A Futures Library and Parallelism Abstractions for a Functional Subset of Lisp

David L. Rager
The University of Texas at Austin
1616 Guadalupe Street
Suite 2.408
Austin, TX 78701
ragerdl@cs.utexas.edu

Warren A. Hunt, Jr.
The University of Texas at Austin
1616 Guadalupe Street
Suite 2.408
Austin, TX 78701
hunt@cs.utexas.edu

Matt Kaufmann
The University of Texas at Austin
1616 Guadalupe Street
Suite 2.408
Austin, TX 78701
kaufmann@cs.utexas.edu

ABSTRACT

This paper discusses Lisp primitives and abstractions developed to support the parallel execution of a functional subset of Lisp, specifically ACL2.

We (1) introduce our Lisp primitives (futures) (2) present our abstractions built on top of these primitives (`spec-mv-let` and `plet+`), and (3) provide performance results.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*

General Terms

Performance, Verification

Keywords

functional language, parallel, plet+, spec-mv-let, granularity, Lisp, ACL2

1. INTRODUCTION

Our project is about supporting parallel evaluation for applicative Common Lisp, specifically the purely functional programming language provided by the ACL2 system [11, 9]. We provide language primitives, three at a low level and two at a more abstract level, for convenient annotation of source code to enable parallel execution.

Our intended application for this parallel evaluation capability is the ACL2 theorem prover, which has been used in some of the largest industrial formal verification efforts [2, 14]. As multi-core CPUs become commonplace, ACL2 users would like to take advantage of the underlying available hardware resources [10]. Since the ACL2 theorem prover is primarily written in its own functional language, it is reasonable to introduce parallelism into ACL2's proof process in a way that takes advantage of the functional programming paradigm.

After discussing some related work, we introduce three Lisp primitives that enable and control parallel evaluation, based on a notion of *futures*. We build on these three primitives to introduce two primitives at a higher level of abstraction.[1] We then demonstrate these primitives' utility by presenting some performance results. We conclude with remarks that include challenges for the Lisp community.

2. RELATED WORK

There is a large body of research in parallelizing functional languages and their applications, including work in automated reasoning. Here, we simply mention some of the pioneers and describe some recent developments in the area.

An early parallel implementation of Lisp was Multilisp, created in the early 1980s as an extended version of Scheme [6]. It implemented the *future* operator, which is often defined as a promise for a form's evaluation result [7, 3]. Other parallel implementations of Lisp include variants such as Parallel Lisp [7], a Queue-based Multi-processing Lisp [4], and projects described in Yuen's book "Parallel Lisp Systems" [16]. Our approach builds upon these approaches by implementing parallelism primitives and abstractions for systems that are compliant with ANSI Common Lisp, and thus, available for use in applications like ACL2. Furthermore, our abstractions have clear logical definitions inside a theorem proving system, making it straightforward to reason about their use.

More recent developments include the Bordeaux Threads project [1], which seeks to unify the multi-threading interfaces of different Lisps. We approach the same problem by providing a multi-threading interface [12, 13] to Clozure Common Lisp (CCL) and Steel Bank Common Lisp (SBCL). We have our own multi-threading interface because we need some different features. For example, our interface exposes

[1]The higher-level primitives are defined within the ACL2 logic, and hence have clear functional semantics that are amenable to formal verification. We avoid further discussion of the ACL2 logic in this paper.

a CCL feature for semaphores, *notification objects*, that we use to determine whether a semaphore was actually signaled (as opposed to returning from a wait due to a timeout or interrupt).

Another recent development is Haverbeke's PCall library [8]. This library is similar to our futures library, in that it provides a way to spawn a thread to evaluate an expression and then use the returned value in the original spawning thread. As a branch of this PCall library, Sedach has initiated the Eager Future2 project, whose web site [15] reports some additional features like error handling and the ability to force the abortion of a future's evaluation. Our latest extension of ACL2, which is described in this paper, has some of these features.

3. FUTURES LIBRARY

There are three Lisp primitives that enable and control parallel evaluation: `future`, `future-read`, and `future-abort`. The `future` macro surrounds a form and returns a data structure with fields including the following: a closure representing the necessary computation, and a slot (initially empty) for the value returned by that computation. This structure can then be passed to `future-read` to access the value field after the closure has been evaluated. The final primitive, `future-abort`, terminates futures whose values are no longer needed.

The following naïve version of the Fibonacci function illustrates the use of `future` and `future-read`.

```
(defun pfib (x)
  (if (< x 33)
      (fib x)
    (let ((a (future (pfib (- x 1))))
          (b (future (pfib (- x 2)))))
      (+ (future-read a)
         (future-read b)))))
```

Even if only a single core is available, we still want to support the delaying of a computation until its result is needed, as illustrated by the `spec-mv-let` primitive discussed in the next section. Of course, in the single-threaded case we need not provide infrastructure for distributing computation to more than one thread. The following discussion of our implementation of futures primitives thus includes optimizations for the single-threaded case.

The multi-threaded implementation of `future` provides the behavior summarized above as follows: when a thread evaluates a call of `future`, it returns a future, F. F contains a closure that is placed on the *work-queue* for evaluation by a *worker* thread. The value returned by that computation may only be obtained by calling the `future-read` macro on F. If a thread tries to read F before the worker thread finishes evaluating the closure, the reading thread will block until the worker thread finishes. However, when the single-threaded implementation is given a future, F, to read, if the future has not previously been read, the closure is evaluated by the reading thread, and the resulting value is saved inside F. The final primitive, `future-abort`, removes a given future, F, from the work-queue; sets an *abort flag* in F; and aborts evaluation (if in progress) of F's closure.

4. ABSTRACTIONS

We build two abstractions on top of the futures primitives. These abstractions avoid the difficult task of introducing futures into the ACL2 programming language and logic. One primitive, `spec-mv-let`, is similar to `mv-let` (ACL2's notion of `multiple-value-bind`). Our design of `spec-mv-let` is guided by the shape of the code where we want to parallelize ACL2's proof process. `Spec-mv-let` calls have the following form.

```
(spec-mv-let
 (v1 ... vn)  ; bind distinct variables
 <spec>       ; evaluate speculatively; return n values
 (mv-let
  (w1 ... wk) ; bind distinct variables
  <eager>     ; evaluate eagerly
  (if <test>  ; ignore <spec> if true
              ;   (does not mention v1 ... vn)
      <abort-form>     ; does not mention v1 ... vn
      <normal-form>))) ; may mention v1 ... vn
```

Evaluation of the above form proceeds as suggested by the comments. First, `<spec>` is executed speculatively (as our implementation of `spec-mv-let` wraps `<spec>` inside a call of `future`). Control then passes immediately to the `mv-let` call, without waiting for the result of evaluating `<spec>`. The variables `(w1 ... wk)` are bound to the result of evaluating `<eager>`, and then `<test>` is evaluated. If the value of `<test>` is true, then the values of `(v1 ... vn)` are not needed, and the evaluation of `<spec>` may be aborted. If the value of `<test>` is false, then the values of `(v1 ... vn)` are needed, and `<normal-form>` blocks until they are available.

The other abstract primitive, intended to be of more general use to the ACL2 programmer, is `plet+`. `Plet+` is similar to `let`, but it has three additional features: (1) it can evaluate its bindings in parallel, (2) it allows the programmer to bind not just single values but also multiple values, and (3) it supports speculative evaluation, only blocking when the bindings' values are actually needed in the body of the form. `Plet+` is an enhanced version of our previous primitive, `plet` [13], and it supports the Lisp declarations for `let` that are allowed by ACL2: `type`, `ignore`, and `ignorable`. An optional *granularity* form (as for `plet`) provides a test for whether the computation is estimated to be of large enough granularity. To date we have restricted our use of `plet+` to small examples, preferring to use `spec-mv-let` in our ACL2 builds and testing. That may change as we further develop `plet+`, for example by reducing the garbage generated when binding multiple values.

5. PERFORMANCE RESULTS

We first present example uses of each primitive with naïve versions of the Fibonacci function, comparing their times for parallel and serial executions. Figure 1 shows the results for these tests. Then, in Subsection 5.4, we compare the performance of the parallelized ACL2 prover to the unmodified (serial) version.

All testing was performed on an eight-core 64-bit Linux machine running 64-bit CCL with the Ephemeral Garbage Collector (EGC) disabled and a 16 gigabyte Garbage Collection

(GC) threshold. See Subsection 5.5 for the reasons behind this decision. All times are reported in seconds, and each speedup factor reported is a ratio of serial execution time to parallel execution time. In each case we report minimum, maximum, and average times for ten consecutive runs of each test, both parallel and serial, in the same environment. The scripts and output from running these tests are available for download at http://www.cs.utexas.edu/users/ragerdl/-els2011/supporting-evidence.tar.gz.

5.1 Testing Futures

Recall the definition of `pfib` in Section 3. In our experiments, calling `(pfib 45)` yielded a speedup factor of 7.62 on an eight-core machine, which is nearly ideal in spite of asymmetrical computations occurring at the end of parallel evaluation.

Thus futures provide an efficient mechanism for parallel evaluation. But they also provide an efficient mechanism for aborting computation. By running the following test, we can see how long it takes to abort computation that has already been added to the work-queue. The following script takes approximately 6 seconds to finish, so it only takes about 60 microseconds to spawn and abort a future. We call function `count-down`, which is designed to consume CPU time. `(Count-down 1000000000)` typically requires about 5 seconds. Since calling `mistake` 100,000 times only requires 6 seconds, we know that we are actually aborting computation.

```
(defun mistake ()
  (future-abort (future (count-down 1000000000))))

(time
 (dotimes (i 100000)
   (mistake)))
```

5.2 Testing Spec-mv-let

We next define a parallel version of the Fibonacci function using the `spec-mv-let` primitive. The support for speculative execution provided by `spec-mv-let` is unnecessary here, since we always need the result of both recursive calls; but our purpose here is to benchmark `spec-mv-let`. The following definition has provided a speedup factor of 7.75 when evaluating `(pfib 45)`.[2]

```
(defun pfib (x)
  (if (< x 33)
      (fib x)
    (spec-mv-let (a)
      (pfib (- x 1))
      (mv-let (b)
        (pfib (- x 2))
        (if nil
            "speculative result is always needed"
          (+ a b))))))
```

5.3 Testing Plet+

The following version of the Fibonacci function, which uses `plet+`, has provided a speedup factor of 7.82 for the evaluation of `(pfib 45)`.

[2]ACL2 users may be surprised to see `mv-let` bind a single variable. However, this definition is perfectly fine in Lisp, outside the ACL2 read-eval-print loop.

Figure 1: Performance of Parallelism Primitives in the Fibonacci Function

Case	Min	Max	Avg	Speedup
Serial	40.06	40.21	40.08	
Futurized	5.15	5.78	5.26	7.62
Spec-mv-let	5.13	5.22	5.17	7.75
Plet+	5.08	5.18	5.12	7.82

Figure 2: Performance of ACL2 Proofs with the EGC Disabled and a High GC Threshold

Proof	Case	Min	Max	Avg	Speedup
Embarrass	serial	36.49	36.53	36.50	
	par	4.58	4.61	4.60	7.93
JVM-2A	serial	229.79	242.40	231.14	
	par	34.42	39.42	35.51	6.51
Measure-2	serial	175.99	179.93	176.53	
	par	47.07	53.71	50.01	3.53
Measure-3	serial	86.63	86.85	86.73	
	par	24.24	25.36	24.90	3.48

```
(defun pfib (x)
  (if (< x 33)
      (fib x)
    (plet+ ((a (pfib (- x 1)))
            (b (pfib (- x 2))))
           (with-vars (a b)
                      (+ a b)))))
```

5.4 ACL2 Proofs

We currently use `spec-mv-let` to parallelize the main part of the ACL2 proof process. We are not interested in speedup for proof attempts that take a small amount of time. However, we have obtained non-trivial speedup for some substantial proofs.

Figure 2 shows the speedup for four proofs. The first proof is a toy proof that we designed to be embarrassingly parallel and test the ideal speedup of our system. The proof named "JVM-2A" is about a JVM model constructed in ACL2. The third and fourth proofs are related to proving the termination of Takeuchi's Tarai function [5]. These proofs are not intended to be representative of all ACL2 proofs. Parallelism does not improve the performance of many ACL2 proofs, and it might even slow down some proofs. Investigating these issues is part of our future work.

5.5 The Effects of GC

We now consider the performance of parallelized ACL2 with different garbage collector configurations. In Figure 3, we report the performance of proof "JVM-2A" with the EGC either enabled or disabled and the GC configured to use either the default threshold or a threshold of 16 gigabytes.

While it is clear that both serial and parallel executions benefit from having the EGC disabled and a high GC threshold, one may wish to make a comparison not presented in the figures. Specifically, one could compare the optimal serial

Figure 3: Performance of Theorem JVM-2A with Varying GC Configurations

EGC & Threshold	Case	Min	Max	Avg	Speedup
on, default	serial	245.52	246.99	246.79	
	par	372.54	482.62	413.42	0.60
on, high	serial	245.38	247.09	246.90	
	par	377.91	524.78	422.20	0.58
off, default	serial	291.57	292.14	291.97	
	par	110.57	117.17	114.77	2.54
off, high	serial	229.79	242.40	231.14	
	par	34.42	39.42	35.51	6.51

configuration that uses the default GC threshold (where the EGC is enabled) to the optimal parallel configuration that uses the default GC threshold (where the EGC is disabled). In this comparison, the serial execution requires an average of 247 seconds, and the parallel execution takes an average of 115 seconds, yielding a speedup factor of 2.15.

Figure 3 shows that for applications running in parallel, it may be beneficial to disable the EGC and use a high GC threshold. Of course, those steps would likely be unnecessary in the presence of parallelized garbage collection.

6. CONCLUSION

We provide parallelism primitives at two levels of abstraction and demonstrate their successful use in speeding up computation. The higher-level library provides abstractions, `spec-mv-let` and `plet+`, which allow significant speedup with little extra annotation in the code. The lower-level library is based on the concept of futures and provides more explicit control of parallel computations. Note that the higher-level primitives fit nicely into the ACL2 applicative programming environment. Indeed, we parallelized the key ACL2 proof process, which is written in the ACL2 programming language, using `spec-mv-let`. Our results to date are promising, obtaining significant reductions in some proof times using this parallelized version.

It is our hope that by bringing the continued development of this library to the attention of the Lisp community: (1) ideas from our library can be reused in other systems, (2) Lisp implementors will be motivated to continue improving multithreading capabilities, for example by parallelizing garbage collection, (3) the Lisp community will continue to think about parallelism standards, and (4) we will gain feedback on ways to improve our implementation and/or design.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0945316. We thank Jared Davis, J Strother Moore, and Nathan Wetzler for helpful discussions.

8. REFERENCES

[1] Bordeaux Threads. *Bordeaux Threads API Documentation*, March 2010. http://trac.common-lisp.net/bordeaux-threads/wiki/ApiDocumentation.

[2] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam K. Srivas and Albert John Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 275–293. Springer-Verlag, 1996.

[3] Marc Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. PhD thesis, Waltham, MA, USA, 1993. UMI Order No. GAX93-22348.

[4] Richard P. Gabriel and John McCarthy. Queue-based multi-processing lisp. In *Conference on LISP and Functional Programming*, pages 25–44, 1984.

[5] David Greve. Assuming termination. In *ACL2 '09: Proceedings of the eighth international workshop on the ACL2 theorem prover and its applications*, pages 121–129, New York, New York, USA, 2009. ACM.

[6] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a microprocessor. In *Conference on LISP and Functional Programming*, pages 9–17, 1984.

[7] Robert H. Halstead, Jr. New ideas in parallel lisp: Language design, implementation, and programming tools. In *Parallel Lisp: Languages and Systems*, pages 2–57, 1989.

[8] Marijn Haverbeke. Idle cores to the left of me, race conditions to the right. June 2008. http://marijnhaverbeke.nl/pcall/background.html.

[9] Matt Kaufmann, Pete Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[10] Matt Kaufmann and J Strother Moore. Some key research problems in automated theorem proving for hardware and software verification. *Spanish Royal Academy of Science (RACSAM)*, 98(1):181–195, 2004.

[11] Matt Kaufmann and J Strother Moore. *ACL2 Documentation*. ACL2, March 2010. http://www.cs.utexas.edu/∼moore/acl2/current/acl2-doc-index.html.

[12] David L. Rager. Adding parallelism capabilities in ACL2. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 90–94, New York, New York, USA, 2006. ACM.

[13] David L. Rager and Warren A. Hunt, Jr. Implementing a parallelism library for a functional subset of lisp. In *Proceedings of the 2009 International Lisp Conference*, pages 18–30, Sterling, Virginia, USA, 2009. Association of Lisp Users.

[14] David Russinoff, Matt Kaufmann, Eric Smith, and Robert Sumners. Formal verification of floating-point RTL at AMD using the ACL2 theorem prover. In Simonov Nikolai, editor, *Proceedings of the 17th IMACS World Congrress on Scientific Computation, Applied Mathematics and Simulation*, Paris, France, 2005.

[15] Vladimir Sedach. *Eager Future2*, January 2011. http://common-lisp.net/project/eager-future/.

[16] C. K. Yuen. *Parallel Lisp Systems: A Study of Languages and Architectures*. Chapman & Hall, Ltd., London, UK, 1992.