

# Satin++: Divide-and-Share on the Grid

Gosia Wrzesinska, Jason Maassen, Kees Verstoep, Henri E. Bal

*Dept. of Computer Science*

*Vrije Universiteit Amsterdam*

*De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*

*{gosia, jason, versto, bal}@cs.vu.nl*

**Keywords:** divide-and-conquer, shared data, consistency models, high-performance applications, GRID

**Technical area:** parallel, cluster and GRID computing

## Abstract

*Divide-and-conquer is a popular and effective paradigm for writing grid-enabled applications. It has been shown to perform well in environments with high network latencies and dynamically changing numbers of processors. However, an important disadvantage of the divide-and-conquer paradigm is its limited applicability due to the lack of a shared data abstraction. We propose a divide-and-share model: the divide-and-conquer model extended with shared objects. Shared objects implement a relaxed consistency model called guard consistency. We have implemented Satin++: a framework for writing divide-and-share applications. With Satin++ we implemented a number of applications including VLSI routing, N-body simulation and a SAT solver. We evaluate the performance of our model on the DAS-2 supercomputer and on the heterogeneous, wide-area Grid'5000 testbed and demonstrate that our applications can achieve high efficiencies on the Grid.*

## 1 Introduction

Divide-and-conquer is a popular and effective paradigm for writing parallel, grid-enabled applications [9, 28, 45]. Divide-and-conquer is a generalization of the master-worker paradigm recommended by the Global Grid Forum as an efficient paradigm for grid programming [31]. Unlike master-worker, divide-and-conquer applications operate

by recursively dividing the problem into smaller subproblems. Divide-and-conquer applications can be easily parallelized by running different subproblems on different processors. Because of their hierarchical structure, divide-and-conquer algorithms can be executed with excellent communication locality on hierarchical platforms. In earlier work, we have shown that divide-and-conquer applications can run efficiently in environments with high network latencies and dynamically changing numbers of available processors [45, 48].

An important disadvantage of the divide-and-conquer paradigm, however, is its limited applicability due to the lack of global state. The only way of sharing data between tasks is by explicit parameter passing. This model is insufficient for many applications [24]. One class of such applications consists of programs that pass large data structures as parameters. With pure divide-and-conquer, those large parameters need to be copied each time a task is executed remotely, while copying the parameters once and reusing them later would be more efficient. Another class of applications consists of programs which need to share data between independent tasks. In pure divide-and-conquer, this form of data sharing is not possible. Branch-and-bound applications belong to this class. Sharing the best known solution between all the processors taking part in the computation allows pruning large parts of the search trees. Another example is game-tree search where a transposition table is shared to avoid evaluating the same position twice.

In this paper, we propose a divide-and-share model: the divide-and-conquer model extended with a shared data abstraction – shared objects. Implementing a shared data abstraction in a distributed system is a challenging problem. Providing a strong form of consistency (e.g., sequential consistency [30]) while maintaining high performance is infeasible even on tightly connected systems like clusters of workstations. In grid environments, it is even more difficult due to the high wide-area latencies and the fact that the set of processors may change during the computation. Many relaxed consistency models have been proposed (e.g., causal consistency [6], release consistency [22], PRAM [32]), but none of them are suitable for divide-and-conquer grid applications. As we will explain in more detail in Section 7, they are either too expensive to implement in grid environments (causal consistency), or do not fit the needs of our applications (release consistency, PRAM). Therefore, we designed a new, relaxed consistency model, which we call *guard consistency*. A programmer can define the consistency requirements of an application by means of *guard functions*. A guard function is associated with a divide-and-conquer task and defines whether the shared objects accessed by this task are in a consistent state. The runtime system allows the replicas to become inconsistent as long as the guards are satisfied. If a guard is unsatisfied, the runtime system brings the local replicas into a consistent state.

As a proof of concept we implemented the shared object model in the Satin [45] system. The extended system is called Satin++. We wrote several applications, such as an N-body simulation, VLSI routing, a SAT-solver and the Traveling Salesman Problem. We evaluated our model on a single cluster of the DAS-2 supercomputer and

the heterogeneous, wide-area Grid'5000 system [1]. Our VLSI routing application, for example, achieved over 80% efficiency on 120 processors in 3 clusters of Grid'5000.

The rest of this paper is organized as follows: in Section 2, we describe the divide-and-share model. In Section 3, we present the API and show a number of examples. Section 4 describes the implementation details. In Section 5, we discuss our experiences with programming divide-and-share applications. Section 6 contains a performance evaluation of our model. In Section 7, we discuss related work. Finally, we draw our conclusions in Section 8.

## 2 The divide-and-share model

The divide-and-share model combines the divide-and-conquer model with shared objects. Divide-and-conquer algorithms operate by subdividing the problem into subproblems (subtasks) and solving them recursively. 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 round, but the subtasks cannot share data with each other. In the divide-and-share model, tasks can share data using *shared objects*. Updates performed on a shared object are visible to all tasks.

Operations on shared objects are executed *atomically*. The runtime systems guarantees that shared object operations do not run concurrently with each other. The runtime system also guarantees that the shared object operations do not run concurrently with divide-and-conquer tasks. An operation performed by a task becomes visible to other tasks only when the system reaches a so-called *safe point*: when a task is creating (spawning) subtasks, when a task is waiting for its subtasks to finish, or when a task completes. Tasks can also explicitly poll for shared object updates. This makes the model clean and easy to use, as the programmer does not need to use locks and semaphores to synchronize access to shared data.

Shared objects are replicated on every processor taking part in the computation. Replication is implemented using an update protocol with function shipping: *write methods*, that is, methods that modify the state of the object, are forwarded to all processors which apply them on their local replicas. *Read methods*, that is, methods that do not change the state of the objects, are executed locally. Our approach differs from the *invalidation* protocols used in most DSMs [10, 11, 26, 25]. It has been shown, that for shared object models, update protocols are often more efficient than invalidation [8].

Our shared objects model provides a relaxed consistency model called *guard consistency*. Under guard consistency, the user can define the application consistency requirements using *guard functions*. Guard functions 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 returns *true* if those objects are in a correct state, or *false* otherwise. The runtime system allows replicas to become inconsistent as long as

guards are satisfied: the updates are propagated to remote replicas on a best effort basis. The runtime system does not guarantee that the updates will not be lost or duplicated. Updates may be applied in different order on different replicas. When a guard is unsatisfied, the runtime system invalidates the local replica of a shared object and fetches a consistent replica from another processor. This will be explained in more detail in Section 4.

### 3 Programming interface and examples

We have incorporated our shared object model into Satin – a Java-based framework for writing grid enabled divide-and-conquer applications. The extended system is called Satin++. In this section, we describe the programming interface of Satin++ and we use simple examples to demonstrate how to write parallel applications with our system.

Like Satin, Satin++ extends sequential Java with two divide-and-conquer primitives: *spawn* and *sync*. Rather than extending Java with new keywords, Satin++ uses *marker interfaces* to indicate that a certain method is *spawnable*, that is, needs to be considered for parallel execution. Spawnable methods must be declared in an interface that extends the *satins.Spawnable* marker interface. A class containing spawnable methods must extend the *satins.SatinObject* class and implement the interface containing spawnable methods. Figure 1 shows an implementation of the Traveling Salesman Problem (TSP) in Satin++. Interface *TspInterface* (line 21) extending *satins.Spawnable* defines the spawnable method *tsp()*. This method is defined in class *Tsp* (line 26). To synchronize with the spawned method invocations the *sync()* function defined in *satins.SatinObject* class is used (line 40, 50).

To define a shared object in Satin++, the programmer has to write a class that extends the special class *satins.so.SharedObject*. The programmer also needs to use the special interface *satins.so.WriteMethodsInterface* to mark write methods. Only methods marked as write methods are propagated to other replicas of the shared object. Read methods are executed only on the local replica. The TSP application (Figure 1) uses two shared objects: *DistanceTable* (line 17) and *Min* (line 5). *DistanceTable* is a static object - it does not change during the execution. Therefore all its methods are read methods (not shown). *Min* has two methods: *get()* and *set()*. *Set()* is declared in the *MinInterface* (line 2) which extends the special *satins.so.WriteMethodsInterface* and is therefore a write method. *Get()* is a read method.

For each spawnable function, the programmer may define a boolean guard function which defines what the state of shared objects should be for the spawnable function to run. A guard function should return *true*, if the shared objects are in a consistent state and the Satin task can be executed, and *false* otherwise. If the guard function returns *false*, the local replicas of shared objects need to be brought into a consistent state before the task can be executed.

The guard function must be defined in the same class as the spawnable function it guards. The name of the

```

1 interface MinInterface extends satin.so.WriteMethodsInterface {
2     public void set(int val);
3 }
4
5 final class Min extends satin.so.SharedObject implements MinInterface {
6     int val = Integer.MAX_VALUE;
7
8     public void set (int new_val) {
9         if (new_val < val) val = new_val;
10    }
11
12    public int get() {
13        return val;
14    }
15 }
16
17 final class DistanceTable extends satin.so.SharedObject {
18     [...]
19 }
20
21 public interface TspInterface extends satin.Spawnable {
22     public int tsp(int hops, byte[] path, int len, DistanceTable dist, Min min);
23 }
24
25 public class Tsp extends satin.SatinObject implements TspInterface {
26     public int tsp(int hops, byte[] path, int len, DistanceTable dist, Min min) {
27         int [] mins = new int [NRTOWNS];
28
29         if (len >= min.get()) return len; /*use the shared object to generate a cutoff*/
30
31         if (hops == NrTowns) {
32             min.set(len); /*update minimum*/
33             return len;
34         }
35
36         for (int city : dist.getCitiesNotOn(path)) {
37             /*spawn a new task for each city not on initial path*/
38             mins[i++] = tsp(hops+1, extendPath(path, city), len+dist.getDistTo(city), dist, min);
39         }
40         sync();
41
42         return getMinimum(mins); /*return the shortest route*/
43     }
44
45     public static void main(String args[]) {
46         DistanceTable dist = new DistanceTable(NRTOWNS);
47         Min min = new Min();
48         Tsp tsp = new Tsp();
49         int result = tsp.tsp(0, new byte[0], 0, dist, min); /*spawned*/
50         tsp.sync();
51         System.out.println ("Shortest path:" + result);
52     }
53 }

```

**Figure 1. Programming TSP in Satin++**

```

1  /*spawnable function*/
2  public LinkedList computeForces(byte[] nodeId, int iteration, Bodies bodies) {
3      LinkedList res[] = new LinkedList[8];
4      BodyTreeNode treeNode;
5
6      treeNode = bodies.findTreeNode(nodeId);
7      if (treeNode.children == null) {
8          return treeNode.computeForcesSeq(bodies); /*leaf node, do sequential computation*/
9      } else {
10         for (int i = 0; i < 8; i++) {
11             if (treeNode.children[i] != null) {
12                 /*spawn child tasks*/
13                 byte[] newNodeId = createNewNodeId(nodeId, i);
14                 res[i] = computeForces(newNodeId, iteration, bodies); /*spawned*/
15             }
16         }
17         sync();
18         return combineResults(res);
19     }
20 }
21
22 /*guard function*/
23 public boolean guard_computeForces(byte[] nodeId, int iteration, Bodies bodies) {
24     return (bodies.iteration+1 == iteration);
25 }
26
27 /*main function, in which the body positions are updated*/
28 public static void main(String[] args) {
29
30     BarnesHut barnesHut = new BarnesHut();
31     Bodies bodies = new Bodies(NUMBODIES);
32
33     for (int iteration = 0; iteration < ITERATIONS; iteration++) {
34         LinkedList results = barnesHut.computeForces(rootNodeId, iteration, bodies); /*spawn*/
35         sync();
36         bodies.update(results, iteration);
37     }
38 }

```

**Figure 2. Using a guard function to enforce shared object consistency in Barnes-Hut**

guard function must be “guard\_<spawnable\_function>”. It must have exactly the same parameter list as the spawnable function and return a boolean value. Therefore, it will have access to exactly the same shared objects and it can check their consistency. It will also have access to other parameters of the Satin++ task on which the definition of consistent state of the objects may depend. Since TSP does not need any consistency guarantees, we use a different application as an example: the Barnes-Hut N-body simulation. Figure 2 shows how a guard function is used Barnes-Hut. The positions of all bodies are stored in a shared object *bodies*. The application performs a number of iterations. At the end of each iteration, the root task updates the positions of the bodies (line 36). Before a processor starts executing a task belonging to a certain iteration, it has to make sure that it received the updates belonging to the previous iteration. This is done by means of the guard function (line 23).

## 4 Implementation

In this section we describe the implementation of Satin++. It consists of a runtime system and a *bytecode rewriter* that transforms the annotated sequential code into a parallel application. The bytecode rewriter generates the communication, load-balancing, and fault-tolerance code needed to execute the application on a Grid.

Satin++ was implemented on top of the Ibis communication library [46]. The core of Ibis is implemented in pure Java, without using any native code. The Satin++ runtime system also is written entirely in Java. The resulting system therefore is highly portable (due to Java’s “write once, run anywhere” property) allowing the software to run unmodified on a heterogeneous grid.

The divide-and-conquer primitives, *spawn* and *sync*, were implemented in the following way: for each spawned method invocation a datastructure containing the method’s parameters (*invocation record*) is created and inserted in the *work queue* of the processor. When the *sync()* call is reached, the processor executes tasks (invocation records) from the work queue. When the work queue is empty, load balancing is performed. Satin++ uses a load balancing strategy called Cluster-aware Random Work Stealing (CRS). With CRS, when a processor runs out of work, it tries to *steal* a task from another processor. Wide-area steals are overlapped with local, intra-cluster steals, so that wide-area latencies are hidden. Because CRS favors local steals over wide-area steals, it uses little wide-area bandwidth. Thus, CRS allows the applications to run efficiently over slow wide-area networks. More details on CRS can be found in [45].

Satin++ provides transparent fault-tolerance and malleability, that is, Satin++ can tolerate processors joining or leaving an on-going computation. A processor that joins the computation starts stealing tasks from other processors and it fetches replicas of shared objects from the processor it stole the first task from. When a processor leaves the computation or crashes, tasks computed by this processor are re-computed by the processors the tasks were stolen from, as described in [48].

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, also while a broadcast takes place, a processor can miss an update or receive it twice. The updates may also arrive in different order 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 period of time specified by the programmer. All updates accumulated during this period are forwarded at the end of it in one big message.

Guard consistency is implemented in the following way. Conceptually, a guard function is evaluated for each task (provided a guard function is defined). In fact, the Satin++ runtime system only needs to evaluate guards

for remote tasks which were obtained from other machines. This approach can be used because the strongest consistency model a divide-and-conquer application may need is DAG-consistency [13], where a child task must see updates that its parent has seen, as well as updates made by the parent. It may or may not see updates made by its siblings. Therefore, when a parent and child tasks 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. Thus, it is sufficient to check the consistency of shared objects (that is, execute guards) for remote tasks.

If a guard evaluates to false, the following actions are taken. First the system waits a certain amount of time for late updates to arrive. If after this time 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.

## 5 Applications

In this section, we will describe our experiences with programming grid applications using Satin++. We implemented four applications: the Traveling Salesman Problem, LocusRoute (a VLSI standard cell router), a Barnes-Hut N-Body simulation and a SAT solver. LocusRoute and Barnes-Hut are applications from the SPLASH suite [39, 47]. For each of the applications, we also discuss if it is possible to program it in a pure divide-and-conquer style (i.e., without the shared data abstraction). For the applications that can be implemented without a shared data abstraction, we show how using shared data can improve the performance and/or the ease of programming.

### 5.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) application computes the shortest path through a set of cities. Each city should be visited exactly once. We use a branch-and-bound algorithm to solve this problem. This algorithm recursively searches all possible paths. However, it can prune large parts of the search space by maintaining a global variable containing the length of the shortest path found so far. If the length of a partial path is bigger than the current minimal length, this path is not expanded further and a part of the search space is pruned.

Implementation of TSP in Satin++ is straightforward (see Figure 1). A new task is spawned for each partial path. The global minimum is implemented as a shared object. Also the static datastructure containing the distances between all cities is implemented as a shared object to reduce communication overhead. The shared objects do not need to be consistent to ensure the correctness of the algorithm. However, delays in update propagation for the minimum object may lead to search overhead.

Implementing TSP in a pure divide-and-conquer style, that is, without a shared data extension, is possible but inefficient. In pure divide-and-conquer, it is impossible to propagate the current best solution and therefore it is impossible to prune any part of the search space. This leads to enormous search overhead and slows down the execution of the program by a factor of 100 or even 1000, depending on the problem size and the number of processors used. Using the Younger Brothers Wait Concept (YBWC) [20] can improve matters. With YBWC, the second and subsequent subjobs are not spawned until the first subjob is executed. The result returned by the first subjob is passed to the subsequent subjobs and is likely to cause pruning in those subjobs. This technique reduces the search overhead but also decreases the amount of parallelism and causes load imbalance. Therefore, a pure divide-and-conquer version of TSP with YBWC optimization is still around 40% slower than the divide-and-share version.

## 5.2 LocusRoute

LocusRoute is a VLSI standard cell router. It routes wires between endpoints so as to minimize the total area of the layout. To minimize the area, the algorithm tries to route wires through regions (routing cells) that have few other wires running through them. It calculates a cost function for each route: the number of wires in the routing cells the route passes, and uses the route with the lowest cost. The total cost of the circuit is the sum of the number of wires running through each routing cell. Because the order of placement of the wires affects the total cost, the program performs a number of iterations. On every iteration except the first one, each wire is 'ripped out' and re-routed.

LocusRoute was implemented in Satin++ by recursively splitting the set of wires into two subsets. The subsets are routed in parallel. A shared object is used for storing the *cost array* - a data structure that keeps track of the number of wires running through each routing cell in the circuit. The data need not be consistent. However, a delay in update propagation may diminish the quality of the resulting circuit. Because the cost array is updated very often, LocusRoute uses message combining with a delay period of 100 milliseconds.

Implementing LocusRoute in a pure divide-and-conquer style is not possible. Without a shared data abstraction it is not possible to implement the cost array data structure on which the placement of wires depends.

## 5.3 Barnes-Hut N-body simulation

Barnes-Hut simulates the evolution of a large set of bodies under the influence of forces, for example gravitational or electrostatic forces. The evolution of N bodies is simulated in iterations of discrete time steps. If all pairwise interactions between bodies were computed, the complexity of the algorithm would be  $O(n^2)$ . The Barnes-Hut algorithm reduces this complexity by approximating far away groups of bodies by a single body at the center of

the mass of the group of bodies. This technique reduces the complexity of the algorithm to  $O(N \log N)$ . For the purpose of this optimization, the simulated bodies are organized in a tree structure that represents the space the bodies are in. The root node represents the whole space, its children the subspaces of this space, etc. For each body, the algorithm traverses the body tree. If a body tree node is far away from the given body, all bodies in this node are approximated with a large body in the center of the node and the force is computed. After computing forces for all bodies, the positions of the bodies and the body tree are updated.

In the Satin++ implementation of the algorithm, a new task is spawned for each node in the body tree. The task calculates forces for all bodies contained in this node. The positions of the bodies and the tree node are stored in a shared object, so that this enormous data structure does not have to be sent over the network each time a task is executed remotely. The shared object is updated at the end of each iteration. The application does have consistency constraints: the updates must be propagated to a processor before it can start working on the next iteration. The consistency of the data is ensured by means of guards, as described above (see Figure 2).

The Barnes-Hut application can be also implemented in a pure divide-and-conquer style. In that case, the positions of the bodies and the body tree have to be passed as task parameters. This means, however, that the body tree has to be sent over the network each time a task is stolen, which typically is thousands to tens of thousands times during the application run. This would cause significant overhead, as the body tree is a large data structure. The amount of data sent over the network can be decreased by passing only a *necessary tree* instead of the full body tree as a parameter. A necessary tree contains only those parts of the body tree that are needed for the bodies in the task's part of the tree. With this optimization it is possible to achieve performance similar to the performance of the divide-and-share version. However, this optimization requires a bigger programming effort, increasing the code size by 10%.

## 5.4 SAT solver

The satisfiability problem (SAT), that is the problem of deciding whether a given boolean formula is satisfiable, is an important NP-complete problem. The solution of a SAT problem is either a boolean variable assignment that makes the given formula true, or the result “unsatisfiable” meaning that no such assignment exists. Solving a SAT problem requires a systematic search over a potentially huge solution space. Various techniques have been developed to make this search more efficient for practical problems, but it is inherently difficult. Satisfiability solvers are commonly used in industry to verify the correctness of complex digital circuits, such as out-of-order execution units in modern processors.

The SAT solver used for this paper, is based on SAT4J [2], a reimplementaion in Java of MiniSAT [19]. Both MiniSAT and SAT4J are “industry strength” solvers, that are competitive with other state-of-the-art implemen-

tations. The solver uses a backtracking search that speculatively assigns boolean values to variables until the problem is satisfied or a conflict is encountered. Upon a conflict the solver backtracks. Parallelizing SAT4J with Satin++ was relatively easy. For each speculative assignment a task is spawned so that alternative assignments are evaluated in parallel.

A challenging issue in parallelizing SAT solvers arises from the fact that it is hard to predict how much execution time is needed to solve a spawned subproblem. It is possible for some subproblems that the time needed to execute the subproblem is so short that it will not amortize the cost of spawning it. Therefore, in our implementation we use the approach taken in the GridSAT solver [18]: each task first performs a certain amount of sequential search before splitting up the remaining search problem. This guarantees that only “hard” tasks will be split.

SAT solvers often implement an iterative strategy to go down the search tree. The purpose of this is to avoid spending too much time in very deep subtrees that might have been cut off more easily if an alternative branch was chosen earlier. In sequential SAT solvers, this can easily be implemented by choosing a certain bound on the total number of assignment conflicts found, and increasing that bound gradually by a certain factor. However, implementing a similar conflict bound with a parallel version is harder, since without communication, a particular branch does not know how many conflicts are generated in other branches, and how the conflicts add up globally. It is possible to make some assumption about the number of conflicts still allowed in a particular subbranch, but this can easily be over- or underestimated, leading either to more iterative restarts, or more searching in fruitless subtrees than the sequential version does. However, with the shared objects supported by Satin++, up-to-date knowledge about the global amount of conflicts remaining can be obtained almost trivially.

Other aspects that are currently implemented using shared objects are the pruning of subproblems in case a truth assignment is found by one of the searches, and an implementation of global learning [18]. In global learning, information about conflicting assignments learned locally in one branch of the search tree is made available to other branches in order to potentially cut off related subtrees. As reported elsewhere [12, 18], sharing these learned clauses can indeed potentially help, but also introduces some overhead that has to be earned back. A simple way to decrease the overhead is by restricting the use of global learning to clauses up to a certain length (in general, the shorter the learned clause, the higher its potential impact). It appears that a good maximal length for learned clauses is rather SAT problem dependent; currently we limit it to clauses of up to ten literals. Since knowledge gained by global learning is basically an additional source of information, it does not have to be implemented with traditional consistency models which would be much too costly on a wide-area distributed system.

It is possible to implement the SAT solver in the pure divide-and-conquer style. Such an implementation, however, is less efficient. The main reason for this inefficiency is that independent branches cannot share the global number of conflicts found, as described above. Also, global learning is not used in the pure divide-and-conquer

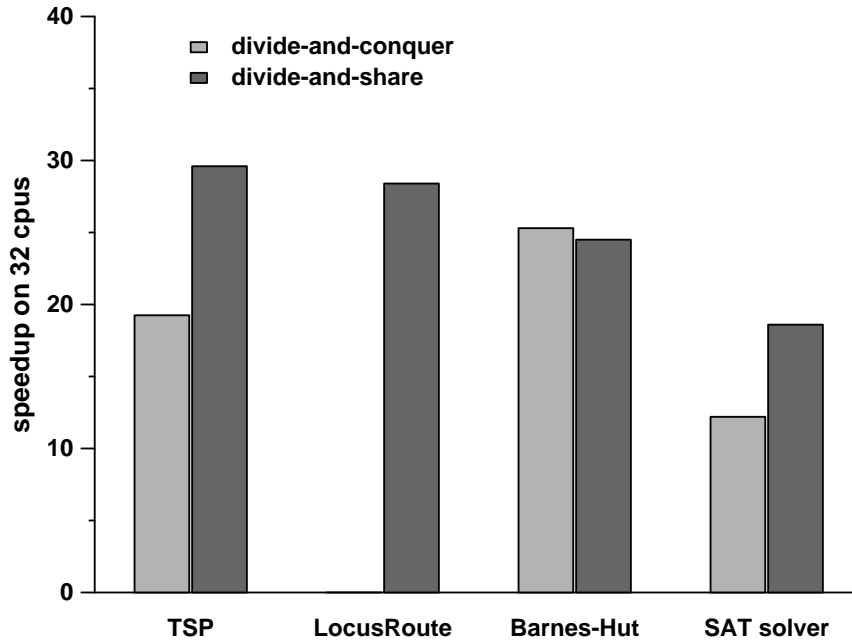


Figure 3. Speedups on 32 DAS-2 processors

version. However, global learning appears to have less influence on the performance of the solver on the SAT problem used by us. Finally, instead of using a shared object to notify other branches that a solution has been found and the computation should terminate, the special *abort mechanism* has to be used [44].

## 6 Performance evaluation

In this section we will evaluate the performance of Satin++. The first part of the evaluation was carried out on a single cluster of the DAS-2 supercomputer. To demonstrate that our model is also suitable for grid environments, the second part of our experiments is performed on the Grid'5000 testbed. Grid'5000 is a wide-area and heterogeneous system which currently consists of 7 clusters located across France.

In the first part of our experiments, we tested the performance of our four applications on a single DAS-2 cluster. For those applications which can be programmed in pure divide-and-conquer style, that is, without shared objects, we compared the performance of the divide-and-conquer version with the performance of the divide-and-share version. We always chose the most efficient divide-and-conquer version, that is, for TSP we chose the Young Brothers Wait version and for Barnes-Hut we chose the Necessary Tree version. We used 32 processors with 1-GHz Pentium-IIIs and 1GB RAM. The processors are connected with Myrinet [16]. Figure 3 shows the speedups the applications achieved on 32 processors of the DAS-2 cluster. All divide-and-share applications achieve good

location	processor	cache size
Sophia	AMD Opteron 246 2 GHz	1024 KB
Rennes	Intel Xeon 2.4 GHz	512 KB
Bordeaux	AMD Opteron 248 2.2 GHz	1024 KB
Orsay	AMD Opteron 246 2 GHz	1024 KB

**Table 1. Processor configurations in the Grid’5000 testbed**

	nr cpus Sophia	nr cpus Rennes	nr cpus Bordeaux	nr cpus Orsay	total nr cpus	normalized nr cpus	speedup	grid efficiency	single cluster efficiency
TSP	50	40	30	-	120	115	97	86%	86%
LocusRoute	50	40	30	-	120	94	81	84%	95%
SAT solver	-	32	40	40	112	98	54	56%	60%

**Table 2. Test results the Grid’5000 testbed**

speedups. The divide-and-share versions of TSP and SAT solver perform much better than their divide-and-conquer versions. For Barnes-Hut, the performance of both versions is comparable. LocusRoute cannot be programmed without shared objects.

The second part of the experiments we carried out on the Grid’5000 system. This experiment shows how our model behaves in a *real* grid environment. Grid’5000 is a wide-area system with latency between clusters ranging from 4 to 10 milliseconds and bandwidth ranging from 200 to 1000 Mb/s. Grid’5000 is also heterogeneous: it contains machines with different architectures and different speeds. Table 1 lists the configuration of the Grid’5000 processors we used. We tested three out of four applications: TSP, LocusRoute and SAT solver. The first two we ran on a testbed consisting of 3 clusters in Sophia Antipolis, Rennes, and Bordeaux. By the time we were ready to test the SAT solver, the Sophia cluster was down. Therefore, the SAT solver we tested on clusters in Orsay, Rennes and Bordeaux. Unfortunately, we did not have enough time to also test the Barnes-Hut application. However, we do not expect that this application will achieve good performance in a grid environment because of its high communication-to-computation ratio.

Since Grid’5000 is a heterogeneous system, it is hard to define efficiency. We used the following methodology: for each of the three applications we ran a smaller benchmark problem on a single processor in each cluster. This way we could compute the relative speeds of the processors in each cluster. The differences in processors speeds are application specific, they depend on the type of application (e.g., if it is cpu- or memory- bound). We normalized the runtimes of the benchmark problems relative to the runtime on a single processor of the Sophia (for TSP and LocusRoute) or Orsay (for SAT solver) cluster. Next, we computed the normalized number of cpus. Those numbers are listed in Table 2. We used the normalized number of cpus to define the efficiency:

$$efficiency = \frac{t_{single\_cpu}}{t_{all\_cpu}} * \frac{1}{normalized\_nr\_cpus} * 100\%$$

	nr tasks executed	nr tasks stolen	nr shared object invocations	amount of data sent for work stealing	amount of data sent for object updates
TSP	2 000 000	7 000	48	20 Mb	8 Kb
LocusRoute	160 000	4 000	120 000	84 Mb	28 Mb
SAT solver	160 000	5 000	130 000	2.1 Gb	15 Mb

**Table 3. Statistics for Grid’5000 runs**

All three applications achieve high efficiencies on the Grid’5000 testbed. We compared these efficiencies with efficiencies of the same applications on a single cluster. The number of processors we used in the single cluster was the same as the normalized number of processors. For all three applications, the efficiencies in the wide-area setting were very close to the efficiencies on a single cluster. Table 2 lists those efficiencies. The amount of data sent by each application is shown in Table 3.

## 7 Related Work

Few other divide-and-conquer frameworks provide shared data abstractions. Cilk [15] provides a shared memory abstraction for divide-and-conquer computations on the Connection Machine CM-5. Cilk’s shared memory implements a relaxed consistency model called DAG-consistency. DAG-consistency is defined on the DAG of threads that make up a parallel computation. Unlike with sequential consistency, different threads are allowed to observe different sequences of writes, but those sequences must respect the dependencies of the DAG. In divide-and-conquer terms: a task must see all writes performed by its ancestor tasks but does not need to see writes of its siblings. DAG-consistency is implemented using the Backer algorithm [14] which performs well on a tightly coupled machine like the CM-5, but is not suitable for wide-area systems. Moreover, Cilk’s shared memory was developed for pure divide-and-conquer applications which use large data structures (such as Barnes-Hut) and not for applications that need to share data between sibling tasks (such as TSP). Updates of shared data are passed only along the edges of the execution tree but not to sibling tasks. Only sibling tasks that execute on the same machine can see each other’s updates. Therefore Cilk’s shared memory is unsuitable for applications such as TSP and SAT solver with learned clause sharing. Peng et al. [37] noticed this shortcoming of Cilk and implemented SilkRoad – an extension to Cilk that provides global user locks and shared memory with lazy release consistency [27]. SilkRoad was designed to run in a single cluster environment and is not suitable for wide-area grid environments. Javelin [35] is a framework for writing branch-and-bound applications. Branch-and-bound is similar to divide-and-share but more restrictive. Javelin provides a very limited possibility of sharing data between tasks for bound propagation. All tasks are sharing the current bound, usually an integer or real number, but Javelin allows it to be of any object type. When a task finds a new bound, it broadcasts it to all processors. There are no consistency guarantees.

The function shipping approach to object replication was inspired by Orca [8]. Orca provides sequential con-

sistency which is implemented using totally ordered broadcast. Orca applications achieve good performance on a single cluster, but because of the restrictive consistency model, Orca is not suitable for wide-area systems. RepMI [33] offers object replication in Java with sequential consistency. The API of RepMI is similar to our API: the programmer uses inheritance and marker interfaces to define replicated objects. The read and write methods, however, are distinguished automatically by the compiler and runtime system. RepMI achieves good performance on a cluster of machines connected with Myrinet [16]. Similar to Orca, however, its restrictive consistency model makes it unsuitable for wide-area computing. Also, read/write analysis, thread scheduling, and indirection in accessing replicated objects adds overhead which is not justified for applications that do not need strong consistency.

Many systems try to achieve better performance by relaxing consistency requirements and tailoring them to the application needs. One approach is to provide multiple consistency levels for the programmer to choose from [4, 5, 21, 23, 34, 36, 38]. Those approaches, however, are inflexible: the requirements of an application often lie “in between” the proposed consistency models. Another approach is to let the programmer numerically *quantify* the amount of inconsistency the application can tolerate, for example by providing the maximum staleness or numerical error of data. Many variations of this approach have been proposed [7, 17, 29, 40, 41, 42, 43]. The most general solution was proposed by Yu and Vadhat in [49]. The authors describe TACT, a system in which a programmer defines the application’s consistency requirements by defining abstract consistency units (*conits*) and specifying how each operation influences conits and what the state of conits should be for the operation to execute. This solution is similar to our approach in the sense that consistency requirements are specified on a per-access basis, only the granularity is smaller than in our system (per single read or write access rather than per task). Also, in our system, the consistency requirements concern only the local replica while in TACT they are global: they specify what should be the maximal difference between any two replicas. TACT, however, was not designed with high-performance applications in mind, but applications such as airline reservation systems. The protocols used in TACT are heavy-weight and less suitable for high performance applications.

Also in the field of concurrency control in databases there were many efforts to exploit application semantics to maximize performance. In [3], Agrawal et al. propose to specify application’s consistency requirements by means of *guarded actions*. In this model, a database consists of objects with high-level operations defined on them. Users access the database through programs in which each operation is preceded by a *consistency assertion* stating what the state of the database objects should be for the operation to be performed. A guarded action is a pair consisting of a consistency assertion and an operation. This approach is similar to ours. However, this consistency model is used for *concurrency control* and not for *replica control* as in our case. Concurrency control algorithms schedule concurrent access to data on a single machine to ensure consistency whereas replica control deals with maintaining

consistency across multiple replicas. Replica control is harder because the information needed to make an optimal decision may not be available locally.

## 8 Conclusions

We presented a divide-and-share programming model which combines the divide-and-conquer paradigm with a shared data abstraction – shared objects. The new divide-and-share model has a broader applicability than the pure divide-and-conquer model.

Shared objects implement a new consistency model, guard consistency, designed especially for grid-enabled divide-and-conquer applications. Under guard consistency, the programmer can define the consistency requirements of the application using guard functions associated with divide-and-conquer tasks. A guard function specifies what the status of an object should be for a task to execute correctly. The runtime system allows replicas of shared objects to become inconsistent as long as guards are satisfied. When a guard is unsatisfied, the system brings the local replica into a consistent state. The guard consistency model is easy to use and allows for efficient implementation in grid environments.

We implemented a number of divide-and-share applications using the Satin++ divide-and-share framework: Locus Route (VLSI routing), SAT solver, Barnes-Hut (N-body simulation) and Traveling Salesman Problem. We evaluated the performance of our applications on the DAS-2 supercomputer and showed that they achieve good speedups on a single cluster. To demonstrate that our model is suitable for *real* Grid environments, we tested it on the wide-area, heterogeneous Grid'5000 testbed and showed that an application using shared data can achieve an efficiency of more than 80% on the Grid.

Satin++ is an excellent platform for developing grid-enabled applications. It provides a clean and easy to use programming model and allows rapid development of grid applications. Writing a parallel application with Satin++ typically requires adding only a few lines of code to the sequential code: annotating spawnable methods and write methods of the shared objects. The Satin++ runtime system then executes the application with high efficiency in wide-area grid environments.

## Acknowledgments

This work was carried out in the context of Virtual Laboratory for e-Science project ([www.vl-e.nl](http://www.vl-e.nl)). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). We would also like to thank the Grid'5000 community for allowing us to use their testbed.

## References

- [1] Grid'5000 website: <http://www.grid5000.fr>.
- [2] SAT4J website: <http://www.sat4j.org>.
- [3] Divyakant Agrawal, Amr El Abbadi, and Ambuj K. Singh. Consistency and orderability: semantics-based correctness criteria for databases. *ACM Trans. Database Syst.*, 18(3):460–486, 1993.
- [4] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj K. Singh. Evaluating Weak Memories with Maya. Technical Report TRCS93-23, University of California at Santa Barbara, 1994.
- [5] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj K. Singh. Mixed Consistency: A Model for Parallel Programming (Extended Abstract). In *Symposium on Principles of Distributed Computing*, pages 101–110, 1994.
- [6] Mustaque Ahamad, James E. Burns, Phillop W. Hutto, and Gil Neiger. Causal memory. In *5th International Workshop on Distributed Algorithms*, pages 9–30, October 1991.
- [7] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.
- [8] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Cerieel Jacobs, Koen Langendoen, Tim Ruehl, and M. Frans Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1), February 1998.
- [9] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, pages 165–172, Connemara, Ireland, September 1996.
- [10] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, 1990.
- [11] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of COMPCON 1993*, pages 528–537, 1993.
- [12] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. A Universal Parallel SAT Checking Kernel. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '03)*, volume 4, pages 1720–1725, Las Vegas, Nevada, USA, 2003. CSREA Press.

- [13] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-Consistent Distributed Shared Memory. In *10th International Parallel Processing Symposium (IPPS '96)*, pages 132–141, Honolulu, Hawaii, April 1996.
- [14] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-Consistent Distributed Shared Memory. In *10th International Parallel Processing Symposium (IPPS '96)*, pages 132–141, Honolulu, Hawaii, USA, April 1996.
- [15] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.
- [16] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles E. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
- [17] DeQuing Chen, Chunqiang Tang, Brandon Sanders, Sandhya Dwarkadas, and Michael L. Scott. Exploiting High-level Coherence Information to Optimize Distributed Shared State. In *Proc. of the 9th ACM Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, June 2003.
- [18] Wahid Chrabakh and Rich Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, Washington, DC, USA, 2003. IEEE Computer Society.
- [19] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, 2003. Springer.
- [20] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Game Tree Search on a Massively Parallel System. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess7*, pages 203–218, Maastricht, The Netherlands, 1994. University of Limburg.
- [21] Roy Friedman. Implementing hybrid consistency with high-level synchronization operations. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 229–240. ACM Press, 1993.
- [22] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings*

of the 17th Annual International Symposium on Computer Architecture, pages 15–26, New York, NY, USA, 1990. ACM Press.

- [23] Abdelsalam Heddaya and Himanshu Sinha. Distributed Parallel Computing in Mermera: Mixing Noncoherent Shared Memories. Technical Report 1996-005, Boston University, 7 1996.
- [24] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, MIT Departement of Electrical Engineering and Computer Science, 1996.
- [25] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 213–226, 1995.
- [26] Pete Keleher, Alan L. Cox, Sanhya Dwarkadas, and Willy Zwaenepoel. ThreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, 1994.
- [27] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.
- [28] Yoshiaki Futakata Kento Aida, Wataru Natsume. Distributed Computing with Hierarchical Master-Worker Paradigm for Parallel Branch and Bound Algorithm. In *3rd International Symposium on Cluster Computing and the Grid*, pages 156–163, Tokyo, Japan, May 2003.
- [29] Narayanan Krishnakumar and Arthur J. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Trans. Database Syst.*, 19(4):586–625, 1994.
- [30] Leslie Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997.
- [31] Craig Lee, Satoshi Matsuoka, Domenico Talia, Alan Sussman, Matthias Mueller, Gabrielle Allen, and Joel Saltz. A Grid Programming Primer. Global Grid Forum, August 2001.
- [32] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.
- [33] Jason Maassen. *Method Invocation Based Communication Models for Parallel Programming in Java*. PhD thesis, Vrije Universiteit Amsterdam, 2003.

- [34] Ronald G. Minnich. *Mether: A Memory System for Network Multiprocessors*. PhD thesis, University of Pennsylvania, 1991.
- [35] Michael O. Neary and Peter Cappello. Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In *Proc. ACM Java Grande/ISCOPE Conference*, pages 56–65, November 2002.
- [36] Gang Peng, Srikant Sharma, and Tzi cker Chiueh. Mixed Consistency Model: Meeting Data Sharing Needs of Heterogeneous Users. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 219–228, 2005.
- [37] Liang Peng, Weng Fai Wong, Ming Dong Feng, and Chung Kwong Yuen. SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters. In *IEEE International Conference on Cluster Computing (Cluster2000)*, pages 243–249, November 2000.
- [38] Karim R. Mazouni Rachid Guerraoui, Benoit Garbinato. The GARF Library Of DSM Consistency Models. In *Proceedings of the 6th ACM SIGOPS European Workshop*, pages 51–56, 1994.
- [39] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News*, 20(1):5–44, 1992.
- [40] Aman Singla, Umakishore Ramachandran, and Jessica Hodgins. Temporal notions of synchronization and consistency in Beehive. In *SPAA '97: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–220. ACM Press, 1997.
- [41] Sai Susarla and John Carter. Khazana: A flexible wide-area data store. Technical Report UUCS03-020, University of Utah, School of Computer Science, 2003.
- [42] Sai Susarla and John Carter. Flexible Consistency for Wide Area Peer Replication. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 199–208, 2005.
- [43] Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed consistency for shared distributed objects. In *PODC '99: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 163–172. ACM Press, 1999.
- [44] Rob V. van Nieuwpoort. *Efficient Java-Centric Grid Computing*. PhD thesis, Vrije Universiteit Amsterdam, 2003.
- [45] Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Satin: Simple and Efficient Java-based Grid Programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2004.

- [46] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Cerial Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency & Computation: Practice & Experience*, 17(7–8):1079–1107, 2005.
- [47] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, New York, NY, USA, 1995. ACM Press.
- [48] Gosia Wrzesinska, Rob V. van Nieuwpoort, Jason Maassen, and Henri E. Bal. Fault-tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. In *International Parallel and Distributed Processing Symposium*, Denver, Colorado, USA, April 2005.
- [49] Haifeng Yu and Amin Vadhat. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *ACM Transactions on Computer Systems*, 20(3):239–282, 2002.