The Satin Divide-and-Conquer System Programmer's Manual

The Ibis Group

May 16, 2008

1 Introduction

Satin is a parallel programming environment for divide-and-conquer parallelization, and master-worker parallelization. Satin extends Java with two simple primitives for divide-and-conquer programming: `spawn` and `sync`. The Satin byte-code rewriter and runtime system cooperate to implement these primitives efficiently on top of the IPL.

This manual explains how to program applications using Satin. For information on how to run applications using Satin, see the Satin user's manual, available in the Satin distribution in the `docs` directory.

2 Satin jobs

To use Satin, the programmer must label one or more methods in his class as Satin jobs. This is done by defining an interface that extends the Satin interface ibis.satin.Spawnable. For example:

```
package search;

interface Searcher extends ibis.satin.Spawnable {
   public int search(int a[], int from, int to, int val);
}
```

All methods that implement Searcher.search() will be Satin jobs, they are marked as spawnable. In general such a marker interface may contain an arbitrary number of methods. When a Satin program is compiled with the Satin compiler, methods marked as spawnable may be executed in parallel. A class that has spawnable methods must extend the special class ibis.satin.SatinObject. The result of a Satin job can only be used after the invocation of the sync method, which lives in ibis.satin.SatinObject. The sync method is guaranteed to only terminate after the earlier job invocations have terminated. Satin imposes one restriction on the type of parameters and return type of a Satin job: they must be of a basic type or must be serializable. Since a SatinObject is serializable, all its subclasses are serializable as well. This restriction ensures that a

```
package search;

import ibis.satin.SatinObject;

class SearchImpl1 extends SatinObject implements Searcher {
   public int search(int a[], int from, int to, int val) {
      for(int i = from; i < to; i++) {
         if (a[i] == val) return i;
      }
      return -1;
   }

   public static void main(String[] args) {
      SearchImpl1 s = new SearchImpl1();
      int a[] = new int[200];

      // Fill the array with random values between 0 and 100.
      for(int i=0; i<200; i++) {
         a[i] = (int) (Math.random() * 100);
      }

      // Search for 42 in two sub-domains of the array.
      // Because the search method is marked as spawnable,
      // Satin can run these methods in parallel.
      int res1 = s.search(a, 0, 100, 42);
      int res2 = s.search(a, 100, 200, 42);

      // Wait for results of the two invocations above.
      s.sync();

      // Now compute an overall result.
      int res = (res1 >= 0) ? res1 : res2;

      if(res >= 0) {
         System.out.println("found at pos: " + res);
      } else {
         System.out.println("element not found");
      }
   }
}
```

Figure 1: Parallel search with Satin.

Satin job can be stored and moved from one processor to another. Note that this means that all fields of a Satin object must be either serializable or transient.

As an example, Figure 1 shows an implementation of the search method. It just looks at a range of elements of the passed array, and tries if one of the elements equals val. If so, it returns the index of the element. If no matching element was found, -1

```
import ibis.satin.SatinObject;

class SearchImpl2 extends SatinObject implements Searcher {
   public int search(int a[], int from, int to, int val) {
      if (from == to) { // The complete array has been searched.
          return -1;    // The element was not found.
      }
      if (to - from == 1) { // Only one element left.
          return (a[from] == val) ? from : -1; // It might be the one.
      }

      // Now, split the array in two parts and search them in parallel.
      int mid = (from + to) / 2;
      int res1 = search(a, from, mid, val);
      int res2 = search(a, mid, to, val);
      sync();
      return (res1 >= 0) ? res1 : res2;
   }

   // main method as in previous figure
}
```

Figure 2: Divide-and-conquer parallel search with Satin.

is returned. Now, we can invoke search as is shown in the main method of Figure 1. In this case, we call search twice, once for the first half of the array, and once for the second half. If we compile the program with javac, the two search invocations will be done sequentially. The sync method in SatinObject does nothing. So we can run the program normally on any JVM. If we modify the class files using the Satin bytecode rewriter, however, the program will be converted to a parallel program. The two calls to search in the main class of Figure 1 will be executed in parallel if two JVMs are available. To run the code on more JVMs, the array can be split into more chunks. In general, an arbitrary number of invocations to Satin job methods may be done.

It is also allowed to recursively invoke job methods from job methods, as is shown in Figure 2. Here the search is recursively divided into smaller problems until a problem remains that is trivially handled. This model of programming is called divide-and-conquer. Satin has special grid-aware load-balancing algorithms built-in to run such programs efficiently on the grid, on any number of machines. This way, the entire search can be done in parallel on a large number of machines. The example shows how easy it is to write a real parallel grid application using Satin.

3 Exceptions and abort

A Satin job may throw exceptions in the usual way. This can be used to rapidly terminate a large set of jobs, e.g., when a search result was found. Terminating jobs that are no longer useful can be done with the abort method from the SatinObject class. Satin has its own exception type: `ibis.satin.Inlet`. By extending the `Inlet` class, the programmer can inhibit the generation of a stack trace, which is an expensive operation. The stack trace is usually not useful in the Satin context, because the exception is not an error, it is used to steer the search. Extending `Inlet` is optional, regular exceptions can also be used. Figures 3 and 4 change the search algorithm we used in the previous examples to use exceptions.

Note that in this version the search method doesn't return a value. It will throw a SearchResultFound object containing the result if the element was found, or it will terminate without throwing an exception if no result is found. This is also the reason the search method implements the Searcher3 interface instead of the Seacher interface. The declaration of the exception in the throws clause and the return type are the only differences. At the top level this exception is caught, and the other search jobs can then be terminated. They have become useless, because the element has already been found in the array. The parallel search can thus be stopped. We call the catch block in the main method that handles the result of the search an inlet.

The way of searching in Figures 3 and 4 is called speculative parallelism: the search speculatively starts on both two parts of the array simultaneously, even though we know from the start that a part of the search may be terminated as soon as the element is found. We might thus do more work than a sequential search. On the other hand, we might also do less work than the sequential version, for instance if the element is in the beginning of the second part of the array.

The inlet in main contains a return statement. This must be there, because the inlet runs in a new thread. The original main thread is still blocked in the sync statement. When the inlet returns, the main thread is unblocked, because both search jobs have finished: one threw an exception, the other has been aborted.

4 Satin job scheduling

Satin offers a choice of three different job scheduling strategies:

Random work-stealing. When looking for work, each Satin instance first examines its own job queue. When this job queue is empty, it randomly selects another Satin instance and takes the first job on its job queue. If that job queue is empty, it randomly selects another Satin instance, et cetera, et cetera. This strategy is the default, and has in the past been proven to be optimal, although that might seem counter-intuitive.

Cluster-aware random work-stealing. The word "cluster" refers to a group of computers that are connected to each other by means of a fast local network. In turn, clusters can be connected to each other, but usually the network between clusters is much slower, and/or has a much higher latency. The cluster-aware random work-stealing strategy is very similar to random work-stealing, except that each participant first tries to steal jobs from other participants in its own cluster (thus using the fast local network). It will only go to participants from other clusters if no participant on the same cluster has work.

```
package search;

import ibis.satin.SatinObject;
import ibis.satin.Inlet;

class SearchResultFound extends Inlet {
   int pos;

   SearchResultFound(int pos) {
      this.pos = pos;
   }
}

interface Searcher3 extends ibis.satin.Spawnable {
   public void search(int a[], int from, int to, int val)
      throws SearchResultFound;
}

class SearchImpl3 extends SatinObject implements Searcher3 {
   public void search(int a[], int from, int to, int val)
         throws SearchResultFound {

      if (from == to) { // The complete array has been searched.
         return;        // The element was not found.
      }
      if (to - from == 1) {    // Only one element left.
         if (a[from] == val) { // Found it!
            throw new SearchResultFound(from);
         } else {
            return; // The element was not found.
         }
      }

      // Now, split the array in two parts and search them in parallel.
      int mid = (from + to) / 2;
      search(a, from, mid, val);
      search(a, mid, to, val);
      sync();
   }

   ...
```

Figure 3: Speculative parallel search with Satin.

To determine to which cluster a participant belongs, Satin uses the parent of the location as obtained from the underlying Ibis IPL. The default for this is the domain part of the hostname. However, the location can be set using the `ibis.location` property.

```
   ...

   public static void main(String[] args) {
      SearchImpl3 s = new SearchImpl3();
      int a[] = new int[200];
      int res = -1;

      // Fill the array with random values between 0 and 100.
      for(int i=0; i<200; i++) {
         a[i] = (int) (Math.random() * 100);
      }

      // Search for 42 in two sub-domains of the array.
      // Because the search method is marked as spawnable,
      // Satin can run these methods in parallel.
      try {
         s.search(a, 0, 100, 42);
         s.search(a, 100, 200, 42);

         // Wait for results of the invocations above.
         s.sync();
      } catch (SearchResultFound x) {
         // We come here only if one of the two jobs found a result.
         s.abort(); // kill the other job that might still be running.
         res = x.pos;
         return; // return needed because inlet is handled in separate thread.
      }

      if(res >= 0) {
         System.out.println("found at pos: " + res);
      } else {
         System.out.println("element not found");
      }
   }
}
```

Figure 4: Speculative parallel search with Satin.

Master-worker This strategy is suitable for applications where all Satin jobs are generated on a single participant, the “master”. All other participants are “workers”, they obtain jobs from the master and execute them.

Making sure that all participants always can find work to do may need some tuning. Jobs must not be too small, because otherwise the mechanism for obtaining jobs, which may involve network traffic, is too expensive. On the other hand, there must be enough jobs to keep everybody busy. The `SatinObject` class has a boolean method `needMoreJobs()`, which indicates whether it would be useful to generate more jobs.

5 Other SatinObject methods

The `pause()` method pauses Satin's operation. When the application contains a large sequential part, this method can be called to temporarily pause Satin's internal load distribution strategies to avoid communication overhead during the execution of sequential code. To resume Satin's operation, the `resume()` method must be used.

6 Satin program arguments

The Satin system accepts the following parameters, which are passed to java as system property values.

`-Dsatin.closed` Only use the initial set of hosts for the computation; do not allow further hosts to join the computation later on.

`-Dsatin.stats` Display some statistics at the end of the Satin run. This is the default.

`-Dsatin.stats=false` Don't display statistics.

`-Dsatin.detailedStats` Display detailed statistics for every member at the end of the Satin run.

`-Dsatin.alg`=*algorithm* Specify the load-balancing algorithm to use. The possible values for *algorithm* are: RS for random work-stealing, CRS for cluster-aware random-work stealing, and MW for master-worker.

7 Satin Shared Objects

The divide-and-conquer model operates 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. Therefore, we have extended the model, and in the process renamed it ”divide-and-share”.

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*. Satin guarantees that shared object operations do not run concurrently with each other. Satin 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.

A shared object type must be marked as such by extending the special class `ibis.satin.SharedObject`. This class exports a single method

```
        public void exportObject();
```

which may optionally be invoked to inform Satin that it might be useful to broadcast the object to all participants in the run.

Write methods must be marked as such, by specifying them in an interface that extends the marker interface `ibis.satin.WriteMethodsInterface`. Note that it is up to the user to specify which methods are write methods. Here is a small example:

```
interface MinInterface extends ibis.satin.WriteMethodsInterface {
    public void set(int val);
}

final class Min extends ibis.satin.SharedObject
        implements MinInterface {
    private int val = Integer.MAX_VALUE;
    public void set(int newVal) {
        if (newVal< val) val = newVal;
    }
    public int get() { return val; }
}
```

Our shared object 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. 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 the updates will not be lost or duplicated. Updates may be applied in different order on different replicas. When a guard is unsatisfied, Satin invalidates the local replica of a shared object and fetches a consistent replica from another processor.

For each spawnable method, the programmer may define a boolean guard method which specifies what the state of shared object parameters should be before actually executing the spawned method. Such a guard function must be defined in the same class as the spawnable method it guards. The name of the guard function is `guard_`*spawnable_function*. It must have exactly the same parameter list as the spawnable method 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 consistency state of the objects may depend. Satin takes care of invoking this method at the right moment. Here is a small example:

```
/* A spawnable method. */
public Result compute(int iteration, Data data) {
    ...
}

/* Guard: make sure that data represents the right iteration. */
public boolean guard_compute(int iteration, Data data) {
    return data.iteration == iteration-1;
}
```

8 Further Reading

The Javadoc included in the `javadoc` directory has detailed information on all classes and their methods.

The Ibis web page `http://www.cs.vu.nl/ibis` lists all the documentation and software available for Ibis, including papers, and slides of presentations.

For detailed information on running a Satin application see the User's Manual, available in the docs directory of the Satin distribution.