Satin: Efficient Parallel Divide-and-Conquer in Java

Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal

Dept. of Mathematics and Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands
rob@cs.vu.nl, kielmann@cs.vu.nl, bal@cs.vu.nl
http://www.cs.vu.nl/albatross/

Abstract. Satin is a system for running divide and conquer programs on distributed memory systems (and ultimately on wide-area metacomputing systems). Satin extends Java with three simple Cilk-like primitives for divide and conquer programming. The Satin compiler and runtime system cooperate to implement these primitives efficiently on a distributed system, using work stealing to distribute the jobs. Satin optimizes the overhead of local jobs using on-demand serialization, which avoids copying and serialization of parameters for jobs that are not stolen. This optimization is implemented using explicit invocation records. We have implemented Satin by extending the Manta compiler. We discuss the performance of ten applications on a Myrinet-based cluster.

1   Introduction

There is currently much interest in divide and conquer systems for parallel programming [2, 6, 10, 11, 15]. Divide and conquer style programs start by dividing the problem into subproblems. Each subproblem is then recursively solved, again by dividing it into smaller subproblems. An example of such a system is Cilk [6], which extends C with divide and conquer primitives. Cilk runs these annotated C programs in parallel, in an efficient way, but is mainly targeted at shared memory machines. Atlas [2], an extension of Java, is a divide and conquer system designed for distributed memory machines. Its primitives have a high overhead, however, so it runs fine-grained parallel programs inefficiently.

In this paper, we introduce a new system, called Satin, which also is a divide and conquer system based on Java. Satin (as the name suggests) was inspired by Cilk. In Satin, single-threaded Java programs are parallelized by annotating methods that can run in parallel. Our ultimate goal is to use Satin for distributed supercomputing applications on hierarchical wide-area clusters (e.g., the DAS [8]). We think that the divide and conquer model will map efficiently on such systems, as the model is also hierarchical. In this paper, however, we focus on the implementation of Satin on a single local cluster computer. In contrast to Atlas, Satin is designed as a compiler-based system in order to achieve high performance. Satin is based on the Manta [12] native compiler, which supports highly efficient serialization and communication. Parallelism is achieved in

Satin by running different spawned method invocations on different machines. The system load is balanced by *work stealing*.

One of the contributions we make in the paper is the use of explicit invocation records, to enable the on-demand serialization of parameters to spawned method invocations. This optimization is possible because of Satin's parameter semantics. Furthermore, we demonstrate that Satin can run efficiently on distributed memory machines. Satin also cleanly integrates divide and conquer programming into Java, and solves some problems that are introduced by this integration (e.g., by garbage collection).

2   The Programming Model

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.

2.1   Spawn and Sync

We have introduced three new keywords to the Java language, `spawn`, `sync`, and `satin`. The `spawn` keyword must be placed in front of a method invocation, which will then be called a *spawned method invocation*. When `spawn` is placed in front of a method invocation, conceptually a new thread is started which will run the method. (The implementation of Satin, however, eliminates thread creation altogether.) The spawned method will run concurrently with the method that executed the `spawn`. In Satin, spawned methods always run to completion. 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 `satin` modifier must be placed in front of a method declaration, if this method is ever to be spawned.

To illustrate the use of `spawn` and `sync`, an example program is shown in Fig. 1. This code fragment calculates Fibonacci numbers, and is a typical example of a divide and conquer program. Note that this is a benchmark, and *not* a suitable algorithm for efficiently calculating the Fibonacci numbers. The program is parallelized just by inserting `spawn` in front of the recursive calls to *fib*. The two subproblems will now be solved concurrently. Before the results are combined, the method must wait until both subproblems have actually been solved, and have returned their value. This is done by the `sync` operation. A well known optimization in parallel divide and conquer programs is to make use of a threshold on the number of spawns. When this threshold is reached, work is executed sequentially. This approach can easily be programmed using Satin.

Satin does not provide shared memory, because this is hard to implement efficiently on distributed memory machines. Moreover, our ultimate goal is to run Satin on wide-area systems, which clearly do not have shared memory. The only way of communicating between threads is via the parameters and the return

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

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

        return x + y;
    }

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

Fig. 1. A Satin example: Fibonacci.

value. The parameter passing mechanism, as described in Sect. 2.2, assures that all data that can be accessed via parameters will be sent to the machine that executes the spawned method invocation.

2.2   The Parameter Passing Mechanism

Because Satin does not provide shared memory, objects passed as parameters in a spawned call to a remote machine will not be available on that machine. Therefore, Satin uses *call-by-value* semantics when the runtime system decides that the method will be spawned remotely. This is semantically similar to the standard Java Remote Method Invocation (RMI) mechanism [17]. Call-by-value is implemented using Java's *serialization* mechanism, which provides a *deep copy* of the serialized objects [16]. For instance, when the first node of a linked list is passed as an argument to a spawned method invocation (or a RMI), the entire list is copied.

It is important to minimize the overhead for work that does not get stolen and is executed by the machine that spawned the work, as this is the common case. For example, in almost all applications we have studied so far, at most 1 out of 400 jobs gets stolen. Because copying all parameter objects (i.e., using call-by-value) in the local case would be prohibitively expensive, parameters are passed by reference when the method invocation is local. Therefore, the programmer cannot assume either call-by-value or call-by-reference semantics for `satin` methods (normal methods are unaffected and have the standard Java semantics). It is therefore erroneous to write Satin methods that depend on the parameter passing mechanism. (A similar approach is taken in Ada for parameters of a structured type.)

An important characteristic of Satin is that when the extensions `satin`, `spawn`, and `sync` are removed, a sequential standard Java program remains.

This program produces the same result as the parallel Satin program. This always holds, because Satin does not specify the parameter passing mechanism. Using call-by-reference in all cases (as normal Java does) is thus correct.

3   The Implementation

The large majority of jobs will not be stolen, but will just run on the machine the jobs were spawned on. 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 executed or stolen, not when the work is generated. To be able to defer this important decision, Satin's runtime system uses invocation records, which will be described below. 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. [13].

When a program executes a `spawn`, Satin redirects the method call to a stub. This stub creates an *invocation record* (see Fig. 2), describing the method to be invoked, the parameters that are passed to the method, and a reference to where the method's return value has to be stored. For primitive types, the value of the parameter is copied. For reference types (objects, arrays, interfaces), only a reference is stored in the record. In the example of Fig. 2, a `satin` method is invoked with an integer, an array, and an object as parameters. The integer is stored directly in the invocation record, but for the array and the object, references are stored, to avoid copying these data structures. The compiler allocates space for a counter on the stack of all methods executing `spawn` operations. This counter is called the *spawn counter*, and counts the number of pending spawns, which have to be finished before this method can return. The *address* of the spawn counter is also stored in the invocation record.


```
SATIN int foo(int i, double[] d, Object o);
int result = SPAWN foo(i, d, o);
```
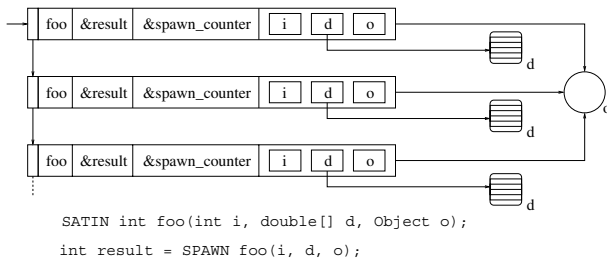
Fig. 2. Invocation records in the job queue.

The stub that builds an invocation record for a spawned method invocation is generated by the Manta compiler, and is therefore very efficient, as no runtime type inspection is required. From an invocation record, the original call can

be executed by pushing the value of the parameters (which were stored in the record) onto the stack, and by calling the Java method.

The invocation record for a `spawn` operation is stored in a queue. The spawn counter (located on the stack of the invoking method) is incremented by one, indicating that the invoking method now has a pending spawned method invocation. The invoking method may then continue running. After the spawned method invocation has eventually been executed, its return value will be stored at the return address specified in the invocation record. Next, the spawn counter (the address of which is also stored in the invocation record) will be decremented by one, indicating that there now is one less pending `spawn`. The `sync` operation executes work stored in the job queue, and waits for the spawn counter to become zero. When this happens, there are no more pending spawned method invocations, so the method may continue.

Serialization is Java's mechanism to convert objects into a stream of bytes. This mechanism always makes a deep copy of the serialized objects: all references in the serialized object are traversed, and the objects they point to are also serialized. The serialization mechanism is used in Satin for marshaling the parameters to a spawned method invocation. Satin implements serialization on demand: the parameters are serialized only when the work is actually stolen. In the local case, no serialization is used, which is of critical importance for the overall performance. In the Manta system, the compiler generates highly-efficient serialization code. For each class in the system a so-called serializer is generated, which writes the data fields of an object of this class to a stream. When an object has reference fields, the serializers for the referenced objects will also be called. Furthermore, Manta uses an optimized protocol to represent the serialized objects in the byte stream. Manta's implementation of the serialization mechanism is described in more detail in [12].

The invocation records describing the spawned method invocations are stored in a double ended job queue. A Dijkstra-like protocol [6] is used to avoid locking in the local case. Satin registers the invocation records at the garbage collector, keeping parameter objects alive when they are referenced only via the invocation record, and not via a Java reference. (Otherwise, the garbage collector might free objects that are needed to execute the spawn operations, but are no longer referenced via the Java program). Satin's work stealing is implemented on top of the Panda communication library [1], primarily using Panda's message passing primitives. On the Myrinet network (which we use for our measurements), Panda is implemented on top of the LFC [3] network interface protocol. Satin uses the efficient, user level locks that Panda provides for protecting the work queue.

4   Performance Evaluation

We evaluated Satin's performance using ten application kernels. All measurements were performed on a cluster of the Distributed ASCI Supercomputer (DAS), each containing 200 MHz Pentium Pros that are locally connected by Myrinet. The machines run the Linux (RedHat 6.2) operating system.

4.1   Basic Spawn Overhead (Fibonacci)

An important indication of the performance of a divide and conquer system is the overhead of the parallel application on one machine, compared to the sequential version of the same application. The sequential version is obtained by filtering the keywords `satin`, `spawn`, and `sync` out of the parallel program. The difference in run times between the sequential and parallel programs is caused by the creation, the en-queuing and de-queuing of the invocation record, and the construction of the stack frame to call the Java method. Fibonacci gives an indication of the worst-case overhead, because it is very fine grained. Cilk is very efficient, the parallel Fibonacci program on one machine has an overhead of only a factor of 3.6 (measured on a Sun Enterprise 5000, with 167 MHz UltraSPARC processors) [6]. Atlas is implemented completely in Java and does not use on-demand serialization. Therefore its overhead is much worse, a factor of 61.5 (hardware unknown) [2]. The overhead of Satin is a factor 7.25, substantially lower than that of Atlas.

These overhead factors can be reduced at the application level by introducing threshold values that spawn only large jobs. For Fibonacci, for example, we tried a threshold value of 20 for a problem of size 45, so all calls to *fib(n)* with $n<20$ are executed sequentially, without using `spawn`. This simple change to the application reduced the overhead to almost zero. Still, $22.8 \cdot 10^6$ jobs were spawned, leaving enough parallelism for running the program on large numbers of machines. For Fibonacci, the threshold can easily be determined by the programmer, while for other applications this may be difficult or impossible. In general, however, it still is important to keep the sequential overhead of a divide and conquer system as small as possible, as it allows the creation of more fine-grained jobs and thus a better load balancing.

The overhead for the other applications we implemented is much lower than for the (original) Fibonacci program, as shown in Table 1. Here, t_s denotes the run time of the sequential program, t_1 the run time of the parallel program on one machine. In general, the overhead depends on the number of parameters to spawned methods. All parameters have to be stored in the invocation record when the work is spawned, and pushed on the stack again, when executed.
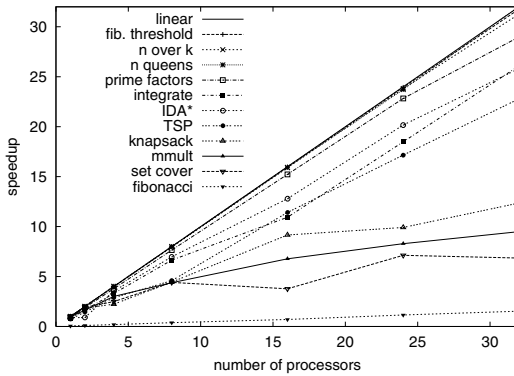
4.2   Parallel Applications

We ran ten applications on the DAS cluster, using up to 32 CPUs. Figure 3 shows the achieved speedups while Table 2 provides detailed information about the parallel runs. All speedup values were computed relative to the sequential applications, with the Satin-specific annotations removed from the code.

There is a strong correlation between measured speedup and the sequential overhead value, as already shown in Table 1: the lower the overhead, the higher the speedup we achieved. In Table 2 we compare the measured speedup with its upper bound, computed as the number of CPUs divided by the overhead on a single CPU. We also show the percentage of this upper bound as actually achieved by the measured speedup. This percentage is very high for most applications,

Table 1. Application overhead factors, times in seconds.

application	problem size	t_s	t_1	overhead
adaptive integration	0, 2.0E5, 1.0E-4	363.137	451.117	1.24
set covering problem	58, 29	1983.723	2071.333	1.04
fibonacci	41	65.517	475.133	7.25
fibonacci threshold	45	473.749	473.834	1.00
Iterative deepening A*	60	220.131	250.001	1.14
knapsack problem	28	1064.220	1150.016	1.08
matrix multiplication	1024 x 1024	137.982	141.742	1.03
n over k	34, 17	971.991	977.847	1.01
n-queens	15	1861.318	1909.942	1.03
prime factorization	1234567890	874.504	930.954	1.06
traveling sales person	17	982.864	1352.617	1.38

Fig. 3. Application speedups

Table 2. Parallel performance breakdown for 32 CPUs

application	overhead	speedup	#CPUs/ overhead	% max speedup	jobs	stolen
integrate	1.24	26.09	25.81	101 %	$63.3 \cdot 10^6$	2187
set cover	1.04	6.85	30.8	22.2 %	$51.0 \cdot 10^6$	579
fibonacci	7.25	4.31	4.4	98.0 %	$536 \cdot 10^6$	2906
fib. threshold	1.00	31.77	32.0	99.3 %	$22.8 \cdot 10^6$	1951
IDA*	1.14	25.82	28.1	91.9 %	$33.6 \cdot 10^6$	3866
knapsack	1.08	12.36	29.6	41.8 %	$33.5 \cdot 10^6$	417
mmult	1.03	9.49	31.1	30.5 %	$37.4 \cdot 10^3$	8567
n over k	1.01	31.27	31.7	98.6 %	$1.05 \cdot 10^6$	2458
n-queens	1.03	31.02	31.1	99.7 %	$2.47 \cdot 10^6$	3027
prime factors	1.06	28.98	30.2	96.0 %	$33.6 \cdot 10^6$	2609
tsp	1.38	22.79	23.2	98.2 %	$200 \cdot 10^6$	3026

denoting that Satin's communication costs are low. The actual percentage depends (like the sequential overhead) on the number of method parameters and their total serialized size. Table 2 also lists the total number of spawned jobs and the number of stolen jobs, which is less than 1 out of 400 for all applications, except for mmult. Because the number of stolen jobs is so small, speedups are mainly determined by sequential overhead. A good example is Fibonacci, which achieves 98% of the upper bound, but still has a low speedup due to the sequential overhead. Satin's sequential efficiency thus is important for the successful deployment of the divide and conquer paradigm for parallel computing.

Mmult does not get good speedups, because the problem size is small due to memory constraints, the run time on 32 cpus is only 14 seconds. Also, much data is transferred, in total over all CPUs, 31 MByte is sent per second. The mediocre speedup of knapsack, a very irregular application, is caused by load imbalance. The search space is pruned by both the weights and the values of the elements in the knapsack, making it difficult to estimate the grain size of a job. Therefore, many small jobs get stolen. The same holds for the set-covering problem, where a large percentage of the time is spent in finding work. On 32 nodes, only 1.2 percent of the work stealing attempts were successful.

5   Related Work

We discussed Satin, a divide and conquer extension of Java. Satin has been designed for distributed memory machines, while most divide and conquer systems use shared memory machines (e.g. Cilk [6]). There is also a version of Cilk for distributed memory machines, called CilkNOW [5], but it only supports functional Cilk programs (without shared memory), and it does not make a deep copy of the parameters to spawned methods. Our own previous work on parallel divide and conquer [9] was based on the C language while having similar restrictions as CilkNOW. Alice [7] and Flagship [18] offer a hardware solution for parallel divide and conquer programs (e.g., a reduction machine with one global address space for the parallel evaluation of declarative languages), while Satin is purely software based, and does not require, or provide, a single address space.

Mohr et al. [13] describe the importance of avoiding thread creation in the common, local case (lazy task creation). Satin also avoids creating threads in the local case, targeting distributed memory adds the problem of copying the parameters of parallel invocations (marshalling). Satin builds on the ideas of lazy task creation, and avoids both the starting of threads and the copying of parameter data by choosing a suitable parameter passing mechanism.

Another divide and conquer system based on Java is Atlas [2]. Atlas is not a Java extension, but a set of Java classes that can be used to write divide and conquer programs. While Satin is targeted at efficiency, Atlas was designed with heterogeneity and fault tolerance in mind, and aims only at a reasonable performance. Because Satin is compiler based, it is possible to generate code to create the invocation records, thus avoiding all runtime type inspection. The

Java classes presented in [11] can also be used for divide and conquer algorithms. However, they are restricted to shared-memory systems.

A compiler-based approach is also taken by Javar [4]. In this system, the programmer uses annotations to indicate divide and conquer and other forms of parallelism. The compiler then generates multi-threaded Java code, which runs on any JVM. Therefore, Javar programs run only on shared memory machines and DSM systems, whereas Satin programs run on distributed memory systems. Java threads impose a large overhead, which is why Satin does not use threads at all, but provides light weight invocation records. There are many other projects which use Java for parallel processing, for instance [14] and the work referenced in this paper.

6   Conclusions and Future Work

We have described our experiences in building a parallel divide and conquer system for Java, which runs on distributed memory machines. We have shown that an efficient implementation is possible by choosing convenient parameter semantics. An important optimization is the on-demand serialization of parameters to spawned method invocations. This was implemented using explicit invocation records. Our Java compiler generates code to create these invocation records for each spawned method invocation. We have also demonstrated that divide and conquer programming can be cleanly integrated into Java, and that problems introduced by this integration (e.g., through garbage collection) can be solved.

Our ultimate goal is to use Satin for distributed supercomputing applications on hierarchical wide-area clusters. We believe that divide and conquer programs will map efficiently on such systems, as the model is also hierarchical. Our intention is to carry out research on the scheduling of divide and conquer programs on hierarchical wide-area systems.

Acknowledgments

References

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

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

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

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

[5] R. Blumofe and P. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *In Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, Anaheim, California, 1997.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 207–216, Santa Barbara, California, July 1995.

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

[8] The Distributed ASCI Supercomputer (DAS). http://www.cs.vu.nl/das/.

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

[10] K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *SuperComputing '99*, November 1999.

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

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

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

[14] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, Nov. 1997.

[15] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 72–83, Atlanta, May 4-6 1999. Massachusetts Institute of Technology.

[16] Sun MicroSystems, Inc. Java (TM) Object Serialization Specification, 1996. ftp://ftp.javasoft.com/docs/jdk1.1/serial-spec.ps.

[17] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, pages 5–7, July–September 1998.

[18] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: A parallel architecture for declarative programming. In *15th IEEE/ACM Symp. on Computer Architecture*, pages 124–130, Honolulu, Hawaii, 1988. ACM SIGARCH newsletter,16(2).