nova. The First International Data Challenge for Finding SuperNovae, held in conjunction with IEEE Cluster2008. *First Prize Winner.*

Marlow, S., Jones, S. L. P., Moran, A., and Reppy, J. H. 2001. Asynchronous Exceptions in Haskell. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 274–285.

Mohr, E., Kranz, D., and Halstead, R. 1990. Lazy Task Creation: a Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.* 185–197.

MPIF, M. P. I. F. 1996. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville.

Neary, M. O. and Cappello, P. 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, 56–65.

Nieuwpoort, R. V. v. 2003. Efficient Java-centric grid-computing. Ph.D. thesis, Vrije Universiteit Amsterdam. `http://www.cs.vu.nl/~rob/`.

Nieuwpoort, R. V. v., Kielmann, T., and Bal, H. E. 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, 34–43.

Nieuwpoort, R. V. v., Kielmann, T., and Bal, H. E. 2007. User-friendly and reliable grid computing based on imperfect middleware. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07).* Reno, NV, USA.

Nieuwpoort, R. V. v., Maassen, J., Agapi, A., Oprescu, A.-M., and Kielmann, T. 2006. Experiences deploying parallel applications on a large-scale grid. In *EXPGRID - Experimental Grid testbeds for the assessment of large-scale distributed applications and tools, workshop in conjunction with the 15th International Symposium on High Performance Distributed Computing (HPDC-15).*

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

Nieuwpoort, R. V. v., Maassen, J., Kielmann, T., and Bal, H. E. 2005-A. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience 6,* 3 (September), 19–32.

Nieuwpoort, R. V. v., Maassen, J., Wrzesińska, G., Hofman, R., Jacobs, C., Kielmann, T., and Bal, H. E. 2005-B. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency & Computation: Practice & Experience 17,* 7-8, 1079–1107.

Peng, L., Wong, W., Feng, M., and Yuen, C. 2000. SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters. In *IEEE International Conference on Cluster Computing (Cluster2000).* Chemnitz, Saxony, Germany, 243–249.

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

Plaat, A., Schaeffer, J., Pijls, W., and de Bruin, A. 1996. Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence 87,* 1-2 (November), 255–293.

Robinson, J., Russ, S., Heckel, B., and Flachs, B. 1996. A task migration implementation of the Message-Passing Interface. In *5th IEEE International Symposium on High Performance Distributed Computing (HPDC'96)*. IEEE Computer Society, Syracuse, NY, USA, 61–68.

Romein, J. W., Bal, H. E., Schaeffer, J., and Plaat, A. 2002. A performance analysis of transposition-table-driven work scheduling in distributed search. *IEEE Transactions on Parallel and Distributed Systems 13,* 5 (May), 447–459.

Seinstra, F., Drost, N., Kemp, R., Maassen, J., Nieuwpoort, R. V. v., Verstoep, K., and Bal, H. 2008. Scalable wall-socket multimedia grid computing. 1st IEEE International Scalable Computing Challenge (SCALE2008), held in conjunction with the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid2008). *First Prize Winner*.

Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., and Casanova, H. 2002. Overview of GridRPC: A Remote Procedure Call API for grid computing. In *Intl. Workshop on Grid Computing (GRID02)*. Springer Verlag, LNCS 2536, Baltimore, MD, 274–278.

Slate, D. and Atkin, L. 1977. Chess 4.5 –The Northwestern Univeristy Chess Program. *Chess Skill in Man and Machine*, 82–118.

Smith, T. and Watherman, M. 1981. Identification of common molecular subsequences. *Journal of Molecular biology 147*, 195–197.

Sun Microsystems. 2008. *Java Remote Method Invocation Specification*. Online at http://java.sun.com/javase/technologies/core/basic/rmi.

Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., and Matsuoka, S. 2003. Ninf-G: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing 1,* 1, 41–51.

Taura, K., Endo, T., Kaneda, K., and Yonezawa, A. 2003. Phoenix : a parallel programming model for accommodating dynamically joining/leaving resources. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*. 216–229.

Vadhiyar, S. S. and Dongarra, J. J. 2005. Self adaptivity in Grid computing. *Concurrency and Computation: Practice and Experience 17,* 2–4, 235–257.

Verstoep, K., Maassen, J., Bal, H. E., and Romein, J. W. 2008. Experiences with fine-grained distributed supercomputing on a 10g testbed. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (ccgrid)*. 376–383.

Waldo, J. 1998. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency 6,* 3 (July), 5–7.

Wollrath, A., Waldo, J., and Riggs, R. 1997. Java-Centric Distributed Computing. *IEEE Micro 17,* 3 (May/June), 44–53.

Wolski, R., Spring, N., and Hayes, J. 1999. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems 15,* 5–6 (October), 757–768.

Wrzesińska, G., Maassen, J., and Bal, H. E. 2007-A. Self-adaptive applications on the grid. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*.

Wrzesińska, G., Maassen, J., Verstoep, K., and Bal, H. E. 2007-B. Satin++: Divide-and-share on the grid. In *2nd IEEE International Conference on e-Science and Grid Computing*. Amsterdam, The Netherlands.

Wrzesińska, G., Nieuwpoort, R. V. v., Maassen, J., and Bal, H. E. 2005. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*. Denver, CO.

Wrzesińska, G., Oprescu, A.-M., Kielmann, T., , and Bal, H. 2007-C. Persistent fault-tolerance for divide-and-conquer applications on the grid. In *CoreGRID Symposium*. Rennes, France, 230–237.

Received October 2008