Fault-Tolerant Distributed Theorem Proving

Jason Hickey*

Cornell University

Abstract. Higher-order logics are expressive tools for tasks ranging from formalizing the foundations of mathematics to large-scale software verification and synthesis. Because of their complexity, proofs in higher-order logics often use a combination of interactive proving together with computationally-intensive tactic applications that perform proof automation. As problems and proof automation become more sophisticated, these proofs represent substantial investments—each interactive step may represent several hours of design time.

We present an implementation of a distributed proving architecture to address the problems of speed, availability, and reliability in tactic provers. This architecture is implemented as a module in the MetaPRL logical framework. The implementation supports arbitrary process joins and all-but-one process failures at any time during a proof. Proof distribution is completely transparent; the existing tactic base is unmodified.

1 Introduction

In recent years, there have been many example of significant formalization efforts in higher-order logics, including Nipkow's formalization of Java [15], Howe's verification of the SCI cache-coherency protocol [7], Miller and Srivas's verification of the AAMP5 avionics processor [14] in PVS [3], the verification and automated optimization of Ensemble protocols [11], and many others. Higher-order logics are often chosen for these endeavors not only because they can formalize meta-principles, but also because they retain the conciseness and intuition of the original design.

Proofs in higher-order logic can be computationally complex. It is easy for proof search to wander into areas of intractability, and many systems like HOL [9], Coq [5], Isabelle [16], and Nuprl [6] combine interactive guidance with automated proving using *tactics*, which are programs that define domain-specific proof procedures and heuristics. While this model has been successful, it is expensive. Many of the examples mentioned above represent years of processing time and human effort. The expense, especially the human part, is the major obstacle preventing widespread adoption of these practices.

* Support for this research was provided by DARPA grant F30602-95-1-0047. Address: Jason Hickey, Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853 USA. Tel: 607-255-0408, Email: `jyh@cs.cornell.edu`.

How can we ease the burden of proof? In this paper we aim at increasing the power of proof automation by exploiting widely available computational resources. We present an architecture, implemented in the MetaPRL logical framework, for distributing tactic proving over large groups of processors using the Ensemble group communication system. To counteract problems of reliability in distributed environments, we use Ensemble's fault recovery support to allow arbitrary reconfigurations at any time in a proof (if at least one process remains running at any given time).

To preserve the large existing tactic base, we replace the existing tactic implementation with a functionally equivalent distributed tactic scheduler. No extra knowledge is required to use distributed proving, and our measurements show significant speedup for medium-sized groups.

In the remainder of the paper, we introduce the distributed architecture in three steps: in Section 2 we give a general overview of tactic proving; in Section 3 we generalize the tactic model to multi-threaded proving; and in Section 4 we present the communication model and the distributed proving architecture. In Section 5 we present some performance results for some of the logical domains in MetaPRL, including first-order logic and the Nuprl type theory. We summarize our results in Section 6 and we finish with a discussion of related work in Section 7.

2 Refinement Architecture and Tactics

Foundational LCF-style tactic provers like Coq [5], Nuprl [6], and the MetaPRL logical framework [12] perform proofs in a backward-chaining goal-directed style. Given a *goal* sentence like the sequent $A, A \Rightarrow B \vdash B$, the user selects a *tactic* to apply to the sentence in an attempt to either prove it, or reduce it to a set of simpler *subgoals*. When a tactic is applied, it reduces the proof to the set of primitive inferences defined by the logic. For example, a proof of the sequent above is given in Figure 1. The `dT` tactic is a generic tactic for applying introduction and elimination rules, and the `trivialT` tactic applies "trivial" rules like the AXIOM rule of propositional logic.

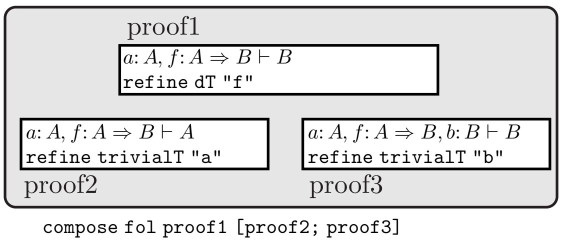


Fig. 1. Example propositional proof

The proving architecture is implemented in several parts, shown in Figure 2. For each logic, there is a module defining the primitive rules of the logic. There is a logic engine, called the *refiner*, for applying a primitive rule to a goal to reduce it to a (possibly empty) set of subgoals, and there is a *primitive tactic* module that defines the basic tactic operations. Proof automation is implemented in the *tactic library*.

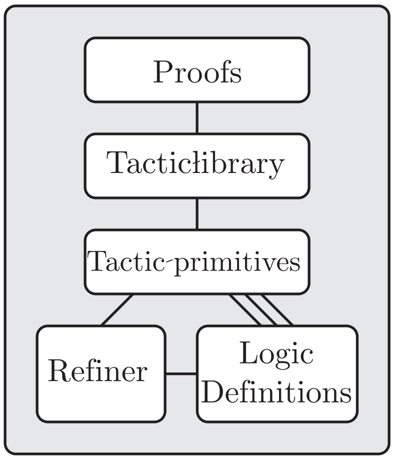


Fig. 2. Tactic-prover architecture

For each primitive rule, the refiner defines a primitive object that can be used to apply the rule. In addition, the refiner exports the following operations.

```
module Refiner : sig
    type term, rule, logic, proof
    exception RefineError
    val refine : logic → rule → term → proof
    val compose : proof → proof list → proof
    val goal_of_proof : proof → term
    val subgoals_of_proof : proof → term list
end
```

The `term` type is used to represent logical sentences. The `rule` and `logic` types are used to represent primitive rules and logics (we omit the construction of logics here, more detail can be found in Hickey [12]). The `proof` type represents *partial* proof trees that have been constructed by refinement; the `refine` function applies a `rule` to a goal to produce a partial proof. The `compose` function is used to compose partial proof trees. It is an error if any of the subgoals of the proof are not the same as the goals of the subproofs. If an error occurs at any time, the `RefineError` exception is raised.

The tactic-primitives module simplifies the application of the refiner's rules by supporting proof search and composition with *tacticals*. The tactic interface is defined as follows:

```
module TacticPrimitives (Refiner) : sig
    type prim_tactic, proof
    type tactic = term → prim_tactic
    val tactic_of_rule : Refiner.rule → tactic
    val proof_of_goal : logic → term → proof
    val refine : proof → tactic → proof
    val andthenT : tactic → tactic → tactic
    val orelseT : tactic → tactic → tactic
end
```

The tactic-primitives module re-implements the `proof` type to contain single node proof trees. It contains a constructor for building tactics from rules, and it implements a `refine` function that works with tactics. The `tactic` type itself is implemented as a function that returns a primitive reasoning operation, given the goal the tactic is being applied to. The tactic module also implements two *tacticals:* the `(anthenT tac1 tac2)` tactic applies `tac1` to the goal, then applies `tac2` to each of the subgoals. The `(orelseT tac1 tac2)` tactic applies `tac1` to the goal. If it succeeds, the proof is returned; otherwise, `tac2` is applied instead.

When embedded in a meta-language like OCaml, these primitives are sufficient to encode proof search. For example, a tactic to prove propositional sentences with implication is shown in Figure 3. The `axiom`. `imp_elim`, and `imp_intro` tactics are constructed from the rules given by the refiner. The `onSomeHypT` tactical searches the hypothesis of the goal sequent for a successful application of the `imp_elim` or `axiom` rules.

```
(* Refiner declarations:
* val imp_elim : int → tactic
* val imp_intro : tactic
* val axiom : int → tactic
*)
let rec autoT t =
    orelseT (onSomeHypT 0 imp_elim) (orelseT (onSomeHypT 0 axiom) imp_intro)
and onSomeHypT tac i t =
    if i = hyp_count t then
        raise RefineError
    else
        orelseT (tac i thenT autoT) (onSomeHypT tac (i + 1)) t
```

Fig. 3. Proof procedure for propositional logic

The implementation of the tactic primitives is straightforward. The `prim_tactic` type is implemented as the refiner type `proof`, the `tactic_of_rule` function calls the refiner with the goal argument, and the `refine` function is just the identity.

The `andthenT tac1 tac2` tactical performs the refinements specified by the tactics `tac1` and `tac2`, then composes the result. The `orelseT tac1 tac2` tactical applies `tac1`, catches the exception and applies `tac2` if `tac1` failed.

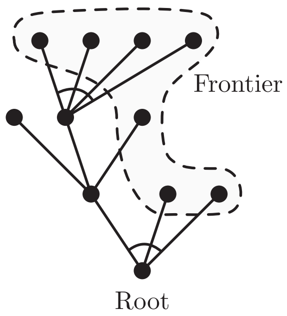
```
module Tactics (Refiner) = struct
    type prim_tactic = logic → proof
    let tactic_of_rule rule t logic = Refiner.refine logic t
    let andthenT tac1 tac2 t logic =
        let p = tac1 t logic in
        let tl = Refiner.subgoals_of_proof p in
        let pl = List.map (fun t → tac2 t logic) tl in
            compose p pl
    let orelseT tac1 tac2 t logic =
        try tac1 t logic with
        RefineError → tac2 t logic
end
```

This simple implementation is sufficient for single-threaded proving; in the next Section we generalize the model to multi-threaded proving.

3 Multithreaded Refinement

Multithreaded refinement is a useful tool for interactive provers because it allows the user to run several tasks at once, making it possible to continue interactive development of unfinished proof nodes while other nodes are being searched by the refiner. In our implementation, multithreading is a prerequisite to distributed refinement. In this Section, we cover the main issues for multithreading: the implementation of a proof tree data type that supports concurrent operations, and a policy for thread scheduling.

3.1 Proof tree data type for concurrent processes



A tactic proof constructs a primitive proof tree, in which each node in the tree is labeled by a goal sentence and a tactic that was applied to the goal to produce the children. While the proof is being performed, the *frontier* of the tree contains pending subgoals. The diagram at the right shows a partial proof tree. The root is at the bottom and nodes labeled with `andthenT` tacticals are noted with an arc. In the Figure, the root is and-branching. The left-most node has been explored first; it is or-branching where the leftmost child has failed, and the second child is and-branching. Since tactics are (by design) functional, the *order* in which the frontier nodes are explored is unimportant, and we can assign threads to multiple nodes in the frontier.

To implement the proof tree data type, we label nodes in a proof tree with seven kinds of labels:

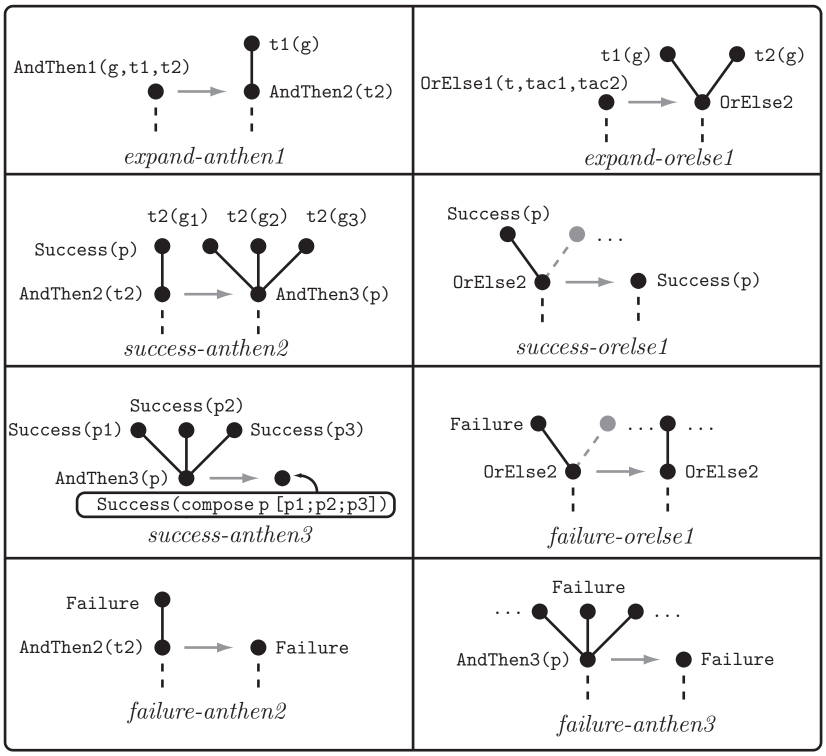


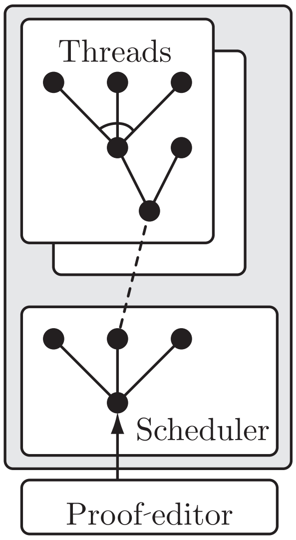
Fig. 4. Proof tree operations

```
AndThen1 (goal, tac1, tac2)
AndThen2 (tac2)
AndThen3
OrElse1 (goal, tac1, tac2)
OrElse2
Success (p)
Failure
```

The `AndThen1` and `OrElse1` labels correspond to applications of the tacticals at leaf nodes in the tree. The `AndThen2`, `AndThen3`, and `OrElse2` labels are used to label interior nodes as the proof tree is expanded. The `Success (p)` labels a leaf node for a successfully completed proof, and the `Failure` label marks frontier nodes that have raised an exception.

The complete set of operations on proof trees is shown graphically in Figure 4. These operations define expansion of `AndThen1` and `OrElse1` nodes, and they define how to back-propagate `Success` and `Failure` labels. A tree is complete when the root node is labeled either `Success(p)` for some proof `p`, or `Failure`. Note that two operations, *failure_andthen* and *success_orelse*, cause pruning of sibling nodes in the proof tree.

3.2 Threaded Refinement



To implement threaded refinement, we need to agree on a policy for the expansion of frontier nodes in the tree. The implementation we use in MetaPRL schedules the expansion into two components, shown in the diagram on the right. There is a centralized scheduler (the *Scheduler*) that manages the root of the proof tree, and there are multiple threads that schedule subtrees chosen by the Scheduler. The Scheduler does not perform node expansion directly; it passes unexpanded frontier nodes to idle threads, and handles responses from completed threads. The scheduling policy used by the Scheduler in MetaPRL is random.

The reason for this design is efficiency. Communication between threads is costly, and it is more likely that proof nodes near the root of the tree have large subtrees. Also, while the Scheduler uses a general scheduling algorithm policy, the individual threads use an efficient left-to-right depth-first search.

There are four operations for communication between the Scheduler and the threads, shown in Figure 5. The Scheduler submits a frontier node to an idle thread with the *submit* operation, and it receives the result from a completed thread with the *done* operation. When a branch of the tree is pruned, a thread may be terminated with the *cancel* operation. If there are idle threads and no available frontier nodes to be scheduled, the scheduler can request that a thread return the root part of its tree to the scheduler with the *request* operation. By assumption, thread expansion is left-to-right, and the thread retains a single root node after the *request*.

Communication between threads is done by the OCaml `Event` module, defined with the following interface.

```
module Event : sig
    type α event
    val create : unit → α event
    val post : α → α event → unit
    val select : α event list → α
    val wrap : α event → (α → β) → β event
end
```

An α `event` is a communication channel for messages of type α. The `post` function sends a message on the channel; the `select` function suspends the calling thread until a message arrives on one of the events in the list and the message is returned.

The thread interface to the scheduler has just three functions: one to `submit` a node to be expanded, another to `request` the root node from the running

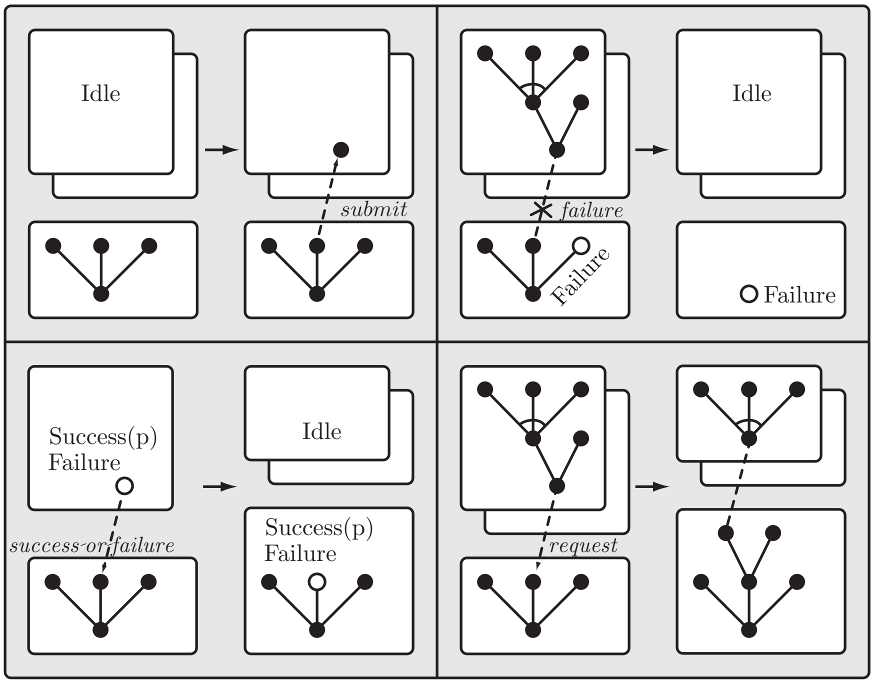


Fig. 5. Scheduling operations

thread, and another to `cancel` the subtree expansion. The result of an expansion is returned through the event produced by the `submit` function; the `cancel` function causes `Failure` to be returned immediately.

```
module type ThreadSig = sig
    val submit : node → node event
    val request : unit → unit
    val cancel : unit → unit
end
```

The Scheduler implements the tactic interface as follows. A tactic produces an unexpanded proof node. The `tactic_of_rule` function provides the base case, returning `Success (p)` if the primitive refinement had proof `p`, `Failure` otherwise. The `andthenT` and `orelseT` tacticals perform no immediate computation; they produce proof nodes, which are scheduled and expanded by the `refine` function.

```
module Scheduler (Refiner) = struct
    type node =
          Success of Refiner.proof
        | Failure
        | AndThen1 of term * tactic * tactic
        | AndThen2 of node * tactic
        | AndThen3 of Refiner.proof * node list
        | OrElse1 of term * tactic * tactic
        | OrElse2 of node * node
    and prim_tactic = node
    and tactic = term → prim_tactic
    let tactic_of_rule rule t logic =
        try Success (Refiner.refine logic rule t) with
        RefineError → Failure
    let andthenT tac1 tac2 t = AndThen1 (t, tac1, tac2)
    let orelseT tac1 tac2 t = OrElse1 (t, tac1, tac2)
    let refine t tac logic =
        match schedule (tac t) with
            Success p → p
            | Failure → raise RefineError
end
```

4 Distributed refinement

In distributed refinement, tactic evaluation occurs on multiple processors, and communication between refinement processes is through message passing. Our implementation of distributed refinement is symmetric; for each refiner process, we add an additional thread, called a *distribution service*, that requests frontier proof nodes from the Scheduler and passes them to remote refiners. There are two parts to the implementation: the *communication* model, and the *distribution service*.

4.1 Distributed Memory Architecture

The communication model that we implement uses the Ensemble group communication system, in which processes form groups. Ensemble provides failure detection when processes can no longer communicate with the group, and it supports broadcast communication in the group with the additional guarantee that all process in the group receive messages in the same order (this is provided by the *total-ordering* protocol).

Ensemble implements an *upcall* interface, where an Ensemble application receives communication events through the following interface.

```
module type EnsembleUpcalls = sig
    type process, message
    val new_view : process list → message list
    val heartbeat : unit → message list
    val install_trigger : (unit → unit) → unit
    val receive : message list → message list
end
```

These *upcalls* are implemented by the application and passed to Ensemble during initialization. Most upcalls return a message list, which Ensemble broadcasts to the group. Messages are received through the `receive` upcall. Messages can be sent using the `heartbeat` upcall, which can be triggered by the function provided by the `install_trigger` upcall, which is called once when the application is initialized. The `new_view` upcall provides information about group membership. If the status of the group changes, the `new_view` function is called with a new list of processes belonging to the group.

We implement a shared-memory model using Ensemble. The memory entries initially correspond to frontier proof nodes. Entries may be *locked* or *unlocked*. Locks are exclusive; if an entry is locked, it is locked by exactly one process. Entries are created in the unlocked state, and the process that created the entry is called its *owner*. When a refiner process has no more unfinished frontier nodes, it may request a lock on an entry, with the intention of expanding it and returning the result value to the owner.

```
module DMA : sig
    type (α,β) handle
    val store : α → (α,β) handle
    val lock : ((α,β) handle → unit) → α handle
    val delete : (α,β) handle → unit
    val return : (α,β) handle → β → unit
    val event_of_handle : (α,β) handle → β event
    val get_value : (α,β) handle → α
end
```

Each process in the group keeps a local version of the shared memory, with entries containing the process that *created* the entry (with the `store` function), the process that *locked* the entry (with the `lock` function), and the value of the entry. The first argument to the `lock` function is a cancellation function to be called if the entry is *deleted* with the `delete` function. Only the owner of an entry is allowed to delete it (unless Ensemble determines that the owner failed, in which all copies of the entry are deleted). Once a `handle` to an entry is obtained, the value associated with the entry is be obtained with the `get_value` function. Once a computation finishes, a process *returns* a value to the owner with the `return` function, which deletes the entry from the memory.

The memory is implemented with Ensemble's broadcast mechanism. When an entry is created with the `store` function, the entry is broadcast to all processes in the group. When a lock is requested, an unlocked entry is selected from the local version of the memory, and a lock request is broadcast to the group. The lock request is *successful* if it is the first lock request to be received for that entry. Locks are granted only to the first requester; the total ordering property of Ensemble guarantees that all processes in the group agree on message ordering. If a lock request fails, and there is another unlocked entry in the shared memory, the request is issued for the new entry. Otherwise, the lock is postponed until a new entry is created.

When a `new_view` is initiated by Ensemble, it means the the group membership may have changed because of failures or the creation of new refiner processes. The `new_view` function delivers a new set of processes that compose the group. When a process fails, the entries it *created* are deleted, and its outstanding locks are removed. When a process joins the group, or when two groups merge, different process may have different local versions of the shared store. At the `new_view` a group coordinator is elected (the oldest surviving process in the group). Each process sends its local store to the coordinator, which merges the copies, deletes any entries for which the owner has failed, and broadcasts the result.

4.2 Distributed Refinement

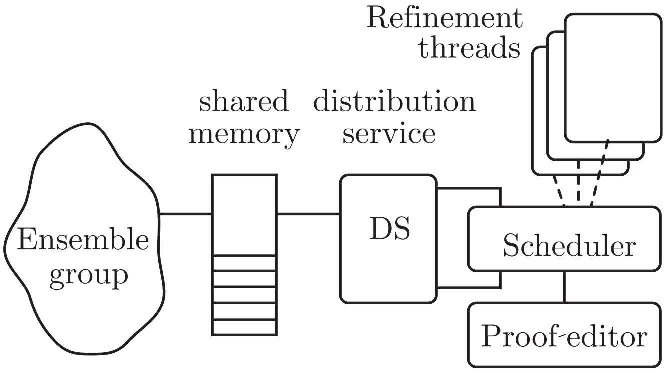


Fig. 6. Distributed refiner block diagram

Distributed refinement is implemented by augmenting the threaded refiner with a *distribution service* (DS). A block diagram of the distributed refiner is shown in Figure 6. To the Scheduler, the DS acts like a thread (it implements the `ThreadSig` interface). When the scheduler `submits` a node to the DS, the DS `stores` the node in the shared memory. If the Scheduler `cancels` the node, the DS `deletes` the entry from the shared memory. The Scheduler treats the `request` event specially. When the Scheduler makes a `request` for a new node, the DS attempts to `lock` an entry in the shared memory. If the lock is successful, the subgoal associated with the entry is passed to the Scheduler as a new frontier node. When the Scheduler completes the evaluation, it passes the result back to the DS, which `returns` the result to the owner.

In the Figure, the proof editor is optional; there may be only one process in the group with a proof editor. When an idle process joins the group, it synchronizes its shared memory, and it may pass new nodes to the scheduler for refinement. As the scheduler runs, new subgoals are passed back to the DS to be performed remotely. As a large proof is developed, a complex dependency graph is established between the proof nodes of different process. When a process fails, a large portion of the proof tree may be pruned.

5 Performance Measurements

We include performance measurements for four domains:

- The `Pigeon` domain is a proof of the propositional pigeonhole principle. The `Pigeon3` problem shows that 4 pigeons do not fit into 3 holes, and the `Pigeon4` problem shows that that 5 pigeons do not fit into 4 holes.
- The `Gen` domain is a proof of heredity in a large geneological database in first-order logic.
- The `Term` domain is a proof of correctness of `term` operations in the Nuprl type theory.
- The `CZF` domain is a proof of the derivation of Aczel's CZF set theory from the Nuprl type theory.

The `Pigeon` and `Gen` domains use fully automated proof search. The `Term` and `Set` domains are automated replays of proof transcripts for interactively generated proofs. The transcripts contain the full search problem posed by the user; the `Term` domain contains about 2700 interactive proof nodes, and the `CZF` domain contains about 2300 (the proof steps in `CZF` are significantly simpler than those in `Term`). These two domains each represent about three weeks of interactive development.

The runtimes for each of these problems are shown in Figure 7 for one to five processors in the Ensemble group, running on Linux 200MHz Pentium 686 machines. We also include data points for the original unthreaded tactic primitives for comparison.[1]

The use of the distributed prover usually incurs a penalty over the the unthreaded prover. The speedup on these problems is fairly consistent, averaging about 3.2 with for the 5 processor case. The GEN problem is an exception; it acheived a superlinear speedup. On this problem, the random scheduling algorithm performs better than the default depth-first-search.

[1] To the reviewers: these numbers reflect times for interpreted OCaml bytecode. We are currently modifying the OCaml marshaler to marshal native-code functions, and we would like to update these numbers before the final version of this paper. We expect the speedup to decrease somewhat as the relative cost of communication increases.

Problem	unthreaded	1 proc	2 proc	3 proc	4 proc	5 proc	1 failure	messages	bytes
`Pigeon3`	7	8.6	7.2	6.2	4.5	3.8			
`Pigeon4`	207	221	141	95	80	67	95	969	2.27×10^6
`Gen`	109	118	73	41	28	20	45	2352	9×10^6
`Term`	422	440	250	201	160	137	180		
`CZF`	63	75	40	32	26	22	47		

Fig. 7. Performance on the five domains

The performance numbers under failure measure the computation for a group of size five, with one failure approximately half way through the computation. Part of the extra time is devoted to failure detection (Ensemble has a default 15 second timeout), part is due to the computation that was lost to the failed process, and part is due to the smaller group size after the failure.

The communication cost on these problems is high—the largest GEN problem generates 0.45MB/sec in communication. The proof nodes that are being sent between machines contain tactics and proof trees that grow to several 100KB as the problem runs. Much of this communication is unnecessary, since the proof trees are often copied and discarded. Total memory requirements also increase with the cost of communication because the OCaml marshaler makes a deep copy of every value that is sent on the network.

Communication cost tends to be the major bottleneck. On these problems, network latency was a greater problem than lock contention. In addition, the larger problems were more likely to undergo false failures. A generic technique for addressing these problems would be message compression: most goals that are passed sequentially across the network share common subterms. A caching marshaler would preserve sharing for common subterms in multiple message, addressing the problem of message size and memory usage.

6 Summary, implementation notes, and future work

The distributed architecture we have presented owes its simplicity to the modular design of MetaPRL and Ensemble. The distributed tactic module in MetaPRL transparently replaces the unthreaded tactic interface between the tactic library and the logic engine, making it unnecessary to modify the tactic library. Ensemble also provides a concise interface—although the internal mechanism for ensuring consistency is quite complex.

Several interesting problems arose in the implementation. One problem was that Ensemble was not designed for a multithreaded environment, and it initially provided poor performance. When our effort to augment Ensemble achieved only modest gains, we separated the MetaPRL and Ensemble processes. Communication between MetaPRL and Ensemble is currently is through (physical) shared memory.

OCaml provided essential communication support for communicating tactics as *functions*. The OCaml marshaler provides native support for marshaling arbitrary OCaml values (including functions) to byte streams. This is not without its pitfalls. From the programming perspective, it is difficult to know which values are part of a function's closure, and the MetaPRL implementation initially attempted to marshal the entire Ensemble data structure into every message! The solution was careful coding to avoid spurious bound variables in the tactic functions. Communication becomes a bottleneck as group sizes increase, and a more sophisticated marshaler would reduce message sizes by caching recurring values.

The speedups we obtain are due to the wealth of parallelism in the proofs. However, efficiency can be greatly increased in some domain with domain-specific scheduling policies. Another issue is proof caching, where commonly occurring subgoals can be identified. We leave these two issues as future work.

7 Related Work

There are several examples of first-order proving and limited higher-order distributed theorem provers [1, 17, 2, 4]. These efforts differ from ours in at least two respects: we focus on general-purpose provers that use a mixture of interactive and automated proof, and we require fault-tolerance. However, the scheduling problem is a common issue in all implementations, and the insight gained from first-order provers may lead to a more general scheduling policy mechanism in our framework.

Our communication model (the DMA) is quite similar to Leiserson's CILK system [8], in which parallel and distributed programs are written using annotated C code. Program distribution shares many of the mechanisms we describe for the Ensemble shared memory.

The Nuprl proof development system includes both a logic and a mechanism for reasoning. An early version of the system is described in Constable et. al. [6]; more recent descriptions can be found in Jackson's thesis [13]. An account of Ensemble can be found in Hayden [10].

References

1. Maria Paola Bonacina and William Mccune. Distributed theorem proving by peers. In *1194 Conference on Automated Deduction (CADE12)*, pages 841–845, 1994.
2. D.J. Cook and R.C. Varnell. Adaptive parallel iterative deepening search. *Journal of Artificial Intelligence Research*. to appear.
3. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. `http://www.csl.sri.com/sri-csl-fm.html`.
4. J. Denzinger and F. Fuchs. Cooperation in theorem proving by loosely coupled heuristics. Technical Report SR-97-04, University of Kaiserslautern, 1997.

5. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The coq proof assistant user's guide. Technical Report Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
6. R.L. Constable et.al. *Implementing Mathematics in the NuPRL Proof Development System*. Prentice–Hall, 1986.
7. A. Felty, D. Howe, and F. Stomp. Protocol verification in Nuprl. In *CAV'98, Lecture Notes on Computer Science*. Springer, 1998.
8. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, 1998.
9. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
10. Mark Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, 1998.
11. Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In *TACAS '99*, March 1999.
12. Jason J. Hickey. Nuprl-Light: An implementation framework for higher–order logics. In *14th International Conference on Automated Deduction*. Springer, 1997.
13. Paul Bernard Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, January 1995.
14. Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995.
15. Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages p. 161–170. ACM Press, New York, 1998.
16. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994.
17. G. Sutcliffe and J. Pinakis. A heterogeneous parallel deduction system. In *Proceedings of the Workshop on Automated Deduction, FGCS'92*, 1992.