A Framework for Distributed Proof Search*

David Renshaw, André Platzer, Chris Martens, and Michelle Mazurek

Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, USA
{renshaw|aplatzer|cmartens}@cs.cmu.edu, mmazurek@andrew.cmu.edu

Abstract. In automated theorem proving, the nondeterminism of proof rule application naturally gives rise to nested and/or branching of independent subgoals. This problem structure should be amenable to parallel exploration, but there are at present few theorem prover implementations which actually exploit this potential. We propose a new programming model to make it easier to write such parallel provers. In it, details such as communication and work distribution among processors are hidden from the programmer. Banyan, our implementation of this model, only needs problem-specific input such as how a tree of proof goals gets constructed for the proof search algorithm and what should be the relative amount of processor time spent at each subgoal. This allows the programmer to focus on high level proof search and finely tune a search strategy, for example by giving more time to branches that are more promising. The Banyan runtime system automatically distributes and dynamically redistributes proof goals among distributed processors and performs weighted round-robin scheduling among active goals. We have used Banyan to write a theorem prover for hybrid systems, DLBanyan. DLBanyan outperforms our existing sequential prover for hybrid systems and achieves a near linear speedup in the number of processors used.

1 Introduction

Efficient proof search is the key to turning theoretically complete proof systems into practically useful automatic proof procedures. For problems like system verification that have no effective axiomatization, efficient proof search is even more important. The efficiency of a prover depends critically upon how it deals with nondeterminisms, which may be choices between possible proof rule applications or between possible invariants and induction hypotheses. Some are *don't care* nondeterminisms, e.g. the order of α-rule applications, and have virtually no practical impact on proving. Others are more significant *don't know* nondeterminisms, e.g. choices between different invariants. It is not easy to come up with good exploration strategies for those nondeterminisms. As an example domain, we consider hybrid systems [1], i.e. systems with joint discrete and continuous dynamics, such as cars and aircraft. In hybrid systems verification, the difference in

* This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-0926181, CNS-0931985, NASA grant NNG-05GF84H, ONR award N00014-10-1-0188.

performance between good and bad proof search strategies can be two orders of magnitude or more [2]. Theorem proving for hybrid systems [3] involves decision procedures for real arithmetic [4]. Because real arithmetic has a high theoretical and practical computational complexity [5], neither depth-first nor breadth-first proof search is practically complete, even for decidable subproblems of hybrid systems verification. Both exploration orders can be stuck for several days on the wrong proof choices without finding an efficient 30 minute proof. When using computationally expensive decision procedures, inefficient proof search can turn a theoretically decidable problem into a practically infeasible problem.

Practically complete procedures need to follow a fair exploration order of all possible proof options for arbitrarily long times, including the time for background decision procedures. Thus proof search has an inherently parallel problem structure, but few theorem provers actually exploit this potential. One reason is that general parallel (and distributed) programming can be tricky. Simplified distributed programming models like MapReduce list processing, on the other hand, do not match the problem structure of proof search very well.

Flat parallel process structures are inefficient for parallel proving problems. Neither eager parallel exploration nor lazy work-stealing exploration really match. Eagerly forking one parallel proving process for each subproblem will quickly clutter the machine with too many processes competing for computing and memory resources, thereby effectively spoiling progress. This effect is particularly counterproductive in the presence of background decision procedures that are computationally expensive and memory intensive. Lazy work-stealing, which proves as many subgoals at once as there are processors and moves on to the next subproblem whenever one subproblem terminates, is also unlikely to work. Again, this problem intensifies when using decision procedures, because they can run for days on one subproblem without terminating, even for decidable theories. For other problem domains, where all subproblems need to be solved, these simple exploration orders still work. The theorem proving problem is inherently different, though, because answers to some subproblems invalidate the need to consider others, but it is not clear which subproblems terminate most quickly. Consequently, static exploration orders are insufficient and automated theorem proving needs a more dynamic exploration order that fits to the proving domain.

In this paper we propose a new programming model that naturally fits the actual problem structure in proof search procedures of automated theorem provers. Our model makes it easier to implement distributed parallel proof search procedures and hides details about communication and work distribution among processors. The proposed model is designed for proof search problems and makes it easy to specify parallel exploration strategies that are sufficiently focused to make progress but sufficiently broad to ensure the fairness required for completeness reasons. In this paper we consider the automated theorem proving problem for hybrid systems [3], but our approach is relevant for other domains as well.

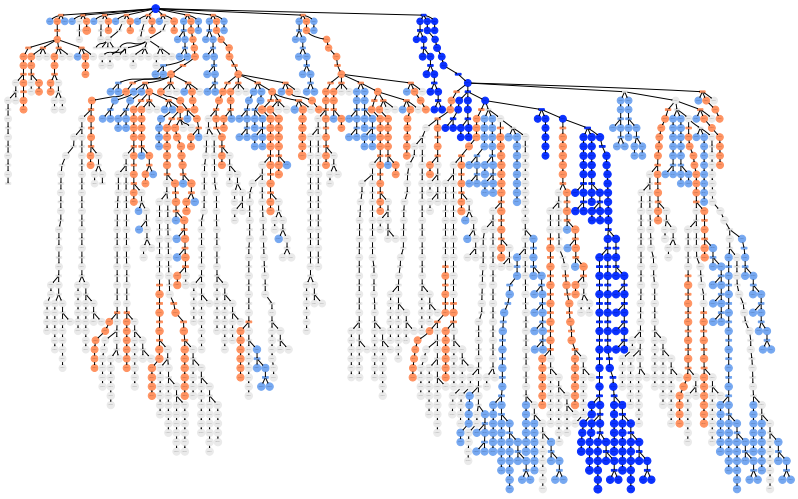


Fig. 1. Proof search for verifying a switching damped oscillator

The basic principle of our programming model allows for distributed parallel proof search in the form of annotated and/or trees. Specifically, we consider a (tableau-style) backwards proof search procedure for a sequent calculus for hybrid systems [3, 2]. A proof is a tree of sequents in which *all* leaves are provable by axioms or arithmetic decision procedures. During proof search, however, the proof structure has multiple choice points, because there are choices in proof rule applications (or-nodes). Only one choice of proof rule applications or invariants needs to succeed for a successful proof. See Fig. 1 for an illustration of the exploration of the proof search space for verifying a switching damped oscillator control. Each node is either an *and-node* (all subgoals need to be proven, illustrated by rectangular bars in Fig. 1) or an *or-node* (at least one subgoal needs to be proven, illustrated by circles in Fig. 1). The dark blue nodes represent the (first) successful proof of the verification conjecture. The light gray nodes have neither been proven nor disproven yet, but became irrelevant after the dark blue proof has been found. The medium color nodes have been proven (medium blue) or disproven (medium orange) during proof search, but no consecutive subset of the proven nodes constitutes a full proof of the conjecture, because other subgoals of its and-nodes have been disproven or are still unknown.

Proof search thus has a concurrent nature. Considering the trend toward multi-core, multi-processor, or even distributed cluster systems, it makes sense to exploit this concurrency and actually explore the concurrent proof search structure in parallel. (Henceforth, we will use the word "processor" to denote a "cpu" or core— a hardware unit, of which there may be several in a single chip, capable of executing code. A distributed three-node computer system with two quad-core chips would have 24 processors.) As long as the total number of processors exceeds the number of subgoals in the proof search structure and

communication cost is low, parallelization is fairly simple: Every subgoal could be explored by one processor. When there is more work than processors (the proof space in Fig. 1 has 2118 nodes) or when locality needs to be respected to avoid unnecessary communication cost, smart parallelization is more difficult, because tradeoffs must be made on how much time to spend on the various subgoals. Even for the extreme case of a single processor system, we show that significant speedups can be obtained by exploiting the concurrent problem structure in theorem proving compared to our previous strictly sequential implementation of KeYmaera [6], a theorem prover for hybrid systems.

Note, in particular, that it is not clear what the best parallel exploration order would be for the various proof nodes in Fig. 1. For each of the proof options, we want to come up with a proven/disproven decision as quickly as possible in order to find the actual proof as early as possible. It is our goal to come up with a framework in which it is easy to specify good parallel exploration orders in a natural way, such that the framework takes care of parallelization, distribution, workload sharing, proof management, propagation of proof status information, and local proof hint communication.

Our contributions are as follows:

- We propose a new model for parallel computation.
- We show how this model fits naturally to the theorem proving problem.
- We identify simple ways how fair parallel exploration strategies can be described with our framework.
- We implement the framework Banyan that makes it easy to develop parallel provers.
- We implement a theorem prover for hybrid systems in Banyan.
- In experimental results we evaluate the performance of our parallel exploration strategies. We show that our new framework outperforms a previous theorem prover for hybrid systems and demonstrate that our new parallel proof framework generally finds proofs much faster.

2 Related Work

Parallel Processing Languages and Frameworks. The primary inspiration for Banyan comes from frameworks for high performance data-parallel computing such as MapReduce [7] and Dryad [8]. Using the "single program, multiple data" programming model, these systems hide the low-level details of parallelization from the application programmer, who need only worry about how to express a problem within the framework. Both have proven to be effective in exploiting the computing power of large clusters for data processing. It is unclear how to use them on problems such as theorem proving that have a tree structure and where the amount of time spent on any particular task is difficult to predict.

Cilk [9] is an extension of the C programming language providing support for nested parallelism with provable efficiency in terms of space, time, and communication. Cilk's primary parallelization mechanism follows a work-stealing model.

Cilk can emulate limited or-parallelism by aborting branches that become irrelevant. But, unlike Banyan, Cilk does ensure a fair exploration of or-branches, does not allow the programmer to specify the relative time share of parallel tasks, and does not allow communication between threads of computation.

NESL [10] is a programming language with support for nested data parallelism. One of its more significant features is a method, called a *vectorization transformation*, for automatically unfolding a recursive computation tree into a flat structure that can easily be executed in parallel. More recently, Data-Parallel Haskell [11] has extended this line of research. However, since vectorization is only useful for and-parallelism, it is of limited use in our problem domain.

Parallel Automated Reasoning. There has been much work on parallel SAT solvers. They come in two main varieties: *divide and conquer* solvers [12] that partition the search space and *strategy-parallel* solvers [13] which try many strategies in parallel. Both of these approaches are typically enhanced by sharing of information between parallel processors. This work has been successful. However, SAT solving has a rather different structure from the problems we are targeting. SAT is a decidable problem with an accepted fasted algorithm where all the gains come from tuning various parameters. We are interested in a more open-ended exploration of an undecidable problem.

There has also been work on Parallel theorem provers. For example, PARTHEO [14] is a parallel prover for first-order logic. It uses work-stealing to keep all processors busy. It does not attempt to schedule branches fairly. Other provers have taken a strategy-parallel approach. RCTHEO [15] lets each processor work on the entire problem, using a different random seed at each processor, and then makes random selection at choice points. Banyan aims to give much more control over exploration.

3 And/Or Branching Structure in Proof Search

Suppose we are trying to prove a formula ϕ. We come to a point where we have two independent subgoals A and B. If both of them must return for us to prove ϕ, then this is an instance of *and-branching* (all subgoals need to be proved). An example is if we want to prove a conjunction $A \wedge B$ then we need to prove both A and B; see Fig. 2. Then the only question is which processor gets to do which work and when. If T_n is the ideal amount of time it takes for ϕ to be solved on n processors, and P is our actual number of processors, then T_1/P and T_∞ are lower bounds on the time it takes to complete. Cilk provides support primarily for this kind of and-parallelism and provides an (expected) running time of $T_1/P + O(T_\infty)$ [16].

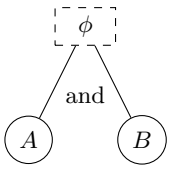


Fig. 2. And branching

Suppose, on the other hand, that when trying to prove a formula we come to a point with a formula ϕ where we have two methods of proceeding, A, B, either of which may or may not give us an answer to the problem of deciding ϕ. The

methods A, B could be, for example, two different proof rules that are applicable to the subgoal ϕ, two different invariants to try, or two different decision procedures. Then this is an instance of *or-branching* (see Fig. 3), and dealing with it is not so straightforward. Depending on what we know about the performance characteristics of A and B, we have several ways of proceeding.

The simplest thing we might do is first to try method A and then to try method B. But what if method A does not terminate and we can only succeed by trying B? To deal with that case, we might try A and B concurrently, for example by spawning a thread for each. Even if these threads are running on the same processor, we have gained something, for now we are guaranteed that we will spend a total amount of time no greater than twice the minimum termination time of A and B (assuming fair scheduling of the two threads).

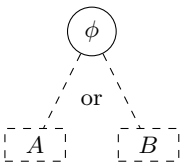


Fig. 3. Or branching

If we know something about the expected running times of A and B, then we can do even better. Suppose that we know that A succeeds 80% of the time with an expected termination time of $t_A = 16$ minutes, and suppose that B succeeds the other 20% of the time with an expected termination time of $t_B = 1$ minute. Assume that when A (or B) does not succeed it does not terminate, which is a fairly common situation in practical decision procedures for arithmetic, for instance. If a is the proportion of our time we spend on A, then the expected number of minutes spent is $\frac{0.80\times16}{a} + \frac{0.20\times1}{1-a}$. In this case it is optimal to spend $\frac{8}{9}$ of our time on rule A and $\frac{1}{9}$ on rule B. This results in an expected amount of work of 16.2 minutes. The naive strategy of devoting half of our time to each rule yields an expected amount of work of 26 minutes. Consequently, we observe that it makes sense to consider time shared parallel exploration for theorem proving problems.

If we have two processors we can of course do even better again. If no other subproblem is using the processors, we do best by trying A on one processor and B on the other. Then we are done in time $\min(t_A, t_B)$. In automated theorem proving problems, we will typically have more tasks than processors. Then our goal should be to distribute tasks among processors and to share time on each processor in such a way that

1. We can specify the proportion of processor time being devoted to each subtask.
2. We maintain locality in the proof exploration in order to minimize communication costs and maximize coherence of the resulting proof.

These are our primary goals in the design of Banyan. If they can be achieved, we argue that our framework provides a solid basis for theorem provers, because and- and or-branching are fundamental in backwards proof search. Choosing which proof rule to apply is an instance of or-branching. Proving all the premises of that rule is an instance of and-branching.

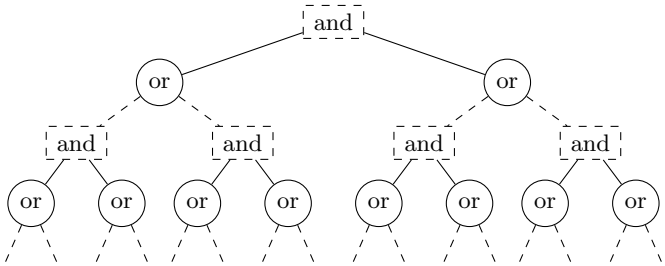


Fig. 4. Nested And/Or branching structure in proof search problems

4 Automated Theorem Proving for Hybrid Systems

The specific problem that we consider is the verification problem for hybrid systems [1], i.e., systems with interacting discrete dynamics and continuous dynamics along differential equations. This is an important but challenging problem [1], with applications in air traffic control, car control, robotics, and many more domains. We consider a proof calculus for hybrid systems that is a complete axiomatization relative to differential equations [3]. Specifically, we use a backwards proof search procedure [2] for a sequent calculus for a logic for hybrid systems that is called *differential dynamic logic* $\mathsf{d}\mathcal{L}$ [3]. Safety properties (and more general properties) of hybrid systems can be stated in $\mathsf{d}\mathcal{L}$ and proved in the $\mathsf{d}\mathcal{L}$ calculus. In this paper, we mostly focus on verifying safety properties of hybrid systems.

The most crucial parts of the proof search procedure for hybrid systems are the search for invariants to prove properties of loops and the search for a generalization called differential invariants [17] to prove properties of differential equations without having to solve them. In a sense made precise by a relative completeness argument and a relative semidecision procedure [3, 2], this search for (differential) invariants is the "only" theoretically difficult part.

In practice, another significant challenge comes from the need to handle real arithmetic for proving properties of the evolution of continuous quantities like positions and velocities in the system dynamics. Real arithmetic is decidable [4] but of high computational complexity [5]. In practice, a considerable percentage of decision procedure [4] calls for real arithmetic does not even terminate after a day or sometimes weeks. Alternatively, decision procedure calls for checking validities can be split at conjunctions and fed into separate decision procedure invocations. For proving the validity of $A \wedge B$, for instance, the proof search procedure can choose between deciding the validity of $A \wedge B$ or, instead, separately deciding the validity of A and the validity of B:

$$\mathit{decide}(A \wedge B) \qquad \text{versus} \qquad \mathit{decide}(A) \text{ and } \mathit{decide}(B)$$

This choice causes nondeterminisms in solving real arithmetic. Because decision procedures for real arithmetic work algebraically and do not operate by logical

proof principles, this nondeterminism can impact the efficiency of proof search by several orders of magnitude [2].

With these nondeterminisms, the $\mathtt{d}\mathcal{L}$ proof calculus fits naturally to the proposed and/or branching model for theorem proving. While the branches resulting from one rule application are and-branches (all of them have to close for the proof to succeed), the various rule alternatives are or-branches (only one of the proof search alternatives has to be successful for the proof). For instance, differential equations can either be handled using their solutions (if it is polynomial) or by searching for differential invariants. Most notably, induction rules give rise to or-branches, as there are several possible formulas (infinitely many) that could be used as (differential) invariants, but one successful invariant is enough for closing the proof. For techniques that generate promising candidates for (differential) invariants, we refer to previous work [18]. Some unsuccessful candidates for invariants can be refuted quickly, but others run for days without a result. Hence, practical and scalable hybrid systems verification requires a fair concurrent exploration of all proof options to be successful in advanced applications.

5 Programming Model for Distributed Proving

Our programming model is designed to expose maximally the concurrent nature of the proof search problem. The main unit of control is the *computation node*. One such node would correspond to, for instance, each and-node and or-node in Fig. 4. Each node (except the root) has a parent and zero or more children, forming a computation tree replacing the traditional control-flow stack.

5.1 Background

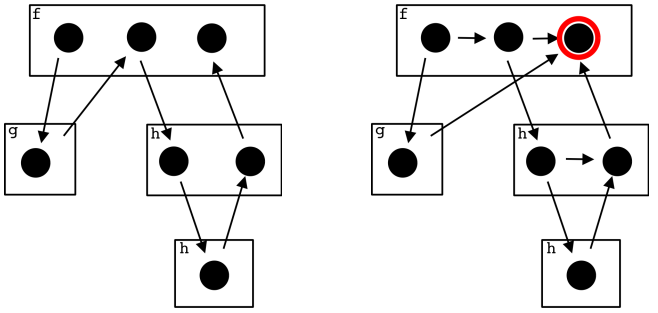
Consider the function in Fig. 5 (written in the C programming language) which attempts to prove a formula `fm` by using two methods, represented by the functions `g` and `h`, where `h` is a recursive function that calls itself.

```
bool f (formula fm ) {
 bool r1, r2;
 r1 = g(fm);
 r2 = h(fm);
 if( r1 || r2 ) return true;
 else return false;
}
```

Fig. 5. A function that could be parallelized

A sequential execution of the code might produce a control-flow graph as in Fig. 6(a). Such an execution would not take advantage of the fact that the calls to `g` and `h` are independent. The same function with a few annotations could be parallelized in parallelization frameworks for C, e.g., Cilk. Cilk would produce a control flow graph like the one in Fig. 6(b).

The function calls are executed in parallel with the rest of the function. Results are collected at a sync point, at which the calling function blocks until



(a) Sequential execution (b) Parallel execution in Cilk

Fig. 6. Execution models

its spawned function calls return. The primary design goal of Cilk is efficiency in the case when *all* of these subtasks need to be performed, i.e. the case of and-branching.

The red circle signifies what is called an *inlet* in Cilk terminology. It is a piece of code that is called whenever a function executing in parallel returns. The inlet can cause the other active function calls to abort. For example, if `g(fm)` returns `true`, it would be possible to write an inlet that aborts the call to `h`. Thus, Cilk can emulate or-branching. However, Cilk is not designed for doing or-branching fairly. It uses a work-stealing system to map tasks to processors but does not share multiple tasks on a single processor. If there are more tasks than processors (e.g., 2118 in Fig. 1) and some of them do not terminate then others may starve, which would render the proof search problem incomplete due to fairness issues.

Our model makes two major improvements, reflected in Fig. 7. The boxes, which represented stack frames in the original Fig. 6, now represent computation nodes.

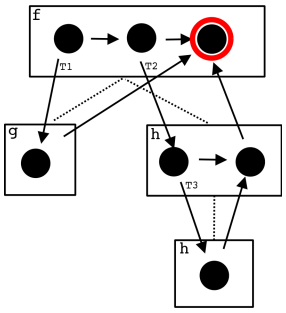


Fig. 7. Parallel execution in Banyan

The first improvement is that the calls to parallel functions are annotated with a number of *tickets* to pass to the call. These tickets are meant to indicate the proportion of total processor time that should be spent on each branch. The application programmer can assume the existence of a scheduler that will give each node an amount of time approximately proportional to its share of tickets. Delivering on this promise is the difficult part of making our programming model useful, as we detail in the next section.

Second, there are message-passing channels, represented by dotted lines, between a parent and its children, allowing local sharing of information. We work under the assumption that nodes can be moved around among different processors

Each node has a `work` function which is where its main functionality will be specified. If the node is a leaf, this function might call a computationally expensive decision procedure. An internal node would compute possible rule applications, or generate (differential) invariants. It might construct new nodes and register them as its children and give them tickets.

A node must have `childReturned` function to deal with a child's result. E.g. an or-node deals with a child returning with a proof by immediately returning with a proof. Finally, a node must also have `handleMessage` function, which specifies what to do when a message arrives. A typical implementation might selectively pass a modified version of the message to some of the node's children.

5.2 Example Proof Strategies

In this model we can easily simulate common sequential search strategies, which we generalize to parallel processing. To perform a depth-first exploration, a node can give all its tickets to its leftmost active child. To perform breadth-first search, a node can evenly split tickets among its active children. We can also use more sophisticated strategies such as what might be termed *staged search*, in which we explore breadth first for nodes that have more than a certain number of tickets, and we explore depth-first otherwise. This strategy might be desirable to prevent a search from spreading itself out too thin.

The message-passing feature allows, for example, sharing of information about successful invariants. It is often the case that a hybrid system will have a differential equations for continuous evolutions inside of a loop. Then, during proof search we will be trying many possible loop invariants, and for each we will be trying many differential invariants. Differential invariants that succeed for one loop invariant might be more likely to succeed for another or may even be guaranteed to work already. By communication through their common ancestor, these branches can explore more promising options first and potentially save time.

In practice we have found that having fairly scheduled or-parallelism is very useful. In our logic for hybrid systems, there are two proof rules that work by solving differential equations. One gives a difficult subgoal. The other gives an easier subgoal but is incomplete. It is difficult to know *a priori* whether the easier one will work. We have found that trying them both in parallel is very effective and can save hours of work.

6 Implementation: Banyan

Banyan is implemented in Scala [19], an object-oriented language built on the Java Virtual Machine, providing extensive support for functional programming. The main logical components of runtime instance of Banyan are a *coordinator*, a *client*, and one or more *workers*.

6.1 Worker implementation

Most of the core functionality of Banyan is provided by the workers. During execution there will be one worker for each processor being utilized. The proof nodes will be distributed among the workers. A worker has two threads: 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 to cause work to get done by calling the `work` function on each node that it owns.

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 notifies the node and waits for it to yield control. If new nodes were created during the timeslice, they are added at the end of the task queue.

If a node needs more than one timeslice to finish 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. If 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 its `work` function. 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.

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*. Each worker strives to keeps its surplus close to zero.

Periodically (once every two seconds for our experiments), 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 then communicates with the coordinatoor to find a good worker to send the subtree to. If such a worker exists, the subtree is shipped there.

We are still investigating possible metrics for assessing the suitability of a subtree (or possibly set of subtrees) to be shipped to other workers. The present

version of Banyan uses a slightly modified version of a "lowest absolute value" metric described. We are also examining what to do when the tickets are not evenly enough distributed, in which case fairly distributing the nodes is impossible.

6.2 Memoization

Banyan provides another feature. The application programmer can specify a key for each node. Before a node is created, Banyan will check the global shared database to see if a node with the same key has already been created. If so, the existing node is used in place of a new one. In this way, the computation tree becomes a DAG. This option is not yet fully operational in our implementation, but there are no fundamental barriers to having it so. We imagine that this feature would be very useful for theorem proving in logics where identical subgoals are often created.

6.3 DLBanyan

On top of the Banyan framework we have built DLBanyan, a theorem prover for hybrid systems. To illustrate a typical Banyan program, here is a slightly simplified code sample from our implementation of and-nodes:

```
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 the node 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 node returns indicating that it also cannot find a proof.

7 Experimental Results

We present experimental results in Table 1. All of the DLBanyan proofs were run on a Mac Pro with two 2.66 GHz quad core Intel Xeon processors and 16 GB of RAM. We show a visualization of the car control proof search problem in Fig. 8. Nodes are drawn with an area proportional to the amount of time spent on them. Blue nodes returned with a proof, orange nodes returned disproved, and gray nodes were aborted after becoming irrelevant to the main proof search. The darker blue nodes are part of the successful complete proof.

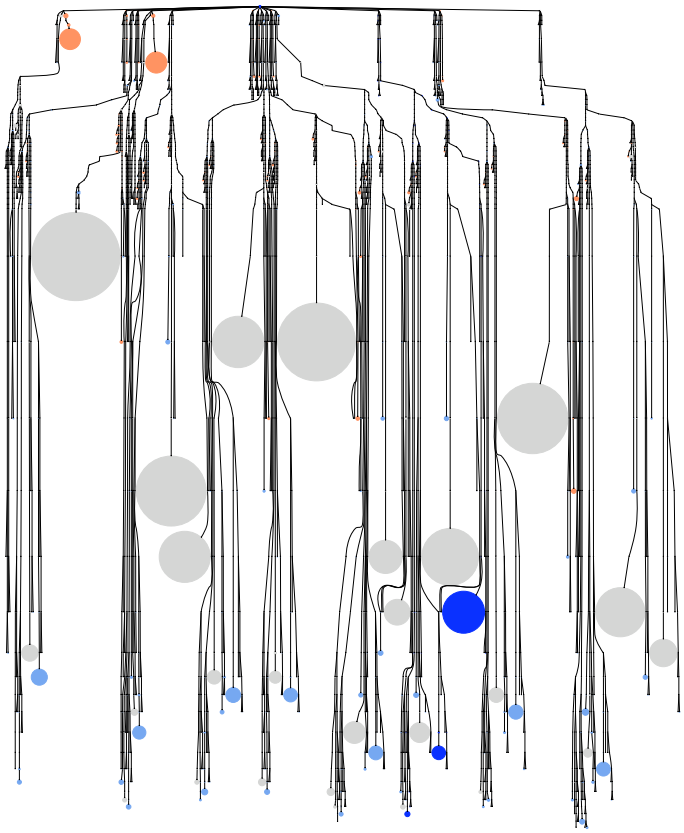


Fig. 8. Proof search for car control (full invariant search)

In the car control problem with full invariant search (Fig. 8) the root node has 16 children, corresponding to 16 choices for an invariant in the first step of the proof. DLBanyan is able to parallelize the work on these branches with near linear speedup for up to at least six workers. We also performed the proof search with a good loop invariant preselected. In that case, the search only has two large chunks of work which could be performed in parallel. As Table 1 shows, DLBanyan does indeed exploit that parallelism.

Observe that DLBanyan can prove the switching damped oscillator example [2], but KeYmaera cannot prove it automatically. (The 148s cited in Table 1 requires a user interaction.) The reason is an inherent limitation in the sequential implementation of the proof search strategies in KeYmaera: KeYmaera follows a sequential proof strategy and uses timeouts for deciding when to move on to a different choice for the current nondeterminism. With its sequential proof search implementation, KeYmaera does not manage the real concurrent proof structure and cannot revisit earlier proof options in different subgraphs of the proof search that had timed out earlier. In KeYmaera we found it extremely hard to

Table 1. Experimental results: time in seconds of verification in KeYmaera compared to Banyan with 1/2/3/4/5/6 workers. The (*) signifies a non-automatic proof.

Hybrid systems	KeYmaera	Banyan workers					
verification problem	Prover	1	2	3	4	5	6
simplified car control	415	3					
car control	6110	26	17	19			
car control (full invariant search)	-	137	67	42	44	36	24
switching damped oscillator	148*	2					

make proof search more complete and automatic without explicitly addressing the parallel structure in proof search as Banyan allows us to do. This is one of the reasons why we are convinced that Banyan is a successful framework for developing automatic proof search procedures. DLBanyan provides less functionality than KeYmaera, but it follows a much better proof exploration strategy.

We must admit that some of the difference in running time between KeYmaera and DLBanyan is due to more mundane reasons. In the course of developing DLBanyan, we discovered several improvements to our proof search methods that seem to significantly lower running time. Two things in particular seem to help significantly: performing substitution to eliminate auxiliary variables, and universally closing a formula before sending it to the real arithemtic decision procedure. However, these are steps that are not always clearly beneficial, and one of the reasons we are developing banyan is so that we can specify strategies that try different choices for such options in parallel.

8 Conclusions and Future Work

We have presented a new programming model that fits naturally to the parallel proof exploration in (backwards style) automated theorem proving. Our framework makes it easy to specify advanced fair parallel proof search strategies. Our approach directly adopts and/or proof structures in theorem proving and uses a time sharing primitive we call tickets to control fair exploration dynamically. Our implementation of this framework, Banyan, takes care of the distribution and redistribution of workload to the processors following the ticket guidelines and provides local communication among proof nodes for dynamic re-prioritization and sharing of partial successes and proof hints. In the Banyan framework, we have implemented a theorem prover for hybrid systems called DLBanyan. Experimental results show that our concurrent representation of the proof search problem enables us to find proofs automatically much faster than in a previous sequential implementation. The fact that Banyan directly allows the exploration of concurrent proof options even enables us to prove two cases automatically that previous provers could not.

At the same time, DLBanyan still falls short of providing means for interactive user guidance, which we have found to be quite useful in proving more advanced verification problems interactively in KeYmaera. A combination of the

parallel proof search framework Banyan with the capabilities of KeYmaera is promising future work.

Banyan is available for download at `www.cs.cmu.edu/~renshaw/banyan`.

References

1. Henzinger, T.A.: The theory of hybrid automata. In: LICS, IEEE CS (1996)
2. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Heidelberg (2010)
3. Platzer, A.: Differential dynamic logic for hybrid systems. J Autom Reas **41**(2) (2008) 143–189
4. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. J. Symb. Comput. **12**(3) (1991) 299–328
5. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. **5**(1/2) (1988) 29–35
6. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In Armando, A., Baumgartner, P., Dowek, G., eds.: IJCAR. Volume 5195 of LNCS., Springer (2008) 171–178
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, Berkeley, CA, USA, USENIX Association (2004)
8. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys, ACM (2007) 59–72
9. Randall, K.H.: Cilk: Efficient Multithreaded Computing. PhD thesis, Department of Electrical Engineering and Computer Science, MIT (1998)
10. Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel language. Journal of Parallel and Distributed Computing **21**(1) (1994) 4–14
11. Jones, S.P., Leshchinskiy, R., Keller, G., Chakravarty, M.M.T.: Harnessing the multicores: Nested data parallelism in haskell. In Hariharan, R., Mukund, M., Vinay, V., eds.: FSTTCS. Volume 2 of LIPIcs., Dagstuhl-Leibniz-Zentrum (2008) 383–414
12. Böhm, M., Speckenmeyer, E.: A fast parallel sat-solver - efficient workload balancing. In: Annals of Mathematics and Artificial Intelligence. (1996) 40
13. Hamadi, Y., Sais, L.: Manysat: a parallel sat solver. Journal on Satisfiability, Boolean Modeling and Computation (JSAT (2009)
14. Schumann, J., Letz, R.: Partheo: A high performance parallel theorem prover. In: CADE, Springer (1990) 40–56
15. Ertel, W.: Or-parallel theorem proving with random competition. In: LPAR '92: Proceedings of the International Conference on Logic Programming and Automated Reasoning, London, UK, Springer-Verlag (1992) 226–237
16. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation. (1998) 212–223
17. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. **20**(1) (2010) 309–352
18. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. Form. Methods Syst. Des. **35**(1) (2009) 98–120
19. Odersky, M.: Scala. http://www.scala-lang.org (2010)