Adding Parallelism Capabilities to ACL2

David L. Rager
Department of Computer Sciences, The University of Texas at Austin
ragerdl@cs.utexas.edu

ABSTRACT

We have implemented parallelism primitives that permit an ACL2 programmer to parallelize execution of ACL2 functions. We (1) introduce logical definitions for these primitives, (2) explain the features of our extension, (3) give an evaluation strategy for our implementation, and (4) use the parallelism primitives in examples to show speedup.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*

General Terms

verification, performance

Keywords

parallel ACL2, functional language, plet, pcall, pand, por, granularity

1. INTRODUCTION

One of ACL2's strengths lies in its ability to efficiently execute industrial-sized models. As multi-core CPUs [1] become commonplace, we want to take advantage of the available hardware resources.

We introduce four parallelism primitives: `pcall`, `plet`, `pand`, and `por`. `Pcall` is logically the identity macro. With `pcall`, the arguments of a function can be evaluated in parallel. `Plet` allows parallel evaluation of variable bindings. `Pand` and `por` are similar to the ACL2 macros `and` and `or` but different in the aspects outlined in sections 2.3 and 2.4.

We then discuss three features of our parallelism extension. First, with recursive use of the parallelism primitives, our extension can adapt to the data so that a function's computation does not serialize when the data is asymmetric. Second, we provide a means to specify a criterion for determining granularity. This helps the system determine when arguments to a function are complex enough to warrant parallel evaluation. Third, when issuing a `pand` or `por`, our system recognizes opportunities for early termination and returns from evaluation.

Next we explain our evaluation strategy. We first describe how we determine when parallelism resources are available. If resources are unavailable, the parallelism primitive expands to its serial equivalent. We also explain our implementation of work consumers.

At the end of the paper, performance results are illustrated by using the parallelism primitives with a naïve Fibonacci function, boolean if normalization, and mergesort.

Much related work has been done in the area of functional language parallelism. Some of this work includes *futures* and primitives like `pcall` [10, 3]. Also Hunt and Moore provide a partial implementation for an ACL2 parallelism extension using futures [private communication] [4].

2. PARALLELISM PRIMITIVES

We consider two goals in the use of our parallelism primitives. First, users need a way to efficiently parallelize computation in functions they execute. Second, the use of parallelism primitives should be as logically transparent as possible. With these goals in mind, we present each primitive's semantics and provide examples of usage.

Any function that uses parallelism primitives must have its guards defined and verified before it can execute with parallelism [8]. To save space, guards have been omitted from most examples in this paper, but the full definitions can be found in the supporting scripts.

2.1 Pcall

The first primitive, `pcall`, is logically the identity macro. `Pcall` takes a function call whose arguments it may evaluate in parallel and then applies the function to the results of that parallel evaluation. A simple example is as follows:

```
(defun pfib (x)
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (pcall (binary-+ (pfib (- x 1))
                            (pfib (- x 2)))))))
```

In this example, `(pfib (- x 1))` and `(pfib (- x 2))` can be evaluated in parallel, and then `binary-+` will be applied to the list formed from their evaluation. If the programmer uses pcall in a function whose argument evaluations always require a large amount of time, the user will experience speedup. As explained in section 3.2, a granularity form can be used to limit when parallelism is introduced.

Since macros can change expressions in unexpected ways, we disable the pcall'ing of macros. While it may be possible to reliably expand macros using the LISP function macroexpand, we have avoided it so far. If a user wishes to parallelize computation of arguments to a macro, we suggest they use plet instead.

2.2 Plet

The second primitive, plet, is logically equivalent to the macro let. Under the hood, plet evaluates the binding computations in parallel and applies a closure created from the body of the plet to the results of these binding evaluations. A simple example is as follows:

```
(defun pfib (x)
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (plet ((fibx-1 (pfib (- x 1)))
                  (fibx-2 (pfib (- x 2))))
                 (+ fibx-1 fibx-2)))))
```

As with pcall, the evaluations for bindings fibx-1 and fibx-2 occur in parallel, and then the closure containing + is applied. A feature of plet is that its body's top level call can be a macro. This is because the closure will have all its arguments evaluated before it is applied.

2.3 Pand

The third primitive, pand, is fundamentally different from and. Pand evaluates its arbitrary number of arguments in parallel, evaluates their conjunction, and returns a boolean result. From this definition, we note two differences. First, we return a boolean result. This makes it consistent with por, which we describe later. The second difference is that the truth or falsity of the evaluation of the first argument does not prevent the evaluation of the second argument.

Consider call: (pand (consp x) (equal (car x) 'foo)). With pand both (consp x) and (equal (car x) 'foo) can execute in parallel. With and, the falsity of (consp x) prevents the evaluation of (car x). Our logical definition of pand does not provide (consp x) as a guard to (car x).

As an example, suppose we have a function valid-tree which traverses a tree, testing each atom to make sure it is a valid-tip. We can write a parallel version as follows:

```
(defun valid-tree (x)
  (if (atom x)  (valid-tip x)
    (pand (valid-tree (car x))
          (valid-tree (cdr x)))))
```

Once one of the arguments evaluates to nil, we can return nil without waiting for the other arguments to finish evaluation. This feature is called early termination and is explained in section 3.3.

2.4 Por

In the same way as pand, the fourth primitive, por, is fundamentally different from or. Por evaluates its arguments in parallel, evaluates their disjunction, and returns a boolean result. Since the evaluation order of parallel computation can be nondeterministic, it is safest to consistently return a boolean value rather than risk providing different results for por calls with the same argument list. Similar to pand, por guards computation in a different way than or.

Suppose we have the macro call: (por (atom x) (equal (car x) 'foo)). We can see that with por both (atom x) and (equal (car x) 'foo) can execute in parallel. With or, the truth of (atom x) prevents the evaluation of (car x). Our logical definition of por does not provide (not (atom x)) as a guard to (car x).

3. FEATURES

A discussion of some user-level features follows.

3.1 Data Dependent Parallelism

When computing results on symmetric data, it is often easy to accurately partition resources. For example, if we have two CPU cores available for processing, we split the computation at the top recursive level and create two pieces of parallel work. However, when the data is asymmetric, the evaluation of one piece may terminate significantly before the other piece, effectively serializing computation. To be more concrete, take the following function that counts the leaves of a tree:

```
(defun naive-pcount (x)
  (if (atom x)  1
    (pcall (binary-+ (acl2-count (car x))
                     (acl2-count (cdr x))))))
```

If we give this function a tree shaped like a list, we realize quickly that splitting computation only at the first recursive level and not parallelizing again after one of the recursions terminates will result in an almost serial computation.

It turns out that the solution fits quite naturally into ACL2. The user must simply define functions to recur into functions that use the parallelism primitives. Whenever a primitive is encountered and resources are available, the system will parallelize computation. See section 4.1 for an explanation of resource availability.

3.2 Granularity Form

When computing functions like the naïve Fibonacci, some inputs are large enough to warrant computing the arguments in parallel, while other inputs are too small to be worth the cost of parallelism overhead. For example, consider the definition of Fibonacci found in section 2.1. Experiments on our machine indicate that whenever x is less than thirty, that we should call a serial version of the Fibonacci function. This could require two definitions of the function, e.g.,

```
(defun fib (x)
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (binary-+ (fib (- x 1))
                     (fib (- x 2))))))

(defun pfib (x)
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        ((< x 30) (binary-+ (fib (- x 1))
                            (fib (- x 2))))
        (t (pcall (binary-+ (pfib (- x 1))
                            (pfib (- x 2)))))))
```

We realize quickly that writing both of these function definitions is both cumbersome and redundant. As such, the user can provide a *granularity form* with each parallelism primitive. When using the granularity form, the system will only parallelize computation if (1) resources are available and (2) the dynamic evaluation of the granularity form returns non-nil. Below is a definition of the Fibonacci function using a granularity form. To conform with LISP standards, the syntax of the granularity-form is a type of pervasive declaration [7].

```
(defun pfib (x)
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (pcall (declare (granularity-form (>= x 30)))
                  (binary-+ (pfib (- x 1))
                            (pfib (- x 2)))))))
```

We can also declare a granularity form with an extra argument that describes the call depth of the function the user is parallelizing. Take mergesort as an example. Mergesort splits the data into symmetric chunks for computation, so we increment the depth argument during the recursive call on both the car and cdr. A parallelized version of mergesort based on Davis's Ordered Sets library [5] can be found in the supportive scripts.

A less intrusive method involves analyzing the data itself for structural properties. When we define the function that performs boolean if normalization, we will define nthcar, a function similar to nthcdr to examine the input if expression and determine granularity.

3.3 Early Termination

When computing an ACL2 and, due to lazy evaluation, some of the arguments to the and may never be evaluated. When we evaluate arguments to a pand in parallel, even more opportunity for skipping work is available. Consider the following function that computes whether a tree is valid:

```
(defun pvalid-tree (x)
  (if (atom x)  (valid-tip x)
    (pand (pvalid-tree (car x))
          (pvalid-tree (cdr x)))))
```

We would like to stop execution as soon as any tip is found to be invalid. So, when computing the conjunction of terms by using pand, once one of those terms evaluates to nil, all sibling computations are aborted and the pand returns nil. The user can experience superlinear speedup when a nil is beyond the first argument to the pand.

The concept of early termination also applies to por, except that the early termination condition is when an argument evaluates to non-nil.

4. EVALUATION STRATEGY

Below is an explanation of some implementation details that make up our evaluation strategy. It is not meant to be complete, in the sense that it enables someone to duplicate our work. It is intended to introduce the user to the underlying system.

4.1 Estimating Resource Availability

There are two resources to manage: CPU cores and work consumers (threads in our implementation). CPU cores can be in one of two states: *busy* and *idle*. Work consumers can be in one of three states: *stalled*, *active*, and *pending*. A work consumer is *stalled* whenever it is waiting either for a CPU core to become available or for work to enter the parallelism system. A work consumer can only be *active* when it has been allocated a CPU core and is actively processing a piece of work. If a work consumer encounters a parallelism primitive and parallelizes its evaluation, it will enter the *pending* state until its children finish, when it becomes *active* again. The goal is to keep CPU cores busy and avoid overwhelming the operating system (OS) with work consumers.

A conceptual *work queue* contains pieces of work and is organized into four sections: *unassigned* (U), *started* (S), *waiting* (W), and *resumed* (R). The first section stores *unassigned* work not yet acquired by a work consumer. The *started* section contains the pieces of work associated with an active thread that have not encountered an opportunity for parallelism. If a piece of work encounters a parallelism primitive and splits its work, the work will enter the *waiting* stage, as it waits for its children to finish. After the work's children finish, it *resumes* computation and return the result. Figure 1 illustrates the relationships between pieces of work, cores, and consumers.

Figure 1: Life Cycle of a Piece of Work

Work State	U	S	W*	R*
Allocated Core	no	yes	no	yes
Consumer State	n/a	active	pending	active

*the writing and resumed states are not always entered.

4.1.1 Limiting Active Consumers

We limit the number of active work consumers to match the number of CPU cores. Once a work consumer finishes a piece of work, if there is work in the unassigned section, it will immediately acquire another piece of work. Limiting the number of active work consumers in this way minimizes context switching overhead [6].

4.1.2 Keeping CPU Cores Busy

Whenever a work consumer acquires a CPU core resource, the consumer will immediately acquire work from the unassigned section and begin processing it. If there is no work in the unassigned section, the work consumer will stall until work is added. If parallelism opportunities had recently occurred but had reverted to their serial equivalents because all CPU cores were busy, this stalling would be wasted time. To avoid this, we treat the unassigned portion as a buffer and aim to keep at least p pieces of work in it at all times. We set p to the number of CPU cores, so that if all consumers finish simultaneously, they can acquire a new piece of work. Figure 2 shows the limits we impose for a system with p CPU cores.

Figure 2: Lengths of Work Queue Sections

unassigned	started + resumed	waiting on children
$- \geq p -$	$— = p —$	
	≤ 50	

4.1.3 Limiting Total Workload

Since the OS supports a limited number of work consumers, we must impose restrictions to ensure application stability. To demonstrate how execution could surpass the limit, suppose we have a function that counts the leaves of a tree, as below:

```
(defun pcount (x)
  (if (atom x)  1
    (pcall (binary-+ (pcount (car x))
                     (pcount (cdr x))))))
```

If we call this function on a heavily right-skewed tree, e.g., a list of length 100,000, then due to the short time required to count the `car`'s, the computation may parallelize every few `cdr` recursions. This creates a deeply nested call stack with potentially thousands of `pcall` parents waiting on their `pcount` children.

Since (1) the stack of parents waiting on children can only unroll itself by the children finishing evaluation and (2) any piece of work allowed into the system must eventually be allocated to a work consumer, unless we build in a limit to prevent additional work from being added when the maximum number of work consumers has been reached, the system will become unstable or deadlock. In OpenMCL, we have found a reasonable limit to be around 50. As shown in figure 2, we limit the total count of work to 50.

4.2 Work Consumer Implementation

We use threads to implement work consumers for the following reasons. First, threads share memory, which is good for our target: the SMP desktop market. Second, threads are lighter-weight than processes, lending themselves to finer-granularity problems.

Of the LISPs that support native threads and build ACL2, OpenMCL and SBCL provide threading primitives sufficient to implement our parallelism extension as described in this paper. Finally, to save time associated with spawning threads, we recycle them.

5. PERFORMANCE RESULTS

We evaluate our parallelism extension on a dual-CPU Power Mac G5, where each CPU contains two 2.5 GHz cores with 1 MB of L2 cache each. This Mac has six gigabytes of DDR2 memory and runs OS X version 10.4.6. We compile our extended ACL2 on a development copy of OpenMCL 1.1. With this setup, "perfect parallelism" would compute parallelized ACL2 functions in one quarter of their serial time. All times reported in this section are an average of three consecutive executions.

We present three tests. First, we define a doubly recursive version of the Fibonacci function. We choose the Fibonacci computation, because it is well known and it demonstrates our ability to adapt to asymmetric parallel computation. By this we mean that the computation for the first recursive call takes more time than the second recursive call. Our double recursive definition is inefficient, but it serves as a baseline for determining whether our system experiences speedup that increases linearly with respect to the number of CPU cores. The Fibonacci function is computation heavy and does not create much garbage, allowing an accurate measurement of parallelism overhead and the effects of granularity.

Running three trials of `(fib 47)` on the machine described above requires an average of 323 seconds, while `(pfib 47)` requires an average of 86 seconds. The speedup factor of 3.78 implies that we gain 95% of our potential speedup.

We also evaluate our parallelism extension using boolean if normalization. Unlike the Fibonacci function, the boolean if normalization algorithm is tied to useful functions [2]. Finding a good granularity form for boolean if normalization is a difficult challenge. In our example, we examine the structure of the if expression. Using this structure-based granularity form, we obtain a non-GC speedup of 1.77 on a ten bit ripple carry adder.

Is it more meaningful to examine the total time or the time spent outside the garbage collector? On one hand, only total execution time is relevant. On the other, since OpenMCL does not have a parallelized garbage collector [9], the best we can hope for is a speedup factor of four for the non-GC'd portion. We therefore focus our efforts on the non-GC time.

These garbage collection issues can lead to surprising results. Even the highly parallelizable algorithm `mergesort` may trigger the garbage collector too often to experience significant overall speedup. Below is a table of results, including measurements for `mergesort` as defined in the supportive scripts.

Table 1: Performance Test Results (seconds)

Case	Total Time	GC Time	Non-GC Time	Total Speedup	Non-GC Speedup
Fib					
Serial	325.88	0.00	325.88		
Parallel	86.11	0.07	86.05	3.78	3.79
If Norm					
Serial	71.28	9.68	61.60		
Parallel	65.84	30.97	34.87	1.08	1.77
Sort					
Serial	24.50	12.41	12.10		
Parallel	23.91	19.79	4.12	1.025	2.934

6. FUTURE WORK

We have ported the parallelism extension to SBCL, but it still functions best in OpenMCL. In the future, we want to make the SBCL implementation as efficient as the OpenMCL implementation. We also plan on integrating the extension into the main ACL2 distribution.

Future applications include integrating parallelism into the theorem proving process, possibly via relieving hypotheses during backchaining in parallel. Another option is to prove subgoals in parallel. ACL2's ability to save proof output for delayed printing is a step towards meeting this goal.

7. CONCLUSION

The four parallelism primitives are: `pcall`, `plet`, `pand`, and `por`. These primitives allow linear speedup of execution for functions that generate little garbage and have large granularity. Functions whose granularity varies can use a granularity form to ensure parallelism only occurs with larger computations. Since OpenMCL has a serial garbage collector, functions whose execution time is dominated by garbage collection will not experience as much speedup.

8. ACKNOWLEDGEMENTS

We thank Warren Hunt Jr. for supervising this work and for contributing the idea of recycling threads. Additionally, we thank Matt Kaufmann for creating an ACL2 compilation flag that makes `mv` and `mv-let` thread safe. We also thank him for looking over the parallelism design and implementation, making detailed suggestions, and reviewing this paper many times. We thank Gary Byers for providing OpenMCL level threading primitives and helping us understand OpenMCL's threading environment. We thank J Strother Moore, Robert Boyer, Sandip Ray, and Jared Davis for their general support and help with other issues. Finally, we recognize the National Science Foundation for funding this work under grant CNS 0429591.

9. REFERENCES

[1] AMD. Introducing multi-core technology. On the Web, April 2006. http://multicore.amd.com/en/Technology/.

[2] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, Inc., 1979.

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

[4] Jr. Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. In *Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, 1977.

[5] Jared Davis. Finite Set Theory based on Fully Ordered Lists. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*, November 2004.

[6] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[7] Guy L. Steele Jr. *Common Lisp the Language*. Digital Press, second edition, 1990.

[8] Matt Kaufmann and J Strother Moore. Miscellaneous remarks about guards. On the Web, April 2006. http://www.cs.utexas.edu/users/moore/acl2/v2-9/GUARD-MISCELLANY.html.

[9] OpenMCL. *The Ephemeral GC*, April 2006. http://openmcl.clozure.com/Doc/The-Ephemeral-GC.html.

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