Banyan: A framework for distributing tree-structured computation

15712: Advanced and Distributed Operating Systems, Spring 2010

Chris Martens   Michelle Mazurek   David Renshaw

ABSTRACT

We present a framework for writing distributed programs that solve tree-structured search problems. Our design decisions are made primarily based on our development of a parallel theorem prover, which demonstrates our target use case as a program that performs search with recursive branching. Using this example as a guidepost, we have strived to invent an interface general enough to capture common patterns in recursive programming, powerful enough to make efficient use of a cluster setting, and simple enough for a programmer to easily adapt sequential solutions to it. Our implementation addresses the low-level problems of resource allocation, communication, and scheduling, leaving the programmer to define only the work distribution strategy and what to do with values returned by recursive calls.

1. INTRODUCTION

In recent years, several frameworks have been developed to help programmers take advantage of parallel execution within a cluster without requiring them to understand all the nuances of task distribution, network communication, fault tolerance, and other issues. Many of these frameworks [1, 6] are tailored to inherently data-parallel applications such as web indexing and data mining. Our work explores a common class of applications that do not match well with these data-parallel solutions.

Consider, for example, a divide-and-conquer algorithm:

```
fun divide_conquer (problem) =
  let
    subproblems = divide(problem)
    subanswers = map divide_conquer subproblems
    answer = combine subanswers
  in
    answer
```

The definitions of `divide` and `combine` are application-specific, and they encompass a variety of search problems, such as decision trees, functional program interpretation, and theorem proving. Critically, such an algorithm calls itself *recursively* on its subproblems, generating an unknown amount more work – the branches of the tree may be very deep, or they may return immediately. The programmer does not know a priori how to divide the task into subtasks, so the MapReduce model does not apply. Our aim is to facilitize this kind of programming pattern in a distributed setting.

A natural way to parallelize such a problem is to assign a goal to one process, called a *worker*, and send all subproblems it creates to separate, possibly remote, processes. These processes need to communicate the fruits of their labor to the parent so that it can combine them into a cumulative answer. Also, the solution to a subproblem may influence the necessity or efficiency of others, so we may desire the ability for a parent to kill threads if they become unneeded and to propagate subtree information if it can benefit efficiency.

Some implementation questions immediately arise:

- When a process creates new subproblems, how should they be divided among workers?
- Should workers pull from a shared job queue or have jobs pushed to them by a parent worker?
- How do we decide when it is worth it to ship a job to another host?
- Once jobs are assigned, how should they be scheduled? What metric should determine their priority?

We answer these questions with our design and implementation of Banyan, a framework designed to facilitate distribution of tree-structured search applications. Banyan abstracts from the application programmer the need to consider where nodes are processed, as well as how to effectively balance them across the available machines in a cluster. The application programmer simply specifies how nodes are generated and coalesced, along with priority *tickets* that express the relative importance of various nodes. Banyan distributes these nodes across the cluster, taking into consideration the balance between the benefit of using more processing power and the cost of network transit overhead.

Our results consist of a working prototype written in Scala that achieves reasonable speedup proportional to the number of indivisble large chunks of work with the addition of extra processors. We argue that these results and the flexibility of our design suggest Banyan can be useful for a general class of tree-structured problems.

The rest of this paper is organized as follows: in Section 3 we discuss related work in the areas of parallelization frameworks and theorem proving; in Section 4 we give an overview of the programmer interface and system architecture; in Section 5 we describe the Scala methods the programmer implements and uses to interface with Banyan; in Section 6 we discuss details of Banyan's implementation; in Section 7 we describe our case study theorem proving application; in Section 8 we provide results from some preliminary evaluation of Banyan; in Section 9 we discuss future work improving and extending Banyan and conclude.

2. BACKGROUND: THEOREM PROVING

Theorem proving is our chosen avenue for investigating the distribution of tree-structured problems. To motivate our design decisions and familiarize the reader with some terminology, we now explain the idea of theorem proving in light of the parallelization problem.

2.1 And/or parallelism

A theorem prover takes as input a *goal*, i.e. a statement whose truth is unknown, and returns a proof or refutation of it. A proof of a goal G is a tree with root G and subtrees $T_1 \dots T_n$ which are proofs of goals $G_1 \dots G_n$ (called *subgoals* of G), where there exists an *inference rule* allowing one to conclude G from $G_1 \dots G_n$. It is important to the generality of our design that we wish to return *proofs*, not merely yes-or-no answers. In pseudocode:

```
fun prove (goal) : proof option =
  for each rule that applies:
    let
      proofs = map prove (premises(rule, goal))
    in
      if all proofs are non-null,
        return Proof(rule, proofs)
  return null
```

The structure of this solution is more intricate than the simple divide-and-conquer algorithm. In particular it is worth observing that the results of iteration over the rules is treated *disjunctively*: as soon as we know one of them works, we can return – whereas the results of the recursive calls to `prove()` are treated *conjunctively*: we require that all of them be valid. Both of these work divisions can be parallelized, and we classify them distinctly as *or-parallelism* and *and-parallelism.* Throughout this exposition we will refer to *and-nodes* and *or-nodes* to refer to nodes that should be and-parallelized and or-parallelized, respectively. These different types of parallelism are relevant when we consider scheduling priority: when an or-node returns with a proof, its parent should indicate that the other children are irrelevant, and similarly for a refutation of an and-node.

What inference rules exist is specific to the logical system for which the proof search is implemented, and different rules (or sets of rules) suggest different search strategies. For example, in *propositional* logic, the search space is finite: all goals have a finite number of finite sets of subgoals. In *first-order* logic, propositions can quantify over terms of which there may be (countably) infinitely many. In *linear logic* [4], where propositions behave like resources rather than knowledge in that they can be consumed, whether a proof will be found depends on how resources are divided among subgoals. Well-known solutions to proof search in these systems employ vastly different strategies that suggest a healthy range of use for us to target. We specifically implement an application for *dynamic logic* theorem proving, the details of which (and our reasons for choosing it) are described below.

2.2 Dynamic Logic Theorem Proving

Dynamic logic [11] is a language for formally specifying properties of hybrid systems, which have discrete and continuous components to reason about, such as control systems for cars or trains. The current state of the art in dynamic logic theorem proving is represented by KeYmaera [14], a theorem prover for differential dynamic logic which has been successful for several applications [15, 13]. KeYmaera builds a proof tree in a step-by-step manner without backtracking. To decide which rule to use on a particular proof obligation, KeYmeara applies a user-specified strategy which may take into account arbitrary global data from the proof tree. The implementation is sequential and due to some early design decisions it cannot be easily mapped to parallel workers. Moreover, *or-branching* is rather awkward to express due to the lack of backtracking.

Theorem proving for differential dynamic logic has several properties that make it particularly interesting to try to distribute. One, it is often the case that a proof find an *invariant.* There may infinitely many possibilities to choose from, and the proof just needs to find one that works—meaning that the recursive subcomputation based on that invariant succeeds. This means that we may have infinite or-parallelism.

Another important property is that the leaves of the proof search are expensive calls to a decision procedure for real arithmetic—essentially a black box that might take anywhere from a fraction of a second to many days to return. If there are several or-parallel branches working on their leaves, it is important for a theorem prover not to spend too much time bogged down in any one call of the procedure.

3. RELATED WORK

3.1 Distributed parallelization frameworks

Many systems have been designed to provide frameworks for parallelizing work and taking advantage of the resources provided by clusters of computers.

MapReduce is a higher-level framework that allows developers to distribute data-parallel computations across a cluster of machines. Computation is divided into two phases: the *map* phase that processes key-value pairs into an intermediate state and the *reduce* phase where related intermediate values are merged. One master node distributes map and reduce jobs among the other nodes, accounting for data locality when possible and restarting jobs in progress if a failure occurs. Although MapReduce provides a conceptually simple framework for parallelization, its restrictive data flow model means it is not appropriate for a task like ours, which involves tree-structured computation problems as well as significant data sharing among jobs.

An alternative parallelization framework is Pig Latin, a data processing language implemented on top of the Hadoop map-reduce framework [10]. Pig Latin provides a compromise position for handling large data sets between declarative SQL queries and a procedural map-reduce approach. Pig Latin contains query-like operations like *filter* and *group*, but because each operation is engineered to specify only one data transformation at a time, they can be combined in a procedural way more familiar to many experienced programmers, while providing a more flexible data flow to programmers than a basic map-reduce framework. Pig Latin programs are compiled into sets of map-reduce jobs that can then rely on Hadoop for parallelism, load-balancing, and fault-tolerance, at the cost of some additional overhead from shoehorning operations into this data flow. Although Pig Latin is more flexible than basic map-reduce, it still targets more strictly data-parallel applications than our framework will support.

Dryad is another parallelization framework that attempts to provide for more flexible data flow than MapReduce [6]. Unlike Pig Latin, Dryad is completely independent of the map-reduce architecture; instead, Dryad requires programmers to specify the data flow of each program as a directed acyclic graph. Unlike MapReduce jobs, which must have exactly one input and one output set, Dryad allows an arbitrary number of inputs and output at each graph vertex. Dryad does not, however, directly support the dynamic tree-based application structure we target, and does not support the extensive inter-job communication our target applications require.

3.2 Distributed shared memory and message passing

Linda is a distributed communications mechanism designed for efficient message passing among nodes [3]. Linda defines a theoretically infinite global tuple-space buffer. Nodes can atomically add to, read from, and remove tuples from the buffer in order to communicate. Message types are specified using matching tuple fields; no specific addressing or routing is required. Our implementation uses a system based on Linda for its shared memory component.

Another mechanism for enabling a global data store is *memcached*, which provides distributed in-memory caching and is often used to relieve database load for Web applications [2]. memcached servers are independent key-value stores distributed among several hosts. Clients use one layer of hashing to compute which server a value can be written to or looked up in; within a server, a second layer of hashing is used the locate the value. Using this framework, clients can parallelize their lookup requests across the servers, and multiple lookups to the same server can be coalesced to reduce traffic. Individual servers never need to communicate with each other, and a server that fails simply reads as a cache miss.

Chord can also be used for distributed data storage [16]. Chord uses consistent hashing [7] to distribute keys among nodes in a balanced and fault-resistant fashion. Using the Chord protocol, the correct node for a given key can be located, in a peer-to-peer manner, by asking increasingly closer nodes in turn.

The Message-Passing Interface (MPI) is a message-passing API standard for distributed communication [8]. MPI supports both point-to-point and collective communication among processes organized into ordered sets called *groups*. MPI is a basic, low-level distributed communications interface; it does not provide higher-level features like built-in parallelization, load balancing, or fault tolerance that we want to support.

4. SYSTEM OVERVIEW

In this section we describe the major features of Banyan's design.

Banyan is designed to minimize the work done by the programmer to parallelize his or her application. Tree-structured computation can be conceptualized as a recursive mechanism for dividing a problem into subproblems and then combining the results. Banyan recognizes this abstraction by requiring the programmer to implement two primary functions: one to generate subproblems and one to process incoming answers from children. The programmer must also write secondary functions supporting Banyan's cooperative scheduling paradigm. Optionally, the programmer may

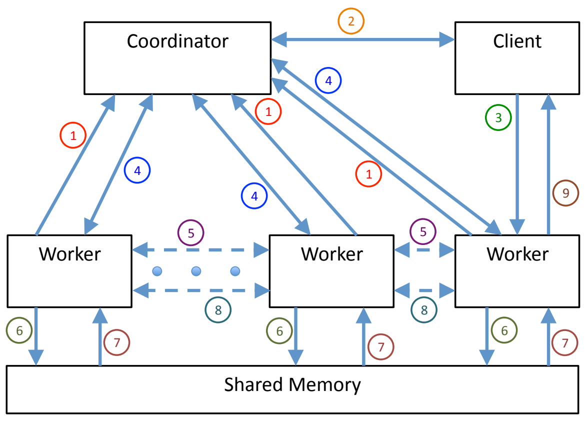

Figure 1: Overview of Banyan architecture. (1) When workers start up, they register with the coordinator. (2) When a client has a job to submit, it asks the coordinator where to send it, and the coordinator supplies the address of a worker. (3) The client submits the root node of the job to a worker. (4) When workers containing splittable subtrees exceed their ticket targets, they ask the coordinator for the address of a remote worker with free resources. (5) Workers distribute subtrees to remote workers as selected by the coordinator. (6) When nodes complete, their results are optionally stored in shared memory. (7) Before new nodes are created, they are checked against the shared memory, if enabled, to see if they have already been resolved. (8) When nodes complete, their parents are notified, either locally or remotely. (9) When the entire job is complete, the client is notified.

specify an additional function for generic message handling between related nodes. The programmer is also responsible for providing a *subproblem identifier*, which is used for indexing into an optional shared memory feature. The programmer meets these requirements by subclassing *TreeNode*, the main unit of computation within Banyan.

Figure 1 presents an overview of the Banyan architecture, which is detailed in the following subsections. First, however, some clarification of terminology: A *node* is one decision point within a tree-structured problem. The problem starts with one node, then expands as that node creates child nodes and those nodes create their own children. A *worker* is one instance of a Banyan process, which may manage many local nodes. A *host* is one independent machine within the network. More than one worker may run on the same host.

4.1 Workers

A *worker* is the basic Banyan process. Workers are responsible for managing and scheduling local nodes according to each node's assigned *tickets*. Each node is allowed to run for a short timeslice proportional to its ticket value. (In the trials described below, each ticket is worth 1 millisecond per scheduling round and each worker has a target of 1000 tickets.) When a worker determines that it has too much work to do locally, it selects a subtree and passes that subtree to another worker with a lighter load.

Workers also manage the basic Banyan infrastructure on behalf of applications. Workers enable the Banyan abstraction that each node can interact directly with its parents and children, without any knowledge of the network or node distribution. Workers ensure that messages from one node to another — for example, to donate tickets, to instruct a child to abandon work, or to inform a parent that a child has completed — are delivered properly, regardless of whether the destination node is local or remote. One worker automatically exports nodes to another, remote worker when the scheduler detects that the load on the current worker has become too heavy; this move is transparent to the client application.

4.2 Coordinator

The *coordinator* is a shared central server that manages the workload across workers. Our system is designed to minimize communication with the coordinator, so as to prevent it from becoming a bottleneck, while still allowing the convenience of one central point of management.

When workers come online, they register with the coordinator, which maintains a list of available workers. If a worker leaves the system cleanly, it sends a de-registration message to remove itself from the list. When a worker decides a given subtree has grown too expensive to continue to process locally, the worker sends a message to the coordinator to request a remote worker. The coordinator determines whether an appropriate remote worker is available, and if so returns its address to the requesting worker. The requesting worker can then pass the subtree to the remote worker to be processed there.

As nodes are assigned to various workers, tickets are transferred among nodes, and nodes complete, workers periodically update the coordinator with their current ticket loads. In this way, the coordinator always has relatively fresh knowledge of the comparative load across workers. Some lag in this information can be tolerated, because the load balancing throughout the system is always approximate.

The coordinator can be run on the same host as one or more workers, or it can run independently on its own host.

4.3 Client

The application programmer is responsible for writing a client application to kick off execution of a tree-structured search computation. The client application must generate a *root* tree node representing the problem starting point. Using Banyan infrastructure, the client submits the root node to an initial worker and processing begins. Children of the root node are scheduled and assigned normally within the Banyan framework.

4.4 Shared memory

To make use of shared memory, the application programmer need only extend a different interface and supply a key for indexing. The key should be something unique to a problem instance, but not to a node, so that Banyan can match on it to avoid duplicating work. Under the hood, Banyan stores problem instances in the Fly space when they register and updates their status when the node working on it changes. It records the value returned for it so that when future nodes register with the same problem instance, it can tell the node to return that value to the parent immediately.

5. USING BANYAN

The following subsections describe how programmers can develop applications using Banyan.

5.1 TreeNode interface

The TreeNode interface requires application programmers to implement five methods.

workHere(): Unit.
This method is called whenever a node is allocated a timeslice. This method should contain the node's internal processing, along with instructions to create new children as needed. This function will be called from the beginning each time the node gets a timeslice, While this method is running, any of the following four methods may be called concurrently.

childReturned(child: Int, childStatus: Status): Unit.
This method will be called when a child node returns to its parent, either because it has completed its work or has given up. Arguments to the function include the child's ID and its completion status. The programmer can use this method to update a node's state with knowledge of its children's results, allowing `workHere()` to make decisions using that information in the future.

timeout(): Unit.
This method is called when a node's timeslice runs out. The programmer should use this method to exit from whatever work is being done in the `workHere()` method, either by saving state of a partially completed task, or by killing an uninterruptible task. In the latter case, the programmer may want to increase the `timeSlicesToUse` field.

abort(): Unit.
This method will be called when a node's status is changed to `Irrelevant`. The programmer should use this method to gracefully exit any work that is being executed.

handleMessage(msg: Any): Unit.
This method will be called if a node receives a message from its parent or one of its children. The main Banyan messages — transferring tickets and reporting completion — are handled separately, but this utility provides a way for the programmer to build in additional communication beyond what Banyan explicitly supports. This method is not used in our theorem proving application.

5.2 Utility methods of TreeNode

As part of implementing the TreeNode interface, application programmers have access to a set of utility methods within the TreeNode class that invoke the Banyan infrastructure.

makeChildIrrelevant(child: Int): Unit.
Tell a child node that it can stop working.

makeOpenChildrenIrrelevant(): Unit.
Tell all children nodes that they can stop working.

returnNode(v: Any): Unit.
Indicate this this node's computation is complete and give the value that it should return.

checkTickets(): Tickets.
Learn how many tickets this node currently holds.

transferTickets(rel: Relative, rsrc: Tickets): Unit.
Donate some of the tickets at this node to a child or the parent of this node.

checkStatus(): Status.
Learn the status of this node.

sendMessageTo(rel: Relative, msg: Any): Unit.
Send a message to a child or the parent.

newChild(childID: NodeID): Unit.
When a node constructs new nodes it can register them as children using this method. The child will be put in the `nodeMap` and added to the task queue.

statusLock.synchronized{ / critical section */}.*
Each node has a field called `statusLock`, which can be used to guarantee that a critical section sees a consistent view of its own status and its children's status.

timeSlicesToUse: Int.
A node will change this field when it wants to save up timeslices in order to avoid interrupting a long computation. The default value is 1. When the scheduler reaches a node, it increments a private field called `timeSlicesSaved` and only lets the node run if `timeSlicesSaved` equals `timeSlicesToUse`. In that case, it runs for the appropriate amount of time and then sets `timeSlicesSaved` to zero.

5.3 Utility functions in BanyanPublic

The application programmer also has access to more general utility methods within the BanyanPublic object.

getShortName(): String.
Return a short printable name that is unique to the current worker, which may be useful for nodes that need to create globally unique names, e.g. for fresh variables in a theorem prover.

data_TreeNode(nd: TreeNode): String.
Dump an easily parsable textual representation of data from the subtree rooted at *nd*, including the name of each node, the amount of time spent there, its status, and its return value.

5.4 Client interface

The application programmer must write a client application that creates the root tree node and submits it to the Banyan system. The client should operate as follows:

1. Call `setCoordinator(addr: String, prt: Int): Unit` to register the location and port of the coordinator.

2. Call `setLocalPort(prt: Int)` to register the port the client should use to communicate with Banyan.

3. Call `getRootParent(): NodeID` to create a stub "parent node" for the root. The stub parent, which remains at the client, provides the line of communication by which computation results are returned.

4. Create an application- and problem-specific tree node that encapsulates the problem to be solved and the strategy for solving it. Use the RootParent stub as the parent argument to the TreeNode's construction.

5. Call `startRoot(rootNode: TreeNode): Unit` to submit the node to Banyan and begin execution.

Our sample theorem prover client takes input from a source file describing the specific sequent to be proved, then creates an OrNode (a theorem-prover implementation of a TreeNode) based on that sequent.

6. IMPLEMENTATION

Banyan is implemented in Scala [9], an object-oriented language built on the Java Virtual Machine, providing extensive support for functional programming. We selected Scala for its concurrency libraries [5] and because we wanted interoperability with some of our existing tools written in Java and Scala.

6.1 Worker implementation

Most of the core functionality of Banyan is provided by the workers. A worker has two threads of control: a listener and a scheduler. The listener's job is to wait for messages from other workers, the coordinator, or the client. The scheduler's job is to loop through the task queue and cause work to get done. The two actors communicate through a small number of shared variables, primarily the node map and the task queue.

6.1.1 Scheduling algorithm

The scheduler performs weighted round-robin cooperative scheduling on the nodes that it owns. That is, in each round each node gets to work for a timeslice proportional to the number of tickets that it holds. After that time, the scheduler calls the timeout method of the node and waits for it to yield control. If new nodes are created, they are added at the end of the task queue.

If a node needs more than one timeslice to complete some piece of indivisible work, it is allowed to save up timeslices for future use. If a node receives a timeout and has to kill a computation, it might save up twice as many timeslices for the next try. It that still fails, it might save twice as many again. Using this strategy, a node will never work more than three times its minimum required time.

Banyan does not provide support for nodes that wish to pause threads or processes between calls of `workHere()`. Allowing a paused thread to persist between timeslices would mean that no longer could any node be shipped off at any time between timeslices, because paused threads are not serializable. This could get in the way of the load balancing. It would be interesting to investigate how allowing nodes to be non-mobile would affect the performance of Banyan.

6.1.2 Node transfer strategy

Each worker keeps track of the total number of tickets held by nodes that it owns and the target number of tickets that the coordinator has indicated that this worker should hold. The total minus the target is called the *ticket surplus*.

Periodically (once every five seconds for the runs documented in our evaluation section), the scheduler pauses between timeslices to assess the ticket situation at this worker. If the surplus is negative, the scheduler sends an update to the coordinator indicating its present total and target. If the surplus is positive, the scheduler is responsible for finding a suitable subtree to try to ship to another worker. To do this, it first computes the total tickets held by and total time spent on each local subtree. It then selects a subtree with number of tickets approximately equal to the surplus and with time-spent above a threshold. (Here we assume that nodes which have required a lot of work in the past are less likely to terminate quickly in the future.) It sends a message to the coordinator indicating the number of tickets in this subtree. It waits for a reply from the coordinator. The reply could be "no" or it could give the address of a worker that would be willing to accept the subtree. In the latter case the scheduler ships off that subtree. The scheduler then updates the coordinator on its ticket situation and then resumes its scheduling loop. During this whole procedure the scheduler holds a lock that prevents the listener from adding nodes or tickets.

One might imagine that the best subtree to ship off would be the one which will minimize the absolute value of the resulting ticket surplus. That is not necessarily true. Consider the following situation with four workers. Each worker has a target of 1000 tickets. The root node starts with 4000 tickets. It creates three subnodes and donates 1333 tickets to each. Assessment takes place. The worker can now either ship off one of these nodes with 1333 tickets and be left with a surplus of 1667 tickets, or it can ship off the whole tree and be left with a deficit of 1000 tickets. If we go by absolute value, this is better than shipping off a subtree. But clearly shipping off the whole tree does not buy us anything.

We are still investigating improved metrics for the suitability of a subtree (or possibly set of subtrees) to be shipped off. The present version of Banyan uses a slightly modified version of the "lowest absolute value" metric described above, favoring subtrees that will leave a surplus over those that will leave a deficit.

6.2 Shared memory

We use the Fly Object Space [17] for sharing solutions among subproblems. Fly is an implementation of Linda tuplespaces for Java and Scala objects. A Fly space is parameterized on a FlyEntry type for storing data, and it has three main methods:

1. *write(e:FlyEntry, timeout:Long)*: Writes *e* to the space with a lease of *timeout*.
2. *read(template:FlyEntry, timeout:Long)*: Returns a copy of any object from the space which matches the template at all non-null fields.
3. *take(template:FlyEntry, timeout:Long)*: Removes an entry matching the template and returns it.

We use the space as shared memory for the workers. Our FlyEntry type consists of four fields: the node ID of a child, a key indicating the problem it represents, its status, and an optional return value (set to `None` until it returns).

When a node generates a child, we first create a template matching on the key and the status `Returned()`. If any nodes are successfully read, we tell the child to return immediately with the value in the returned object. If no nodes are read, we create a new FlyEntry with a `Working()` status and place it in the Fly space.

When a node returns to its parent, it `take()`s the entry matching its node ID, sets the status field to `Returned()`, and sets the value field to whatever value it is about to return to its parent.

In our current setup, we only ever really need to insert nodes at the return to the parent, because no worker ever uses information from objects with any status other than `Returned()`. However, we have imagined a couple of extensions to this use, described in 9.

A couple limitations of Fly emerged during our use of it. For one thing, it does not seem to allow for infinite leases. For another, we could not find sufficient documentation for running it as a distributed store – currently the Fly server runs on just one host.

7. CASE STUDY

In our case study we implemented a theorem prover for a small fragment of differential dynamic logic using a simplified set of proof rules based on those found in [12]. For the real-arithmetic backend we used our own implementation of the Cohen-Hörmander algorithm.

The Banyan part of the prover consists of three implementations of the `TreeNode` interface: `andNode`, `orNode`, and `ArithmeticNode`. The `workHere()` function for `AndNode`s and `OrNode`s just creates the appropriate children, tranfers tickets to them, and yields control. The `workHere()` function for `ArithmeticNode`s calls Cohen-Hörmander backend. The `timeout` method of `ArithmeticNode`, sets a shared flag that tells the backend to quit (it doesn't save any state) and doubles the value of `timeSlicesToUse`.

Here is a slightly simplified code sample from `AndNode`:

```
def childReturned(child: Int, v: ReturnType)
  : Unit = v match {
     case Proved(rl) =>
       numOpenChildren -= 1
       if(numOpenChildren <= 0)
         returnNode(Proved(rule))
     case GaveUp() =>
       returnNode(GaveUp())
    }
```

This says that if an `AndNode` learns that one of its children has returned with a proof, then it will return if all the rest of its children have already returned with a proof. If one of its children returns indicating that it cannot find a proof, then the `AndNode` returns indicating that it also cannot find a proof.

8. EVALUATION

We evaluated Banyan using the differential dynamic logic theorem prover application. We tested two example problems, one modeling the temperature of a water tank and one modeling the height of a bouncing ball. For the water tank example, we deliberately tested a *bad hints* problem description that provides the prover with incorrect starting invariants, leading it to try some expensive wrong paths before finding the correct proof.

All evaluations were run on four virtual machines running Debian Lenny with 2 GB of memory and one virtual CPU each. The virtual machines themselves were located on a single Xen 3.4 host, a dual 2.66 GHz Intel Xeon E5430 quad-core system with 16 GB of memory.

Completion time, in seconds			
Test	1 Worker	4 Workers	Speedup
Water tank, bad hints	2978	1213	2.4x
Bouncing ball (run 1)	209	50	4.2x
Bouncing ball (run 2)	147	62	2.4x

Table 1: Overall Banyan performance. Moving from one to four workers decreased completion time by varying degrees, based in part on the decomposition of the problem. For tests using four workers, the coordinator and client were each co-located with one worker.

8.1 Performance

First, we compare the performance of Banyan with different numbers of workers. Table 1 shows the completion time for different problems, in seconds. For each example, we tested configurations with one worker and with four workers. In the four-worker configuration, the coordinator and the client were each co-located with one worker for a total of four machines.

Figures 2 and 3 illustrate the computation of the water tank and bouncing ball examples, drawn as directed graphs of tree nodes. Each node is sized in proportion to the amount of time spent doing work there. Most nodes are small; the few very large nodes represent the arithmetic computation leaves. In the water tank problem, there are only two large computational leaves; this helps to explain why we don't see a bigger speedup when increasing to four workers. The work in the computational leaves cannot be further divided among workers. The graph of the bouncing ball problem shows similar limitations to the benefits of parallelism.

We show two runs of the bouncing ball problem illustrating a large performance range for solving the same problem. We have identified two possible explanations for this difference. First, our cooperative scheduling timeouts can artificially exaggerate differences in node running time; a node that times out just before it finishes must run again from the beginning, potentially taking twice as long to complete as it could have if it finished just before timing out. Second, our experimental setup of four hosts required co-locating the coordinator and the client with one worker each. If during a given run the worker that receives an expensive arithmetic node is co-located with the coordinator, task switching between the worker and coordinator process may increase the overall task latency.

These experiments were run on Banyan with shared memory disabled.

8.2 Scheduling overhead

We attempted to measure how much overhead the Banyan framework adds to the total computation time for these problems. Within each worker, the two main sources of overhead are *task scheduling* and *ticket assessment*, described in Sections 6.1.1 and 6.1.2 respectively. Task scheduling manages the weighted round-robin invocation of active local nodes, and ticket assessment keeps the worker in balance with its ticket target. For each test run, we measured the average time (in ms) spent in task scheduling and in ticket assessment at each worker. Table 2 details the results.

Scheduling time appears to be longer, on average, on workers running fewer local nodes. This may occur because these

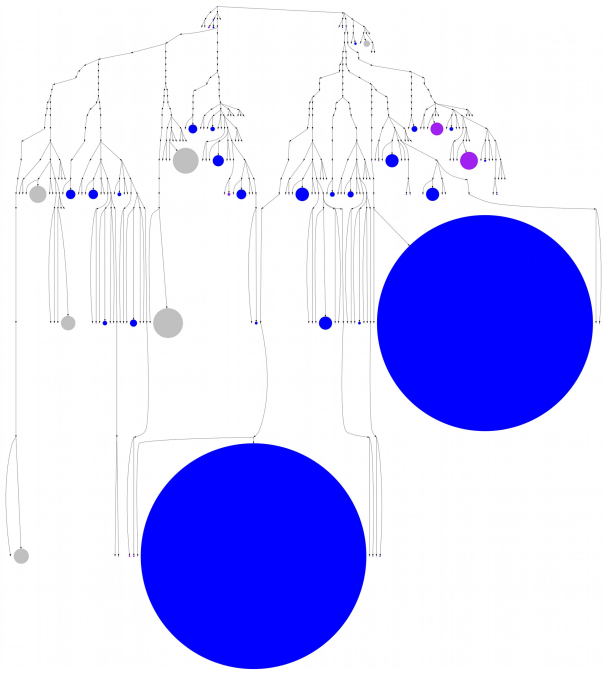

Figure 2: Directed graph of computation for the water tank (bad hints) example. In this graph, node sizes are proportional to the total time spent working in each node. The problem contains mainly small, quick nodes, with a few larger nodes representing subproblems requiring extensive arithmetic computation.

workers often house just a few heavyweight nodes as compared to many fast nodes; heavyweight nodes are more likely to be skipped as part of timeslice hoarding, meaning the scheduler must load them, update their hoarding status, put them back in the queue, and load a different node before beginning work. In some instances, if there is only one large active node, the same node may be cycled through the queue repeatedly as its timeslice accumulates. This behavior could probably be improved by reconsidering how timeslice hoarding is handled.

Ticket assessment, by contrast, is much faster on workers with fewer nodes. This may be because, if a worker is too busy, a large set of local nodes must be searched to find a good candidate subtree for offloading.

We compare these results to the amount of time spent in a node during one working instance. The scheduler must run in between each node instance, so the ratio of scheduling time to node working time should be low to minimize overhead. Figure 4 shows a histogram of node working times (both axes are log scale). More than half of all node working instances run for less than 1 ms, while a few nodes run for tens to hundreds of seconds. This means that working time will frequently be comparable to our average scheduling time of 0.18 ms. To make Banyan more practical, the scheduler may need optimization. We can also see that ticket assessment is comparatively very expensive, at almost 24 ms on average. Because ticket assessment occurs no more than once every 5 seconds, however, this probably does not have a large impact on overall overhead. It could be interesting to examine the tradeoffs of more frequent ticket assessment,

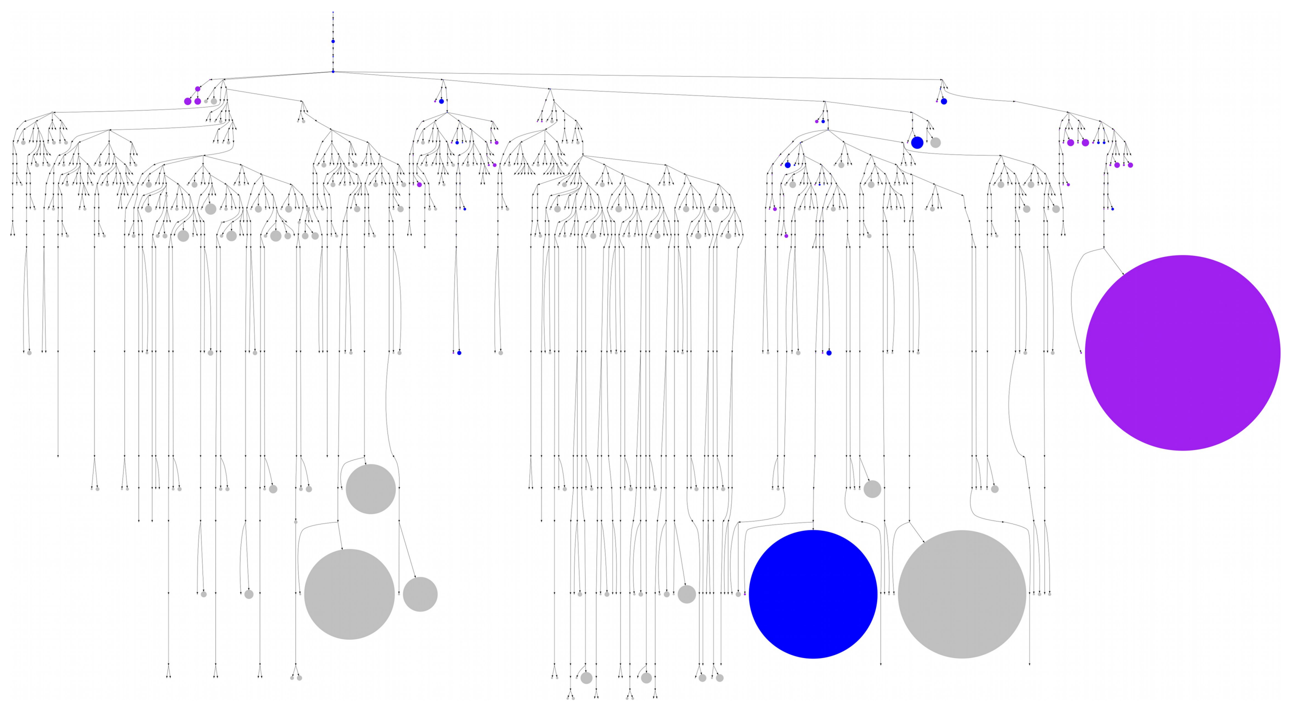

Figure 3: Directed graph of computation for the bouncing ball example. In this graph, node sizes are proportional to the total time spent working in each node. The problem contains mainly small, quick nodes, with a few larger nodes representing subproblems requiring extensive arithmetic computation.

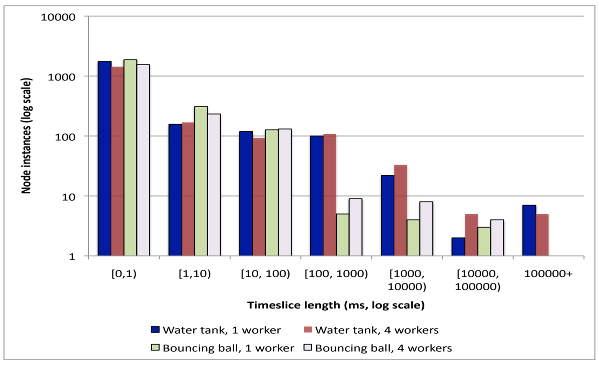

Figure 4: Histogram of node working time. This figure shows the distribution of node working time across several experiments (both axes are log scale). More than half of all nodes run for less than 1 ms; a few run for more than 100 seconds.

which could maintain better balance across workers, at the cost of more overhead processing time.

8.3 Comparison to KeYmaera

To see how well our prover compares with the current state of the art, we ran the water tank example on KeYmaera using the same Cohen-Hörmander backend for real arithmetic. As of yet, KeYmaera still beats us handily, finding a proof in 131 seconds (on a single processor) compared to our best case of 1213 seconds (using four distributed processors).

This is mainly because KeYmaera is using a much more sophisticated set of proof rules, so the real-arithmetic problems that is asks the backend are often much simpler.

Consider the following sequent, which is a simplified version of the sequent appearing at one of the two large nodes in Figure 2.

$$s = 1, x = 0, y = 10 \vdash (s = 3 \rightarrow y \geq 5-2x), (s = 1 \rightarrow y \leq 10+x)$$

Our Banyan prover needs to spend 1084 seconds on this node, which corresponds to the backend needing somewhere between 328 and 542 seconds (due to our timeslice doubling scheme).

KeYmaera, on the other hand, knows how do deal with substitution and has some built-in rules for dealing with arithmetic. It is therefore able to reduce the corresponding node to

$$y = 10 \vdash y \leq 10$$

before calling the backend.

8.4 Generality

To argue that our framework is more general than the particular use case of a theorem prover, we sketch a implementation of using Banyan for a distributed minimax algorithm.

Minimax finds the best move in a two-player game tree based on the metric that the opponent will always choose the move that's worst for the other player, also based on minimax. There is usually a finite cutoff on the depth the program will search in the tree.

	Instances	Avg. time (ms)
Scheduling overhead		
Water tank (1 worker)	2160	0.10
Water tank (4 workers), A	1047	0.19
Water tank (4 workers), B	444	0.48
Water tank (4 workers), C	217	0.06
Water tank (4 workers), D	137	2.24
Bouncing ball (1 worker)	2326	0.08
Bouncing ball (4 workers), A	936	0.06
Bouncing ball (4 workers), B	922	0.13
Bouncing ball (4 workers), C	66	2.28
Bouncing ball (4 workers), D	15	0.20
Average		**0.18**
Task assessment overhead		
Water tank (1 worker)	25	2.12
Water tank (4 workers), A	12	43.50
Water tank (4 workers), B	11	47.09
Water tank (4 workers), C	12	14.33
Water tank (4 workers), D	9	37.56
Bouncing ball (1 worker)	7	6.57
Bouncing ball (4 workers), A	4	55.75
Bouncing ball (4 workers), B	3	76.33
Bouncing ball (4 workers), C	4	2.25
Bouncing ball (4 workers), D	2	4.50
Average		**23.81**

Table 2: Scheduling and task assessment overhead measured at each worker. For each test run, we show how many times scheduling and task assessment were required at each worker, as well as the average time spent on each activity at each worker. The overall average duration of scheduling and task assessment activities is also shown.

The pseudocode for the sequential version looks like:

```
fun minimax(gamestate, depth) : Int
  if gamestate is a leaf or depth = 0:
    return heuristic_value(gamestate)
  else:
    a = -infinity
    foreach m in possible_moves(gamestate)
      a = max(a, -minimax(m, depth-1))
    return a
```

To use Banyan to solve this problem, we would implement the *workHere()* method to generate the possible moves and register each one as a new child node. We would have an extra field in our node type to indicate its depth, and child nodes would be created with a decremented depth. *childReturned(v)* would compare the negation of v to some stateful value (initialized to a minimum integer value) and update that value to v if v is greater. Once all children return, it would return to its parent with the final value.

9. CONCLUSION AND FUTURE WORK

We have designed and implemented Banyan, a working prototype framework for distributing tree-structured computation across a cluster. As far as we know, we are the first to develop a system to enable easy parallelization for this pervasive class of applications.

We have several ideas for improvements to the next version of Banyan, described below.

Support for multiple jobs.

Modest changes to the Banyan coordinator would enable running one continuous instance of Banyan, dynamically adding new jobs as needed even if existing jobs are already running. Currently, the coordinator assigns the client a total number of tickets for its job based on the number of available workers, and ticket targets are set to evenly divide that value among the workers. To support multiple jobs, the coordinator would need to enable "ticket inflation" – that is, reorganize ticket targets to evenly split the total number of tickets for all jobs in the system, not just the first job the coordinator sees. For example, if the coordinator assigns each new job an initial value of 1000 tickets, then when one job is running each of four workers should target 250 tickets. When a new job is submitted, there will be 2000 tickets worth of work available in the system, so each worker should target 500 tickets. Most of the infrastructure needed to implement this change is already implemented in Banyan.

Fault tolerance.

Banyan currently has no graceful response to the failure of a worker or the coordinator during processing.

As a first step, we could add heartbeat communication between workers containing subtrees that have parent-child relationships. If a worker discovers that a node's parent is no longer available, the child node (and all of its children recursively) should be aborted. If a worker discovers that a node's child is no longer available, that child should be recreated and considered for shipping to a remote worker as normal. This mechanism would ensure that duplication of work is limited to only those nodes falling below a fault within the tree structure. However, because subtrees are distributed pseudo-randomly, there is a good chance that any failed worker would contain a relatively high-up node, resulting in cascading node cancellations and significant duplication of work. A stronger solution that could avoid this waterfall effect would be preferable and would require a more thoughtful design.

Failure of the coordinator could potentially be handled as follows: when a new coordinator starts, it broadcasts a message asking workers to re-register. Once workers have re-registered and updated the coordinator with their current ticket states, the coordinator can determine and propagate a new set of ticket targets and then resume advising workers about where to ship surplus subtrees. While the coordinator is unavailable, workers will be unable to offload work, so processing may slow down but should remain correct.

Another important aspect of fault tolerance will be providing redundancy within the shared memory system, so that failure of one shared-memory host does not destroy stored node results. Strategies for fault-resistant distributed storage have been explored in a variety of systems, and we would expect to adapt an existing approach.

Improve shared memory.

Currently, no worker ever uses the Fly space for objects with any status other than `Returned()`. However, our design is flexible enough to allow for some possible extensions:

- A node may under some conditions want to wait for a `Working()` node to finish rather than beginning another branch of the same work. This may depend on how long the node has been working, which could be indicated with a time field. We are not sure what a proper heuristic would be to determine whether to wait, however.

- An alternate work distribution scheme could use Fly in a way that most tuple-spaces are used: as a global store for workers to place untouched jobs. This use would entail more drastically rethinking our system design.

Additionally, our original design intended for the user to supply a comparator function to allow for *inexact matching* on the Fly space key. That is, rather than search for exact matches of a subproblem, we would search for *stronger* instances of that problem, for a user-supplied definition of stronger.

Also, we currently only run Fly on one host that everyone knows about. In the future, we would like to distribute the shared store.

Compare to simpler models.
How does our computation model compare to having, say, a global shared queue of tasks to be worked on?

Our code is available at `http://www.cs.cmu.edu/~renshaw/banyan`.

10. ACKNOWLEDGMENTS

The authors would like to acknowledge Rob Simmons, Jamie Morgenstern, William Lovas, Andre Platzer, Guy Blelloch, and Shafeeq Sinnamohideen for helpful conversations, as well as 15-712 course staff Dave Andersen and Iulian Moraru for their comments and guidance. We would also like to thank the Parallel Data Lab for the use of their cluster.

11. REFERENCES

[1] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[2] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004:5–, August 2004.

[3] D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *PODC '82: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 10–18, New York, NY, USA, 1982. ACM.

[4] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[5] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202 – 220, 2009. Distributed Computing Techniques.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[7] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[8] MPI Forum. MPI: A message-passing interface standard, version 2.2, September 2009. http://www.mpi-forum.org.

[9] M. Odersky. Scala. http://www.scala-lang.org.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[11] A. Platzer. Differential dynamic logic for hybrid systems. *J Autom Reas*, 41(2):143–189, 2008.

[12] A. Platzer and E. M. Clarke. Computing differential invariants of hybrid systems as fixedpoints. *Form. Methods Syst. Des.*, 35(1):98–120, 2009.

[13] A. Platzer and E. M. Clarke. Formal verification of curved flight collision avoidance maneuvers: A case study. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009.

[14] A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[15] A. Platzer and J.-D. Quesel. European train control system: A case study in formal verification. In K. Breitman and A. Cavalcanti, editors, *ICFEM*, volume 5885 of *LNCS*, pages 246–265. Springer, 2009.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[17] Zink Digital Ltd. The fly objectspace. `http://flyobjectspace.com/`.