Using Parallelism to Improve Theorem Prover Interactivity

David L. Rager
ragerdl@cs.utexas.edu
May 17, 2010

Abstract

Multi-core CPUs have become commonplace in desktop computers, but theorem provers often do not take advantage of the additional resources these CPUs provide in an interactive setting. This PhD proposal focuses on automatically using these additional resources to lessen the delay between when a user submits a conjecture to the ACL2 theorem prover and when the user receives feedback from the prover useful for learning how to be successful in completing a failed proof.

Research contributions include: (1) maintaining the ability of users to interact with the theorem prover despite the use of parallel execution in its proof process, (2) mechanisms for providing early feedback to the user, (3) improving support for multi-threaded programming in an implementation language (Lisp) and a language used for reasoning (ACL2), and (4) evaluating the usefulness of parallelizing a theorem prover at the subgoal level.

1 Introduction

ACL2 is a theorem prover for first-order logic based on applicative Common Lisp. It has been used in some of the largest industrial formal verification efforts [8, 30]. As multi-core CPUs become commonplace [3], ACL2 users would like to take advantage of the underlying available hardware resources (see section 4.5 in [17]). Since the ACL2 theorem prover is primarily written in its own functional language and one can introduce parallelism into functional languages without concerns about safety (see section 1.4 in [2]), it is reasonable to introduce parallelism into ACL2’s proof process in a way that fits naturally into the ACL2 programming paradigm. By using parallelism in the proof process, we hope to lessen the delay between when a user submits a conjecture to the prover and when the user receives useful feedback from the prover concerning the conjecture’s provability.

We plan on achieving this goal through the following steps:

1. Create a version of ACL2 equivalent to the current version, which removes most sequential dependencies within the ACL2 proof process.\(^1\) Also disable the use of features that depend on sequential behavior.

\(^1\)The waterfall is the primary ACL2 proof process and further explained in section 3.1.
2. Introduce the primitives and abstractions necessary to execute the waterfall in parallel.

3. The first step requires removing I/O from the waterfall. Create a mechanism to re-incorporate this I/O.

4. Provide the user a mechanism to enable and disable parallel evaluation of the waterfall. We may eventually enable such parallel evaluation by default.

This proposal begins by outlining some of the work related to our project. We then discuss how we plan to modify ACL2 in a way that preserves soundness. We next describe the primitives and abstractions we anticipate to be necessary for introducing parallelism into the waterfall. And finally, we outline our plans for evaluating the soundness and performance of our work. The scientific contributions are as follows:

1. *Maintain Interactivity*: We will continue to support the ways users interact with ACL2 even though there will now be parallel evaluation in its proof process.

2. *Consider Mechanisms for Providing Early Feedback*: We will consider providing proof-tool users with non-deterministic feedback that could result in super-linear speedup. By this, we mean that an ACL2 user could see feedback for a later subgoal dramatically sooner in a parallelized proof process than they would receive feedback for an earlier subgoal in a serial proof process.

3. *Provide Parallelism Abstractions*: We will enrich the built-in theory with more primitives that allow programmers to create and manage parallel execution.

4. *Improve Support for Lisp-Level Programming*: We will enrich the multi-threading primitives available to Lisp programmers, upon which the parallelism abstractions are built.

5. *Evaluate Our Approach*: We will evaluate the feasibility and efficiency of parallelizing a semi-automatic and interactive theorem prover at the subgoal level.

## 2 Previous Work

There are three main bodies of previous work directly related to this project: (1) previous multi-threading Lisp abstractions, (2) multi-threading interfaces for other functional and procedural languages, and (3) the use of parallelism already available in theorem provers.

### 2.1 Parallelism in Lisp

The first body of work concerns relatively old parallel implementations of Lisp, such as Multilisp. Multilisp was created in the early 1980s as an extended version of Scheme [11]. It implemented the `future` operator, which is often defined as a promise for a form’s evaluation result (see section 4 in [12]). While our first major parallelism extension to ACL2 did not implement the future operator, Halstead’s work inspired exploration of issues like garbage collection, granularity, and the ordering of parallel evaluation. Here, we propose a parallelism extension to ACL2 that employs a future operator, which is discussed later in section 4.1.
Other parallel implementations of Lisp include variants such as Parallel Lisp [12], a Queue-based Multi-processing Lisp [10], and projects described in Yuen’s book “Parallel Lisp Systems” [38]. Our approach is different from previous approaches, in that we provide a level of logical abstraction that permits parallel evaluation without having to manage low level primitives like futures.

As stated in the introduction, one contribution of our work will be the further development of a multi-threading library for Lisp systems, specifically Clozure Common Lisp [9] and Steel Bank Common Lisp [31]. The Bordeaux Threads project is an example of a library that already attempts to unify the multi-threading interfaces of different Lisps [7]. However, since the Bordeaux-threads project makes different decisions than we wish to make,\(^2\) we continue the development of our multi-threading interface.

### 2.2 Parallelism in other Functional Languages

The second body of work concerns other functional and procedural parallelism implementations. Haskell is a well-used pure functional programming language that has parallelism variants. Haskell is a close relative to Lisp, as both support the functional paradigm, and it also has the ability to spawn threads to perform computation. This is elucidated by the Haskell manual:

There are two implementations of Parallel Haskell: SMP parallelism, which is built-in to GHC... and supports running Parallel Haskell programs on a single multiprocessor machine, and Glasgow Parallel Haskell (GPH), which supports running Parallel Haskell programs on both clusters of machines or single multiprocessors. GPH is developed and distributed separately from GHC [13].

Ordinary single-threaded Haskell programs do not benefit from enabling SMP parallelism alone. You must expose parallelism to the compiler by either explicitly creating threads or using a `par` operator (`par` has the type signature `par :: a -> b -> b` and when used in an expression such as `(x ‘par‘ y)` returns the value of `y`).

Orc is a functional mashup [20] language developed with parallelism included as a first-class feature. There are four combinators in the Orc language: the parallel combinator `|`, the sequential combinator `>>x`, the pruning combinator `<x<`, and the less well-known `otherwise` combinator `. The parallel combinator is used to spawn threads to evaluate expressions in parallel, and the other combinators help manage the flow of results from those parallel evaluations. Further details can be found in the paper of Kitchin et al. [19].

### 2.3 Parallelism in Procedural Languages

The multi-threading paradigms of C++ and Java are well known and focus on synchronization mechanisms like condition variables, locks, shared memory, and threads. Cilk is a

\(^2\)E.g., the Bordeaux-threads definition of `condition-wait` for CCL is simply to wait on a semaphore. Since a semaphore effectively “stores” a signal when there is no thread already waiting on it, this results in slightly different behavior than a traditional call to `condition-wait`, returning immediately rather than waiting, as specified by the POSIX standard [14]
C extension that provides parallelism with a very fine level of granularity [25]. Cilk operates enough threads to keep each CPU core busy, and once a thread finishes evaluation, it “grabs” another portion by stealing work from another thread. Some optimizations related to using the stack frame allow the implementation to be quick and effective. Since Lisps do not typically allow direct access to stack frames, Cilk’s techniques are not directly related to this work. However, the ability to provide parallelism for such a low level of granularity is noteworthy.

Several frameworks enable distributing computation across networks in C++, including Unified Parallel C (UPC) [21] and Message Passing Interface (MPI) [4]. Since our immediate goal is to improve the ACL2 theorem prover user’s experience, and utilizing the CPU cores already on the machine would be an improvement, we do not yet plan to distribute ACL2’s computation across the network.

2.4 Parallelism in Theorem Provers

A parallelized theorem prover is a theorem prover that develops portions of its proof in parallel with each other. Kapur and Vandevoorde developed DLP, a distributed version of the Larch Prover, as a framework for parallel interactive theorem proving. Like ACL2, DLP is a rewrite-based theorem prover with many opportunities for the parallelization of subgoal proofs [15]. They recognize both the potential for speedup and the need to support user interaction. DLP provides a primitive named spec (short for speculate) which applies a user-specified proof strategy to a conjecture or subgoal. These strategies include case splitting and induction. ACL2 provides a similar facility, with or-hints (see “hints” in [1]), except that while ACL2 currently uses or-hints in a single-threaded environment, DLP attempts their use in parallel.

In 1990 Schumann and Letz presented Partheo, “a sound and complete or-parallel theorem prover for first order logic” [33]. Partheo was written in parallel C and ran sequential theorem provers [22] based on Warren’s abstract machine on a network of 16 transputers. The authors discuss or-parallelism in section 4.1 of their paper, which is also used in their later work.

The multi-process theorem prover SiCoTHEO is a further extension of running multiple SETHEO-based provers in parallel [32]. SiCoTHEO starts multiple copies of the sequential theorem prover SETHEO [22], except with different configuration parameters. Once one of these copies finds the solution to the problem, SiCoTHEO aborts the other copies’ searches and returns the result. This approach is also similar to ACL2’s or-hints, except the search is parallelized and distributed over processes.

Wolf and Fuchs discuss two types of theorem proving parallelism: cooperative and non-cooperative [37]. A cooperative approach can use information from one subgoal’s proof in the proof of the next subgoal. A non-cooperative approach proves subgoals independently of one another. ACL2’s current approach is largely non-cooperative.3

---

3An example of a cooperative ACL2 proof technique occurs when the main goal pushes two subgoals, named *1 and *2, for proof by induction and then *2 pushes two more subgoals, named *2.1 and *2.2, and finds that *2.2 is subsumed by *1. This is one of the reasons we intentionally abstain from parallelizing the use of induction.
Other examples include Moten’s parallel interactive theorem prover MP refiner [26], Maude’s concurrent rewriting logic [24], the Peers distributed theorem proving prototype [6], and parallel Isabelle/HOL [23, 36]. The Isabelle/HOL work is best explained with a quotation from one of their papers [23]:

Isabelle proof documents follow a certain structure that allows various parallel scheduling strategies.... The main observations are as follows.

1. Large Isabelle applications consists of a DAG-structured collection of theories. Independent nodes in that graph can be loaded in parallel. This is analogous to a parallel make tool....

2. Theorem statements are explicit and proofs are irrelevant, in the sense that a theorem can be accepted as correct and used elsewhere without having checked its proof yet. It is, of course, necessary to finish proofs at some point but this can be done independently via futures....

3. Isar proofs have a rich sub-structure, where most runtime is spent in terminal justifications (small local proofs, involving potentially complex automated reasoning tools). Here is a stylized Isar proof text for illustration:

   lemma A and B
   proof
   show A by auto
   show B by blast
   qed

   These by steps can be parallelized implicitly, without having to reimplement proof tools like auto or blast involved here.

Our work is different from the above Isabelle/HOL work in the following ways. The first item above is akin to certifying the ACL2 test suite with the process-level parallelism available via GNU Make’s “make -j” feature. As previously discussed, our concern is the interactive time, not the time it takes to certify the test suite. The second item above would be akin to parallelizing the proofs of multiple ACL2 theorems. So, if an ACL2 book defined 20 theorems, you could prove each of those 20 theorems in parallel with one another. Again, this would reduce the time it takes to certify a book, but it would have no effect upon the time it takes to attempt the proof of just one conjecture. And the third and final item above would be akin to running the steps of ACL2’s proof checker (see “proof-checker” in [1]) in parallel. Isar provides a way to manually direct a proof and allows the use of heuristics within that manual direction to be parallelized. While it is possible for ACL2 users to use the proof checker, the common practice is instead to guide ACL2’s proof attempts with hints specifying the heuristics it should choose. As such, ACL2 users do not use the proof checker that often, and parallelizing the proof checker is likely to be of little value.

We are unaware of any use of parallelism in the proof processes of the Coq [5] and HOL4 [35] theorem proving systems.
2.4.1 Parallelism in ACL2

Process-level parallelism is currently available to ACL2 users via GNU Make’s “make -j” option. This option allows the certification of ACL2 libraries across multiple cores on one machine. While this option optimizes the most common benchmark for improving ACL2 performance, it does nothing for improving the interactive delay involved in using the ACL2 theorem prover. It is this interactive delay that this dissertation seeks to improve.

Additionally, we have fully implemented and integrated four parallelism primitives designed to allow an ACL2 user to evaluate expressions in parallel: plet, pargs, pand, and por [27, 28, 29]. As explained in section 4.2, we anticipate improving plet to be more versatile.

3 Preparing the Waterfall for Parallel Evaluation

ACL2’s top priority is soundness [16], which we describe as never incorrectly displaying “Q.E.D”. Modifying ACL2’s code requires scrutiny, intelligence, and knowledge of the ACL2 system. Even the authors sometimes have difficulty remembering the coding invariants necessary to soundly modify the system. Despite the authors’ extreme attention to detail and disciplined approach to modifying the ACL2 code, soundness bugs have still occurred. These bugs are corrected with each new release (see topic “release-notes” in [1]), but their existence is evidence that modifying ACL2 is difficult.

As such, and in concert with good software practice, when we prepare ACL2 for parallel execution, we must take a systematic approach with levels of abstraction that are easy to read and understand. This section outlines our systematic approach.

3.1 Introduction to the Waterfall

As part of an introduction to the waterfall, we explain our motivation for parallelizing the waterfall, instead of another aspect of the theorem prover. It is useful to view the ACL2 waterfall from two angles. The first angle introduces the flow of the waterfall’s proof process (see Figure 1). We plan on parallelizing the part that performs simplification, destructor elimination, fertilization, generalization, and the elimination of irrelevance.\(^4\) By parallelizing the proof process at this level, we hope to separate it into pieces large enough in granularity to warrant the overhead associated with parallel execution.\(^5\)

The second view of the waterfall comes from examining the functions themselves. Each of the proof steps that we want to refactor is defined by functions called by waterfall-step1. The call graph for the waterfall can be found in Figure 2. The indentations represent calls to the indented function name from the function name relative to its indent. As an example, waterfall1-lst is called from waterfall1, and since waterfall1-lst has no other function names at the same indentation level, we know waterfall1 calls no other function that we deemed to be significant enough to warrant listing. Waterfall1-lst is shown as calling itself, because it recurs on all but the first element of the given clause list. The use of the infix XOR indicates that the function calls are part of a Lisp cond that results

\(^4\)Induction occurs at a higher functional level, and as such, is currently omitted from our plans.

\(^5\)See section 5.2 for some preliminary performance results.
in calling exactly one of the functions. The optional call is on the same conceptual level as
the function call following it, so \texttt{waterfall0} is not indented. Since \texttt{waterfall0} can directly
call five distinct waterfall functions, it is the longest and most complex function listed.

\texttt{Waterfall1-lst} accepts a list of clauses to prove. It begins the proof of each clause by
calling \texttt{waterfall1} on the first element in that list of clauses. \texttt{Waterfall1-lst}'s recursive
call is on the rest of that list of clauses. In ACL2 vernacular, each of these clauses is known
as a \textit{subgoal}. Our objective is to attempt the proof of each of these subgoals in parallel.

\begin{verbatim}
waterfall
  waterfall1-lst
    waterfall1
      waterfall0-with-hint-settings [optional]
      waterfall1
        waterfall-step
          waterfall-step1
          waterfall0-or-hit XOR waterfall0-with-hint-settings
            XOR waterfall0
            XOR waterfall1-lst

waterfall1-lst
\end{verbatim}

Figure 2: Function Call Graph for the ACL2 Waterfall
3.2 Removing the Modification of State from the Waterfall

Before discussing the removal of the modification of state from the waterfall, we must first introduce the concept of state itself.

The variable state (see “state” in [1]) is intended to provide a way to record and model the modification of ACL2 global variables in a functional language (see “assign” in [1]), printing output to the screen (see “fmt” in [1]), and performing other actions with side-effects like checking the status of system calls (see “sys-call” in [1]). We are able to reliably tell where state is modified because of the following two mechanisms:

1. ACL2 restricts the name state from being used as a variable name anywhere that it would not represent this special functional wrapper for functions with side-effects.

2. State is declared as an ACL2 single-threaded object (stobj). ACL2 requires that if a function modifies a stobj, that it returns the stobj as part of that function’s return value.

Through these two mechanisms, it is not possible to modify state without the caller’s knowledge. At a higher and more meaningful level, we will know that if a function does not return state, that there were no logical side-effects during that function’s evaluation, and thus, it is safe to call that function in parallel with itself. Therefore, we can deduce that if we remove the need to return state from waterfall1-lst, that waterfall1-lst will have no side-effects and be safe to evaluate in parallel.

Note that while ACL2 is a functional language, there is some “magic” in raw Lisp involving how the special variable *ld-level* is bound and the implementation of ACL2 arrays. Part of this work involves fixing the soundness and performance issues related to these types of implementation details.

3.2.1 How State is Used in the Waterfall

All of the functions shown in Figure 2 take state as an argument and return a possibly-modified version of state. We separate the returning of state during the waterfall into three cases:

1. Sometimes state remains unmodified. In this case, it is trivial to remove state from the return value.

2. State is modified whenever the waterfall performs printing. As a first step, we have simply discarded such printing. These printings can be subdivided into “progress reports” and “error reports.” Progress reports do not affect the prover’s control logic and as such can be temporarily discarded.

---

6The lack of logical side-effects is useful because almost all side-effects (including most of the printing that occurs) are included in the logic. Some counter-examples include printing with cw (“comment window printing”) and printing via ACL2 “wormholes”.

7It is only safe to call the waterfall functions in parallel with one another, because there is no guarantee that other functions will not write to the data these waterfall functions read. We must also know that the waterfall functions are the only functions running in parallel before we obtain this second guarantee.
without concern for soundness. However, error reports occur when encountering an error and should result in halting of the current proof attempt. These error signals should continue to be propagated up to the logic that results in such halting, and we will need to carefully inspect our changes to make sure we do not omit such propagation.

3. **State** is also modified when using certain features of the theorem prover, such as clause processors and computed hints. For now, we simply disable those features and cause a hard error whenever a proof attempts to use them. Disabling these features results in being unable to verify 7.3% of the regression suite.

After removing output from the waterfall and disabling the features that require modifying **state** during the waterfall, we will be able to remove the modification of **state** from the waterfall. Section 3.4 elaborates upon how we perform this removal in a manner designed to be easy to check and not prone to errors. After removing **state** from the waterfall, it is possible to parallelize the call of **waterfall1** on one subgoal with the call of **waterfall1-lst** on the remaining subgoals. After these parallelized subgoal proofs finish, we will need a way to combine the results. These results are contained in the **pspv**, which we discuss next.

### 3.3 Limiting PSPV Dependence

**State** is not the only single-threaded variable in the waterfall; there is also the variable **pspv**, which is short for **prover-special-variables**. Despite the name, the prover-special-variables are not actually **special** in the Lisp sense. They are, however, intended to be a place to store assignments to variables that would be global if ACL2 were written in a more imperative manner.

**Prove-spec-var** is an ACL2 record that contains thirteen fields, including the tag-tree, pool of clauses to prove, hint settings, gag state, etc. Of these fields, only the tag-tree and the pool of clauses to prove are modified by a call to the waterfall.\(^8\)

#### 3.3.1 Accounting for Changes to PSPV’s Tag-Tree and Pool

This results in a need to account for changes to the **tag-tree** and pool of clauses, which we will henceforth refer to as the **pool**. We omit a discussion of how these two sub-variables are changed and instead outline how we plan to combine these changes. Combining the changes to tag-trees is straightforward, as we simply use the time-tested function **cons-tag-trees** to combine the tag-trees. The order of the arguments to **cons-tag-trees** matters, and we “cons” the second set of changes onto the first set of changes. This results in tag-trees similar enough to the tag-trees generated by an unaltered ACL2, such that the flow and generation of proofs is unaltered. Additionally, since tag-trees are sets, combining elements in this way does not present a soundness issue.

Accounting for changes to the **pool** is more difficult. Combining pools requires three inputs: the original pool, the first new pool, and the second new pool. In the common case, the first and second pool are both extensions of the original pool. In this case, we determine

---

\(^8\)We test for this property by adding a dynamically-checked assertion that checks for equality of the non-tag-tree/pool portions of the **pspv**.
the differences between the original and first pool, the original and second pool, and then just append those differences to each other and the original pool.

There is currently a second case that occurs when we do not know how to combine the changes to the \texttt{pspv} pools. We hope to eventually always be able to combine these changes, but we are still experimenting. An example of a case we do not yet know how to combine occurs when one of the pools contains a \texttt{to-be-proved-by-induction} tag that is not stored in the same element slot as in the other pool. When this happens, we realize that we can not combine the pools, and we re-call the function that generated the second set of pool changes, passing in the intermediate \texttt{pspv} instead of just the original \texttt{pspv}. In the ACL2 implementation, the function that needs to be re-called is the \texttt{cdr} recursion of \texttt{waterfall1-lst}.

### 3.4 Obtaining Abstraction with Macros

One bold approach to modifying ACL2 would be to simply replace the definitions of the waterfall functions with versions that remove the modification of \texttt{state} from the waterfall, combine the \texttt{pspv}'s, disable problematic ACL2 features, and prove the subgoals in parallel. While faster to deploy, such changes would likely be unincorporated into the main ACL2 development branch. A more incremental and modular approach is desired.

Our approach semi-automatically transforms the current waterfall and potentially sequential-\texttt{pspv} ACL2 code to avoid modifying \texttt{state} and soundly combine \texttt{pspv}'s in a way that is easy to check. The macros introduced in this section are \textit{not} intended to introduce parallelism. These macros are instead intended to help construct a version of ACL2's waterfall that can be evaluated in parallel by introducing a parallelism primitive into its definition.$^9$

#### 3.4.1 A Modular Approach

One aspect of our modular approach involves creating macros that define their bodies differently, depending upon which compile-time features are enabled. Take the macro \texttt{mv}. We define a macro \texttt{stateless-mv}, which, when feature \texttt{stateless-waterfall} is missing, simply converts the form to a call of \texttt{mv}. However, when feature \texttt{stateless-waterfall} is present, it returns an \texttt{mv} that discards its final argument. As such, we can change our calls of \texttt{(mv ... state)} inside the waterfall to \texttt{(stateless-mv ... state)} without worrying about affecting the stateful version of the waterfall. This nets us two properties: (1) an observer can look at the differences between the two definitions of the modified waterfall function and very quickly see that the change was trivial and (2) we did not have to pollute our waterfall code with conditionals based on compile-time features. Thus far we have defined stateless versions of ACL2 macros \texttt{value}, \texttt{er}, \texttt{er-let*}, \texttt{mv}, \texttt{pprogn}, \texttt{warning$\$}, \texttt{er-progn}, \texttt{io?}, \texttt{io?-prove}, and \texttt{mv-let}. Additionally, we have a macro that returns \texttt{state} or \texttt{nil}, depending upon the compile-time feature.

Most of the macro definitions are straightforward, but we highlight one interesting definition, that of \texttt{pprogn}. Its definition is simple:

$^9$An example of such a code transformation would be the changing of a \texttt{let} to a \texttt{plet}
(defmacro stateless-pprogn (&rest rst)
  #+stateless-waterfall (car (last rst))
  #-stateless-waterfall '(pprogn ,@rst))

The interesting part is that the stateless version discards all but its last form. Since the intermediate forms of pprogn can only return state, and since we are removing the modification of state from the waterfall, there is no longer a reason to execute those forms. As such, we just return the value computed by the last form. The decision to simply discard forms means we need to perform careful inspection of each use of stateless-pprogn to ensure that we are only omitting output. Before this project completes, we will likely need to modify stateless-pprogn to combine intermediate output messages for printing.

3.4.2 An Incremental Approach

Our incremental approach is outlined as the following:

1. Create a version of ACL2 equivalent to the current version, which removes some sequential dependencies involving pspv and state, by combining returned pspv values and removing the modification of state from the waterfall. Also develop an automated mechanism for running the regression suite with fake certificates for books that are expected not to certify.

2. Introduce the primitives necessary to spawn parallel evaluation from within the waterfall. The first step is to do this in raw Lisp, but then this will be done within the logic. See sections 4.1 and 4.2 for a discussion of these primitives.

3. The first step above removes I/O from the waterfall. Now we will attempt to re-incorporate this I/O, for example by accumulating messages to print with a pipelining parallelism primitive. Such an approach would provide output that agrees with the output when parallelism is disabled. As a potentially more useful alternative, we are heavily considering providing output as soon as possible, even if in a non-deterministic order, by using stateless printing (for example through a “comment window” or “wormhole”), as a way to provide early feedback to the user.

4. Provide the user a mechanism to enable and disable parallel evaluation of the waterfall. We may eventually enable such parallel evaluation by default for interactive sessions.

Most of the above steps will be controlled with compile-time features and macros.

4 Introducing Parallelism

There are four main challenges involved in introducing parallelism into ACL2. First, we need to extend the logic with the primitives necessary to evaluate the waterfall in parallel. The primitives we introduce are similar in spirit to and build upon plet, par, pand, and por, which were introduced as part of my master’s work. Second, we need to use those primitives in the waterfall. Third, we need to manage the output, and finally we need a way to manage user interrupts.
4.1 Raw Lisp Primitives

A *future* is often thought of as a promise for the resulting value of a computation. A future’s corresponding computation is usually evaluated in a separate thread, and any thread that needs the future’s value blocks until that computation is finished evaluating.

We introduce three raw Lisp primitives for creating, reading, and aborting future evaluations: *future*, *future-read*, and *future-abort*. *Future* will return a structure that represents the evaluation of some computation in another thread. *Future-read* will know how to read the value from that structure. And *future-abort* will know how to terminate the evaluation of a future.

The raw Lisp definition of *future* creates a closure containing the work to evaluate and stores that closure on a queue. Whenever an evaluation needs the computed value of the future, it must call the function *future-read*. Because incorporating these raw Lisp version of futures into the ACL2 logic would also mean extending the logic to comprehend a new data type, we do not plan on incorporating these raw Lisp versions of futures into the ACL2 logic. However, by having different definitions for macros that use futures in raw Lisp and simply perform a logical equivalent in the ACL2 loop, we are able to use *futures* in the waterfall and develop proofs of subgoals in parallel.

Our raw Lisp versions of futures also support rapid termination of speculatively generated futures, which, in our application, are typically proof goals. This is useful when the *pspv*’s can not be combined. In this case, the *cdr* recursion of *waterfall1-lst* must be terminated and restarted with the *pspv* value returned from calling *waterfall1* on the *car* of the clause list. This rapid termination feature is also used when *waterfall1* returns a value that indicates we should stop the proof attempt.

4.2 Logical Abstractions

One logical abstraction we have developed and used is *spec-mv-let*. The current raw Lisp and logical ACL2 definitions of *spec-mv-let* are shown in the appendix. The displayed version of *spec-mv-let* takes three arguments: (1) a list of potentially-useful variables for which to compute values, (2) an expression that computes those values, and (3) another *mv-let* form whose body contains an *if-then-else* expression. By using *spec-mv-let*, two computations can be evaluated in parallel: the expression in (2) above and the expression included with the inner *mv-let* that calculates values for the inner sets of bindings. The intent is to write forms known to evaluate to needed values inside the inner *mv-let*. The system then performs a test, and when the values from (1) above are found to be irrelevant, it will know that it can abort the evaluation associated with (2) above and return the value computed by the *then* expression. However, if those values are found to be relevant, the primitive blocks until they are finished being computed. After waiting, the *else* branch of the *if-then-else* expression evaluates and returns.

We are imagining changing the macro *plet* to support both (1) multiple values and (2) speculative evaluation. Supporting multiple values is simply a matter of figuring out the appropriate syntax. However, enabling speculative evaluation requires more strategy. For now, we think that requiring the ACL2 programmer to specify which variables’ computations can be aborted will suffice. The ACL2 programmer performs this specification by calling
a macro similar to check-vars-not-free. If we extend plet to have these additional functionalities, one could implement the previously mentioned spec-mv-let macro in terms of plet. An example translation of the form in Appendix A.1 can be found in Appendix A.4.

We do not want to limit ourselves to the two above logical abstractions. However, our long-term hope is that plet will serve as the connection between the ACL2 logic and the underlying raw Lisp primitives. Consequently, we hope that macros like spec-mv-let can be written exclusively in ACL2 user-space, without needing a distinct raw Lisp definition.

4.3 Managing Output

As previously mentioned, the simplest solution is to disable all output. However, since we are trying to lessen the time required to develop, as opposed to certify, a proof, the output is likely crucial to the user.

An approach we could pursue involves printing output as it becomes available, in a manner consistent with an unaltered ACL2. We know the machine is capable of storing output temporarily without running out of memory, because ACL2 already does so with gag-mode (see “set-gag-mode” in [1]). We would need to invent a means to (1) let a thread know where to store its output (2) signal when new output is added and (3) signal when the thread is finished generating output.

A third possibility involves non-deterministically printing a subset of the output as it becomes available. Suppose that an ACL2 user operates by default with gag-mode enabled, so that they only see the proof checkpoints. In this case, it may be reasonable to simply print the checkpoints as they become available. Kaufmann and Moore suggest that the ACL2 user is unlikely to think about the ACL2 narrative that brings the prover to a particular checkpoint. Instead, they suggest that an ACL2 user would prefer to have something meaningful to think about as soon as possible. Printing checkpoints as they arrive would satisfy such a preference. Furthermore, Kaufmann and Moore suggest that since a user wants to see the full narrative in only a rare case, they would likely be satisfied replaying the proof in a sequential mode with full output enabled.

When an ACL2 user’s conjecture fails to prove, they will often submit a lemma intended to help the prover make more progress when attempting to prove that conjecture. In the face of non-deterministic printing, it would be more difficult than it is currently for a user to determine whether the addition of that lemma helped prove a previously-given checkpoint. One potential solution to this involves creating a new mechanism for presenting a conjecture to the prover. This new mechanism would allow a user to assert that a particular checkpoint was proved. After printing a conjecture’s checkpoints, the parallel version of ACL2 could provide the user with a form that introduced such an assertion. This would give the ACL2 user valuable feedback about whether a lemma helped. In fact, even though it is likely more useful in a non-deterministic environment, this type of feedback might even be useful in a single-threaded environment. An investigation into mechanisms like this are a planned part of this research.
4.4 Managing Interrupts

In terms of allowing the user to interact with an ACL2 proof, we must ask the question, “What does a user want to happen when they interrupt a proof?”

The most common answer to this question is simply that the current proof attempt should be terminated, and the underlying parallelism threads should be reset so that ACL2 is not actively consuming CPU core resources.

A second answer to this question involves just breaking the main thread and letting the parallelized subgoal computations continue to evaluate. Then if a user cancels the proof via the main thread, ACL2 would abort the child computations. An advantage to this approach is that while the main thread waits for user input, the computer still has something potentially useful to compute. A disadvantage of letting the child threads continue evaluation is that the state of the system will continue to change, so the debugging that a raw Lisp prompt lends itself to could be less useful.

As another alternative, we could pause the subgoals’ threads when the main thread is interrupted. We could accomplish this by creating a function that sends breaks to all of the parallelism threads. Then we would call this function from ACL2’s `our-abort`, which is called whenever the user interrupts a proof. We would then order the parallelism threads to resume execution when the user resumes the proof attempt. Even if we omit the calling of the function that pauses all of the threads from `our-abort`, the user would still be able to call it from the raw Lisp prompt of the broken main thread. Therefore, the ACL2 user could still pause the proof and inspect an unchanging program state.

5 Evaluation

We break down the evaluation of our project into two main areas: soundness and performance.

5.1 Evaluation of Soundness

Our methods are intended to treat soundness as a first-class objective. We have discussed the removal of features from the waterfall, intentionally causing books that use these features to fail. Of the approximately 2800 books in the regression suite, only about 200 of them fail with those features disabled. As such, we still have 93% of the original regression suite as a test bench.

Additionally, as discussed earlier, we have created a system of macros that should make our changes to the code clear and concise.

There is one more point worth mentioning. Since the version of ACL2 we are developing is used to decrease the interactive time it takes to generate a proof, and not the time it takes to certify a book, it is feasible to certify a suite of books with a different ACL2 image than used to generate the books. As such, we could improve the version of ACL2 that proves books in parallel to “99.9%” soundness and then require that books be certified with the parallel waterfall disabled. Since the user can still enable process-level parallelism with “make -j”, it is possible that nothing would be lost.
5.2 Evaluation of Performance

As described earlier, the goal is to reduce the latency between when a user submits a conjecture and when the proof attempt returns useful feedback to the user (whether it be via success, failure, information that lets a user start thinking about relevant lemmas, or information that convinces the user to terminate the proof attempt). As such, our goal is not to reduce the time it takes to certify the regression suite while using “make -j”. We could instead focus on the single-process time it takes to certify the regression suite. A preliminary performance result for one of the books that takes longer to certify follows.

Summary for making books/ordinals/ordinal-addition.cert:

Average sequential time was: 6.22s
Average parallel time was: 5.09s
Sequential minimum was: 6.12s
Parallel minimum was: 4.87s

Of 10 iterations, the parallel version was faster than the sequential version 10 times.

However, even the time it takes to certify a book with a single process would only be a guide towards determining the speedup relevant to helping ACL2 users develop proofs more quickly. A potentially more meaningful metric involves removing lemmas from libraries that certify and then examining the time it takes to obtain feedback that guides the user towards the missing lemma. The steps to such an approach follow.

1. Remove a random lemma from a random book in the regression suite.

2. Load the book, up to the point of the removed lemma.

3. Find the next theorem in the book that no longer proves.

4. For that theorem, attempt its proof in both the serial and parallelized versions of ACL2.

5. Compare the time it takes to reach a checkpoint\(^{10}\) in each of those ACL2 versions. Since users tend to hardly notice interactive delays less than a second in duration (see chapter 5 in [34]), we are much more interested in comparing proof failures that require more than a second to finish than those that take less than a second. As the number of CPU cores climbs into the thousands, a very long-term goal for ACL2 would be to reduce all meaningful proof attempts to take less than a second. Investigating the scalability of such proof attempts is another aspect of this work.

\(^{10}\)An ACL2 checkpoint is a point in the proof where ACL2 has applied most of its heuristics and is about to apply a heuristic with a relatively low chance of success. As such, ACL2 users consider these checkpoints to be where the prover “gets stuck” and to be where user intervention is often required.
If after optimizing the futures library and considering varying levels of granularity, we are unable to obtain useful speedup, we will be required to consider whether parallelizing the waterfall’s proof of subgoals is the right strategy to take advantage of the multi-core CPUs now common in most machines.

5.2.1 Subjective Evaluation

There are also subjective approaches to evaluating our solution. Included in these approaches are performing some proofs of system level functions to get a feel for the parallelism system’s usefulness and asking experienced ACL2 users like Jared Davis and Sandip Ray for feedback. Sandip already suggests that the ability to automatically receive feedback sooner could be helpful. Sandip’s current approach is to sometimes use the :reorder hint mechanism. However, the use of the :reorder hint mechanism requires the user to identify in advance expensive subgoals in the proof of the working conjecture and reorder them appropriately. According to Sandip, this is a difficult exercise in general, but particularly difficult early in the verification project when the necessary domain insights have not yet been gleaned. Attempting the subgoals in parallel obviates this requirement while automatically providing this type of reordered feedback.

6 A Final Motivating Example

One clear way of achieving our goal of reducing the latency of our system is to demonstrate a faster wall clock time. Another mechanism is to examine the time it takes to receive feedback that causes the user to understand their current problem. Suppose you have the following function, which determines whether a list ends with nil:

```
(defun truer-listp (x)
  (if (atom x)
      (eq x nil)
      (truer-listp (cdr x))))
```

Then suppose that you are trying to prove the following theorem, which is false:

```
(thm (truer-listp x))
```

The proof will immediately break into two subgoals, one for the inductive case (which succeeds) and one for the base case (which fails). In this simple example, neither of these cases takes a significant amount of time. However, if you had a more complicated example, where the inductive proof required a significant amount of time (or never finished at all), then immediately receiving the information about the base case proof attempt would be useful. This is because the problem lies in trying to prove the base case, not trying to prove the inductive case. Therefore, the sooner we show the base case failure to the user, the sooner they are able to correct their theorem and progress with their work.
References


A Spec-mv-let

A.1 Example Usage

(spec-mv-let (x y)
  ;; speculatively evaluate (mv 3 4)
  (mv 3 4)
  (mv-let (q r)
    (mv 7 8)
    (if (equal q 8)
      ;; the speculative evaluation is irrelevant, return
      ;; a value that doesn’t use those results
      (+ q r)
      ;; the speculative evaluation is useful, return a
      ;; value that uses those results
      (+ x y q r))))

A.2 Logical ACL2 Definition

(defun spec-mv-let (bindings computation body)
  (assert$ (and (true-listp body)
    (equal (length body) 4)
    (or (equal (car body) 'stateless-mv-let)
      (equal (car body) 'mv-let)))
  (let* ((inner-let (car body))
    (inner-bindings (cadr body))
    (inner-body (caddr body))
    (ite (cadddr body)))
    (assert$ (and (true-listp ite)
      (equal (length ite) 4)
      (equal (car ite) 'if))
    (let* ((test (cadr ite))
      (true-branch (caddr ite))
      (false-branch (cadddr ite)))
      '(check-vars-not-free
        (the-very-obscure-future)
        ,inner-let
        ,inner-bindings
        ,inner-body
        (if (check-vars-not-free ,bindings ,test)
          (check-vars-not-free ,bindings ,true-branch)
          (mv-let ,bindings
            ,computation
            ,false-branch))))))))
### A.3 Raw Lisp Definition

`(defmacro spec-mv-let (bindings computation body)
 (assert (and (true-listp body)
              (equal (length body) 4)
              (or (equal (car body) 'stateless-mv-let)
                  (equal (car body) 'mv-let))))
(let* ((inner-let (car body))
       (inner-bindings (cadr body))
       (inner-body (caddr body))
       (ite (cadddr body)))
(assert (and (true-listp ite)
              (equal (length ite) 4)
              (equal (car ite) 'if)))
(let* ((test (cadr ite))
       (true-branch (caddr ite))
       (false-branch (cadddr ite)))
  '(let ((the-very-obscure-feature (future ,computation)))
    (,inner-let
     ,inner-bindings
     ,inner-body
     (if ,test
        (progn (future-abort the-very-obscure-feature)
               ,true-branch)
        (mv-let ,bindings
               (future-read the-very-obscure-feature)
               ,false-branch))))))

### A.4 Translation of Example A.1 into Plet

`(plet (((x y) (mv 3 4))
        ((q r) (mv 7 8)))
       (if (equal q 8)
           (check-vars-not-used (x y)
                                (+ q r))
           (+ x y q r)))
B Helpful Definitions

**Early Termination** - Terminating an evaluation before its result is computed. Once an evaluation result is determined to be useless, the thread computing that result can stop that evaluation and start something else. Used in implementing `pand` once an argument evaluates to `nil`. Used in `por` once an argument evaluates to `non-nil`.

**Future** - A “promise” for a form’s evaluation result. Implemented by us in Lisp by returning a data structure that can be read with the function `future-read`.

**Linear Speedup** - Occurs when the time an evaluation requires is inversely proportional to the number of available CPU cores. E.g., if there are four available CPU cores and we have linear speedup, the parallel evaluation takes a quarter the amount of time that the serial evaluation requires.

**Multiple Values** (abbrev. `mv`) - A mechanism for returning more than one value as part of a function’s return signature.

**Pand** - A version of `and` that evaluates its arguments in parallel and returns their conjunction as a Boolean value.

**Pargs** - An identity function that evaluates the arguments to the surrounded function call in parallel.

**Plet** - A version of `let` that evaluates its bindings in parallel.

**Por** - A version of `or` that evaluates its arguments in parallel and returns their disjunction as a Boolean value.

**Progn** - A mechanism to perform a series of computations in a functional language (Lisp).

**Prover Special Variables** (abbrev. `pspv`) - A collector (really an ACL2 record) for assignments to variables that would be global if ACL2 were written in an imperative manner. Used in the Waterfall.

**Speedup** - For a given computation, the ratio of the time it takes to compute serially to the time it takes to evaluate in parallel. E.g., if a computation takes 15 seconds in the serial case and 3 seconds in the parallel case, the speedup is a factor of 5.

**State** - Provides a way to record and model the modification of ACL2 global variables in a functional language, print output to the screen, and perform other actions with side-effects like checking the status of system calls.

**Super-Linear Speedup** - Occurs when an algorithm permits speedup that surpasses the number of available CPU cores. E.g., assuming 4 available CPU cores, if a computation takes 20 seconds serially and 1 second in parallel, super-linear speedup is said to have occurred.

**Waterfall** - The ACL2 proof process that applies all proof heuristics except for induction.