

Copyright

by

David Lawrence Rager

2008

**Implementing a Parallelism Library for ACL2 in
Modern Day LISPs**

by

David Lawrence Rager, B.B.A

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

May 2008

**Implementing a Parallelism Library for ACL2 in
Modern Day LISPs**

**Approved by
Supervising Committee:**

Acknowledgments

Many people deserve and have my thanks for the help and support they have offered along the way. Warren Hunt Jr. has my gratitude for supervising this work. Matt Kaufmann created an ACL2 compilation flag that makes `mv` and `mv-let` thread safe. Since then he has helped refine the parallelism extension design, implementation, and documentation. Gary Byers has provided Closure Common Lisp (CCL, formerly known as OpenMCL) threading primitives in support of this project and has continually stepped up to the CCL implementation challenges this work has produced. I thank J Strother Moore, Robert Boyer, Sandip Ray, Jared Davis, and the ACL2 user group for their valuable discussions and technical advice. Finally, I thank the National Science Foundation and DARPA for funding this work under grant CNS-0429591, the National Security Agency, and Rockwell Collins, Inc.. With these groups' support, I was able to focus on the construction of this extension, and I am grateful to all involved parties.

I thank Sharon Kuhn, Sandip Ray, and Valerie Karplus for reading and providing thorough feedback on drafts of this paper. I thank Brent and Diane Rager for being excellent examples of work ethic and encouraging me to finish the work I started. And I thank Warren Hunt Jr. and Greg Lavender for being the

primary and secondary reviewers for this thesis.

DAVID LAWRENCE RAGER

The University of Texas at Austin

May 2008

Implementing a Parallelism Library for ACL2 in Modern Day LISPs

David Lawrence Rager, M.A.

The University of Texas at Austin, 2008

Supervisor: Warren A. Hunt Jr.

This thesis discusses four parallelism primitives that have been incorporated into the ACL2 theorem prover. These primitives permit an ACL2 programmer to parallelize evaluation of ACL2 functions. This paper (1) introduces logical definitions for these primitives, (2) explains three features that improve performance of these primitives, (3) gives an explanation of the implementation, and (4) uses the parallelism primitives in examples to show improvement in evaluation time.

Contents

Acknowledgments	iv
Abstract	vi
Contents	vii
List of Tables	xi
List of Figures	xii
Chapter 1 Introduction	1
Chapter 2 Related Work	3
2.1 Functional Language Parallelism	3
2.2 Parallelized Theorem Provers	5
Chapter 3 Parallelism Primitives	7
3.1 Pargs	8
3.2 Plet	9
3.3 Pand	11

3.4	Por	12
Chapter 4 User-level Parallelism Features		13
4.1	Data Dependent Parallelism	13
4.2	Granularity Form	14
4.3	Early Termination	16
Chapter 5 Implementation		18
5.1	LISP Threading Interface	19
5.1.1	Mutual Exclusion and Signaling	20
5.1.2	Controlling Threads	21
5.2	Producing Work	22
5.2.1	Estimating Resource Availability	23
5.2.2	Examining the Work Producer Code	24
5.3	Consuming Work	29
5.3.1	Defining a Worker Thread	30
5.3.2	Limiting the Number of Worker Threads	30
5.3.3	The Birth of a Worker	34
5.3.4	Retrieving Work and Claiming Resources	35
5.3.5	Evaluating a Piece of Parallelism Work	38
5.3.6	Cleaning Up After Work Evaluation	38
5.3.7	The Death of a Worker	40
5.4	Optimizations	42
5.4.1	Semaphore Recycling	42
5.4.2	Thread Recycling	43

5.4.3	Resumptive Heuristics	44
5.4.4	Granularity Test Costs	45
Chapter 6	Performance Results	46
6.1	Measuring Overhead	47
6.2	Doubly Recursive Fibonacci	49
6.3	Matrix Multiplication	50
6.4	Mergesort	52
6.5	Results Summary	52
Chapter 7	Conclusions and Future Work	54
7.1	Parallelism Extension Improvements	55
7.2	LISP Improvements	56
7.3	ACL2 Improvements	57
Appendices		57
Appendix A	Test Machine Specifications	58
A.1	BBH	58
A.2	Megaera	58
A.3	LHug-7	59
Appendix B	Software Specifications	60
B.1	CCL 32 bit PPC	60
B.2	CCL 64 bit x86 MAC	60
B.3	CCL 64 bit Linux x86	61
B.4	SBCL 64 bit threaded Linux	61

B.5 ACL2 3.2.1	61
Appendix C Parallel.lisp	62
Bibliography	136
Vita	141

List of Tables

6.1	Fibonacci Test Results (seconds)	50
6.2	Matrix Multiplication Test Results (seconds)	51
6.3	Mergesort Wall-time Test Results (seconds)	53

List of Figures

5.1	Pseudo Code for Parallelism Producers (parallelism primitives) . . .	30
5.2	Life Cycle of a Piece of Work	32
5.3	Ideal Amounts of Work by Classification	32
5.4	Pseudo Code for Worker Thread	41

Chapter 1

Introduction

This work is about adding parallelism capabilities to the ACL2 theorem prover [ACL07]. ACL2 is a theorem prover for first-order logic based on applicative common LISP. It has been used in some of the largest industrial systems that have been formally verified [BKM96]. Completing these projects critically depends on ACL2's ability to efficiently execute industrial-sized models [GWH02]. As multi-core CPUs [AMD06] become commonplace, ACL2 users would like to take advantage of the underlying available hardware resources [KM04, section 4.5]. Since ACL2 is a functional language and one can introduce parallelism into functional languages without worrying about safety [Agh86, section 1.4], it is reasonable to create a parallelism library with an interface that fits naturally into the ACL2 programming paradigm.

Parallelism has been introduced into ACL2 through four primitives: `pargs`, `plet`, `pand`, and `por`. `Pargs` (short for *parallelize-args*) is logically the identity macro, but it enables parallel evaluation of arguments to a function. `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 3.3 and 3.4.

Following the parallelism primitive introduction is a discussion on three features of the parallelism extension. First, with recursive use of the parallelism primitives, computation can adapt to the data so that a function's computation does not serialize on asymmetric data. Secondly, the parallelism extension provides 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. Thirdly, when issuing a `pand` or `por`, the extension recognizes opportunities for early termination and returns from evaluation.

An explanation of the implementation follows next. Included in this are discussions of (1) the LISP-level threading interface, (2) how parallelism work is produced and under what conditions, (3) how work is consumed and evaluated, and (4) some optimizations.

At the end of the paper, performance gains from using the above parallelism primitives can be seen with a naïve Fibonacci function and matrix multiplication. Mergesort evaluation times are also included to showcase the effects of garbage collection.

Chapter 2

Related Work

The parallelism extension derives guidance from two main bodies of research: (1) the work done to parallelize functional programs and (2) the efforts made to parallelize theorem provers. There has been extensive research in both directions; to give the reader a flavor of such work, only a sampling is provided below.

2.1 Functional Language Parallelism

Some of the work on parallelism began in the 1980s and includes ideas such as *futures* and primitives like *pcall* [RHH84, GM84]. More recent work includes an MPI library for GCL and Hunt and Moore’s partial ACL2 parallelism library implementation.

Multilisp was created in the early 1980s as an *extended version of scheme* [RHH84]. It implemented the *future* operator, which is often thought of as a promise for a form’s evaluation result [RHH89, section 4]. While the parallelism extension to ACL2 does not implement the *future* operator, this work inspired exploration of several issues like garbage collection, granularity, and the ordering of parallel

evaluation. As such, the notion of *future* is described briefly.

When a form was surrounded with the *future* primitive, the system would queue the form for parallel evaluation. The current thread could then proceed with computation, possibly queueing even more forms for parallel evaluation. The current thread would continue computation until it actually needed the evaluation result from one of the parallelized forms. At this point, the thread was said to *touch* [RHH89, section 6.4] the variable that represented the parallelized form, and the current thread's computation would halt until that parallelized form's evaluation was complete.

Hunt and Moore[private communication] have provided a partial implementation for an ACL2 parallelism extension using futures. The `future` primitive is omitted from the current library, but one could modify the LISP to provide it.

Further analysis of the ordering of parallel evaluation is discussed in Jones' book [Jon87]. In section 24.4.5, Jones states that "the tasks generated by a divide-and-conquer program can be thought of as a tree, in which each node is a task and the descendants of a node are the subtasks which it sparks." While Jones does not explicitly state that the computations higher in the tree should take priority over those further down the tree, he does recognize that the computations higher up are larger in granularity, and he states that parallelizing computations with larger granularity is better than parallelizing computations with smaller granularity. As a result, it can be concluded that it is better to first evaluate work higher in the tree, and to delay the pieces with smaller granularity for later. As such, the pieces of parallelism work, described in paragraph four of section 5.2.2, are added to the work queue in a FIFO manner. This FIFO ordering is implemented in the

function `add-work-to-work-queue` and also described in section 5.2.2. Halstead also supports a FIFO ordering on memory-rich machines [RHH89, section 6.2].

In somewhat more recent times, Cooperman provides bindings to MPI for GCL [Coo95]. His library provides typical MPI primitives like `send-message` and `receive-message`. Cooperman also provides parallel evaluation mechanisms such as `send-receive-message` and `par-eval`. Once the ACL2 parallelism extension has been applied to many large-granularity problems, it would be good to extend it to work on multi-machine platforms like par-GCL.

2.2 Parallelized Theorem Provers

A parallelized theorem prover is a theorem prover that evaluates portions of its proof in parallel with one another.

Kapur and Vandevoorde present 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 [DV96]. They both recognize 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` [ACL07, :doc hints], except that while ACL2 tries `or-hints` serially, DLP tries them in parallel.

In 1990 Schumann and Letz presented Partheo, “a sound and complete or-parallel theorem prover for first order logic” [SL90]. Partheo was written in parallel C and ran sequential theorem provers [LSBB92] 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 SETHO-based provers in parallel [Sch96]. SiCoTHEO starts multiple copies of the sequential theorem prover SETHEO [LSBB92], 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 [WF97]. 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. While ACL2's current approach is non-cooperative, if ACL2 began to use information from one subgoal's proof in a sibling's subgoal proof, this paper should be further examined.

Moten presents the parallel interactive theorem prover *MP refiner* [Mot98]. Although Moten's work was not directly used in this library, the introduction section of his paper provides motivation for exploring parallelism in theorem proving.

Chapter 3

Parallelism Primitives

Two goals are considered in the design of the ACL2 parallelism primitives. First, users need a way to efficiently parallelize computation in functions they execute. Secondly, the use of parallelism primitives should be as logically transparent as possible. As each parallelism primitive is introduced, a logical definition will be accompanied by a specification describing which arguments to the primitive are evaluated in parallel. Examples are also included for each parallelism primitive's definition.

Before defining each primitive, a brief overview of ACL2's evaluation mechanisms is provided. Each function defined in ACL2 has two versions: (1) the logic version (also known as the **1** function) and (2) the raw LISP version (also known as the "under the hood" definition).

The logical version allows modeling without the complexity of threads, while the threads spawned by the raw LISP version allow parallel evaluation. By default, when a function is defined without guards or with unverified guards, the logical,

or **1**, version is evaluated. The user can then use `guard verification` to allow execution of the raw LISP version once the type specification for the function has been satisfied [ACL07, :doc guard-miscellany]. Since parallel evaluation only occurs in the raw LISP version, every function that uses a parallelism primitive must have its guards defined and verified before it will execute in parallel.

When examining the examples, the reader will observe two ACL2 specific features. The first has already been briefly described, the use of guards. The second feature is `mbe` [ACL07, :doc mbe]. MBE takes two keyword arguments, a `:logic` form, and an `:exec` form. Simply put, `mbe` is a macro that expands in the ACL2 loop to the `:logic` form and expands in raw LISP to the `:exec` form. The definition of `pfib`, found in section 4.2, demonstrates how `mbe` allows different definitions at the **1** and raw LISP levels.

3.1 Pargs

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

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((mbe :logic (or (zp x) (<= x 0))
            :exec (<= x 0))
        0)
        ((= x 1) 1)
        (t (pargs (declare (granularity (> x 27)))
                  (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 eval-

uation. If the programmer uses `pargs` in a function whose argument evaluations always require a large amount of time, the user will likely experience speedup. It is better to parallelize evaluations that require larger amounts of time, because one wants to minimize the number of times the parallelism overhead is encountered. For experimental results that measure parallelism overhead, see section 6.1. To avoid parallelizing computations that take shorter amounts of time, the user can use a granularity form, as described in section 4.2.

Since macros can change expressions in unexpected ways and sometimes do not even evaluate their arguments, the user is disallowed from `pargs`'ing macros. While it may be possible to reliably expand macros using the LISP function `macroexpand`, it has so far been avoided. If a user wishes to parallelize evaluation of arguments to a macro, they can use `plet` instead.

3.2 Plet

The second primitive, `plet`, is logically equivalent to the macro `let`. `Plet` makes use of closures [Wik07], which contain all the information necessary to do a computation and allow the computation to be postponed, or in this case, assigned to another thread. In its raw LISP version, `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)
  (declare (xargs :guard (natp x)))
  (cond ((mbe :logic (or (zp x) (<= x 0))
          :exec (<= x 0))
        0)
        ((= x 1) 1)
```

```
(t (plet (declare (granularity (> x 27)))
        ((fibx-1 (pfib (- x 1)))
         (fibx-2 (pfib (- x 2))))
        (+ fibx-1 fibx-2))))
```

As with `pargs`, 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. Since the closure will have all its arguments evaluated before being applied, this can be done reliably.

One can also use `plet` to write a function that behaves like the Scheme [AHT07, section 4.3.1] function `map`. `Map` takes a function symbol and a list as arguments. `Map` then applies this function to each element of the list and returns a list of the results of each application. The following macro, `defmap` defines a function named `name` that recursively calls the predefined function `f` in a way similar to `map`. The `element-guard` is used to satisfy the guards of function `f`. Satisfying the guards permits the function to evaluate in raw LISP, a requirement for parallel evaluation. A use of `defmap` is also included.

```
(defmacro defmap (f name element-guard)
  '(defun ,name (lst)
    (declare (xargs :guard (and (true-listp lst)
                                (,element-guard lst))))
    (if (atom lst)
        nil
        (plet ((car-result (,f (car lst)))
                (cdr-result (,name (cdr lst))))
              (cons car-result cdr-result))))))

(defmap fib fib-plist nat-listp)
```

3.3 Pand

The third primitive, `pand`, is fundamentally different from `and`. `Pand` evaluates its arbitrary number of arguments in parallel, evaluates their conjunction, and unlike `and`, returns a Boolean result. Based on this definition, there are two differences between `pand` and `and`. First, `pand` returns a Boolean result. This makes it consistent with `por`, which is described later. The second difference is that due to parallel evaluation, the truth or falsity of the evaluation of the first argument does not prevent the evaluation of the second argument. Why does this matter? Consider the following call:

```
(pand (consp x)
      (equal (car x) 'foo))
```

With `and`, the falsity of `(consp x)` prevents the evaluation of `(car x)`, whereas with parallel execution, both `(consp x)` and `(equal (car x) 'foo)` can evaluate in parallel. As such, the logical definition of `pand` does not provide `(consp x)` as a guard to evaluating `(car x)`.

To demonstrate a feature called *early termination*, suppose there is a function, `valid-tree`, that traverses a tree to test each atom and make sure it is a `valid-tip`. A parallel version could be as follows:

```
(defun valid-tree (x)
  (declare (xargs :guard t))
  (if (atom x)
      (valid-tip x)
      (pand (valid-tree (car x))
            (valid-tree (cdr x))))))
```

Once one of the arguments evaluates to `nil`, the `pand` of that argument can return `nil`. In turn, that parent's parent can return `nil`, and so forth. The feature

that allows this is named early termination and is explained in section 4.3.

3.4 Por

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 is 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` prevents evaluation in a different way than `or`.

As an example, suppose the user has a function with the following macro call:

```
(por (atom x)
     (equal (car x) 'foo))
```

With `por` both `(atom x)` and `(equal (car x) 'foo)` can execute in parallel. Where as with `or`, the veracity of `(atom x)` prevents the evaluation of `(car x)`. This library's implementation of `por` does not use earlier arguments as logical guards to the evaluations of later arguments.

Chapter 4

User-level Parallelism Features

The parallelism library provides three features anticipated to enhance its usefulness to the ACL2 community. A discussion of these user-level features follows.

4.1 Data Dependent Parallelism

When performing computation on symmetric data, it is often relatively easy to partition work and allocate CPU resources efficiently. For example, when sorting a tree, the computation is split at the top recursive level and two pieces of parallelism work are created. However, when the data is asymmetric (similar to a list), the evaluation of one piece may terminate significantly before the evaluation of the other piece, effectively serializing computation. To be more concrete, consider the following function that counts the leaves of a tree:

```
(defun naive-pcount (x)
  (declare (xargs :guards t))
  (if (atom x)
      1
      (pargs (binary++ (acl2-count (car x))
```

```
(acl2-count (cdr x))))))
```

If this function is given a degenerate tree shaped like a list, parallelizing the computation at only `binary+` subsequent subroutine calls results in an almost serial computation.

So how can computation parallelize again after the `car` call terminates? The solution to this problem fits quite naturally into functional programming, and thus, ACL2. The user must simply define functions to call or recur into functions that use the parallelism primitives. Since whenever a primitive is encountered the computation has the chance to parallelize, if and when resources are available at deeper recursive levels, parallelism can be reintroduced. See section 5.2.1 for an explanation of determining resource availability.

4.2 Granularity Form

When computing many functions, for example 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 parallelism overhead. For example, consider the definition of Fibonacci found in section 3.1. Experiments on LHug-7, an eight CPU core machine at the University of Texas (see appendix A.3 for further details), indicate that whenever `x` is less than thirty, that a serial version of the Fibonacci function is often more efficient. This could require introducing two definitions of the function, e.g.,

```
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (cond ((mbe :logic (or (zp x) (<= x 0))
```

```

      :exec (<= x 0))
    0)
    ((= x 1) 1)
    (t (binary-+ (fib (- x 1))
                 (fib (- x 2))))))

(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((mbe :logic (or (zp x) (<= x 0))
            :exec (<= x 0))
        0)
        ((= x 1) 1)
        (< x 30) (binary-+ (fib (- x 1))
                           (fib (- x 2))))
        (t (pargs (binary-+ (pfib (- x 1))
                            (pfib (- x 2)))))))

```

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 the dynamic evaluation of the granularity form does not return `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 [Ste90].

```

(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((mbe :logic (or (zp x) (<= x 0))
            :exec (<= x 0))
        0)
        ((= x 1) 1)
        (t (pargs (declare (granularity (>= x 30)))
                  (binary-+ (pfib (- x 1))
                            (pfib (- x 2)))))))

```

The user can also declare a granularity form that uses an extra argument to describe the call depth of the current function. Take `mergesort` as an example. `Mergesort` splits the data into symmetric chunks for computation, so there is no

asymmetric splitting of data. As such, mergesort can use a `depth` argument to help determine when to parallelize computation. In this example, the depth is incremented in both the `car` and the `cdr` recursive calls. A parallelized version of `mergesort-exec` based on Davis's Ordered Sets library [Dav04] follows:

```
(defun mergesort-exec (x depth)
  (declare (xargs :guard (and (true-listp x) (integerp depth))))
  (cond ((endp x) nil)
        ((endp (cdr x)) (insert (car x) nil))
        (t (mv-let (part1 part2)
                  (split-list x)
                  (pargs (declare (granularity (< depth 4)))
                        (union (mergesort part1 (1+ depth))
                               (mergesort part2 (1+ depth))))))))))
```

To avoid altering function definitions to include depth parameters, one can analyze the data itself for structural properties. For example, one could define an auxiliary function that tested whether both the `car` and `cdr` of a particular input were `consp`'s.

4.3 Early Termination

When computing an ACL2 `and`, due to lazy evaluation, some of the arguments to the `and` may never be evaluated. Furthermore, if a user employs `pand`, even more evaluation may be skipped. Consider the following function that computes whether a tree is valid:

```
(defun pvalid-tree (x)
  (declare (xargs :guard t))
  (if (atom x)
      (valid-tip x)
      (pand (pvalid-tree (car x))
            (pvalid-tree (cdr x)))))
```

It is possible to stop execution as soon as any tip is found to be invalid. Therefore when computing the conjunction of terms by using `pand`, once any of those terms evaluates to `nil`, all sibling computations are aborted and the `pand` returns `nil`. In this example, if a `nil` result is returned beyond the first argument to the `pand`, the user may experience superlinear speedup.

The concept of early termination also applies to `por`, with the exception that the early termination condition is when an argument evaluates to non-`nil`. Some of the complexities involved in implementing early termination are discussed in section 5.2.2.

Chapter 5

Implementation

A sample of the implementation details that make up the extension’s evaluation strategy is provided below. This discussion is intended both to enable duplication of the work and to explain the actual implementation. As such, the reader may find it useful to examine “parallel.lisp” (see Appendix C) while reading this section.

¹ Kaufmann helped guide this initial implementation and its documentation, and hopefully this implementation will soon be part of the ACL2 distribution.

There are three main components that implement the parallelism strategy:

1. In order to avoid LISP-specific reader macros, the extension provides a LISP threading interface; this interface provides mechanisms for mutual exclusion, signaling, and controlling threads.
2. Once the threading interface was created, it became possible to implement parallelism work producers (parallelism primitives) and work consumers (threads

¹To make reading simpler, figure5.3.6 shows pseudo-code for the function `consume-work-on-work-queue-when-its-there`.

for evaluation); a discussion of the strategy and implementation of these producers and consumers follows after the interface explanation.

3. Finally, there are some optimizations anticipated to make the library higher performance. These optimizations are explained in section 5.4.

A modicum of familiarity with different LISP implementations and multi-threading mechanisms is assumed in this section.

5.1 LISP Threading Interface

The extension contains a threading interface that unifies different LISP parallelism features into one set of functions and macros. This interface provides mechanisms for mutual exclusion, signaling, and controlling threads. More specifically, it implements semaphores, condition variables, locks, and the ability to start, interrupt, and kill threads.

Since SBCL and CCL provide primitives sufficient to implement these mechanisms in a relatively straightforward way, the interface only supports SBCL and CCL. When this interface is used outside SBCL and CCL, the function and macro calls either turn into no-ops or perform operations aimed to help with compile-time checking. For example, `(semaphorep (make-semaphore))` is true in every LISP. Other LISPs are certainly considered, but since they do not offer the same threading primitives found in CCL and SBCL, porting to their platforms has been left for a later date.

5.1.1 Mutual Exclusion and Signaling

The interface provides locks, semaphores, and condition variables. Since CCL and SBCL support locks, their interface implementation is only a matter of translation. While CCL provides semaphores, SBCL does not. As such, the SBCL implementation of semaphores uses a structure that contains a counter, a lock, and a condition variable. The interface also provides an extra semaphore feature called a semaphore notification object, which is described in the next paragraph. Lastly, the library provides a condition variable interface. Since the basic parallelism primitives do not require locking for condition variable signaling and receiving, the provided condition variable methods do not require the use of locks.

Guaranteeing safety and liveness properties in the parallelism library requires a way to test whether a semaphore signal was received. Under normal thread execution, this only involves checking the return value of the `semaphore-wait` function. However, if the execution of a thread is aborted while it is waiting on a semaphore, there is no return value to check. In support of this project, the CCL maintainers created a *semaphore notification object* that can be set atomically whenever a semaphore signal is received. Often when a thread receives a semaphore signal, it needs to react by doing something. By placing the appropriate action and the clearing of the notification object inside a `without-interrupts` (a LISP wrapper that disables interrupts), the program is guaranteed that the semaphore value is updated iff the notification object is also updated. In the event that a particular execution is aborted, the surrounding `unwind-protect` can check the notification object and know whether that action occurred. The parallelism extension uses this feature to

keep an accurate record of threads, pieces of parallelism work, and other data.

A LISP `unwind-protect` takes two arguments: a body and a cleanup form. After the body finishes executing, the cleanup form evaluates. In the event that an error occurs, the cleanup form will always run. Furthermore, interrupts are disabled while evaluating the cleanup form. These properties prove useful for implementing many functions in the parallelism library, including the function `parallelize-closure-list`, described in section 5.2.2, and function `consume-work-on-work-queue-when-its-there`, described in section 5.3.4.

While using semaphores and condition variables is almost always more efficient [RHH89, section 5.2.1], the threading interface also provides a busy wait function, `thread-wait`. A thread calls this function by passing a function and argument to evaluate. If this evaluation does not return `nil`, `thread-wait` returns and the thread unblocks. Otherwise, the thread sleeps for 50 milliseconds and then reevaluates this closure. Since CCL already contains a busy waiting function, the interface uses it, and the amount of elapsed time between evaluations is decided by CCL.

5.1.2 Controlling Threads

The interface allows a programmer to create new threads, interrupt threads, and kill threads. Since CCL and SBCL both support thread creation, interruption, and termination, the implementation of these functions only involves translation. As explained in section 5.3.3, the parallelism library uses thread interruption to gracefully abort parallelized work evaluation.

5.2 Producing Work

Any thread that encounters a parallelism primitive has the potential to become a parallelism work producer. The implementation provides a parallelism *work queue* (named `*work-queue*`), in which pending work is collected. Three conditions must be true if the thread is to add parallelism work to the work queue. If any of the following three conditions is false, the parallelism primitive converts to its serial equivalent.

1. First, resources must be available. See section 5.2.1 for a discussion of how this property is determined.
2. Secondly, the current function call must operate on data large enough to warrant parallel computation. As previously described in section 4.2, the programmer can use a *granularity form* to meet this need. If the granularity form is omitted, this condition is assumed to be true.
3. Thirdly, the parallelism library must be enabled. In the current implementation, the parallelism library is enabled by default in threaded SBCL and CCL and disabled in the other LISPs.

Once a decision to parallelize a computation is made, pieces of work containing the information necessary to evaluate in parallel are added to the work queue²; these pieces of work have a relationship with the thread that creates them similar to the parent-child relationship and will be called *child work* throughout the remainder of this paper. Then the thread that parallelizes evaluation signals consumer threads

²An exact description of these pieces is found in 5.2.2

to evaluate the work and waits (on a signaling mechanism) until the parallel evaluations are finished. After the parallel evaluation finishes, the producer resumes execution as if the computation had been serialized all along.

5.2.1 Estimating Resource Availability

Conceptually pieces of parallelism work are organized into four states: *unassigned* (U), *started* (S), *pending* (W), and *resumed* (R). The first section refers to *unassigned* work not yet acquired by a work consumer. Pieces of work in the *unassigned* section are stored in the `*work-queue*`. The second state, *started*, describes the pieces of work being evaluated by an active thread that have not themselves encountered an opportunity for parallelism. The third section, *pending*, refers to work that itself encountered a parallelism primitive and decided to parallelize evaluation. After the work's children are finished evaluating, the work enters a fourth state, *resumed*, finishes evaluation, and returns the result. A piece of work can also be labeled as *active* or *inactive*. It is considered *active* if it is in either the *started* or *resumed* state and *inactive* if in the *unassigned* or *pending* state.

For reasons explained in section 5.3.2, it is necessary to limit two counts: (1) the total number of pieces of parallelism work in the system, and (2) the number of pieces unassigned or active at any given time. Limiting these two counts helps ensure the stability of the system and can therefore be thought of as a safety property. Another goal is to keep at least p (where p is the number of CPU cores) pieces of work in the *unassigned* section. This goal is a type of liveness property since it ensures there is work for parallel evaluation whenever a work consumer is looking for

work. The function `parallelism-resources-available` tests whether the above safety and liveness properties are satisfied. If they are met and parallel evaluation is enabled, the function returns `t`.

5.2.2 Examining the Work Producer Code

Once the system decides to evaluate a particular parallelism primitive in parallel, the system begins gathering and creating data necessary to distribute the computations to work consumers. This involves creating closures for each child computation, creating arrays necessary for communication between threads, and synthesizing this data into structures that will be placed on a global work queue, namely `*work-queue*`. Once these pieces of work are assembled, they are ready to be placed in the work queue. This is done by the function `parallelize-fn` and its subroutines, which are described below.

Parallelize-fn

`Parallelize-fn` requires two arguments and accepts a third optional argument. The first argument, `parent-fun-name`, is the symbol representing the name of the function to apply to the parallel evaluation of the closures. The second argument, `arg-closures`, is the list of closures to evaluate in parallel. The third and last optional argument, `terminate-early-function`, is used by `pand` and `por`. When non-`nil`, this argument is a function that can be applied to a term and returns a result indicating whether or not the term yields an early termination condition. For `pand`, this function is created with the expression `(lambda (x) (null x))`. `Por`'s `terminate-early-function` is even simpler, as the `lambda` only need return `x`. This

function is similar to the combining function described by Goldman et al. [GGS189, section 2.5].

`Parallelize-fn` in turn passes the list of closures and the early termination function to `parallelize-closure-list`, which will be explained in the next section. After `parallelize-closure-list` returns, `parallelize-fn` does one of two things, depending on the function name:

1. If the function name is equal to the symbol `and-list` or `or-list`, `parallelize-fn` applies the function `funcall` to the given symbol and the results of the call to `parallelize-closure-list`. Since both `and-list` and `or-list` are defined to return Boolean values, the `funcall` is not wrapped with a call to `if` that would convert the result to a Boolean a second time.
2. Otherwise, it applies the function `parent-fun-name` to the results of the call to `parallelize-closure-list`.

Parallelize-closure-list

`Parallelize-closure-list` accepts one required argument and a second optional argument. The first argument, `closure-list`, is the list of closures for parallel evaluation. The second argument, `terminate-early-function`, has been explained above.

`Parallelize-closure-list` should always be given a list of two or more closures to evaluate in parallel. This decision is justified by the one-to-many relationship that exists between the current thread and the child work it will create. If there is only one child closure, it would be better for the current thread to ap-

ply that closure to nil and return the result than to involve the parallelism library. Since `parallelize-closure-list` is intended to only work with parallelism calls, it does not perform this application but instead `assert`'s that the `closure-list` is of length greater than one.

Since parallelism consumers can themselves be parallelism producers, `parallelize-closure-list` must accommodate interruptions. In the event the thread is interrupted and instructed to throw the `:result-no-longer-needed` tag, it exits gracefully by removing its `unassigned` child work from the `*work-queue*` and signaling the threads associated with its child work to abort. With this in mind, the producer establishes three data structures to ensure safety and liveness:

1. `Work-list-setup-p` tracks whether the pieces of parallelism work have been added to the `*work-queue*`.
2. `Semaphore-notification-object` is used to ensure the accuracy of the `children-left-ptr`.
3. `Children-left-ptr` contains an array storing exactly one element, the number of child pieces of parallelism work that have not finished evaluation. Immediately before the pieces of parallelism work are added to the queue, the variable stored in `children-left-ptr` equals the number of children for that parent. During `parallelize-closure-list`'s evaluation, the count of children-left will decrement each time the parent receives a signal indicating a child finished. Since `parallelize-closure-list` does not exit until each child evaluation is accounted for, there is an invariant that whenever this function exits, that the number of children left to evaluate equals zero.

The call to `generate-work-list-from-closure-list` assembles a piece of parallelism work for each closure in `closure-list` and returns a list of these pieces of work. Each piece of parallelism work is defined by the structure `parallelism-piece`. `Generate-work-list-from-closure-list` creates one `thread-array`, one `result-array`, and one `semaphore-to-signal-as-last-act`. These three data structures are shared among all the current parent's children. Each child is also assigned a unique index into these arrays, a unique closure taken from `closure-list`, and a copy of a function formed from the `terminate-early-function`.

`Parallelize-closure-list` next establishes an `unwind-protect` form. While executing the body, the child pieces of work are added to the `*work-queue*`, and the evaluating thread waits for their parallel evaluation to complete. Conceptually, so long as early termination does not occur, the `children-left-ptr` will be 0 after the body finishes executing. If the producer receives an early termination request from its parent or one of its siblings, it sends all of its children an early termination request and waits for the incomplete children inside the cleanup form of the aforementioned `unwind-protect`.

To ensure that the next three operations either never start or all complete, the producer next establishes a `without-interrupts`. Once inside the `without-interrupts`, the producer spawns a number of worker threads that will both keep the CPU cores busy and not overwhelm the underlying Operating System. Although these threads begin as work consumers, they are named worker threads, because if the work itself has a parallelism primitive, they can also become producers. The label *worker* allows this dual role.

The producer next adds its child pieces of work to the `*work-queue*`. Interrupts are disabled to ensure that if the addition of work is successful, the `setup-work-list` variable is set. At this point, the producing thread is about to wait for its children. Before the thread enters the `pending` state, it calls the function `prepare-to-wait-for-children`, which (1) decrements the number of active threads (since the current thread is about to wait), (2) signals that a CPU core is available, and (3) signals that work has been added to the `*work-queue*`. Thus ends the `without-interrupts` block.

At this point, the current thread waits for the worker threads to finish evaluating its child pieces of work. One could have the parent continue useful computation by evaluating for one of its children. However, due to the complexities of early termination, this idea is difficult to implement. As such, the design is kept simpler by making the producer wait on a semaphore. Since the CPU cycle cost of waiting on a semaphore is low, this waiting likely does not interfere with performance from a timing perspective. The decision to wait instead of evaluate a piece of work on the `*work-queue*` requires more worker threads in total, and further discussion of this matter can be found in section 5.3.2.

If the children finish evaluation or the current thread receives an early termination request from its sibling or parent, it exits the body of the `unwind-protect` and enters the clean-up form. Assuming the work was added to the parallelism system (as determined by the earlier setting of the flag `work-list-setup-p`), the clean up form calls `terminate-children-early-and-rewait`, which, if appropriate, terminates the current thread's children early and again waits for them to finish

evaluation. An invariant maintained by the implementation is that if the current thread is able to exit the call to `terminate-children-early-and-rewait`, that all the current thread's children will have either finished evaluation or been successfully aborted and removed from the parallelism system.

Once the pieces of child work are finished, the current thread waits for parallelism resources. However, the thread uses a special set of heuristics for determining when CPU core resources are available (called resumptive heuristics in this implementation). See section 5.4.3 for an explanation of why the thread uses this different set of heuristics.

Due to the need for semaphore recycling, as explained in section 5.4.1, the `children-done-semaphore` is freed. Finally, the results are retrieved from the `results-array` and returned in the form of a list. As explained at the beginning of the current section, `parallelize-fn` will finish the evaluation of the parallelism primitive and return the result of that evaluation.

5.3 Consuming Work

Threads implement work consumers for the following two reasons. First, threads share memory, which is good for ACL2's target system, the SMP desktop market. Secondly, threads are lighter weight than processes [SGG03, page 131], lending themselves to finer-granularity problems.

Of the LISPs that support native threads and build ACL2, CCL and SBCL provide threading primitives sufficient to implement the parallelism extension as described in this paper.

```

(if (parallelism resources are unavailable)
  (evaluate the primitive serially)
  (progn
    (create appropriate closures for the primitive)
    (create pieces of work from the closures)
    (spawn worker threads if necessary)
    (add pieces of work to the work queue)
    (free parallelism resources since the current thread is
      about to wait on children)
    (wait for child work to finish being evaluated by worker
      threads)
    (when the current thread is aborted by one of its siblings
      or parent, it early terminates its children)
    ;; now the child pieces of work are finished being evaluated
    (wait for parallelism resources in a ‘‘resumptive’’ manner)
    (finish computing evaluating the primitive)
    (return the result)))

```

Figure 5.1: Pseudo Code for Parallelism Producers (parallelism primitives)

5.3.1 Defining a Worker Thread

A worker thread is created for the purpose of consuming and evaluating work placed on the **work-queue**. While a worker thread begins as a consumer, it will also become a producer if its piece of parallelism work encounters a parallelism primitive when parallelism resources are available.

5.3.2 Limiting the Number of Worker Threads

Since parallel evaluation consumes resources, the extension seeks to optimize the use of two resources: CPU cores and worker threads. To further explain how the resources are managed, classifications for each resource are described below.

CPU cores are said to be in one of two states: *active* and *idle*. A CPU

core is said to be *active* when the operating system (OS) executes a thread on it. Likewise, a CPU core is *idle* when the OS does not assign it a thread to execute. Since the library does not have access to the OS scheduler, it assumes that ACL2 is the only application consuming significant CPU cycles. Given this assumption and the CCL function that returns the total number of CPU cores, the library can track internally the number of CPU cores given code to execute (in SBCL the number of CPU cores is assumed to be a constant specified in `parallel.lisp`). As such, the extension need not interact with the Operating System to make a thread runnable, but instead, it can have threads that should not run block on LISP-level signaling mechanisms. This has worked well for applications thus far, but if the need arises, further investigation can occur.

Worker threads are said to be in one of three states: *idle*, *active*, and *pending*. A worker thread is *idle* whenever it is waiting either for a CPU core to become available (as according the parallelism library) or for work to enter the parallelism system. The parallelism library assumes that a worker thread is *active* only when it has been allocated a CPU core, is evaluating a piece of work, and not waiting on child computations. If a worker thread encounters a parallelism primitive and opts to parallelize evaluation, it enters the *pending* state until its children finish, when it becomes *active* again.

Figure 5.2 illustrates the relationships between pieces of work and their possible states, CPU cores, and worker threads.

Figure 5.2: Life Cycle of a Piece of Work

Work State	Unassigned	Started	Pending*	Resumed*
Allocated Core	no	yes	no	yes
Worker Thread State	n/a	active	pending	active

*the pending and resumed states are not always entered.

Limiting the Number of Active Worker Threads

The number of active worker threads is limited to match the number of CPU cores. Once a worker thread 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 [Jon89].

Keeping CPU Cores Busy

Whenever a worker thread acquires a CPU core, it immediately attempts to acquire work from the unassigned section, and if successful, begins evaluation. If there is no work in the unassigned section, the worker thread will go idle until work is added. If parallelism opportunities had recently occurred and performed serial evaluations because all CPU cores were busy, this idleness would be wasted time. To avoid this, the unassigned portion of `*work-queue*` is treated as a buffer and attempts to keep p pieces of work in it at all times. The number P is set to the number of CPU cores, so that if all worker threads finish evaluation simultaneously, they can acquire a new piece of work. Figure 5.3 shows the limits imposed for a system with p CPU cores.

Figure 5.3: Ideal Amounts of Work by Classification

unassigned	started + resumed	pending on children
$\cong p$	$p \leq \&\& \leq 2p$	
≤ 50		

Limiting Total Workload

Since the OS supports a limited number of worker threads, restrictions must be imposed to ensure application stability. It is not sufficient to simply set a limit on the number of threads spawned. The basis for this is as follows: (1) the stack of parents waiting on a deeply recursive nest of children can only unroll itself when they finish evaluating and (2) any piece of work allowed into the system must eventually be allocated to a worker thread. Further knowledge of the architecture of the parallelism system is required to understand why observation (2) is mentioned. Every parent that produces child work will usually spawn worker threads to evaluate this work.³ Therefore, before a parent can decide to add work, it must first check to see if the addition of work would require it to spawn more threads than are stable. If parallel evaluation would require spawning too many threads, then the parallelism primitive evaluates serially. As shown in figure 5.3, if the total count of already existing work is greater than fifty, the primitive opts for serial evaluation. When systems with more than sixteen cores become commonplace, the constant that contains the number fifty should be made larger.

The following example demonstrates how execution can result in generating deeply nested parallelism calls that can require a large number of parent threads waiting on their child threads (who are in turn parents). Suppose there is a function that counts the leaves of a tree, as below:

```
(defun pcount (x)
  (declare (xargs :guard t))
  (if (atom x)
```

³In the current implementation, the producer spawns a number of threads such that the total number of active and idle threads is equal to twice the number of CPU cores.

```
1
(pargs (binary-+ (pcount (car x))
                 (pcount (cdr x))))))
```

If this function is called 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 `pargs` parents waiting on their `pcount` children. Limiting the total amount of parallelism work prevents the need for these thousands of threads, and thus, the system maintains stability.

5.3.3 The Birth of a Worker

As mentioned in section 5.3.2, any time a producer adds parallelism work to the work queue, it determines whether there are enough active or idle worker threads in existence to consume and evaluate its work. If there are not enough worker threads available, the producer spawns n new threads, where the summation of n and the current count of active and idle threads is equal to twice the number of CPU cores in the system.

These new threads are initialized to run the function `consume-work-on-queue-when-its-there`. This function establishes many `catch` blocks during its execution. It first establishes a block that catches the `:worker-thread-no-longer-needed` tag. The worker threads waits for the condition that more work is added or a CPU core becomes available to be true. If this condition is false for a period of time,⁴ the worker thread will `throw` the

⁴Fifteen seconds in the CCL implementation

`:worker-thread-no-longer-needed` tag and expire. This particular `throw` and `catch` pair ensures that the parallelism extension frees OS resources (threads) after they are no longer needed. This process of freeing resources is currently only implemented in CCL. Should the SBCL maintainers introduce a timeout mechanism to the SBCL function `condition-wait`, it would be straightforward to extend the SBCL implementation in a similar way.

The special and thread-local variable `*throwable-worker-thread*` also needs justification. It is possible for a thread to come into existence and not enter the first `catch` block. As such, if that thread is thrown by the `(send-die-to-all-except-initial-threads)` function, the user would receive a cryptic error message. With the addition of this variable, the `send-die-to-all-except-initial-threads` function first tests the variable and only throws the consumer thread if it is not `nil`. This prevents the throwing of the `:worker-thread-no-longer-needed` tag outside of a block that can `catch` it. In reality, this is a rare occurrence, but the parallelism extension developers supposedly encountered it a few times. Since ACL2 is intended to not entertain unexpected errors, this variable is used to protect the user against obscure error messages.

5.3.4 Retrieving Work and Claiming Resources

`Consume-work-on-work-queue-when-its-there` next enters a seemingly infinite loop. In practice, this loop executes until the tag `:worker-thread-no-longer-needed` is thrown. The details of this outer loop are explained in sections 5.3.3 and 5.3.7.

Once inside the infinite and outer loop, the function has a chance to enter an inner loop. This second loop tests the condition that there is (1) work to consume and (2) at least one idle CPU core available for processing. If either of these conditions is false, the thread waits on the condition variable `*check-work-and-core-availability-cv*`. As explained in section 5.2.2 and later in this section, this condition variable is signaled after work is added to the work queue and whenever a CPU core is about to be available. As a result, whenever either of these two conditions is true, a worker thread awakens and checks for the `and` of these two conditions. This check occurs in the `while` test of the inner loop, after receiving the signal associated with `*check-work-and-core-availability-cv*`.

Since the `while` test ensures there is a good chance that both of the above conditions are true, the worker thread enters an `unwind-protect` form and optimistically claims an idle CPU core by decrementing `*idle-core-count*`. If the decrement results in a negative number of idle CPU cores, the `when` will fail, `work` will never be set, the body will unwind (incrementing `*idle-core-count*`), and the thread will loop to the top and again wait on `*check-work-and-core-availability-cv*`. If the decrement results in a non-negative number of idle CPU cores, the current thread establishes four scopes: (1) a `catch` block to receive the `:result-no-longer-needed` tag, (2) an `unwind-protect` to ensure proper cleanup, (3) a `progn` to allow a sequential `unwind-protect` body, and (4) a `without-interrupts` to prevent the thread from aborting inside the `pop-work-and-set-thread` function. For now, the discussion focuses on the `pop-work-and-set-thread` function.

The `pop-work-and-set-thread` function is responsible for correctly popping a piece of work from the `*work-queue*` and setting a pointer in a thread array to the consumer thread calling it. Once the consumer sets the thread pointer, other threads can interrupt it with a function that throws the `:result-no-longer-needed` tag. To prevent other threads from prematurely interrupting it, interrupts are disabled during the call to `pop-work-and-set-thread`.

Since worker threads can be spawned at any point in time, it is possible for the `*work-queue*` to be non-nil during the `(loop while ...)` test and execute until evaluating the `(pop *work-queue*)` inside `pop-work-and-set-thread`. If by the time it actually pops from the `*work-queue*`, another thread has performed a pop from the `*work-queue*`, the current thread is left with no work to consume. Under this scenario, the current thread binds `nil` to `work`, bypasses the two remaining `when` blocks, and increments `*idle-core-count*`.

When the current thread successfully acquires a piece of work, it goes through two phases, each controlled by a `(when work ...)` conditional. In the first phase, the current thread is interruptible, evaluates the work (see section 5.3.5), and aborts sibling computations when appropriate. In the second phase, the current thread cleans up state associated with the current thread's piece of parallelism work. After these two phases are both complete, the consumer unwinds its stack and loops around to the top-most loop. A detailed discussion of each of these phases follows in sections 5.3.5 and 5.3.6.

5.3.5 Evaluating a Piece of Parallelism Work

A consumer thread evaluates a piece of parallelism work by calling the function `eval-and-save-result`. `Eval-and-save-result` breaks the piece of work into the array that will store the evaluation result, an index into that array, and a closure to evaluate. After evaluating the closure, a `cons` of `t` and the result are stored in the indexed position of the array.

Storing a `cons` pair helps determine an early termination condition. Since it is possible for a closure to evaluate to `nil`, and since `nil` represents an early termination condition for `pand`, `nil` does not suffice as a representation for “evaluation not finished yet.” By storing a `cons` pair, a thread can test for atomicity⁵ and determine that the evaluation of a piece of parallelism work is finished if and only if its associated array location consists of a `cons` pair.

5.3.6 Cleaning Up After Work Evaluation

Completing the first `(when work ...)` block involves clearing the thread-array’s pointer to the current worker thread. Then, if appropriate, the worker thread aborts its siblings’ computations. The aborting thread first removes every now-irrelevant parallelism piece from the `*work-queue*`. It does this by comparing each piece of work’s `thread-array` for equality with the `thread-array` in the aborting thread’s piece of work. If they are equal, that piece of parallelism work is removed from the `*work-queue*`. After removing all related child work from the `*work-queue*`, the aborting thread interrupts every sibling thread whose pointer

⁵The thread tests for atomicity in the LISP/ACL2 sense, not in the sense of disallowing interrupts.

in the `thread-array` is set and tells them to evaluate a function that throws the `:result-no-longer-needed` tag. By `throwing` and `catching` the `:result-no-longer-needed` tag, the interrupted thread can abort evaluation of a piece of parallelism work in a controlled manner. This `throwing` effectively aborts the active evaluation of unnecessary sibling pieces of parallelism work.

Since the second (`when work ...`) block is part of the clean-up form of the `unwind-protect`, it is guaranteed to always execute. This allows the consumer to record the freeing of parallelism resources and signal other threads that resources are available. Why record the freeing of parallelism resources before signaling the condition variables? Once the other threads awaken, they should have an accurate view of the program state. If the program state has not yet been updated because the finishing thread signaled first, the awakening threads will make decisions based off data soon to be outdated. To prevent this, the current thread changes the state first and then signals the condition variables.

Since it is possible for a consumer thread to abort evaluation before reaching the `(setf (aref thread-array array-index) nil)` command of the `unwind-protect`'s body, the current thread must again clear the `thread-array` pointer inside the clean up form.

The current thread next increments the `*idle-thread-count*`. The current thread is about to go “idle”, because it has finished its piece of work, and it must be recorded.

As discussed in section 5.2.1, the variable `*total-work-count*` helps determine whether parallelism resources are available. Since the `*total-work-count*`

will now be one less, the current thread decrements it. For the same reasons, it decrements `*unassigned-and-active-work-count*`.

Now that the consumer thread has recorded the freeing of resources, it begins to signal different threads. The consumer first signals its parent, indicating that it has finished evaluating one of the parent's children. It next signals two condition variables. The first condition variable, `*check-work-and-core-availability-cv*`, tells worker threads waiting for the two conditions described in section 5.3.4 to be true to retest the conditions. While these potentially beginning consumer threads certainly need CPU core time, they are given a lower priority than another class of consumer threads, namely threads resuming execution after waiting for child work to finish evaluation. The signaling of condition variable `*check-core-availability-for-resuming-cv*` allows a thread waiting on this set of heuristics to check for an available CPU core. See section 5.4.3 for an explanation of why there are two sets of heuristics.

The current thread next reenters the top of the loop that enables the thread to run until it throws the `:worker-thread-no-longer-needed` tag. See section 5.3.3 for more details on this tag.

5.3.7 The Death of a Worker

When a thread expires, it must decrement the `*idle-thread-count*`. After this safely occurs, the consumer thread expires, and the underlying LISP can garbage collect its resources.

```

(setup catch block for when a thread is no longer needed
 (setup the variable *throwable-worker-thread* for safety
  reasons)
(while there's no work available or until the thread is
 told it's no longer needed
 (wait for work to supposedly be added to the
  *work-queue*)
 (presume that work was added and allocated to the current
  thread and claim cpu-core resources))
(try to acquire a piece of work)
(if there wasn't work, unclaim the cpu-core resources, skip to
 the end and repeat the waiting just done)
;; the current thread now has a piece of work
(setup data structures so that the current thread is associated
 with the piece of work in that work's thread-array)
(evaluate the piece of work)
(if the evaluation results in an early termination condition,
 terminate the siblings)
(signal the parent that the work has been evaluated)
(unclaim the cpu-core resources and rewait for work by looping
 back to the beginning))

```

Figure 5.4: Pseudo Code for Worker Thread

5.4 Optimizations

While different optimizations have been previously mentioned, this section serves as a more detailed explanation for some of them.

5.4.1 Semaphore Recycling

In some versions of CCL, the rapid allocation of semaphores causes the LISP to become unstable. The current version of the parallelism extension creates one semaphore per parallelism parent. If the user executes a poorly parallelized function without using a granularity form, such as in the example below, many parallelism parents can spawn. Since each of these parents requires a semaphore, and since CCL and OS-X can only reliably handle around 30,000 semaphore allocations (even with a large number of them being available for garbage collection), it is necessary to manually recycle the semaphores after they became unused. In the below example, this optimization limits required semaphore allocations to the low thousands (as opposed to upwards of 55,000 semaphore allocations required without this optimization).

Example:

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((mbe :logic (or (zp x) (<= x 0))
         :exec (<= x 0))
        0)
        ((= x 1)
         1)
        (t
         (plet
          (declare (granularity (> x 11)))
          ((a (pfib (- x 1)))
           (b (pfib (- x 2)))))))))
```

```
(+ a b))))))
(pfib 45)
```

5.4.2 Thread Recycling

Initial implementations spawned a fresh thread for each piece of parallelism work. The current implementation, which allows threads to not expire but instead wait for a new piece of work performs better.⁶ There is overhead associated with setting up the loops and data structures necessary to recycle threads. However these costs can be compared to the time it takes an operating system to create a new thread.

To do this comparison, consider the following script. It first spawns two fresh threads to evaluate the arguments to the call `(binary-+ 3 4)`. The second part times the evaluation of `(pargs (binary-+ 3 4))`. The timing results suggest that it takes about 4-5 times longer to evaluate a parallelism call that spawns fresh threads instead of using a `pargs` that recycles threads. This script is run on LHug-7 (see A.3).

```
(defvar *x* 0)
(defvar *y* 0)

(defun parallelism-call ()
  (let* ((semaphore-to-signal (make-semaphore))
        (closure-1 (lambda ()
                     (progn (setf *x* 3)
                            (signal-semaphore semaphore-to-signal))))
        (closure-2 (lambda ()
                     (progn (setf *y* 4)
                            (signal-semaphore semaphore-to-signal))))
        (ignore1 (run-thread "closure-1 thread" closure-1))
        (ignore2 (run-thread "closure-2 thread" closure-2))
        (ignore3 (wait-on-semaphore semaphore-to-signal))
        (ignore4 (wait-on-semaphore semaphore-to-signal)))
```

⁶Suggestions for the current implementation came from Calvin Lin, Warren Hunt, and Jared Davis.

```

      (declare (ignore ignore1 ignore2 ignore3 ignore4))
      (+ ** *y*))

(time (dotimes (i 1000)
      (assert (equal (parallelism-call) 7))))

; Result from the LISP session
; (DOTIMES (I 1000) (ASSERT (EQUAL (PARALLELISM-CALL) 7)))
; took 689 milliseconds (0.689 seconds) to run

(time (dotimes (i 1000) (pargs (binary-+ 3 4))))

; Result from the LISP session
; (DOTIMES (I 1000) (PARGS (BINARY-+ 3 4)))
; took 150 milliseconds (0.150 seconds) to run

```

5.4.3 Resumptive Heuristics

Until now, the explanation of the implementation has focused exclusively on the case of a producer thread producing parallelism work and having many consumers evaluate those pieces of parallelism work. But what happens when the consumer itself encounters a parallelism primitive and becomes a producer? Once these consumer-producers' children finish, they could have a higher priority than the consumer threads that have not yet been evaluated. Consider the following simplified scenario. Suppose all the consumer-producer needs to do before finishing is apply a fast function like `binary-+` to the results of evaluating its arguments. Since hopefully the only work on the `*work-queue*` is of reasonably large granularity, surely the application of `binary-+` would be faster than evaluating a new piece of parallelism work. Also, by allowing the worker thread to finish a piece of work, an operating resource, a thread, becomes idle and available for more work. While there are scenarios that favor both priority schemes, a setup that favors the resuming

thread will likely free resources sooner and has been chosen for this implementation.

Implementing this scheme requires a second condition variable, namely `*check-core-availability-for-resuming-cv*`. It is named so, because only the consumer threads that have spawned children wait on it. When a waiting thread receives the condition variable signal, it will claim an idle core in a more liberal fashion. Instead of waiting for the `*idle-core-count*` to be positive, it will wait for `*idle-core-count*` to be greater than or equal to the negation of the `*core-count*`. For example, if there are 8 CPU cores and 8 active threads, then there can be up to 8 additional active threads that have become active through the resumptive heuristics. This would make a total of 16 active threads. After a resumming thread claims an idle core in this way, they are said to be *active* once again.

5.4.4 Granularity Test Costs

Ideally, testing granularity would be constant in time, and not dependent on the data. For the Fibonacci function defined in section 4.2, this is the case. However, since Fibonacci itself is fast (two branches, two recursive calls, and an addition), even the additional branch that tests for granularity and parallelism resources affects performance. Fortunately, the performance degradation is small enough so that measurable speedup still occurs. See the results in section 6.2 for exact numbers.

Since the evaluation of granularity forms must be fast in any case, the granularity form is tested before resource availability. If future applications demonstrate that more expensive granularity forms are common, this decision should be revisited.

Chapter 6

Performance Results

The parallelism extension is evaluated on three machines: BBH (see A.1), Megaera (see A.2), and LHug-7 (see A.3). Since BBH is a 32 bit PPC and Megaera and LHug-7 are 64 bit x86, the tests are run on two architectures. CCL is used on all three machines, and SBCL is run on LHug-7. With this setup, “perfect parallelism” would compute parallelized ACL2 functions in one half their serial time on BBH, one quarter their serial time on Megaera, and one eighth of their serial time on LHug-7. Unless stated otherwise, all times reported in this section are an average of three consecutive executions, and the test scripts and log files can be found in the supporting evidence file pointed to below.

Three tests demonstrate some capabilities and weaknesses of the parallelism system. First, a doubly recursive version of the Fibonacci function demonstrates near-linear speedup with respect to the number of CPU cores. Second, matrix multiplication exhibits noticeable speedup. There is some preprocessing and garbage collection time that limits speedup to a factor of five and a half on an eight core

machine, but this speedup is still anticipated to be useful to the user. Finally, mergesort demonstrates speedup on the non-garbage collection portion of execution, but the garbage collection time limits speedup, regardless of the number of CPU cores. Its results are included as a motivating example towards future work in parallelizing garbage collectors.

The scripts and output from running these tests are available for download at: <http://www.cs.utexas.edu/users/ragerdl/masters/supporting-evidence.tar.gz>

6.1 Measuring Overhead

Threading and communication overhead occurs when a parallelism primitive is encountered and the decision to parallelize computation is made (instead of evaluating serially). The following examples seek to answer the question, “How many trivial parallelism calls can be processed per second?”

The first example evaluates two atoms (3 and 4) in parallel and then applies a relatively simple function (`binary--+`) to the results of their parallel evaluation. In an effort to avoid logical issues pertaining to ACL2 and only benchmark performance, the example is run from the raw loop of ACL2.¹

Why consider such trivial examples as below? They provide a basis for calculating the cost in time and memory for parallelizing computation, measured both per parallelism primitive and per argument. These examples are run on LHug-7 (see appendix for specifications).

```
(time (dotimes (i 100000) (pargs (binary--+ 3 4))))
```

Evaluation Result:

¹raw LISP mode can be entered by typing `:q` from an ACL2 prompt.

```
(DOTIMES (I 100000)
  (PARGS (BINARY-+ 3 4))) took 19,325 milliseconds
  (19.325 seconds) to run with 8 available CPU cores.
During that period,
  21,085 milliseconds (21.085 seconds) were spent in user mode
  19,421 milliseconds (19.421 seconds) were spent in system mode

9,166 milliseconds (9.166 seconds) was spent in GC.
51,701,821 bytes of memory allocated.
417 minor page faults, 0 major page faults, 0 swaps.
NIL
```

On LHug-7, this call takes 19.325 seconds of wall clock time. Also, 51,701,821 bytes of memory were allocated. By division, each `pargs` evaluation requires 193 microseconds and 517 bytes of memory. In turn, each argument requires 97 microseconds and 259 bytes of memory.

Next, consider what happens to the overhead when more than two arguments are passed into the function being parallelized:

```
(defun octal-+ (a b c d e f g h)
  (+ a b c d e f g h))

(time (dotimes (i 100000) (pargs (octal-+ 1 2 3 4 5 6 7 8))))
```

Evaluation Result:

```
(DOTIMES (I 100000)
  (PARGS (OCTAL-+ 1 2 3 4 5 6 7 8))) took 65,381 milliseconds
  (65.381 seconds) to run with 8 available CPU cores.
During that period,
  40,691 milliseconds (40.691 seconds) were spent in user mode
  41,483 milliseconds (41.483 seconds) were spent in system mode
7,415 milliseconds (7.415 seconds) was spent in GC.
148,005,482 bytes of memory allocated.
374 minor page faults, 0 major page faults, 0 swaps.
```

Note that the execution time of 65.381 seconds is approximately 3.38 times the times of the `binary-+` case (19.325 seconds). Also, note that despite a similar

increase in memory allocation, that the garbage collection time is reduced. The reasons for this are currently unknown, and nondeterminism in the garbage collector is suspected (although this decrease has been found to be repeatable). By division, each `pargs` requires 654 microseconds and generates about 1,480 bytes of garbage. In turn, each parallelized argument requires 82 microseconds and about 185 bytes.

6.2 Doubly Recursive Fibonacci

The doubly recursive definition of Fibonacci (see section 3.1 or supporting evidence for definition) is intended to be a simple example for the user to follow. Additionally, the Fibonacci computation demonstrates an ability to adapt to asymmetric data, as each time that `Fib` parallelizes, it will parallelize on data of different size. For example, an evaluation of `(fib 44)` takes less time than an evaluation of `(fib 45)`. As such, once `(fib 44)` finishes evaluating, other subcomputations of `(fib 45)` have a chance to parallelize computation, using idle CPU cores. On a similar note, Fibonacci demonstrates the usefulness of the granularity form. Since evaluation of `(fib 12)` takes less time than an evaluation of `(fib 45)`, the example code specifies that parallelism will not occur during the `(fib 12)` call. This feature is often called data-dependent parallelism and requires an efficiently executable granularity form to be effective. This double recursive definition of Fibonacci is inefficient, but it serves as a basis for determining whether the parallelism system experiences speedup that increases linearly with respect to the number of CPU cores. The Fibonacci function requires no setup, is computation heavy, and does not create much garbage, allowing an accurate measurement of parallelism overhead and the effects of granularity.

The following table shows the speedup for the above-mentioned platforms and machines. Two of the serial evaluation times for (fib 48) were abnormally high for this particular LHug-7 SBCL run, so the minimum of the three times is used. This minimum is consistent with other test results not included in the supporting evidence section and believed to most accurately portray the parallelism library's speedup.

Table 6.1: Fibonacci Test Results (seconds)

Case	Total Time	GC Time	Non-GC Time	Total Speedup	Non-GC Speedup
<i>BBH CCL PPC 32</i>					
Serial	279.05	0.00	279.05		
Parallel	161.23	0.02	161.21	1.73	1.73
<i>Megaera CCL x86-64</i>					
Serial	160.87	0.00	160.87		
Parallel	46.15	0.01	46.14	3.49	3.49
<i>LHug-7 CCL x86-64</i>					
Serial	192.31	0.00	192.31		
Parallel	28.78	0.09	28.69	6.68	6.70
<i>LHug-7 SBCL</i>					
Serial	462.89	0.00	492.07		
Parallel	65.76	0.01	65.75	7.04	7.04

6.3 Matrix Multiplication

The second test demonstrates speedup on matrix multiplication. When multiplying matrices that stay within fixnum representations, small amounts garbage are generated, and noticeable speedup can be achieved. The tests below involve multiplying matrices that are 2048x2048 in dimension. In the tests run on LHug-7, for a matrix

computation that requires 172 seconds to evaluate serially in SBCL, it takes 46 seconds to evaluate in parallel. This is a speedup factor of 3.77, which is 47% of the ideal speedup. The serial and parallel versions both take advantage of cache effects, as described in an algorithm by Bryant and O'Hallaron [BO03, page 518]. Even the serial component, the transposition that occurs before the actual multiplication, takes advantage of these cache effects.

A large portion of the speedup is likely lost in the parallel version due to the overhead from splitting the problem hierarchically. This hierarchical splitting is in contrast to the serial version, which can just cdr down lists. For more details, see the matrix multiplication implementations found in the supporting evidence.

Table 6.2: Matrix Multiplication Test Results (seconds)

Case	Total Time	GC Time	Non-GC Time	Total Speedup	Non-GC Speedup
<i>BBH CCL PPC 32</i>					
Serial	154.72	7.90	146.83		
Parallel	85.73	7.75	77.98	1.80	1.88
<i>Megaera CCL x86-64</i>					
Serial	82.99	4.21	78.78		
Parallel	32.30	4.51	27.79	2.57	2.83
<i>LHug-7 CCL x86-64</i>					
Serial	145.08	6.58	138.50		
Parallel	53.30	6.83	46.46	2.72	2.98
<i>LHug-7 SBCL</i>					
Serial	172.43	3.31	169.13		
Parallel	45.73	3.52	42.21	3.77	4.01

6.4 Mergesort

Finally, consider another well-known algorithm, mergesort. While highly parallel, the applicative version of mergesort generates significant garbage. Before presenting results, it is necessary to discuss whether it is meaningful to examine the total execution time or whether the time spent outside the garbage collector is more meaningful. On one hand, the user only experiences total execution time (wall-clock time). On the other, since CCL does not have a parallelized garbage collector [Ope06], the best the parallelism extension can hope for is a speedup within the non-GC'd portion. Due to this subjective evaluation, both speedup with GC and without GC are reported. Below is a table of results, including measurements for `mergesort` as defined in the supportive scripts.

Megaera consistently halts the LISP when evaluating the parallel portion of mergesort. As such, its results for this test are omitted. A discussion of this problem can be found in section 7.1.

6.5 Results Summary

In short, Fibonacci is a problem that experiences near-linear speedup on the version of CCL that has been around for many years (32 bit PPC) and demonstrates an ability to adapt to asymmetric data. Matrix multiplication's speedup is not linear, but still useful to the user. Finally, mergesort's speedup could be useful but is heavily limited by the garbage it generates.

Table 6.3: Mergesort Wall-time Test Results (seconds)

Case	Total Time	GC Time	Non-GC Time	Total Speedup	Non-GC Speedup
<i>BBH PPC 32</i>					
Serial	26.13	13.92	12.21		
Parallel	25.46	19.69	5.80	1.03	2.11
<i>LHug-7 CCL x86-64</i>					
Serial	181.50	155.49	26.01		
Parallel	390.63	379.73	10.90	0.46	2.39
<i>LHug-7 CCL x86-64 with GC Disabled</i>					
Serial	30.81	0.00	30.81		
Parallel	6.62	0.00	6.62	4.65	4.65
<i>LHug-7 SBCL</i>					
Serial	124.77	92.47	32.30		
Parallel	529.85	495.45	34.40	0.24	0.94

Chapter 7

Conclusions and Future Work

The four parallelism primitives described in this thesis are: `pargs`, `plet`, `pand`, and `por`. These primitives allow significant speedup of execution for functions that generate small amounts of garbage and have large granularity. Functions whose granularity varies can use a granularity form to ensure parallelism only occurs with larger computations. Since CCL and SBCL have sequential garbage collectors, functions whose execution time is dominated by garbage collection do not experience as much speedup as those functions that generate small amounts of garbage. The provided ACL2 parallelism implementation is an example of a parallelism library implemented in modern day LISPs. Additionally, this implementation is anticipated to be of use to a community of users, specifically the ACL2 community.

The remainder of this section focuses on (1) how the parallelism extension itself can be improved, (2) how the underlying LISPs could better support parallelism, and (3) how ACL2 might benefit from using parallelism in the theorem proving process.

7.1 Parallelism Extension Improvements

The parallelism extension is ported to SBCL, but it still functions most reliably in CCL. With the advent of certain SBCL features (like semaphores notification objects and a function that returns the number of CPU cores in the system), the SBCL implementation could be made as reliable or useful as the CCL implementation. Additionally, the parallelism extension will hopefully be integrated into the main ACL2 distribution.

The parallelism library and tests still manages to occasionally “break” CCL x86-64 and SBCL, usually in the form of halting the LISP. If it turns out to be a bug in the parallelism code, the CCL debugging process should help find it and make the SBCL debugging process easier. One contribution of this work is that CCL and SBCL were stressed, and as a result, the implementors have improved the multithreading aspects of these LISPs. Future work includes further stressing of these multithreading components and fixing the bugs that cause the system to halt, wherever they may be.

The ACL2 parallelism paper from 2006 [Rag06] documents an average speedup factor of 3.8x on four cores for the parallelized Fibonacci function. The results in the current implementation show a 1.7x and 3.5x speedup on dual and quad-core machines. Further investigation of the cause of the disparity between the 2006 and current results could generate more efficient parallel evaluation for the ACL2 user.

7.2 LISP Improvements

The LISP community could focus on two main threading components to more effectively support the parallelization of LISP programs. First, the LISP community could create a standard for LISP multiprocessing implementations. This parallelism extension is implemented in SBCL and CCL because of their accessibility, the fact that ACL2 already has users for these LISPs, and the fact that their multiprocessing libraries were relatively straightforward and simple to use. LispWorks does provide a threading interface [Lis06b, section 14] [Lis06a, section 11] and can probably be used in this application. However, since Lispworks threads do not evaluate LISP code (as opposed to foreign function calls) concurrently in Linux [Lis06b, section 14.5], Lispworks parallelism has been avoided thus far. Allegro does support threading [Fra07a, section 8.1], but since the threads can not run concurrently [Fra07b], the library has not been implemented for Allegro. While GCL does not provide a threading library, it does provide a process-level fork-based `plet`. This implementation of `plet` may be useful for the future, but it is not currently used by this parallelism library.

Secondly, the mergesort tests suggest that performance gains would occur if LISP implementations provide a concurrent garbage collector, such as that in MultiScheme [JSM89] or in JVM implementations like pSemispaces and pMarkcompact [FDSZ01]. Even if LISP and ACL2 programmers do not rewrite their code to use these parallelism primitives, a garbage collector that can run in the background without blocking other threads would be a good way to transparently boost performance and use the extra CPU cores soon to be standard in most desktops.

7.3 ACL2 Improvements

Future applications include integrating parallelism into the theorem proving process. While initial experiments that introduce parallelism into rewriting of terms do not result in significant speedup,¹ perhaps relieving hypotheses during backchaining or other applications of relatively small granularity would be fruitful. However, in the opinion of the author, more significant performance gains should be realized when subgoals can be proved in parallel [KM04, section 4.5]. Since managing proof output is one of the more difficult parts of parallelizing subgoal proofs, ACL2's ability to save proof output for delayed printing is a step towards meeting the goal of subgoal parallelization [ACL07, :doc set-saved-output].

In the future, one might envision making use of a network of computers to discharge different subgoals generated in a monolithic proof in parallel. To make this practical, it will be necessary to (1) find a mechanism for keeping the ACL2 `state` in sync among the different computers, (2) discover a metric for detecting “time consuming” subgoals, and (3) develop a user interface that orders the outputs generated from the concurrent proof attempt in an intelligible manner. The provided SMP implementation is a stepping stone to finding proof problems large enough to warrant the overhead of network communication.

¹In fact, enabling parallelism in this way results in a slow down. This is anticipated to be acceptable since these experiments' real intention is to demonstrate a level of robustness for the parallelism extension.

Appendix A

Test Machine Specifications

A.1 BBH

DNS Name: bbh.csres.utexas.edu

Processors: 2.7 GHz PowerPC G5 (2)

Total Number of Cores: 2

Memory: 8 Gigabytes

Architecture: 64 bit PPC

Marketing Name: Mac PowerPC G5

LISP version(s) used: CCL 32 bit PPC

A.2 Megaera

DNS Name: megaera.csres.utexas.edu

Processors: 3.0 GHz Dual-core Intel Xeon (2)

Total Number of Cores: 4

Memory: 8 Gigabytes

Architecture: 64 bit x86

Marketing Name: Mac Pro Quad Xeon 64-bit Workstation

LISP version(s) used: CCL 64 bit x86

A.3 LHug-7

DNS Name: lhug-7.csres.utexas.edu

Processors: 2.2 GHz AMD Dual Core Opteron 850 (4)

Total Number of Cores: 8

Memory: 32 Gigabytes

Architecture: 64bit x86

Marketing Name: N/A LISP version(s) used: CCL 64 bit x86, SBCL 64 bit threaded

Appendix B

Software Specifications

B.1 CCL 32 bit PPC

File to download: `openmcl-darwinppc-snapshot-070722.tar.gz`

Available for ftp download from: `anonymous@clozure.com/pub/testing`

B.2 CCL 64 bit x86 MAC

File to download: `openmcl-darwinx8664-snapshot-070722.tar.gz`

Available for ftp download from: `anonymous@clozure.com/pub/testing`

B.3 CCL 64 bit Linux x86

File to download: [openmcl-linuxx8664-snapshot-070722.tar.gz](#)

Available for ftp download from: [anonymous@clozure.com/pub/testing](ftp://anonymous@clozure.com/pub/testing)

B.4 SBCL 64 bit threaded Linux

File to download: “the source”

Available for http download from: <http://www.sbcl.org/platform-table.html>

B.5 ACL2 3.2.1

File to download: [acl2.tar.gz](#)

Available for http download from: <http://www.cs.utexas.edu/users/moore/acl2/v3-2/new/v3-2-1/index.html>

This version of ACL2 was modified to include the parallelism extension. For instructions on how to download the complete modified version, please contact the author.

Appendix C

Parallel.lisp

```
; ACL2 Version 3.2.1 -- A Computational Logic for Applicative Common
; Lisp Copyright (C) 2006 University of Texas at Austin

; This version of ACL2 is a descendent of ACL2 Version 1.9,
; Copyright (C) 1997 Computational Logic, Inc. See the
; documentation topic NOTE-2-0.

; This program is free software; you can redistribute it and/or
; modify it under the terms of the GNU General Public License as
; published by the Free Software Foundation; either version 2 of the
; License, or (at your option) any later version.

; This program is distributed in the hope that it will be useful,
; but WITHOUT ANY WARRANTY; without even the implied warranty of
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
; General Public License for more details.

; You should have received a copy of the GNU General Public License
; along with this program; if not, write to the Free Software
; Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

; Written by: Matt Kaufmann          and J Strother Moore
; email:      Kaufmann@cs.utexas.edu and Moore@cs.utexas.edu
; Department of Computer Sciences
; University of Texas at Austin
; Austin, TX 78712-1188 U.S.A.

; The initial version of this file is contributed by David Rager.
; During this period Matt Kaufmann consulted with David Rager and
; helped guide this initial implementation and documentation with
; specific suggestions. Warren Hunt, Jared Davis, and Calvin Lin
; suggested recycling threads.

(in-package "ACL2")

; This file is divided into the following sections.

; Section: Enabling and Disabling Interrupts
; Section: Threading Interface
; Section: Parallelism Basis
; Section: Work Consumer Code
; Section: Work Producer Code
; Section: Parallelism Primitives

; In particular, see the Essay on Parallelism Definitions and the
; Essay on Parallelism Strategy in Section "Parallelism Basis" for
```

```

; overviews on this implementation of parallel evaluation.

; The following check should never fail, since we set the feature
; acl2-par in a perfectly corresponding manner in acl2-init.lisp.

#+(and acl2-par (not openmcl) (not (and sbcl sb-thread)))
(error "It is currently illegal to build the parallel ~
       version of ACL2 in this Common Lisp.")

;-----
; Section:  Enabling and Disabling Interrupts

; "Without-interrupts" means that there will be no interrupt from
; the Lisp system, including ctrl+c from the user or an interrupt
; from another thread/process.  For example, if *thread1* is running
; (progn (without-interrupts (process0)) (process1)), then execution
; of (interrupt-thread *thread1* (lambda () (break))) will not
; interrupt (process0).

; But note that "without-interrupts" does not guarantee atomicity;
; for example, it does not mean "without-setq".

#-acl2-loop-only
(defmacro without-interrupts (&rest forms)

; This macro prevents interrupting evaluation of any of the
; indicated forms in a parallel lisp.  In a non-parallel environment
; (#-acl2-par), we simply evaluate the forms.  This behavior takes
; priority over any enclosing call of with-interrupts.  See also
; with-interrupts.

  #+(and openmcl acl2-par)
  '(ccl:without-interrupts ,@forms)
  #+(and sbcl sb-thread acl2-par)
  '(sb-sys:without-interrupts ,@forms)
  #-acl2-par
  '(progn ,@forms))

#-acl2-loop-only
(defmacro with-interrupts (&rest forms)

; This macro allows interrupting evaluation of any of the indicated
; forms in a parallel lisp.  In a non-parallel environment
; (#-acl2-par), we simply evaluate the forms.  This behavior takes
; priority over any enclosing call of without-interrupts.  See also
; without-interrupts.

```

```

#+(and openmcl acl2-par)
  '(ccl:with-interrupts-enabled ,@forms)
#+(and sbcl sb-thread acl2-par)
  '(sb-sys:with-interrupts ,@forms)
#-acl2-par
  '(progn ,@forms))

#-acl2-loop-only
(defmacro unwind-protect-disable-interrupts-during-cleanup
  (body-form &rest cleanup-forms)

; As the name suggests, this is unwind-protect but with a guarantee
; that cleanup-form cannot be interrupted. Note that OpenMCL's
; implementation already disables interrupts during cleanup (1.1pre
; and later).

#+(and openmcl acl2-par)
  '(unwind-protect ,body-form ,@cleanup-forms)
#+(and sbcl sb-thread acl2-par)
  '(unwind-protect ,body-form (without-interrupts ,@cleanup-forms))
#-acl2-par
  '(unwind-protect ,body-form ,@cleanup-forms))

;-----
; Section: Threading Interface
;
; The threading interface is intended for system level programmers.
; It is not intended for the ACL2 user. When writing system-level
; multi-threaded code, we use implementation-independent interfaces.
; If you need a function not covered in this interface, create the
; interface!

; Many of the functions in this interface (lockp, make-lock, and so
; on) are not used elsewhere, but are included here in case we find
; a use for them later.

; We take a conservative approach for implementations that do not
; support parallelism. For example, if the programmer asks for a
; semaphore or lock in an unsupported Lisp, then nil is returned.

; We employ counting semaphores. For details, including a
; discussion of ordering, see comments in the definition of function
; make-semaphore.

; Note: We use parts of the threading interface for our

```

```

; implementation of the parallelism primitives.

#-acl2-loop-only
(defun lockp (x)
  #+(and openmcl acl2-par) (cl:typep x 'ccl::recursive-lock)
  #+(and sbcl sb-thread acl2-par) (cl:typep x 'sb-thread::mutex)
  #-acl2-par

; We return nil in the uni-threaded case in order to stay in sync
; with make-lock, which returns nil in this case. In a sense, we
; want (lockp (make-lock x)) to be a theorem if there is no error.

  (null x))

#-acl2-loop-only
(defun make-lock (&optional lock-name)

; See also deflock.

; Even though OpenMCL nearly always uses a FIFO for threads blocking
; on a lock, it does not guarantee so: no such promise is made by
; the OpenMCL documentation or implementor (in fact, we are aware of
; a race condition that would violate FIFO properties for locks).
; Thus, we make absolutely no guarantees about ordering; for
; example, we do not guarantee that the longest-blocked thread for a
; given lock is the one that would enter a lock-guarded section
; first. However, we suspect that this is usually the case for most
; implementations, so assuming such an ordering property is probably
; a reasonable heuristic. We would be somewhat surprised to find
; significant performance problems in our own application to ACL2's
; parallelism primitives due to the ordering provided by the
; underlying system.

  #-acl2-par
  (declare (ignore lock-name))
  #+(and openmcl acl2-par) (ccl:make-lock lock-name)
  #+(and sbcl sb-thread acl2-par) (sb-thread:make-mutex
                                   :name lock-name)
  #-acl2-par

; We return nil in the uni-threaded case in order to stay in sync
; with lockp.

  nil)

```

```

#-acl2-loop-only
(defmacro deflock (lock-symbol)

; Deflock defines what some Lisps call a "recursive lock", namely a
; lock that can be grabbed more than one time by the same thread,
; but such that if a thread outside the owner tries to grab it, that
; thread will block.

; Note that if lock-symbol is already bound, then deflock will not
; re-bind lock-symbol.

  '(defvar ,lock-symbol
      (make-lock (symbol-name ',lock-symbol))))

#-acl2-loop-only
(defmacro reset-lock (bound-symbol)

; This macro binds the given global (but not necessarily special)
; variable to a lock that is new, at least from a programmer's
; perspective.

; Reset-lock should only be applied to bound-symbol if deflock has
; previously been applied to bound-symbol.

  '(setq ,bound-symbol (make-lock ,(symbol-name bound-symbol))))

#-acl2-loop-only
(defmacro with-lock (lock-name &rest forms)

; Grab a lock, blocking until it is acquired; evaluate forms; and
; then release the lock. This macro guarantees mutual exclusion.

  #-acl2-par
  (declare (ignore lock-name))
  #+(and openmcl acl2-par)
  '(ccl:with-lock-grabbed (,lock-name) ,@forms)
  #+(and sbcl sb-thread acl2-par)
  '(sb-thread:with-recursive-lock (,lock-name) ,@forms)
  #-acl2-par
  '(progn ,@forms))

#-acl2-loop-only
(progn

; We declare some variables here that are necessary for the
; threading interface. These support semaphore recycling via

```

```
; functions allocate-lock and free-lock, which are useful for
; efficient implementation of wait-on-condition-variable-lockless.
```

```
(defvar *lock-allocation-lock* (make-lock))
(defvar *lock-freelist*
  #+openmcl
  (ccl::%cons-pool)
  #-openmcl
  nil)
)
```

```
#+acl2-loop-only
(defun allocate-lock ()
  (with-lock *lock-allocation-lock*
    (or (pop
        #+openmcl
        (ccl::pool.data *lock-freelist*)
        #-openmcl
        *lock-freelist*)
        (make-lock))))
```

```
#+acl2-loop-only
(defun free-lock (lock)
```

```
; Warning: This function requires that lock is unacquired.
```

```
; We considered creating a user-settable limit on the length of the
; *lock-freelist*. In favor of simplicity, we have not implemented
; this, but if OS lock resources become an issue, this is worth
; further consideration.
```

```
; We test that the lock is not currently acquired. Note that we do
; not test that the lock isn't already on the *lock-freelist*. It's
; questionable whether this check is worth it, since the user can
; get into trouble in many other ways.
```

```
(without-interrupts
  #+(and openmcl acl2-par)
  (when (not (ccl:try-lock lock))
    (error "A lock was freed while still being held."))
  #+(and sbcl sb-thread acl2-par)
  (if (sb-thread:get-mutex lock nil nil)
      (sb-thread:release-mutex lock)
      (error "A lock was freed while still being held.")))
```

```

(with-lock *lock-allocation-lock*
  (push lock
    #+openmcl
    (ccl::pool.data *lock-freelist*)
    #-openmcl
    *lock-freelist*)))

#-acl2-loop-only
(defun reset-lock-free-list ()

; We provide the user the ability to clear the locks stored for
; recycling. If an ACL2 programmer wants to free OS resources for
; garbage collection, they can use this method to free the locks
; stored in our system.

  (with-lock *lock-allocation-lock*
    (setf *lock-freelist*
      #+openmcl
      (ccl::%cons-pool)
      #-openmcl
      nil)))

#-acl2-loop-only
(defun run-thread (name fn-symbol &rest args)

; Apply fn-symbol to args. We follow the precedent set by LISP
; machines (and in turn OpenMCL), which allowed the user to spawn a
; thread whose initial function receives an arbitrary number of
; arguments.

; We expect this application to occur in a fresh thread with the
; given name. When a call of this function returns, we imagine that
; this fresh thread can be garbage collected; at any rate, we don't
; hang on to it!

; Note that run-thread returns different types in different Lisps.

  #-acl2-par
  (declare (ignore name))
  #+(and openmcl acl2-par)
  (ccl::process-run-function name (lambda () (apply fn-symbol args)))
  #+(and sbcl sb-thread acl2-par)
  (sb-thread:make-thread (lambda () (apply fn-symbol args))
    :name name)

; We're going to be nice and let the user's function still run, even

```

```

; though it's not split off.

#-acl2-par
  (apply fn-symbol args))

#-acl2-loop-only
(defun interrupt-thread (thread function &rest args)

; Interrupt the indicated thread and then, in that thread, apply
; function to args. Note that function and args are all evaluated.
; When this function application returns, the thread resumes from
; the interrupt (from where it left off).

#-acl2-par
  (declare (ignore thread function args))
  #+(and openmcl acl2-par)
  (apply #'ccl:process-interrupt thread function args)
  #+(and sbcl sb-thread acl2-par)
  (if args
      (error "Passing arguments to interrupt-thread not supported ~
              in SBCL.")
      (sb-thread:interrupt-thread thread function))
#-acl2-par
  nil)

#-acl2-loop-only
(defun kill-thread (thread)
  #-acl2-par
  (declare (ignore thread))
  #+(and openmcl acl2-par)
  (ccl:process-kill thread)
  #+(and sbcl sb-thread acl2-par)
  (sb-ext:process-kill thread)
  #-acl2-par
  nil)

#-acl2-loop-only
(defun all-threads ()
  #+(and openmcl acl2-par)
  (ccl:all-processes)
  #+(and sbcl sb-thread acl2-par)
  (sb-thread:list-all-threads)
  #-acl2-par
  (error "We don't know how to list threads in this lisp (or ~
         acl2-par is not on the features list.))")

```

```

#-acl2-loop-only
(defun current-thread ()
  #+(and openmcl acl2-par)
  ccl:*current-process*
  #+(and sbcl sb-thread acl2-par)
  sb-thread:*current-thread*
  #-acl2-par
  nil)

#-acl2-loop-only
(defun thread-wait (fn &rest args)

; Thread-wait provides an inefficient mechanism for the current
; thread to wait until a given condition, defined by the application
; of fn to args, is true. When performance matters, we advise using
; a signaling mechanism over this hacker's function.

  #+openmcl
  (apply #'ccl:process-wait "Busy-waiting on condition for thread"
    fn args)

; Sleep outside of openmcl.

  #-openmcl
  (loop while (not (apply fn args)) do (sleep 0.05)))

#+(and sb-thread (not acl2-loop-only))
(defstruct sbcl-semaphore
  (lock (sb-thread:make-mutex))
  (cv (sb-thread:make-waitqueue)) ; condition variable
  (count 0))

#-acl2-loop-only
(defun make-semaphore (&optional name)

; Make-semaphore, signal-semaphore, and semaphorep work together to
; implement counting semaphores for the threading interface.

; This function creates "counting semaphores", which are data
; structures that include a "count" field, which is a natural
; number. A thread can "wait on" a counting semaphore, and it will
; block in the case that the semaphore's count is 0. To "signal"
; such a semaphore means to increment that field and to notify a
; unique waiting thread (we will discuss a relaxation of this
; uniqueness shortly) that the semaphore's count has been
; incremented. Then this thread, which is said to "receive" the

```

```

; signal, decrements the semaphore's count and is then unblocked.
; This mechanism is typically much faster than busy waiting.

; In principle more than one waiting thread could be notified
; (though this seems rare in practice). In this case, only one
; would be the receiving thread, i.e., the one that decrements the
; semaphore's count and is then unblocked.

; If semaphore usage seems to perform inefficiently, could this be
; due to ordering issues? For example, even though OpenMCL nearly
; always uses a FIFO for blocked threads, it does not make such a
; guarantee: no such promise is made by the OpenMCL documentation or
; implementor. Thus, we make absolutely no guarantees about
; ordering; for example, we do not guarantee that the
; longest-blocked thread for a given semaphore is the one that would
; receive a signal. However, we suspect that this will usually be
; the case for most implementations, so assuming such an ordering
; property is probably a reasonable heuristic. We would be somewhat
; surprised to find significant performance problems in our own
; application to ACL2's parallelism primitives due to the ordering
; provided by the underlying system.

; OpenMCL provides us with semaphores for signaling. SBCL provides
; condition variables for signaling. Since we want to code for one
; type of signaling between parents and children, we create a
; semaphore wrapper for SBCL's condition variables. The structure
; sbcl-semaphore implements the data for this wrapper.

; Followup: SBCL has recently implemented semaphores, and the
; parallelism code should maybe be changed to reflect this. It
; probably depends on whether their implementation provides
; previously discussed semaphore-notification-object's.

(declare (ignore name))
#+(and openmcl acl2-par)
(ccl:make-semaphore)
#+(and sbcl sb-thread acl2-par)
(make-sbcl-semaphore)
#-acl2-par

; We return nil in the uni-threaded case in order to stay in sync
; with semaphorep.

nil)

#-acl2-loop-only

```

```

(defun semaphorep (semaphore)

; Make-semaphore, signal-semaphore, and semaphorep work together to
; implement counting semaphores for our threading interface.

; This function recognizes our notion of semaphore structures.

#+(and openmcl acl2-par)
  (typep semaphore 'ccl::semaphore)
#+(and sbcl sb-thread acl2-par)
  (and (sbcl-semaphore-p semaphore)
        (typep (sbcl-semaphore-lock semaphore) 'sb-thread::mutex)
        (typep (sbcl-semaphore-cv semaphore) 'sb-thread::waitqueue)
        (integerp (sbcl-semaphore-count semaphore))))
  #-acl2-par

; We return nil in the uni-threaded case in order to stay in sync
; with make-semaphore, which returns nil in this case. In a sense,
; we want (semaphorep (make-semaphore x)) to be a theorem if there
; is no error.

  (null semaphore))

#-acl2-loop-only
(defun signal-semaphore (semaphore)

; Make-semaphore, signal-semaphore, and semaphorep work together to
; implement counting semaphores for our threading interface.

; This function is executed for side effect; the value returned is
; irrelevant.

  #-acl2-par
    (declare (ignore semaphore))
  #+(and openmcl acl2-par)
    (ccl:signal-semaphore semaphore)
  #+(and sbcl sb-thread acl2-par)
    (sb-thread:with-recursive-lock
      ((sbcl-semaphore-lock semaphore))
      (without-interrupts
        (incf (sbcl-semaphore-count semaphore))
        (sb-thread:condition-notify (sbcl-semaphore-cv semaphore))))
  #-acl2-par
    nil)

#-acl2-loop-only

```

```

(progn

; We declare some variables here that are necessary for the
; threading interface. These support semaphore recycling via
; functions allocate-semaphore and free-semaphore, which is useful
; for efficiency (especially when there are thousands of
; semaphores) and for avoiding errors in early versions (at least)
; of OpenMCL.

(defvar *semaphore-allocation-lock* (make-lock))
(defvar *semaphore-freelist*
  #+openmcl
  (ccl::%cons-pool)
  #-openmcl
  nil)
)

#-acl2-loop-only
(defun allocate-semaphore ()
  (without-interrupts
   (with-lock *semaphore-allocation-lock*
    (or (pop
        #+openmcl
        (ccl::pool.data *semaphore-freelist*)
        #-openmcl
        *semaphore-freelist*)
        (make-semaphore))))))

#-acl2-loop-only
(defun free-semaphore (s)

; Warning: This function assumes that s is properly initialized. In
; particular, it must have a count field of 0, and for SBCL, the
; lock field must be free (which is guaranteed if we only use the
; threading interface).

; We test that the semaphore is properly initialized. Note that we
; do not test that the semaphore isn't already on the
; *semaphore-freelist*. It's questionable whether this check is
; worth it, since the user can get into trouble in many other ways.

; We considered creating a user-settable limit on the length of the
; *semaphore-freelist*. In favor of simplicity, we have not
; implemented this, but if OS semaphore resources again become an
; issue, this is worth further consideration. If this limit is
; implemented, once reached, probably all semaphores should be

```

```
; discarded and a full gc called. This full gc is required to truly
; free the OS-level semaphores underneath the LISP implementation.
```

```
(without-interrupts
  #+openmcl
  (loop while (timed-wait-on-semaphore s 0))
  #+(and sbcl sb-thread)
  (let ((lock (sbcl-semaphore-lock s)))
```

```
; Note that we must test the count before the mutex, so that we
; don't acquire the mutex and then not release it.
```

```
  (if (or (not (equal (sbcl-semaphore-count s) 0))
          (not (sb-thread:get-mutex lock nil nil)))
      (error "It is a design violation to free a semaphore ~
             while its lock is acquired.")
      (sb-thread:release-mutex lock))))
```

```
(with-lock *semaphore-allocation-lock*
  (push s
    #+openmcl
    (ccl::pool.data *semaphore-freelist*)
    #-openmcl
    *semaphore-freelist*)))
```

```
#-acl2-loop-only
(defun reset-semaphore-free-list ()
```

```
; We provide the user the ability to clear the semaphores stored for
; recycling. If an ACL2 programmer wants to free OS resources for
; garbage collection, they can use this method to free the
; semaphores stored in our system.
```

```
(with-lock *semaphore-allocation-lock*
  (setf *semaphore-freelist*
    #+openmcl
    (ccl::%cons-pool)
    #-openmcl
    nil)))
```

```
#-acl2-loop-only
(defun wait-on-semaphore (semaphore
  &optional semaphore-notification-object)
```

```

; This function always returns t. It only returns normally after
; receiving a signal for the given semaphore, setting the
; notification status of semaphore-notification-object (if supplied
; and not nil) to true; see semaphore-notification-status. But
; control can leave this function abnormally, for example if the
; thread executing a call of this function is interrupted (e.g.,
; with interface function interrupt-thread) with code that does a
; throw, in which case semaphore-notification-object is unmodified.

; We need the ability to know whether we received a signal or not.
; OpenMCL provides this through a semaphore-notification-object. As
; it turns out, SBCL does not provide this mechanism currently, so
; we modify our wrapper to "unreceive the signal" in the semaphore.
; We do this by not decrementing the count of it unless we also
; modify the semaphore-notification object. This means we have to
; resignal the semaphore if we were interrupted while signaling, but
; we would have to do this anyway.

#-acl2-par
(declare (ignore semaphore semaphore-notification-object))

#+(and openmcl acl2-par)
(ccl:wait-on-semaphore semaphore semaphore-notification-object)

#+(and sbcl sb-thread acl2-par)
(let ((supposedly-did-not-receive-signal-p t))

  (sb-thread:with-recursive-lock
    ((sbcl-semaphore-lock semaphore))

    (unwind-protect-disable-interrupts-during-cleanup
      (progn
        (loop while (<= (sbcl-semaphore-count semaphore) 0) do

; The current thread missed the chance to decrement and must await.

          (sb-thread:condition-wait (sbcl-semaphore-cv semaphore)
                                     (sbcl-semaphore-lock semaphore)))
          (setq supposedly-did-not-receive-signal-p nil))
        (if supposedly-did-not-receive-signal-p

; The current thread may have received the signal but been unable to
; record it. In this case, the current thread will signal the
; condition variable again, so that any other thread waiting on the
; semaphore can have a chance at acquiring the said semaphore.

```

```

        (sb-thread:condition-notify
         (sbcl-semaphore-cv semaphore)
         (sbcl-semaphore-lock semaphore))

; The current thread was able to record the reception of the signal.
; The current thread will decrement the count of the semaphore and
; set the semaphore-notification-object.

        (progn
         (decf (sbcl-semaphore-count semaphore))
         (when semaphore-notification-object
          (set-semaphore-notification-status
           semaphore-notification-object))))))

#-acl2-par
t) ; default is to receive a semaphore/lock

#-acl2-loop-only
(defun timed-wait-on-semaphore (semaphore length-in-seconds)

; It would be possible to manually implement a
; timed-wait-on-semaphore in SBCL, but it requires an extra thread
; every time a current thread waits. This extra thread could be
; given a pointer to the current thread, sleep for 60 seconds, and
; throw the thread if the current thread hadn't set a variable in a
; shared array. This variable would be set whenever the current
; thread had successfully received the signal to wake up. The
; problem with this idea is that it requires double the number of
; threads. As a result, we leave the spawned worker threads to
; forever exist in SBCL, with the hope that SBCL will one day
; implement a timed-condition-wait.

; We do not simply reduce timed-wait-on-semaphore to
; wait-on-semaphore outside of OpenMCL, because the timeout version
; has different behavior than the non-timeout version. We want the
; hard run-time error here.

#+openmcl
(ccl:timed-wait-on-semaphore semaphore length-in-seconds)
#-openmcl
(error "Timed waiting not supported outside openmcl" nil))

#-acl2-loop-only
(defun make-semaphore-notification ()

; This function returns an object that records when a corresponding

```

```

; semaphore has been signaled (for use when wait-on-semaphore is
; called with that semaphore and that object).

#+(and openmcl acl2-par)
(ccl:make-semaphore-notification)
#+(and sbcl sb-thread acl2-par)
(make-array 1 :initial-element nil)
#-acl2-par
nil)

#-acl2-loop-only
(defun semaphore-notification-status (semaphore-notification-object)
  (declare (ignorable semaphore-notification-object))
  #+(and openmcl acl2-par)
  (ccl:semaphore-notification-status semaphore-notification-object)
  #+(and sbcl sb-thread acl2-par)
  (aref semaphore-notification-object 0)
  #-acl2-par)

; t may be the wrong default, but we don't have a use case for this
; return value yet, so we postpone thinking about the "right" value
; until we are aware of a need.
t)

#-acl2-loop-only
(defun clear-semaphore-notification-status
  (semaphore-notification-object)
  (declare (ignorable semaphore-notification-object))
  #+(and openmcl acl2-par)
  (ccl:clear-semaphore-notification-status
   semaphore-notification-object)
  #+(and sbcl sb-thread acl2-par)
  (setf (aref semaphore-notification-object 0) nil)
  #-acl2-par
  nil)

; We implement this only for SBCL, because even a system-level
; programmer is not expected to use this function. We use it only
; from within the threading interface to implement wait-on-semaphore
; for SBCL.

#-acl2-loop-only
(defun set-semaphore-notification-status
  (semaphore-notification-object)
  (declare (ignorable semaphore-notification-object))
  #+(and sbcl sb-thread)

```

```

(setf (aref semaphore-notification-object 0) t)
#-(and sbcl sb-thread)
(error "Set-semaphore-notification-status not supported outside ~
      SBCL"
      nil))

; Essay on Condition Variables

; A condition variable is a data structure that can be passed to
; corresponding "wait" and "signal" functions. When a thread calls
; the wait function on a condition variable, c, the thread blocks
; until "receiving a signal" from the application of the signal
; function to c. Only one signal is sent per call of the signal
; function; so, at most one thread will unblock. (There is a third
; notion for condition variable, namely the broadcast function,
; which is like the signal function except that all threads blocking
; on the given condition variable will unblock. But we do not
; support broadcast functions in this interface, in part because we
; use semaphores for OpenMCL, and there's no way to broadcast when
; you're really using a semaphore.)

; The design of our parallelism library is simpler when using
; condition variables for the following reason: Since a worker must
; wait for two conditions before consuming work, it is better to use
; a condition variable and test those two conditions upon waking,
; rather than try and use two semaphores.

; Implementation Note: As of March 2007, our OpenMCL implementation
; does not yield true condition variables. A condition variable
; degrades to a semaphore, so if one thread first signals a
; condition variable, then that signal has been stored. Then later
; (perhaps much later), when another thread waits for that signal,
; that thread will be able to proceed by decrementing the count. As
; a result the later thread will "receive" the signal, even though
; that signal occurred in the past. Fortunately, this isn't a
; contradiction of the semantics of condition variables, since with
; condition variables there is no specification of how far into the
; future the waiting thread will receive a signal from the
; signalling thread.

; Note: Condition variables should not be used to store state. They
; are only a signaling mechanism, and any state update implied by
; receiving a condition variable's signal should be checked. This
; usage is believed to be consistent with traditional condition
; variable semantics.

```

```

#-acl2-loop-only
(defun make-condition-variable ()

; If OpenMCL implements condition variables, we will want to change
; the OpenMCL expansion and remove the implementation note above.

#+(and openmcl acl2-par)
(ccl:make-semaphore)

#+(and sbcl sb-thread acl2-par)
(sb-thread:make-waitqueue)

#-acl2-par

; We may wish to have assertions that evaluation of
; (make-condition-variable) is non-nil. So we return t, even though
; as of this writing there are no such assertions.

t)

#-acl2-loop-only
(defmacro signal-condition-variable (cv)
  #-acl2-par
  (declare (ignore cv))

  #+(and openmcl acl2-par)
  '(ccl:signal-semaphore ,cv)

  #+(and sbcl sb-thread acl2-par)

; According to an email sent by Gabor Melis, of SBCL help, on
; 2007-02-25, if there are two threads waiting on a condition
; variable, and a third thread signals the condition variable twice
; before either can receive the signal, then both threads should
; receive the signal. If only one thread unblocks, it is considered
; a bug.

  '(sb-thread:condition-notify ,cv)

  #-acl2-par
  t)

#-acl2-loop-only
(defun wait-on-condition-variable-lockless (cv &optional s)

; Wait-on-condition-variable-lockless takes a required condition

```

```

; variable and an optional amount of timeout value. Since
; timeout-bound waiting is unsupported in SBCL, an error occurs when
; the user tries to include a timeout value.

; Here, s is the number of seconds allowed to elapse before
; unblocking occurs (in essence a timeout). This function returns t
; if we acquire the semaphore (i.e., we don't time out), and
; otherwise returns nil.

#-acl2-par
(declare (ignore cv s))

#+(and openmcl acl2-par)
(if s
    (ccl:timed-wait-on-semaphore cv s)
    (ccl:wait-on-semaphore cv))

#+(and sbcl sb-thread acl2-par)
(if s
    (error "Timed waiting on condition variables unsupported ~
           in SBCL")

; Since we do not want lock notification to be a bottleneck, we
; create a new lock each time we wait. This lock could be defined
; as a part of an SBCL condition variable structure, but we've
; done the inefficient thing for now.

    (let ((lock (make-lock)))
        (with-lock lock (sb-thread:condition-wait cv lock)
            t))

#-acl2-par
nil) ; the default is to never receive a signal

; End of threading interface

;-----
; Section: Parallelism Basis

; In this section we outline definitions and strategies for parallel
; evaluation and define constants, structures, variables, and other
; basic parallelism infrastructure.

; Essay on Parallelism Definitions

; Core

```

```

;
; A core is a unit inside a computer that can do useful work. It
; has its own instruction pointer and usually accesses shared
; memory. In the old days, we had "dual processors." This is an
; example of a two core system. A 2006-vintage example of a four
; core system is "dual sockets" with "dual core technology."

; Process
;
; We generally use the term "process" as a verb, meaning: run a set
; of instructions. For example, the system can process a closure.

; Thread
;
; We use the OS definition of a thread as a lightweight process that
; shares memory with other threads in the same process. A thread in
; our system is in one of the following three states.
;
; 1. Idle - The thread is waiting until both a piece of work (see
; below) and a core are available.
;
; 2. Active - The thread has been allocated a core and is
; processing some work.
;
; 3. Pending - This state occurs iff the thread in this state is
; associated with a parent piece of work, and it is waiting for
; the children of that piece of work to complete and for
; sufficient CPU core resources. A thread in this state is often
; waiting on a signaling mechanism.

; Closure
;
; We use the term "closure" in the Lisp sense: a function that
; captures values of variables from the lexical environment in which
; it is formed. A closure thus contains enough information to be
; applied to a list of arguments. We create closures in the process
; of saving work to be performed.

; Work
;
; A piece of work contains all the data necessary for a worker
; thread to process one closure, save its result somewhere that a
; parent can read it, and communicate that it is finished. It also
; contains some data necessary to implement features like the early
; termination of parallel and/or. Comments at parallelism-piece
; give implementation details.

```

```

; Roughly, work can be in any of four states: unassigned, starting,
; pending, or resumed. A piece of work will be processed by a single
; worker thread (not including, of course, child work, which will be
; processed by other worker threads). When a core becomes available,
; a thread can grab an unassigned piece of work, at which time the
; thread and the piece of work leave their initial states together.
; From that point forward until the piece of work is complete, the
; piece of work and its associated worker thread are considered to be
; in corresponding states (active/started,resumed or pending).
; Initially they are in their active/started states. Later, if child
; work is created, then at that time the thread and its associated
; piece of work both enter the pending state. When all child work
; terminates and either a CPU core becomes available or a heuristic
; allows an exception to that requirement, the piece of work enters
; the resumed state and its associated worker thread re-enters the
; active state. This heuristic (implemented in
; wait-for-resumptive-parallelism-resources) gives priority to such
; resumptions over starting new pieces of work.

; Parallelism Primitive
;
; A macro that enables the user to introduce parallelism into a
; computation: one of plet, pargs, pand, and por.

; End of Essay on Parallelism Definitions

; Essay on Parallelism Strategy

; Whenever a parallelism primitive is used, the following steps
; occur. The text between the < and > describes the state after the
; previous step finishes.

; 1. If there is a granularity form, the form is evaluated. If the
;    form returns nil, the parallelism primitive expands to the
;    serial equivalent; otherwise we continue.

; < granularity form has returned true or was omitted - the system
; was given a "large" amount of work >

; 2. If we heuristically determine that the system is already
;    overwhelmed with too much work (see
;    parallelism-resources-available for details), then the
;    primitive expands to its serial equivalent; otherwise we
;    continue.

```

```

; 3. Create closures for each primitive's arguments, as follows.
;   - Plet: one closure for each form assigned to a bound variable
;   - Pargs: one closure for each argument to the function call
;   - Pand/Por: one closure for each argument
; < have closures in memory representing computation to parallelize >

; 4. Create the data structures for pieces of work that worker
; threads are to process. One such data structure (documented
; in *work-queue* below) is created for each computation to be
; spawned. Among the fields of each such structure is a closure
; that represents that computation. Siblings have data
; structures that share some fields, such as a result-array that
; is to contain the values returned by the sibling computations.
;
; The system then adds these pieces of work to the global
; *work-queue* for worker threads to pop off the queue and
; process.
;
; Note that Step 2 avoids creating undesirably many pieces of
; work. (Actually the heuristics used in Step 2 don't provide
; exact guarantees, since two computations that reach Step 2
; simultaneously might both receive the go-ahead even though
; together, they create work that exceeds the heuristic work
; limit).

; < now have unassigned work in the work-queue >

; 5. After the parent thread adds the work to the queue, it will
; check to see if more worker threads are needed and spawn them
; if necessary. Note however that if there are more threads
; than cores, then any newly spawned thread will wait on a
; semaphore, and only begins evaluating the work when a core
; becomes available. Each core is assigned to at most one
; thread at any time (but if this decision is revisited, then it
; should be documented here and in the Parallelism Variables
; section. Note that this decision is implemented by setting
; *idle-core-count* to (1- *core-count*) in
; reset-parallelism-variables).

; Note that by limiting the amount of work in the system at Step
; 2, we avoid creating more threads than the system can handle.

; < now have enough worker threads to process the work >

```

```

; 6. The parent thread waits for its children to signal their
; completion. It is crucial for efficiency that this waiting be
; implemented using a signaling mechanism rather than as busy
; waiting.

; < the parent is waiting for the worker threads to process the
; work >

; 7. At this point, the child threads begin processing the work on
; the queue. As they are allocated resources, they each pull
; off a piece of work from the work queue and save their results
; in the associated result-array. After a child thread finishes
; a piece of work, it will check to see if its siblings'
; computations are still necessary. If not, the child will
; remove these computations from the work queue and interrupt
; each of its running sibling threads with a primitive that
; supplies a function for that thread to execute. This function
; throws to the tag :result-no-longer-needed, causing the
; interrupted sibling to abort evaluation of that piece of work,
; signal the parent (in an unwind-protect's cleanup form on the
; way out to the catch), catch that tag, and finally reenter the
; stalled state (where the controlling loop will find it
; something new to do). We take care to guarantee that this
; mechanism works even if a child receives more than one
; interrupt. Note that when a child is interrupted in this
; manner, the value stored for the child is a don't-care.

; < all of the children are done computing, the required results are
; in the results-array, and the parent has been signaled a number
; of times equal to the number of children >

; 8. The parent thread (from steps 1-6) resumes. It finds the
; results stored in its results array. If the primitive is a:
; - Plet: it executes the body of the plet with the calculated
;       bindings
; - Pargs: it applies the called function to the calculated
;       arguments
; - Pand, Por: it applies a functionalized "and" or "or" to the
;       calculated arguments. The result is Booleanized.

; End of Essay on Parallelism Strategy

; Parallelism Constants

#-acl2-loop-only
(progn

```

```

(defun core-count-raw ()
  #+openmcl (ccl:cpu-count)
  #-openmcl

; If the host Lisp does not provide a means for obtaining the number
; of cores, then we simply estimate on the high side. A high
; estimate is desired in order to make it unlikely that we have
; needlessly idle cores. We thus believe that 8 cores is a
; reasonable estimate for early 2007; but we may well want to
; increase this number later.

  8)

; *Core-count* is the total number of cores in the system.

(defvar *core-count*
  (core-count-raw))

(defvar *unassigned-and-active-work-count-limit*

; The *unassigned-and-active-work-count-limit* limits work on the
; *work-queue* to what we think the system will be able to process
; in a reasonable amount of time. Suppose we have 8 CPU cores.
; This means that there can be 8 active work consumers, and that
; generally not many more than 24 pieces of parallelism work are
; stored in the *work-queue* to be processed. This provides us the
; guarantee that if all worker threads were immediately to finish
; their piece of parallelism work, that each of them would
; immediately be able to grab another piece from the work queue.

; We could increase the following coefficient from 4 and further
; guarantee that consumers have parallelism work to process, but
; this would come at the expense of backlogging the *work-queue".
; We prefer simply to avoid the otherwise parallelized computations
; in favor of their serial equivalents.

  (* 4 *core-count*))

(defvar *total-work-limit* ; unassigned, started, resumed AND pending

; The number of pieces of work in the system,
; *parallelism-work-count*, must be less than *total-work-limit* in
; order to enable creation of new pieces of work. (However, we
; could go from 49 to 69 pieces of work when encountering a pand;
; just not from 50 to 52.)

```

```

; Why limit the amount of work in the system? :Doc
; parallelism-how-to (subtopic "Another Granularity Issue Related to
; Thread Limitations") provides an example showing how cdr recursion
; can rapidly create threads. That example shows that if there is
; no limit on the amount of work we may create, then eventually,
; many successive cdrs starting at the top will correspond to
; waiting threads. If we do not limit the amount of work that can
; be created, this can exhaust the supply of Lisp threads available
; to process the elements of the list.

```

```

(let ((val

```

```

; Warning: It is possible, in principle to create (+ val
; *max-idle-thread-count*) threads. Presumably you'll get a hard
; Lisp error (or seg fault!) if your Lisp cannot create that many
; threads.

```

```

    50)
    (bound (* 2 *core-count*))
  (when (< val bound)
    (error "The variable *total-work-limit* needs to be at ~
          least ~s, i.e., ~%~
          at least double the *core-count*. Please redefine ~%~
          *total-work-limit* so that it is not ~s."
          bound
          val))
  val))

```

```

; We don't want to spawn more worker threads (which are initially
; idle) when we already have sufficiently many idle worker threads.
; We use *max-idle-thread-count* to limit this spawning in function
; spawn-worker-threads-if-needed.

```

```

(defvar *max-idle-thread-count* (* 2 *core-count*))

```

```

; *initial-threads* stores a list of threads that are considered to
; be part of the non-threaded part of ACL2. When terminating
; parallelism threads, only those not appearing in this list will be
; terminated. Warning: If ACL2 uses parallelism during the build
; process, this variable could incorrectly record parallelism
; threads as initial threads.

```

```

(defvar *initial-threads* (all-threads))

```

```

) ; end of constant list

```

```

; Parallelism Structures

; If the shape of parallelism-piece changes, update the *work-queue*
; documentation in the section "Parallelism Variables."

#-acl2-loop-only
(defstruct parallelism-piece ; piece of work

; A data item in the work queue has the following contents, and we
; often call each a "piece of work."

; thread-array - the array that holds the threads spawned for that
; closure's particular parent

; result-array - the array that holds the results for that closure's
; particular parent, where each value is either nil (no result yet)
; or a cons whose cdr is the result

; array-index - the index into the above two arrays for this
; particular closure

; semaphore-to-signal-as-last-act - the semaphore to signal right
; before the spawned thread dies

; closure - the closure to process by spawning a thread

; throw-siblings-when-function - the function to funcall on the
; current thread's result to see if its siblings should be
; terminated. The function will also remove work from the
; work-queue and throw the siblings if termination should occur.

(thread-array nil)
(result-array nil)
(array-index -1)
(semaphore-to-signal-as-last-act nil)
(closure nil)
(throw-siblings-when-function nil))

; Parallelism Variables

#-acl2-loop-only
(progn

; Keep this progn in sync with reset-parallelism-variables, which

```

```
; resets the variables defined here. Note that none of the
; variables are initialized here, so reset-parallelism-variables
; must be called before evaluating parallelism primitives (an
; exception is *throwable-worker-thread* since it is first called in
; reset-parallelism-variables).
```

```
; *Idle-thread-count* is updated both when a thread is created and
; right before it expires. It is also updated when a worker thread
; gets some work to do and after it is done with that work.
```

```
(defvar *idle-thread-count*)
(deflock *idle-thread-count-lock*)
```

```
; *Idle-core-count* is only used to estimate resource availability.
; The number itself is always kept accurate using a lock, but
; because we read it without a lock when calculating resource
; availability, the value actually read is only an estimate. It
; defaults to (1- *core-count*), because the current thread is
; considered active.
```

```
; There are two pairs of places that *idle-core-count* is updated.
; First, whenever a worker thread begins processing work, the count
; is decremented. This decrement is paired with the increment that
; occurs after a worker thread finishes work. It is also
; incremented and decremented in eval-and-save-result, before and
; after a parent waits for its children.
```

```
; Note: At different stages of development we have contemplated
; having a "*virtual-core-count*", exceeding the number of CPU
; cores, that bounds the number of active threads. Since our
; initial tests did not show a performance improvement by using this
; trick, we have not employed a *virtual-core-count*. If we later
; do employ this trick, the documentation in step 5 of the Essay on
; Parallelism Strategy will need to be updated.
```

```
(defvar *idle-core-count*)
(deflock *idle-core-count-lock*)
```

```
; *Unassigned-and-active-work-count* tracks the amount of
; parallelism work in the system, other than pending work. It is
; increased when a parallelism primitive adds work to the system.
; This increase is paired with the final decrease in
; consume-work-on-work-queue-when-its-there, which occurs when a
; piece of work finishes. It is decremented and incremented
; (respectively) when a parent waits on children and when it resumes
; after waiting on children.
```

```

(defvar *unassigned-and-active-work-count*)
(deflock *unassigned-and-active-work-count-lock*)

; *Total-work-count* tracks the total amount of parallelism work.
; This includes unassigned, started, pending, and resumed work.

(defvar *total-work-count*)
(deflock *total-work-count-lock*)

; We maintain a queue of work to process. See parallelism-piece for
; documentation on pieces of work. Even though *work-queue* is a
; list, we think of it as a structure that can be destructively
; modified -- so beware sharing any structure with *work-queue*!

(defvar *work-queue*)
(deflock *work-queue-lock*)

; An idle thread waits for the condition variable
; *check-work-and-core-availability-cv* to be signaled, at which
; time it looks for work on the *work-queue* and an idle core to
; use. This condition can be signaled by the addition of new work
; or by the availability of a CPU core.

; Warning: In the former case, a parent thread must always signal
; this semaphore *after* it has already added the work to the queue.
; Otherwise, a child can attempt to acquire work, fail, and then go
; wait on the semaphore again. Since the parent has already
; signaled, there is no guarantee that the work they place on the
; queue will ever be processed. (The latter case also requires
; analogous care.)

; Why are there two condition variables, one for idle threads and
; one for resuming threads? Suppose that idle and resuming threads
; waited on the same condition variable. We would then have no
; guarantee that resuming threads would be signaled before the idle
; threads (which is necessary to establish the priority explained in
; wait-for-resumptive-parallelism-resources). Using separate
; condition variables allows both an idle and resuming thread to be
; signaled. Then whichever thread's heuristics allow it to execute
; will claim access to the CPU core. There is no problem if both
; their heuristics allow them to continue.

(defvar *check-work-and-core-availability-cv*)
(defvar *check-core-availability-for-resuming-cv*)

```

```

; When we terminate threads due to a break and abort, we need a way
; to terminate all threads. We implement this by having them throw
; the :worker-thread-no-longer-needed tag. Unfortunately, sometimes
; the threads are outside the scope of the associated catch, when
; throwing the tag would cause a warning. We avoid this warning by
; maintaining the dynamically-bound variable
; *throwable-worker-thread*. When the throwable context is entered,
; we let a new copy of the variable into existence and set it to T.
; Now, when we throw :worker-thread-no-longer-needed, we only throw
; it if *throwable-worker-thread* is non-nil.

```

```

(defvar *throwable-worker-thread* nil)

```

```

; *total-parallelism-piece-historical-count* tracks the total number
; of pieces of parallelism work processed over the lifetime of the
; ACL2 session. It is reset whenever the parallelism variables
; are reset. It is only used for informational purposes, and the
; system does not depend on its accuracy in any way. It therefore
; does not have an associated lock.

```

```

(defvar *total-parallelism-piece-historical-count*)

```

```

) ; end of parallelism variables

```

```

; Following are definitions of functions that help us restore the
; parallelism system to a stable state after an interrupt occurs.

```

```

#-acl2-loop-only

```

```

(defun throw-all-threads-in-list (thread-list)

```

```

; We interrupt each of the given threads with a throw to the catch
; at the top of consume-work-on-work-queue-when-its-there, which is
; the function called by run-thread in
; spawn-worker-threads-if-needed.

```

```

; Compare with kill-all-threads-in-list, which kills all of the
; given threads (typically all user-produced threads), not just
; those self-identified as being within the associated catch block.

```

```

(if (endp thread-list)
    nil
    (progn
      (interrupt-thread
       (car thread-list)
       #'(lambda () (when *throwable-worker-thread*
                      (throw :worker-thread-no-longer-needed nil))))))

```

```

        (throw-all-threads-in-list (cdr thread-list))))))

#-acl2-loop-only
(defun kill-all-threads-in-list (thread-list)

; Compare with throw-all-threads-in-list, which uses throw instead
; of killing threads directly, but only affects threads self
; identified as being within an associated catch block.

  (if (endp thread-list)
      nil
      (progn
        (kill-thread (car thread-list))
        (kill-all-threads-in-list (cdr thread-list))))))

#-acl2-loop-only
(defun all-threads-except-initial-threads-are-dead ()
  #+sbcl
  (<= (length (all-threads)) 1)
  #-sbcl
  (null (set-difference (all-threads) *initial-threads*)))

#-acl2-loop-only
(defun send-die-to-all-except-initial-threads ()

; This function is evaluated only for side effect.

  (let ((target-threads #+sbcl
                        (cdr (all-threads))
                        #-sbcl
                        (set-difference (all-threads)
                                       *initial-threads*)))

    #+acl2-par
    (throw-all-threads-in-list target-threads)
    #-acl2-par nil)
  (thread-wait 'all-threads-except-initial-threads-are-dead))

#-acl2-loop-only
(defun kill-all-except-initial-threads ()

; This function is evaluated only for side effect.

  (let ((target-threads #+sbcl
                        (cdr (all-threads))
                        #-sbcl

```

```

                (set-difference (all-threads)
                               *initial-threads*))
      (kill-all-threads-in-list target-threads))
      (thread-wait 'all-threads-except-initial-threads-are-dead))

#-acl2-loop-only
(defun reset-parallelism-variables ()

; We use this function (a) to kill all worker threads, (b) to reset
; "most" of the parallelism variables, and (c) to reset the lock and
; semaphore recycling systems. Keep (b) in sync with the progn
; above that declares the variables reset here, in the sense that
; this function assigns values to exactly those variables.

; If a user kills threads directly from raw Lisp, for example using
; functions above, then they should call
; reset-parallelism-variables. Note that
; reset-parallelism-variables is called automatically when an ACL2
; user interrupts with control-c and then aborts to get back to the
; ACL2 top-level loop.

; (a) Kill all worker threads.

      (send-die-to-all-except-initial-threads)

; (b) Reset "most" of the parallelism variables.

      (setf *idle-thread-count* 0)
      (reset-lock *idle-thread-count-lock*)

      (setf *idle-core-count* (1- *core-count*))
      (reset-lock *idle-core-count-lock*)

      (setf *unassigned-and-active-work-count* 1)
      (reset-lock *unassigned-and-active-work-count-lock*)

      (setf *total-work-count* 1)
      (reset-lock *total-work-count-lock*)

      (setf *work-queue* nil)
      (reset-lock *work-queue-lock*)

      (setf *check-work-and-core-availability-cv*
            (make-condition-variable))
      (setf *check-core-availability-for-resuming-cv*
            (make-condition-variable))

```

```

(setf *throwable-worker-thread* nil)

(setf *total-parallelism-piece-historical-count* 0)

(setf *initial-threads* (all-threads))

; (c) Reset the lock and semaphore recycling system.

(setf *lock-allocation-lock* (make-lock))
(reset-lock-free-list)

(setf *semaphore-allocation-lock* (make-lock))
(reset-semaphore-free-list)

)

; We reset parallelism variables as a standard part of compilation
; to make sure the code behind declaring variables and resetting
; variables is consistent and that we are in a stable state.

#-acl2-loop-only
(reset-parallelism-variables)

;-----
; Section:  Work Consumer Code

; We develop functions that assign threads to process work.

#-acl2-loop-only
(defun eval-and-save-result (work)

; Work is a piece of parallelism work.  Among its fields are a
; closure and an array.  We evaluate this closure and save the
; result into this array.  No lock is required because no other
; thread will be writing to the same position in the array.

; Keep this in sync with the comment in parallelism-piece, where we
; explain that the result is the cdr of the cons stored in the
; result array at the appropriate position.

(assert work)
(let ((result-array (parallelism-piece-result-array work))
      (array-index (parallelism-piece-array-index work))
      (closure (parallelism-piece-closure work)))
  (setf (aref result-array array-index)

```

```

        (cons t (funcall closure))))))

#-acl2-loop-only
(defun pop-work-and-set-thread ()

; Once we exit the without-interrupts that must enclose a call to
; pop-work-and-set-thread, our siblings can interrupt us so that we
; execute a throw to the tag :result-no-longer-needed. The reason
; they can access us is that they will have a pointer to us in the
; thread array.

; There is a race condition between when work is popped from the
; *work-queue* and when the current thread is stored in the
; thread-array. This race condition could be eliminated by holding
; *work-queue-lock* during the function's entire execution. Since
; (1) we want to minimize the duration locks are held, (2) the
; chance of this race condition occurring is small and (3) there is
; no safety penalty when this race condition occurs (instead an
; opportunity for early termination is missed), we only hold the
; lock for the amount of time it takes to ready and modify the
; *work-queue*

  (let ((work (with-lock *work-queue-lock*
                     (when (consp *work-queue*)
                         (pop *work-queue*))))
        (thread (current-thread)))
    (when work
      (assert thread)
      (assert (parallelism-piece-thread-array work))

; Record that the current thread is the one assigned to do this
; piece of work:

      (setf (aref (parallelism-piece-thread-array work)
                  (parallelism-piece-array-index work))
            thread))
    work))

#-acl2-loop-only
(defun consume-work-on-work-queue-when-its-there ()

; This function is an infinite loop. However, the thread running it
; can be waiting on a condition variable and will expire if it waits
; too long.

; Each iteration through the main loop will start by trying to grab

```

```

; a piece of work to process. When it succeeds, then it will
; process that piece of work and wait again on a condition variable
; before starting the next iteration. But ideally, if it has to
; wait too long for a piece of work to grab then we return from this
; function (with expiration of the current thread); see below.

```

```

(catch :worker-thread-no-longer-needed
  (let* ((*throwable-worker-thread* t))
    (loop ; "forever" - really until
      ; :worker-thread-no-longer-needed thrown

```

```

; Wait until there are both a piece of work and an idle core. In
; openmcl, if the thread waits too long, it throws to the catch
; above and returns from this function.

```

```

(loop while (not (and *work-queue* (< 0 *idle-core-count*)))

```

```

; We can't grab work yet, so we wait until somebody signals us to
; try again, by returning a non-nil value to the call of not, just
; below. If however nobody signals us then ideally (and in OpenMCL
; but not SBCL) a timeout occurs that returns nil to this call of
; not, so we give up with a throw.

```

```

do (when (not #+(and openmcl acl2-par)
  (wait-on-condition-variable-lockless
    *check-work-and-core-availability-cv* 15)
  #+(and sbcl sb-thread acl2-par)
  (wait-on-condition-variable-lockless
    *check-work-and-core-availability-cv*)
  #-acl2-par
  t)
  (throw :worker-thread-no-longer-needed nil)))

```

```

; Now very likely there are both a piece of work and an idle core to
; process it. But a race condition allows either of these to
; disappear before we can claim a piece of work and a CPU core,
; which explains the use of 'when' below.

```

```

(unwind-protect-disable-interrupts-during-cleanup
  (when (<= 0 (with-lock *idle-core-count-lock* ; allocate CPU
    ; core

```

```

; We will do a corresponding increment of *idle-core-count* in the
; cleanup form of this unwind-protect. Note that the current thread
; cannot be interrupted (except by direct user intervention, for
; which we may provide only minimal protection) until the call of

```

```
; pop-work-and-set-thread below (see long comment above that call),
; because no other thread has a pointer to this one until that time.
```

```
                (decf *idle-core-count*))
  (catch :result-no-longer-needed
    (let ((work nil))
      (unwind-protect-disable-interrupts-during-cleanup
        (progn
          (without-interrupts
            (setq work
```

```
; The following call has the side effect of putting the current
; thread into a thread array, such that this presence allows the
; current thread to be interrupted by another (via interrupt-thread,
; in throw-threads-in-array). So until this point, the current
; thread will not be told to do a throw.
```

```
; We rely on the following claim: If any state has been changed by
; this call of pop-work-and-set-thread, then that call completes and
; work is set to a non-nil value. This claim guarantees that if any
; state has been changed, then the cleanup form just below will be
; executed and will clean up properly. For example, we would have a
; problem if pop-work-and-set-thread were interrupted after the
; decrement of *idle-thread-count*, but before work is set, since
; then the matching increment in the cleanup form below would be
; skipped. For another example, if we complete the call of
; pop-work-and-set-thread but not the enclosing setq for work, then
; we miss the semaphore signaling in the cleanup form below.
```

```
                (pop-work-and-set-thread))
      (when work (with-lock *idle-thread-count-lock*
                        (decf *idle-thread-count*)))
    (when work
```

```
; The consumer now has a core (see the <= test above) and a piece of
; work.
```

```
      (eval-and-save-result work)

      (let*
        ((thread-array
          (parallelism-piece-thread-array work))
         (result-array
          (parallelism-piece-result-array work))
         (array-index
          (parallelism-piece-array-index work))
```

```

        (throw-siblings-when-function
         (parallelism-piece-throw-siblings-when-function
          work)))
        (setf (aref thread-array array-index) nil)

; The nil setting just above guarantees that the current thread
; doesn't interrupt itself by way of the early termination function.

        (when throw-siblings-when-function
          (funcall throw-siblings-when-function
                   (aref result-array array-index))))))

        (when work ; process this cleanup form if we acquired
; work
        (let*
          ((semaphore-to-signal-as-last-act
            (parallelism-piece-semaphore-to-signal-as-last-act
             work))
           (thread-array
            (parallelism-piece-thread-array work))
           (array-index
            (parallelism-piece-array-index work)))
            (incf *total-parallelism-piece-historical-count*)
            (setf (aref thread-array array-index) nil)
            (with-lock *idle-thread-count-lock*

; Above we argued that if *idle-thread-count* is decremented, then
; work is set and hence we get to this point so that we can do the
; corresponding increment. In the other direction, if we get here,
; then how do we know that *idle-thread-count* was decremented? We
; know because if we get here, then work is non-nil and hence
; pop-work-and-set-thread must have completed.

                    (incf *idle-thread-count*)))

; Each of the following two decrements undoes the corresponding
; increment done when the piece of work was first created and
; queued.

        (with-lock *total-work-count-lock*
          (decf *total-work-count*))
        (with-lock
         *unassigned-and-active-work-count-lock*
         (decf *unassigned-and-active-work-count*))
        (assert
         (semaphorep semaphore-to-signal-as-last-act)))

```

```

        (signal-semaphore
          semaphore-to-signal-as-last-act))))))
      ) ; end catch :result-no-longer-needed
    ) ; end when CPU core allocation

    (with-lock *idle-core-count-lock*
      (incf *idle-core-count*))
    (signal-condition-variable
      *check-work-and-core-availability-cv*)
    (signal-condition-variable
      *check-core-availability-for-resuming-cv*))))

  ) ; end catch :worker-thread-no-longer-needed

; The current thread is about to expire because all it was given to
; do was to run this function.

  (with-lock *idle-thread-count-lock* (decf *idle-thread-count*))

#-acl2-loop-only
(defun spawn-worker-threads-if-needed ()

; This function must be called with interrupts disabled.  Otherwise
; it is possible for the *idle-thread-count* to be incremented even
; though no new worker thread is spawned.

  (loop while (< *idle-thread-count* *max-idle-thread-count*)

; Note that the above test could be true, yet *idle-thread-count*
; could be incremented before we get to the lock just below.  But we
; want as little bottleneck as possible for scaling later, and the
; practical worst consequence is that we spawn extra threads here.

; Another possibility is that we spawn too few threads here, because
; the final decrement of *idle-thread-count* in
; consume-work-on-work-queue-when-its-there has not occurred even
; though a worker thread has decided to expire.  If this occurs,
; then we may not have the expected allotment of idle threads for
; awhile, but we expect the other idle threads (if any) and the
; active threads to suffice.  Eventually a new parallelism primitive
; call will invoke this function again, at a time when the
; about-to-expire threads have already updated *idle-thread-count*,
; which will allow this function to create the expected number of
; threads.  The chance of any of this kind of issue arising is
; probably extremely small.

```

; NOTE: Consider coming up with a design that's easier to understand.

```
do
  (progn (with-lock *idle-thread-count-lock*
                (incf *idle-thread-count*))
         (run-thread
          "Worker thread"
          'consume-work-on-work-queue-when-its-there))))
```

; Section: Work Producer Code

; We develop functions that create work, to be later processed by
; threads. Our main concern is to keep the work queue sufficiently
; populated so as to keep CPU cores busy, while limiting the total
; amount of work so that the number of threads necessary to evaluate
; that work does not exceed the number of threads that the
; underlying Lisp supports creating. (See also comments in
; *total-work-limit*.)

```
#+acl2-loop-only
(defun add-work-list-to-queue (work-list)
```

; Call this function inside without-interrupts, in order to maintain
; the invariant that when this function exits, the counts are
; accurate.

; WARNING! This function destructively modifies *work-queue*.

```
(let ((work-list-length (length work-list)))
  (with-lock *work-queue-lock*
```

; Via naive performance tests using a parallel version of Fibonacci,
; we determine that (pfib 45) takes about 19.35 seconds when we
; (nconc *work-queue* work-list) (as opposed to 19.7 seconds when we
; reverse the argument order). We do not yet understand exactly why
; this is, but Halstead supports this in his 1989 paper "New Ideas
; in Parallel Lisp: Language Design, Implementation, and Programming
; Tools."

```
      (setf *work-queue*
            (nconc *work-queue* work-list)))
  (with-lock *total-work-count-lock*
    (incf *total-work-count* work-list-length))
  (with-lock *unassigned-and-active-work-count-lock*
    (incf *unassigned-and-active-work-count*
```

```

        work-list-length))
(dotimes (i work-list-length)
  (signal-condition-variable
   *check-work-and-core-availability-cv*)))

#-acl2-loop-only
(defun combine-array-results-into-list (result-array
  current-position acc)
  (if (< current-position 0)
      acc
      (combine-array-results-into-list
       result-array
       (1- current-position)
       (cons (cdr ; entry is a cons whose cdr is the result
              (aref result-array current-position))
              acc))))

#-acl2-loop-only
(defun remove-thread-array-from-work-queue-rec
  (work-queue thread-array array-positions-left)

; The function calling remove-thread-array-from-work-queue must hold
; the lock *work-queue-lock*.

; This function must be called with interrupts disabled.

  (cond ((equal array-positions-left 0)
         work-queue)
        ((atom work-queue)
         nil)
        ((equal thread-array
                 (parallelism-piece-thread-array (car work-queue)))
         (progn
          (with-lock *total-work-count-lock*
                (decf *total-work-count*))
          (with-lock *unassigned-and-active-work-count-lock*
                (decf *unassigned-and-active-work-count*)))

; we must signal the parent

          (assert
           (semaphorep
            (parallelism-piece-semaphore-to-signal-as-last-act
             (car work-queue))))
          (signal-semaphore
           (parallelism-piece-semaphore-to-signal-as-last-act

```

```

        (car work-queue)))
      (remove-thread-array-from-work-queue-rec
       (cdr work-queue)
       thread-array
       (1- array-positions-left))))
    (t (cons (car work-queue)
             (remove-thread-array-from-work-queue-rec
              (cdr work-queue)
              thread-array
              (1- array-positions-left))))))

#-acl2-loop-only
(defun remove-thread-array-from-work-queue (thread-array)
  (without-interrupts
   (with-lock *work-queue-lock*
    (setf *work-queue*
          (remove-thread-array-from-work-queue-rec
           *work-queue*
           thread-array
           (length thread-array))))))

#-acl2-loop-only
(defun terminate-siblings (thread-array)

; This function supports early termination by eliminating further
; computation by siblings. Siblings not yet assigned a thread are
; removed from the work queue. Siblings that are already active are
; interrupted to throw with tag :result-no-longer-needed. The order
; of these two operations is important: if we do them in the other
; order, then we could miss a sibling that is assigned a thread (and
; removed from the work queue) just inbetween the two operations.

  (remove-thread-array-from-work-queue thread-array)
  (throw-threads-in-array thread-array (1- (length thread-array))))

#-acl2-loop-only
(defun generate-work-list-from-closure-list-rec
  (thread-array result-array children-done-semaphore closure-list
   current-position &optional
   throw-siblings-when-function)
  (if (atom closure-list)
      (assert (equal current-position (length thread-array)))
      (cons
       (make-parallelism-piece
        :thread-array thread-array
        :result-array result-array

```

```

      :array-index current-position
      :semaphore-to-signal-as-last-act children-done-semaphore
      :closure (car closure-list)
      :throw-siblings-when-function throw-siblings-when-function)
(generate-work-list-from-closure-list-rec
 thread-array
 result-array
 children-done-semaphore
 (cdr closure-list)
 (1+ current-position)
 throw-siblings-when-function))))

#-acl2-loop-only
(defun generate-work-list-from-closure-list
  (closure-list &optional terminate-early-function)

; Given a list of closures, we need to generate a list of work data
; structures that are in a format ready for the work queue. Via mv,
; we also return the pointers to the thread, result, and semaphore
; arrays.

  (let* ((closure-count (length closure-list))
         (thread-array
          (make-array closure-count :initial-element nil))
         (result-array
          (make-array closure-count :initial-element nil))
         (children-done-semaphore (allocate-semaphore)))
    (progn ; warning: avoid prog2 as we need to return multiple value
           (assert (semaphorep children-done-semaphore))
           (mv (generate-work-list-from-closure-list-rec
                thread-array
                result-array
                children-done-semaphore
                closure-list
                0
                (if terminate-early-function
                    (lambda (x) ; x is (t . result)
                      (when (funcall terminate-early-function (cdr x))
                        (terminate-siblings thread-array)))
                    nil))
                thread-array
                result-array
                children-done-semaphore))))))

#-acl2-loop-only
(defun parallelism-resources-available ()

```

```

; This function is our attempt to guess when resources are
; available. When this function returns true, then resources are
; probably available, and a parallelism primitive call will opt to
; parallelize. We say "probably" because correctness does not
; depend on our answering exactly. For performance, we prefer that
; this function is reasonably close to an accurate implementation
; that would use locks. Perhaps even more important for
; performance, however, is that we avoid the cost of locks to try
; to remove bottlenecks.

; In summary, it is unnecessary to acquire a lock, because we just
; don't care if we miss a few chances to parallelize, or parallelize
; a few extra times.

  (and (f-get-global 'parallel-evaluation-enabled *the-live-state*)
        (< *unassigned-and-active-work-count*
            *unassigned-and-active-work-count-limit*)
        (< *total-work-count* *total-work-limit*)))

#-acl2-loop-only
(defun throw-threads-in-array (thread-array current-position)

; Call this function to terminate computation for every thread in
; the given thread-array from position current-position down to
; position 0. We expect that thread-array was either created by the
; current thread's parent or was created by the current thread (for
; its children).

; We require that the current thread not be in thread-array. This
; requirement prevents the current thread from interrupting itself,
; which could conceivably abort remaining recursive calls of this
; function, or cause a hang in some Lisps since we may be operating
; with interrupts disabled (for example, inside the cleanup form of
; an unwind-protect in OpenMCL 1.1pre or later).

  (assert thread-array)
  (when (<= 0 current-position)
    (let ((current-thread (aref thread-array current-position)))
      (when current-thread
        (interrupt-thread
         current-thread))))

; The delayed evaluation of (aref thread-array...) below is crucial
; to keep a thread from throwing :result-no-longer-needed outside of
; the catch for that tag. Consume-work-on-work-queue-when-its-there

```

```
; will set the (aref thread-array...) to nil when the thread should
; not be thrown.
```

```
(lambda ()
  (when (aref thread-array current-position)
    (throw :result-no-longer-needed nil))))
(throw-threads-in-array thread-array (1- current-position)))
```

```
#-acl2-loop-only
```

```
(defun decrement-children-left
  (children-left-ptr semaphore-notification-obj)
```

```
; This function should be called with interrupts disabled.
```

```
(when (semaphore-notification-status semaphore-notification-obj)
  (decf (aref children-left-ptr 0))
  (clear-semaphore-notification-status
   semaphore-notification-obj)))
```

```
#-acl2-loop-only
```

```
(defun wait-for-children-to-finish
  (semaphore children-left-ptr semaphore-notification-obj)
```

```
; This function is called both in the normal case and in the
; early-termination case.
```

```
(assert children-left-ptr)
(when (<= 1 (aref children-left-ptr 0))
  (assert (not (semaphore-notification-status
                semaphore-notification-obj)))
  (unwind-protect-disable-interrupts-during-cleanup
    (wait-on-semaphore semaphore
                       semaphore-notification-obj)
    (decrement-children-left children-left-ptr
                             semaphore-notification-obj))
  (wait-for-children-to-finish semaphore
                              children-left-ptr
                              semaphore-notification-obj)))
```

```
#-acl2-loop-only
```

```
(defun wait-for-resumptive-parallelism-resources ()
```

```
; A thread resuming execution after its children finish has a higher
; priority than a thread just beginning execution. As such,
; resuming threads are allowed to "borrow" up to *core-count* CPU
; cores. That is implemented by waiting until *idle-core-count* is
```

```

; greater than the negation of the *core-count*. This is different
; from a thread just beginning execution, which waits for
; *idle-core-count* to be greater than 0.

```

```

(loop while (<= *idle-core-count* (- *core-count*)))

```

```

; So, *idle-core-count* is running a deficit that is at least the
; number of cores: there are already *core-count* additional active
; threads beyond the normal limit of *core-count*.

```

```

      do (wait-on-condition-variable-lockless
          *check-core-availability-for-resuming-cv*)
    (with-lock *unassigned-and-active-work-count-lock*
      (incf *unassigned-and-active-work-count*))
    (with-lock *idle-core-count-lock*
      (decf *idle-core-count*)))

```

```

#-acl2-loop-only

```

```

(defun terminate-children-early-and-rewait
  (children-done-semaphore children-left-ptr
   semaphore-notification-obj
   thread-array)

```

```

; This function performs three kinds of actions.

```

```

; A. It signals children-done-semaphore once for each child that is
; unassigned (i.e. still on the work queue) and removes that child
; from the work queue.

```

```

; B. It interrupts each assigned child's thread with a throw that
; terminates processing of its work. Note that we must do Step B
; after Step A: otherwise threads might grab work after Step B but
; before Step A, resulting in child work that is no longer available
; to terminate unless we call this function again.

```

```

; C. The above throw from Step B eventually causes the interrupted
; threads to signal children-done-semaphore. The current thread
; waits for those remaining signals.

```

```

  (when (< 0 (aref children-left-ptr 0))
    (remove-thread-array-from-work-queue ; A

```

```

; Signal children-done-semaphore, which is in each piece of work in
; closure-list.

```

```

    thread-array)

```

```

    (throw-threads-in-array thread-array ; B
      (1- (length thread-array)))
    (wait-for-children-to-finish      ; C
      children-done-semaphore
      children-left-ptr
      semaphore-notification-obj)))

#+(and (not acl2-loop-only) acl2-par)
(defun prepare-to-wait-for-children ()

; This function should be executed with interrupts disabled, after
; all child work is added to the work queue but before the current
; thread waits on such work to finish.

; First, since we are about to enter the pending state, we must free
; CPU core resources and notify other threads.

  (with-lock *idle-core-count-lock*
    (incf *idle-core-count*))
  (signal-condition-variable *check-work-and-core-availability-cv*)
  (signal-condition-variable
    *check-core-availability-for-resuming-cv*))

; Second, record that we are no longer active. (Note: We could
; avoid the following form (thus saving a lock) by incrementing
; *unassigned-and-active-work-count* by one less in
; add-work-list-to-queue.)

  (with-lock *unassigned-and-active-work-count-lock*
    (decf *unassigned-and-active-work-count*)))

#+(and (not acl2-loop-only) acl2-par)
(defun parallelize-closure-list (closure-list
                                &optional terminate-early-function)

; Given a list of closures, we:

; 1. Create a list of pieces of work (see defstruct
; parallelism-piece).

; 2. If there aren't enough idle worker threads, we spawn a
; reasonably sufficient number of new worker threads, so that CPU
; cores are kept busy but without the needless overhead of useless
; threads. Note that when a thread isn't already assigned work, it
; is waiting for notification to look for work to do.

```

```

; 3. Add the work to the work queue, which notifies the worker
; threads of the additional work.

; 4. Free parallelism resources (specifically, a CPU core), since we
; are about to become idle as we wait on our children to finish.
; Issue the proper notifications (via condition variables) so that
; other threads are aware of the freed resources.

; 5. Wait for the children to finish. In the event of receiving an
; early termination from our parent (a.k.a. the grandparent of our
; children) or our sibling (a.k.a. the uncle of our children), we
; signal our children to terminate early, and we wait again.

; Note that if the current thread's children decide the remaining
; child results are irrelevant, that the current thread will never
; know it. The children will terminate early amongst themselves
; without any parent intervention.

; 6. Resume when resources become available, reclaiming parallelism
; resources (see wait-for-resumptive-parallelism-resources).

; 7. Return the result.

; It's silly to parallelize just 1 (or no!) thing. The definitions
; of pargs, plet, pand, and por should prevent this assertion from
; failing, but we have it here as a check that this is true.

(assert (and (consp closure-list) (cdr closure-list)))

(let ((work-list-setup-p nil)
      (semaphore-notification-obj (make-semaphore-notification))
      (children-left-ptr (make-array 1 :initial-element
                                     (length closure-list))))

; 1. Create a list of pieces of work.
(mv-let
 (work-list thread-array result-array children-done-semaphore)
 (generate-work-list-from-closure-list closure-list
                                       terminate-early-function)
 (assert (semaphorep children-done-semaphore))
 (unwind-protect-disable-interrupts-during-cleanup
  (progn
   (without-interrupts

; 2. Spawn worker threads so that CPU cores are kept busy.
(spawn-worker-threads-if-needed)

```

```

; 3. Add the work to the work queue.
      (setq work-list-setup-p
            (progn (add-work-list-to-queue work-list) t))

; 4. Free parallelism resources.
      (prepare-to-wait-for-children))

; 5a. Wait for children to finish. But note that we may be
; interrupted by our sibling or our parent before this wait is
; completed.

; Now that the two operations under the above without-interrupts are
; complete, it is once again OK to be interrupted with a function
; that throws to the tag :results-no-longer-needed. Note that
; wait-for-children-to-finish is called again in the cleanup form
; below, so we don't have to worry about dangling child threads even
; if we don't complete evaluation of the following form.

```

```

      (wait-for-children-to-finish children-done-semaphore
                                   children-left-ptr
                                   semaphore-notification-obj))

```

```

; We are entering the cleanup form, which we always need to run (in
; particular, so that we can resume and return a result). But why
; must we run without interrupts? Suppose for example we have been
; interrupted (to do a throw) by the terminate-early-function of one
; of our siblings or by our parent. Then we must wait for all child
; pieces of work to terminate (see
; terminate-children-early-and-rewait) before we return. And this
; waiting must be non-abortable; otherwise, for example, we could be
; left (after Control-c and an abort) with orphaned child threads.

```

```

      (progn
        (when work-list-setup-p ; children were added to *work-queue*

```

```

; If we were thrown by a sibling or parent, it's possible that our
; children didn't finish. We now throw our children and wait for
; them.

```

```

; 5b. Complete processing of our children in case we were
; interrupted when we were waiting the first time.

```

```

      (terminate-children-early-and-rewait
        children-done-semaphore
        children-left-ptr
        semaphore-notification-obj)

```

```

        thread-array)

; AS OF *HERE*, ALL OF THIS PARENT'S CHILD WORK IS "DONE"

; 6. Resume when resources become available.
      (wait-for-resumptive-parallelism-resources)
      (assert (eq (aref children-left-ptr 0) 0))

; Free semaphore for semaphore recycling.
      (free-semaphore children-done-semaphore))))

; 7. Return the result.
      (combine-array-results-into-list
        result-array
        (1- (length result-array))
        nil))))

(defun and-list (lst)
  (declare (xargs :guard (true-listp lst)))
  (if (endp lst)
      t
      (and (car lst)
            (and-list (cdr lst)))))

(defun or-list (lst)
  (declare (xargs :guard (true-listp lst)))
  (if (endp lst)
      nil
      (if (car lst)
          t
          (or-list (cdr lst)))))

; Parallelize-fn Booleanizes the results from pand/por.

#+(and (not acl2-loop-only) acl2-par)
(defun parallelize-fn (parent-fun-name arg-closures &optional
                      terminate-early-function)

; It's silly to parallelize just 1 (or no!) thing. The definitions
; of pargs, plet, pand, and por should prevent this assertion from
; failing, but we have it here as a check that this is true.
  (assert (cdr arg-closures))
  (let ((parallelize-closures-res
        (parallelize-closure-list arg-closures
                                   terminate-early-function))))

```

```

      (if (or (equal parent-fun-name 'and-list)
              (equal parent-fun-name 'or-list))
          (funcall parent-fun-name parallelize-closures-res)
          (apply parent-fun-name parallelize-closures-res))))

;-----
; Section: Parallelism Primitives

(defdoc parallelism
  ":Doc-Section Parallelism

  executing forms in parallel~/

  One of ACL2's strengths lies in its ability to execute industrial
  models efficiently. ACL2 provides the capability to execute models
  in parallel. This capability can increase the speed of explicit
  calls to simulators written in ACL2, and it can also decrease the
  time required for proofs that make heavy use of the evaluation of
  ground terms. The parallelism primitives allow for limiting
  parallel evaluation (spawning of so-called "threads") depending
  on resource availability. Specifically, the primitives allow
  specification of a size condition to control the granularity under
  which threads are allowed to spawn. If you use the parallelism
  primitives in recursive functions, you can get data-dependent
  parallelism in the program's execution.

  Our API consists of the ACL2 primitives ~ilc[plet], ~ilc[pargs],
  ~ilc[pand], and ~ilc[por], which all accept an optional granularity
  condition. ~ilc[Pand] and ~ilc[por] terminate early when an
  argument is found to evaluate to ~c[nil] or non-~c[nil],
  respectively, thus potentially improving on the efficiency of lazy
  evaluation.~/

  We recommend that in order to learn to use the parallelism
  primitives, you begin by reading examples: ~pl[parallelism-how-to].
  That section will direct you to further documentation topics.~/")

(defmacro set-parallel-evaluation (value)
  ":Doc-Section Parallelism

  enable and disable the parallelism library and the ability to
  submit parallelism primitives at the top level~/

  ~bv[]
  Example Forms:
  (set-parallel-evaluation nil)

```

```
(set-parallel-evaluation t)
(set-parallel-evaluation :bogus-parallelism-ok)
~ev[]
~/
```

The user can enable and disable parallel evaluation of parallelism primitives. When the argument equals `~c[nil]`, all parallelism primitives degrade to their serial equivalent. A setting of `~c[nil]` also permits the serial evaluation of parallelism primitives at the top level. `~l[parallelism-at-the-top-level]`.

When the argument equals `~c[t]`, all parallelism primitives encountered in raw Lisp are given the opportunity to execute in parallel. With a setting of `~c[t]`, the use of parallelism primitives at the top level is disabled. `~l[parallelism-at-the-top-level]` for an explanation.

When the argument equals `~c[:bogus-parallelism-ok]`, both parallel evaluation and the use of parallelism primitives at the top level are enabled. `~l[parallelism-at-the-top-level]` for why we require the user to manually allow parallelism primitives at the top level.~/

```
(cond ((null value)
      '(progn
        (f-put-global 'parallel-evaluation-enabled ,value state)
        (fms "Disabling parallel evaluation. The user can still ~
              use parallelism primitives, but during evaluation ~
              they will degrade to their serial equivalent.~%"
              nil *standard-co* state nil)
        (value-triple nil)))

      ((equal value t)
       #+acl2-par
       '(progn
         (f-put-global 'parallel-evaluation-enabled ,value state)
         (fms "Enabling parallelism for primitives used within ~
               functions that execute in raw Lisp. See :doc ~
               topic parallelism-at-the-top-level for more ~
               information.~%"
               nil *standard-co* state nil)
         (value-triple t)))

      #-acl2-par
      '(progn
        (fms "Parallelism can only be enabled in OpenMCL or ~
```

```

        threaded SBCL. The current Lisp is neither of the ~
        two and parallelism will remain disabled.~%"
        nil *standard-co* state nil)
    (value-triple (@ parallel-evaluation-enabled))))

((equal value :bogus-parallelism-ok)
 #+acl2-par
 '(pprogn
  (f-put-global 'parallel-evaluation-enabled ,value state)
  (fms "Enabling the use of primitives at the top-level. ~
       Note that top-level parallelism primitives execute ~
       serially. See :doc topic ~
       parallelism-at-the-top-level for more ~
       information.~%"
       nil *standard-co* state nil)
  (value-triple :bogus-parallelism-ok))

 #-acl2-par
 '(pprogn
  (fms "Parallelism can only be enabled in OpenMCL or ~
       threaded SBCL. The current Lisp is neither of the ~
       two and parallelism will remain disabled. Note ~
       that you can submit parallelism primitives at the ~
       top level when parallel evaluation is disabled.~%"
       nil *standard-co* state nil)
  (value-triple (@ parallel-evaluation-enabled))))

(t
 '(pprogn
  (fms "Set-bogus-parallelism's argument must be either ~
       nil, t, or :bogus-parallelism-ok.~%"
       nil *standard-co* state nil)
  (value-triple (@ parallel-evaluation-enabled))))))

(defdoc parallelism-at-the-top-level

  ":Doc-Section Parallelism

  triggering parallel execution~/

  The user must use parallelism primitives inside a raw Lisp context
  to gain parallel execution.~/

  The parallelism library is written in raw Lisp. As such, any
  function call expected to run in parallel must execute its
  functions in raw Lisp. There are a few conditions under which the

```

ACL2 user may expect to be in raw lisp but will instead be executing at the ACL2 level. `~l[guard-evaluation-table]` for more details.

In the following example, we submit `~c[pargs]` at the top-level. Instead of executing the raw-Lisp version of `~c[pargs]`, the primitive macroexpands away inside the ACL2 loop. For example:

```
~bv[]
(pargs (binary-+ (bar 4) (bar 5)))
~ev[]
```

Turns immediately into:

```
~bv[]
(binary-+ (bar 4) (bar 5))
~ev[]
```

Since the user could think they are executing in parallel, when instead they are executing serially, we disable the use of parallelism primitives at the top-level. For information on enabling and disabling parallel execution `~pl[set-parallel-evaluation]`.

A trivial way to enable parallel evaluation is to define the function `foo`, such that `foo` does the parallelism call. Since `foo` has its guards verified, it will execute immediately in raw lisp, and it may compute `(bar x)` and `(bar y)` in parallel.

```
~bv[]
(defun foo (x y)
  (declare (xargs :guard (and (acl2-numberp x) (acl2-numberp y))))
  (pargs (binary-+ (bar x) (bar y))))
~ev[]
```

If you want to test a parallelized function whose guards you do not want to specify or verify, then you could do a temporary trick as below. You should not use this trick for anything but testing, because `~ilc[skip-proofs]` can render your session unsound.

```
~bv[]
(skip-proofs
 (defun foo (x y)
  (declare (xargs :guard t))
  (pargs (binary-+ (bar x) (bar y)))))
~ev[]~/")
```

```
(defdoc parallelism-how-to
```

```
  ":Doc-Section Parallelism
```

```
  a tutorial on how to use the parallelism library.~/
```

```
  As explained elsewhere (~pl[parallelism-at-the-top-level]),
  parallelism is only implemented in the raw Lisp versions of
  functions. We therefore write and verify the guards of all the
  functions in this tutorial.~/
```

```
  In this topic we introduce the ACL2 parallelism primitives using
  the example of the doubly-recursive Fibonacci function, whose
  basic definition is as follows.
```

```
  ~b[Serial Fibonacci]
```

```
  ~bv[]
```

```
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (let ((a (fib (- x 1)))
                  (b (fib (- x 2))))
              (+ a b))))))
```

```
  ~ev[]
```

```
  ~b[Introducing] ~ilc[Pargs]
```

```
The simplest way to introduce parallelism to this function
definition is by using ~ilc[pargs]. We simply wrap the addition
expression with a ~ilc[pargs], and the arguments to the addition
will be computed in parallel whenever resources are available. As
a result, we end up with a very similar and thus intuitive
function definition. Note that we replaced ~ilc[+] by
~ilc[binary-+], since ~ilc[pargs] expects a function call, not a
macro call.
```

```
  ~bv[]
```

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (pargs (binary-+ (pfib (- x 1))
                             (pfib (- x 2)))))))
```

~ev[]

~b[Introducing the Problem of Granularity]

After you enter the above two versions of Fibonacci, test them both with variations of the forms:

```
~bv[]  
(time$ (fib 10))  
(time$ (pfib 10))  
~ev[]
```

Start with the small number 10 and then increase the argument by increments of 5 to 10 until you find your curiosity satisfied or your patience wearing thin. You can interrupt evaluation if necessary and return to the ACL2 loop. You will immediately notice that you have not saved much speed by introducing parallelism.

First let us consider the computation of ~c[(pfib 4)]. Assuming resources are available, ~c[(pfib 4)] will create a thread for computing ~c[(pfib 3)] and another thread for ~c[(pfib 2)]. The computation for ~c[(pfib 3)] and ~c[(pfib 2)] will take the same amount of time as in the serial case, but now we have wasted a lot of time setting up two threads (which must somehow be freed later). It is easy to imagine that setting up each thread takes much longer than the entire computation of ~c[(fib 4)].

Second, we must realize that if we have two threads available for computing ~c[(fib 10)], then the evaluation of ~c[(fib 8)] will probably complete before the evaluation of ~c[(fib 9)]. Once ~c[(fib 8)] finishes, parallelism resources will become available for the next recursive call made on behalf of ~c[(fib 9)]. If for example that call is ~c[(fib 3)], we will waste a lot of cycles just handing work to the thread that does this relatively small computation. We need a way to ensure that parallelism resources are only used on problems of a \"large\" size. Ensuring that only \"large\" problems are spawned is called the “granularity problem.”

In summary, we want to have a way to tell ACL2 that when the parameter ~c[x] of ~c[pfib] is “smaller” than some threshold, that we want to disable parallelism. Through testing on OpenMCL, we discovered 27 to be a reasonable threshold.

~b[Explicit Programming for the Problem of Granularity]

One way to avoid the problem of granularity is to duplicate code as below: ~bv[]

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (if (> x 27) ; this is the granularity test
                (pargs (binary-+ (pfib (- x 1))
                                  (pfib (- x 2))))
                (binary-+ (pfib (- x 1))
                          (pfib (- x 2)))))))
```

~ev[]

Duplicating code is fundamentally a bad design principle, because it can double the work for future maintenance. There is thus a special ~c[granularity] declare form that allows you to specify a criterion for spawning threads without duplicating code. The syntax for the ~il[granularity] form can be found in the documentation for each parallelism primitive, but here is what ~c[pfib] looks like using this feature. ~bv[]

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (pargs
             (declare (granularity (> x 27)))
             (binary-+ (pfib (- x 1))
                       (pfib (- x 2)))))))
```

~ev[]

Test each version of the function with variations of the forms:

```
~bv[]
  (time$ (fib 33))
  (time$ (pfib 33))
```

~ev[]

~b[Another Granularity Issue Related to Thread Limitations]

Our implementation of parallelism primitives has the property that once a thread is assigned a computation, that thread stays assigned to that computation until its completion. In particular, if a thread encounters a parallelism primitive that spawns child threads, that thread must wait until the child computations complete before it can continue its own computation. In the meantime, that (parent) thread reduces the number of additional threads that Lisp can provide by 1.

How can this limitation affect the user? Consider, for example, the application of a recursive function to a long list. Imagine that this function is written so that the non-base case of the body is something like `~c[(pargs (process (car x)) (recur (cdr x))))]`. Each such `~ilc[pargs]` that spawns child work must wait for its children, one of which is `~c[(recur (cdr x))]`, to complete. There is an ACL2 limit on how much pending work can be in the system, limiting the number of `pargss` that result in parallel evaluation. For example, if the ACL2 limit is `k` and each `~ilc[pargs]` actually parallelizes, then after `k ~c[cdr]`s there will be no further parallel evaluation.

Possible solutions may include reworking of algorithms (for example to be tree-based rather than list-based) or using appropriate granularity forms. We hope that future implementations will allow thread ‘re-deployment’ in order to mitigate this problem.

`~b[Introducing] ~ilc[Plet]`

We can use a `~ilc[let]` binding to compute the recursive calls of `~c[fib]` and then add the bound variables together, as follows.

```
~bv[]
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (let ((a (fib (- x 1)))
                  (b (fib (- x 2))))
              (+ a b))))))
```

`~ev[]`

By using `~ilc[plet]`, we can introduce parallelism in much the same way we used `~ilc[pargs]`, with an optional granularity form, as follows.

```
~bv[]
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (plet
              (declare (granularity (> x 27)))
              ((a (pfib (- x 1)))
               (b (pfib (- x 2))))
              (+ a b))))))
```

~ev[]

Notice that this time, we were able to use ~c[+] rather than ~c[binary-+]. Unlike ~ilc[pargs], which expects a function call (not a macro call), ~ilc[plet] allows macros at the top level of its body. Note that non-granularity declare forms are disallowed.

~b[Introducing] ~ilc[Pand] ~b[with Tree Validation]

Consider ‘‘genetic trees’’ that contains leaves of DNA elements, in the sense that each tip is one of the symbols ~c[A], ~c[G], ~c[C], or ~c[T]. First we define the function ~c[valid-tip] which recognizes whether a tip contains one of these symbols.

~bv[]

```
(defun valid-tip (tip)
  (declare (xargs :guard t))
  (or (eq tip 'A)
      (eq tip 'G)
      (eq tip 'C)
      (eq tip 'T)))
```

~ev[]

Now we define our function that traverses the tree. Note that a tree does not have the typical structure of a list in Lisp, as it has non-~c[nil] tips.

~bv[]

```
(defun valid-tree-serial (tree)
  (declare (xargs :guard t))
  (if (atom tree)
      (valid-tip tree)
      (and (valid-tree-serial (car tree))
           (valid-tree-serial (cdr tree)))))
```

~ev[]

We also define a parallel version.

~bv[]

```
(defun valid-tree-parallel (tree)
  (declare (xargs :guard t))
  (if (atom tree)
      (valid-tip tree)
      (pand (valid-tree-parallel (car tree))
            (valid-tree-parallel (cdr tree)))))
```

~ev[]

Before we can time the functions, we need to create test trees. We have found that test trees need to be approximately 1 gigabyte

before we clearly see speedup, and we make them asymmetric to demonstrate the ability of our implementation to adapt to asymmetric data. We can create the trees with the execution of the following forms.

```
~bv[]
(defun make-valid-binary-tree (x)
  (declare (xargs :mode :program))
  (if (< x 0)
      (cons (cons 'C 'G) (cons 'A 'T))
      (cons (make-valid-binary-tree (- x 2)) ; 2 to make asymmetric
            (make-valid-binary-tree (- x 1)))))

(defconst *valid-tree* (make-valid-binary-tree 30)) ; takes awhile
(defconst *invalid-tree* (cons *valid-tree* nil)) ; nil is invalid
; tip
~ev[]
```

We can time our functions with the forms:

```
~bv[]
(time$ (valid-tree-serial *valid-tree*))
(time$ (valid-tree-parallel *valid-tree*))
~ev[]
```

Unfortunately, the serial version runs faster than the parallelized version; however, there is more to this story.

~b[[Demonstrating Early Termination with an Invalid Tree](#)]

Now observe this magic:

```
~bv[]
(time$ (valid-tree-serial *invalid-tree*))
(time$ (valid-tree-parallel *invalid-tree*))
~ev[]
```

The serial version took awhile, while the parallel version finished almost immediately. The test for validation was split into testing the `~ilc[car]` and the `~ilc[cdr]` of the `~c[*invalid-tree*]` root, and since the `~c[cdr]` was equal to `~c[nil]`, its test returned immediately. This return immediately interrupted the computation associated with the `~c[car]`, and returned the result.

~b[[Granularity with](#) `~ilc[Pand]`]

We can also define a parallel version that takes advantage of the granularity form:

```
~bv[]
(defun valid-tree-parallel (tree depth)
  (if (atom tree)
```

```

      (valid-tip tree)
    (pand
      (declare (granularity (< depth 5)))
      (valid-tree-parallel (car tree) (1+ depth))
      (valid-tree-parallel (cdr tree) (1+ depth))))
  ~ev[]

```

We can test this form by executing our previous forms. Note by the way that we are unlikely to get speed-up with the parallel version here, because the individual `~c[valid-tip]` tests are so fast compared to the cost of spawning an execution.

```

~bv[]
(time$ (valid-tree-serial *valid-tree*))
(time$ (valid-tree-parallel *valid-tree* 0))
~ev[]

```

```
~b[Introducing] ~ilc[Por]
```

`~ilc[Por]` can be used in the same way as `~ilc[pand]`, but with early termination occurring when an argument evaluates to a non-`~c[nil]` value, in which case the value returned is `~c[t]`. Finally, `~ilc[por]` also features the `~il[granularity]` form."

```
(defdoc granularity
```

```

  ":Doc-Section Parallelism
  limit the amount of parallelism~/

```

Some function calls are on arguments whose evaluation time is long enough to warrant parallel evaluation, while others are not. A granularity form can be used to make appropriate restrictions on the use of parallelism.~/

For example, consider the Fibonacci function. Experiments have suggested that whenever the argument is less than 27, then we should call a serial version of the Fibonacci function. One solution is to write two definitions of the Fibonacci function, one serial and one parallel.

```

~bv[]
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (binary-+ (fib (- x 1))
                      (fib (- x 2))))))

```

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        ((< x 27) (binary+ (fib (- x 1))
                           (fib (- x 2))))
        (t (pargs (binary+ (pfib (- x 1))
                           (pfib (- x 2)))))))
```

~ev[]

We realize quickly that writing both of these function definitions is cumbersome and redundant. This problem can be avoided by use of a `~c[granularity]` form declaration when employing a parallelism primitive. This form ensures that a call is parallelized only if resources are available and the granularity form evaluates to a non-`~c[nil]` value at the time of the call. Below is a definition of the Fibonacci function using a granularity form.

~bv[]

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (pargs (declare (granularity (>= x 27)))
                  (binary+ (pfib (- x 1))
                          (pfib (- x 2)))))))
```

~ev[]

A granularity form can use an extra parameter that describes the call depth of the function the user is parallelizing. Consider for example the following parallel `~c[mergesort]` function, based on Davis's Ordered Sets library. It splits the data into symmetric chunks for computation, so we increment the `~c[depth]` argument during the recursive call on both the `~c[car]` and `~c[cdr]`.

~bv[]

```
(defun SETS::pmergesort-exec (x depth)
  (declare (xargs :mode :program))
  (cond ((endp x) nil)
        ((endp (cdr x)) (SETS::insert (car x) nil))
        (t (mv-let (part1 part2)
                   (SETS::split-list x nil nil)
                   (pargs
                    (declare (granularity (< depth 2)))
                    (SETS::union
                     (SETS::pmergesort-exec part1
                                             (1+ depth))
```

```

(SSETS::pmergesort-exec part2
  (1+ depth)))))))))
~ev[]

```

A less intrusive method involves analyzing the data itself for structural properties. For example:

```

~bv[]
(defun some-depth-exceeds (x n)
  (declare (xargs :guard (natp n)))
  (if (atom x)
      nil
      (if (zp n)
          t
          (or (some-depth-exceeds (car x) (1- n))
              (some-depth-exceeds (cdr x) (1- n)))))))

```

```

(defun valid-tip (x)
  (declare (xargs :guard t))
  (or (eq x 'A)
      (eq x 'T)
      (eq x 'C)
      (eq x 'G)))

```

```

(defun pvalid-tree (x)
  (declare (xargs :guard t))
  (if (atom x)
      (valid-tip x)
      (pand (declare (granularity (some-depth-exceeds x 3)))
            (pvalid-tree (car x))
            (pvalid-tree (cdr x)))))

```

```

~ev[]
~/")

```

```

(defdoc early-termination
  ":Doc-Section Parallelism
  early termination for ~ilc[pand] and ~ilc[por].~/~/

```

The evaluation of `~c[(and expr1 expr2)]` returns `~c[nil]` if `~c[expr1]` evaluates to `~c[nil]`, avoiding the evaluation of `~c[expr2]`. More generally, the evaluation of `~c[(and expr1 expr2 ... exprk)]` terminates with a return value of `~c[nil]` as soon as any `~c[expri]` evaluates to `~c[nil]` ~-[] no `~c[exprj]` is evaluated in this case for `~c[j > i]`. This so-called ‘‘lazy evaluation’’ of `~ilc[and]` terms can thus save some computation; roughly speaking, the smaller the `~c[i]`, the more computation is saved.

If the above call of `~ilc[and]` is replaced by its parallel version, `~ilc[pand]`, then there can be even more opportunity for skipping work. The arguments to `~ilc[pand]` can be evaluated in parallel, in which case the first such evaluation that returns with a value of `~c[nil]`, if any, causes the remaining such evaluations to abort.

Consider the following functions that compute whether a tree is valid (`~pl[granularity]` for a discussion of the granularity form).

```

~bv[]
(defun valid-tip (x)
  (declare (xargs :guard t))
  (or (eq x 'A)
      (eq x 'T)
      (eq x 'C)
      (eq x 'G)))

(defun pvalid-tree (x depth)
  (declare (xargs :guard (natp depth)))
  (if (atom x)
      (valid-tip x)
      (pand (declare (granularity (< depth 10)))
            (pvalid-tree (car x) (1+ depth))
            (pvalid-tree (cdr x) (1+ depth)))))

~ev[]

```

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 `~ilc[pand]`, once one of those terms evaluates to `~c[nil]`, all sibling computations are aborted and the `~ilc[pand]` returns `~c[nil]`. By doing so, we can in principle attain speedup in greater proportion to the number of available cores.

The concept of early termination also applies to `~ilc[por]`, except that early termination occurs when an argument evaluates to `non-~c[nil].~/`

```

(defun car-is-declarep (x)
  (declare (xargs :guard t))
  (and (consp x)
       (consp (car x)) ; (declare (granularity (< depth 5)))
       (equal (caar x) 'declare)))

(defun car-is-declare-with-granularityp (x)
  (declare (xargs :guard t))
  (and (consp x)

```

```

      (consp (car x)) ; (declare (granularity (< depth 5)))
      (equal (caar x) 'declare)
      (consp (cдар x))
      (let ((gran-form (cadar x)))
        (and (true-listp gran-form)
              (equal (length gran-form) 2)
              (equal (car gran-form) 'granularity))))))

(defun check-and-parse-for-granularity-form (x)

; X is a list of forms that may begin with a granularity declaration
; such as (declare (granularity (< depth 5))). The return signature
; is (erp msg granularity-form-exists granularity-form
; remainder-forms). If there is no declaration then we return (mv
; nil nil nil nil x). If there is error then we return (mv t
; an-error-message nil nil x). Otherwise we return (mv nil nil t
; granularity-form (cdr x)).

; It is necessary to return whether the granularity form exists. If
; we did not do so, there would be no mechanism for distinguishing
; between a non-existent granularity form and one that was nil.

; A granularity form declaration is the only acceptable form of
; declaration. Some examples of unaccepted declarations are type
; and ignore declarations.

      (cond ((not (car-is-declarep x))
             (mv nil nil nil nil x))
            ((car-is-declare-with-granularityp x)
             (let* ((granularity-declaration (cadar x))
                    (granularity-form (cadr granularity-declaration)))
               (mv nil nil t granularity-form (cdr x))))
            (t
             (mv t
                  "Within a parallelism primitive, a granularity form ~
                  declaration is the only acceptable form of ~
                  declaration. Some examples of unaccepted ~
                  declarations are type and ignore declarations."
                  nil
                  nil
                  x))))

;;;
;;;
;;; pargs ;;;

```

```

;;;      ;;;
;;;      ;;;
;;;      ;;;

#-acl2-loop-only
(defmacro closure-for-expression (x)
  '(function (lambda () ,x)))

#-acl2-loop-only
(defmacro closure-list-for-expression-list (x)
  (if (atom x)
      nil
      '(cons (closure-for-expression ,(car x))
              (closure-list-for-expression-list ,(cdr x)))))

#+(and (not acl2-loop-only) acl2-par)
(defmacro pargs (&rest forms)

; This is the raw lisp version for threaded Lisps.

  (mv-let
    (erp msg gran-form-exists gran-form remainder-forms)
    (check-and-parse-for-granularity-form forms)
    (declare (ignore msg))
    (assert (not erp))
    (let ((function-call (car remainder-forms))
          (if (null (cddr function-call)) ; whether there are two or more
              ; arguments
              function-call
              (list 'if
                    (if gran-form-exists
                        '(and ,gran-form (parallelism-resources-available))
                        '(parallelism-resources-available))
                    (list 'parallelize-fn
                          (list 'quote (car function-call))
                          (list 'closure-list-for-expression-list
                                (cdr function-call))
                          function-call))))))

#+(or acl2-loop-only (not acl2-par))
(defmacro pargs (&rest forms)
  ":Doc-Section Parallelism

parallel evaluation of arguments in a function call~/

~bv[]

```

Example Forms:

```
(pargs (binary-+ (fibonacci (- x 1)) (fibonacci (- x 2))))  
  
(pargs (declare (granularity (> x 35)))  
       (binary-+ (fibonacci (- x 1)) (fibonacci (- x 2))))~/
```

General Form:

```
(pargs (declare (granularity expr)) ; optional granularity  
       ; declaration  
       (f arg1 ... argN))  
~ev[]
```

where $\sim c[N \geq 0]$ and each $\sim c[\text{argi}]$ and $\sim c[\text{expr}]$ are arbitrary terms. Logically, $\sim c[\text{pargs}]$ is just the identity macro, in the sense that the above forms can logically be replaced by $\sim c[(f \text{arg1} \dots \text{argN})]$. However, this $\sim c[\text{pargs}]$ form may parallelize the evaluation of arguments $\sim c[\text{arg1}]$ through $\sim c[\text{argN}]$ before applying function $\sim c[f]$ to those results. If the above $\sim \text{ilc}[\text{granularity}]$ declaration is present, then its expression ($\sim c[\text{expr}]$ above) is first evaluated, and if the result is $\sim c[\text{nil}]$, then such parallelism is avoided. Even if parallelism is not thus avoided, parallelism may be limited by available resources.

Since macros can change expressions in unexpected ways, we disable the $\sim c[\text{pargs}]$ 'ing of macro calls. While it may be possible to reliably expand macros using the Lisp function $\sim c[\text{macroexpand}]$, we avoid it in an effort to keep the user in a sensible state. To parallelize computation of arguments to a macro, $\sim \text{pl}[\text{plet}]$.

$\sim 1[\text{parallelism-at-the-top-level}]$ for a clarification on evaluating parallelism primitives from within the ACL2 loop.~/

```
(mv-let  
  (erp msg gran-form-exists gran-form remainder-forms)  
  (check-and-parse-for-granularity-form forms)  
  (cond (erp (er hard 'pargs msg))  
        ((or (and (equal (length forms) 1) (not gran-form-exists))  
              (and (equal (length forms) 2) gran-form-exists))  
         (let ((function-call (car remainder-forms)))  
           (if gran-form-exists  
               '(prog2$ ,gran-form ,function-call)  
               function-call)))  
        (t  
         (er hard 'pargs  
          "Pargs was passed the wrong number of arguments. ~  
          Without a granularity declaration, pargs takes one ~
```

argument. With a granularity declaration, pargs ~ requires two arguments, the first of which must be ~ a declare form. See :DOC topic pargs for the pargs ~ granularity form syntax."))))))

```
;;;;;;;;;;;;;
;;;      ;;;
;;; plet ;;;
;;;      ;;;
;;;;;;;;;;;;;
```

```
; Given a list of let bindings, closures-for-bindings returns a list
; of function closures. It does not store the variable names with
; the closures.
```

```
#+-acl2-loop-only
(defmacro closures-for-bindings (x)
  (if (atom x) nil
      (list 'cons (list 'closure-for-expression (cadar x))
            (list 'closures-for-bindings (cdr x))))))
```

```
#+-acl2-loop-only
(defun do-let (results body-closure)
  (apply body-closure results))
```

```
#+-acl2-loop-only
(defun identity-list (&rest rst) rst)
```

```
#+(and (not acl2-loop-only) acl2-par)
(defun plet-fn (closures body-closure)
  (apply body-closure (parallelize-fn 'identity-list closures)))
```

```
#+-acl2-loop-only
(defun make-variable-list (x)
  (if (atom x)
      nil
      (cons (caar x)
            (make-variable-list (cdr x)))))
```

```
#+-acl2-loop-only
(defmacro closure-for-expression-with-variables-as-args
  (bindings body)
  '(function (lambda ,(make-variable-list bindings) ,body)))
```

```
#+(and (not acl2-loop-only) acl2-par)
(defmacro plet (&rest forms)
```

; This is the raw Lisp version for threaded Lisps.

```
(mv-let
  (erp msg gran-form-exists gran-form remainder-forms)
  (check-and-parse-for-granularity-form forms)
  (declare (ignore msg))
  (assert (not erp))
  (let ((bindings (nth 0 remainder-forms))
        (body (nth 1 remainder-forms)))
    (if (null (cdr bindings)) ; whether there is more than one
        ; binding
        (list 'let bindings body)
        (list 'if
              (if gran-form-exists
                  '(and ,gran-form (parallelism-resources-available))
                  '(parallelism-resources-available))
              (list 'plet-fn
                    '(closures-for-bindings ,bindings)
                    '(closure-for-expression-with-variables-as-args
                      ,bindings
                      ,body))
              (list 'let bindings body))))))
```

```
#+(or acl2-loop-only (not acl2-par))
```

```
(defmacro plet (&rest forms)
```

```
  ":Doc-Section Parallelism
```

```
  parallel version of ~ilc[let]~/
```

```
  ~bv[]
```

```
  Example Forms:
```

```
(plet ((first (fibonacci (- x 1)))
       (second (fibonacci (- x 2))))
      (+ first second))
```

```
(plet (declare (granularity (> x 35)))
      ((fib-x-minus-1 (fibonacci (- x 1)))
       (fib-x-minus-2 (fibonacci (- x 2))))
      (+ fib-x-minus-1 fib-x-minus-2))~/
```

```
  General Form:
```

```
(plet (declare (granularity expr)) ; optional granularity
      ; declaration
      ((var1 val1)
       ...
```

```

      (varN valN))
      (declare ...) ... (declare ...) ; optional declarations
      body)
~ev[]

```

The syntax of ~c[plet] is identical to the syntax of ~ilc[let], except that ~c[plet] may have the optional granularity declaration in the position shown above, where ~c[expr] above is an arbitrary term. Moreover, in the logic, ~c[plet] macroexpands to ~c[let]. ~l[let].

~c[Plet] may parallelize the evaluation of each ~c[vali] above before processes the ~c[body] of the ~c[plet]. If the above ~ilc[granularity] declaration is present, then its expression (~c[expr] above) is first evaluated, and if the result is ~c[nil], then such parallelism is avoided. Even if parallelism is not thus avoided, parallelism may be limited by available resources.

~l[parallelism-at-the-top-level] for clarification on evaluating a parallelism primitive within the ACL2 loop.~/

```

(mv-let
 (erp msg gran-form-exists gran-form remainder-forms)
 (check-and-parse-for-granularity-form forms)
 (cond (erp (er hard 'plet msg))
       ((or (and (equal (length forms) 2) (not gran-form-exists))
            (and (equal (length forms) 3) gran-form-exists))
        (let ((bindings (car remainder-forms))
              (function-call (cadr remainder-forms)))
          (if gran-form-exists
              '(prog2$ ,gran-form
                  (let ,(car remainder-forms)
                    ,(cadr remainder-forms)))
              (list 'let bindings function-call))))
 (t (er hard
      'plet
      "Plet was passed the wrong number of arguments. ~
      Without a granularity declaration, plet takes two ~
      arguments. With a granularity declaration, plet ~
      requires three arguments, the first of which must ~
      be a granularity declare form. For the plet ~
      granularity form syntax see :DOC topic plet."))))))

```

```

;;;;;;;;;;;;;
;;;      ;;;
;;; pand ;;;

```

```

;;;      ;;;
;;;      ;;;
;;;      ;;;

; The following reasons support the decision to Booleanize pand.
; 1. It should be consistent with por, which must be Booleanized.
; 2. If users employ pand, it's important that they realize it's
; not exactly like and. Anything that helps them become aware of
; the guard discrepancies is likely helpful toward their goal.

#+(and (not acl2-loop-only) acl2-par)
(defmacro pand (&rest forms)

; This is the raw Lisp version for threaded Lisps.

(mv-let
  (erp msg gran-form-exists gran-form remainder-forms)
  (check-and-parse-for-granularity-form forms)
  (declare (ignore msg))
  (assert (not erp))
  (if (null (cdr remainder-forms)) ; whether pand has only one
    ; argument
    (list 'if (car remainder-forms) t nil)
    (let ((and-early-termination-function
          '(lambda (x) (null x))))
      (list 'if
        (if gran-form-exists
            '(and ,gran-form (parallelism-resources-available))
            '(parallelism-resources-available))
        (list 'parallelize-fn 'and-list
          (list 'closure-list-for-expression-list
            remainder-forms)
            and-early-termination-function)
        (list 'if (cons 'and remainder-forms) t nil))))))

(defun binary-pand (x y)

; Booleanized binary and.

  (declare (xargs :guard t :mode :logic))
  (and x y t))

#+(or acl2-loop-only (not acl2-par))
(defmacro pand (&rest forms)
  ":Doc-Section Parallelism

parallel, Boolean version of ~ilc[and]~/

```

```

~bv[]
Example Forms:
(pand (subset-equalp x y)
      (subset-equalp y x))

(pand (declare
      (granularity
       (and (> (length x) 500)
            (> (length y) 500))))
      (subset-equalp x y)
      (subset-equalp y x))
~ev[]~/

```

```

~bv[]
General Form:
(pand (declare (granularity expr)) ; optional granularity
      ; declaration
      arg1 ... argN)
~ev[]
where ~c[N >= 0] and each ~c[argi] and ~c[expr] are arbitrary
terms.

```

~c[Pand] evaluates its arguments in parallel. It returns a Boolean result: ~c[nil] if any argument evaluates to ~c[nil], else ~c[t]. Note that while ~ilc[and] returns the last non-~c[nil] value from evaluating its arguments left-to-right, ~ilc[pand] always returns a Boolean result. This makes it consistent with ~ilc[por]. The second difference is that the falsity of the evaluation of the first argument does not prevent evaluation of the second argument.

Consider the following form.

```

~bv[]
(pand (consp x) (equal (car x) 'foo))\
~ev[]

```

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

~l[parallelism-how-to] for another example. Also

~pl[parallelism-at-the-top-level] for a clarification on evaluating parallelism primitives within the ACL2 loop. Finally ~pl[early-termination] to read how ~c[pand] can offer more efficiency than ~ilc[and] by avoiding evaluation of some of its arguments.~/"

; Since we use &rest, we know forms is a true-list.

```
(mv-let
 (erp msg gran-form-exists gran-form remainder-forms)
 (check-and-parse-for-granularity-form forms)
 (cond (erp (er hard 'pand msg))
       ((atom remainder-forms) t) ; (pand) == t
       ((null (cdr remainder-forms)) ; same as length == 1
        (list 'if (car remainder-forms) t nil)) ; booleanize
       (gran-form-exists
        (list 'prog2$
              gran-form
              (xxxjoin 'binary-pand remainder-forms)))
       (t (xxxjoin 'binary-pand remainder-forms))))
```

```
;;;;;;;;;;;;;
;;;      ;;;
;;; por ;;;
;;;      ;;;
;;;;;;;;;;;;;
```

; Note that por must be Booleanized if we are to support early termination, i.e., so that any non-nil value can cause por to return.

```
#+(and (not acl2-loop-only) acl2-par)
(defmacro por (&rest forms)
```

; This is the raw Lisp version for threaded Lisps.

```
(mv-let
 (erp msg gran-form-exists gran-form remainder-forms)
 (check-and-parse-for-granularity-form forms)
 (declare (ignore msg))
 (assert (not erp))

 (if (null (cdr remainder-forms)) ; whether por has one argument
     (list 'if (car remainder-forms) t nil)
     (let ((or-early-termination-function
           '(lambda (x) x))))
```

```

      (list 'if
            (if gran-form-exists
                '(and ,gran-form (parallelism-resources-available))
                '(parallelism-resources-available))
            (list 'parallelize-fn 'or-list
                  (list 'closure-list-for-expression-list
                        remainder-forms)
                  or-early-termination-function)
            (list 'if (cons 'or remainder-forms) t nil))))))

(defun binary-por (x y)

; Booleanized binary or.

  (declare (xargs :guard t :mode :logic))
  (if x t (if y t nil)))

#+(or acl2-loop-only (not acl2-par))
(defmacro por (&rest forms)
  ":Doc-Section Parallelism

parallel, Boolean version of ~ilc[or]~/

~bv[]
Example Forms:
(por (subset-equalp x y)
      (subset-equalp y x))

(por (declare
      (granularity
       (and (> (length x) 500)
            (> (length y) 500))))
      (subset-equalp x y)
      (subset-equalp y x))
~ev[]~/

~bv[]
General Form:
(por (declare (granularity expr)) ; optional granularity
      ; declaration
      arg1 ... argN)
~ev[]
where ~c[N >= 0] and each ~c[argi] and ~c[expr] are arbitrary
terms.

~c[Por] evaluates its arguments in parallel. It returns a Boolean

```

result: `~c[t]` if any argument evaluates to non-`~c[nil]`, else `~c[nil]`. Note that while `~ilc[or]` returns the first non-`~c[nil]` value from evaluating its arguments left-to-right, `~ilc[por]` always returns a Boolean result. This prevents the nondeterminism that arises when one argument's evaluation sometimes finishes before another. The second difference is that the truth of the evaluation of the first argument does not prevent the evaluation of the second argument.

Consider the following form.

```
~bv[]
(por (atom x) (equal (car x) 'foo))
~ev[]
```

With `~c[por]` both `~c[(atom x)]` and `~c[(equal (car x) 'foo)]` can execute in parallel. With `~c[or]`, the truth of `~c[(atom x)]` prevents the evaluation of `~c[(car x)]`. Our logical definition of `~c[por]` does not provide `~c[(not) ~c[(atom x)]]` as a guard to `~c[(car x)]`.

`~l[parallelism-how-to]` for another example. Also `~pl[parallelism-at-the-top-level]` for a clarification on evaluating parallelism primitive within the ACL2 loop. Finally `~pl[early-termination]` to read how `~c[pand]` can offer more efficiency than `~ilc[and]` by avoiding evaluation of some of its arguments.~/

```
(mv-let
 (erp msg gran-form-exists gran-form remainder-forms)
 (check-and-parse-for-granularity-form forms)
 (cond (erp (er hard 'por msg))
       ((atom remainder-forms) nil) ; (por) == nil
       ((null (cdr remainder-forms)) ; same as length == 1
        (list 'if (car remainder-forms) t nil))
       (gran-form-exists
        (list 'prog2$
              gran-form
              (xxxjoin 'binary-por remainder-forms))))
 (t (xxxjoin 'binary-por remainder-forms))))
```

Bibliography

- [ACL07] ACL2. *ACL2 Documentation*, November 2007.
- [Agh86] Gul Agha. An overview of actor languages. *SIGPLAN Not.*, 21(10):58–67, 1986.
- [AHT07] Stephen Adams, Chris Hansen, and The MIT Scheme Team. *MIT/GNU Scheme User's Manual*. Massachusetts Institute of Technology, May 2007.
- [AMD06] AMD. Introducing multi-core technology. On the Web, April 2006. <http://multicore.amd.com/en/Technology/>.
- [BKM96] 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.
- [BO03] Randal E. Bryant and David O'Hallaron. *Computer Systems, a Programmer's Perspective*. Prentice Hall, first edition, 2003.

- [Coo95] Gene Cooperman. Star/mpi: Binding a parallel library to interactive symbolic algebra systems. In *Proc. of Int. Symposium on Symbolic and Algebraic Computation*, pages 126–132. ACM Press, 1995.
- [Dav04] 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.
- [DV96] D.Kapur and M. Vandevoorde. Kapur, d. and vandevoorde, m.t., dlp: a paradigm for parallel interactive theorem proving, submitted to cade., 1996.
- [FDSZ01] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM 2001)*, Monterey, CA, 2001.
- [Fra07a] Franz Inc. *Allegro Common Lisp Documentation*, July 2007.
- [Fra07b] Franz Inc. *Allegro Common Lisp FAQ*, 2007.
- [GGSI89] Ron Goldman, Richard P. Gabriel, Carol Sexton, and Lucid Inc. Qlisp: An interim report. In *Parallel Lisp: Languages and Systems*, pages 161–181, 1989.
- [GM84] Richard P. Gabriel and John McCarthy. Queue-based multi-processing lisp. In *Conference on LISP and Functional Programming*, pages 25–44, 1984.

- [GWH02] David Greve, Matthew Wilding, and David Hardin. High-speed analyzable simulators. In Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*, pages 89–106, 2002.
- [Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [Jon89] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.
- [JSM89] Barbara S. Epstein James S. Miller. Garbage collection in multischeme. In *Parallel Lisp: Languages and Systems*, pages 138–160, 1989.
- [KM04] 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.
- [Lis06a] LispWorks. *LispWorks Reference Manual*, July 2006.
- [Lis06b] LispWorks. *LispWorks User Guide*, July 2006.
- [LSBB92] Reinhold Letz, Johann Schumann, Stefan Bayerl, and Wolfgang Bibel. Setho: A high-performance theorem prover. In *Journal of Automated Reasoning*, volume 8(2), pages 183–212, 1992.
- [Mot98] Roderick Moten. Exploiting parallelism in interactive theorem provers. In *Theorem Proving in Higher Order Logics*, pages 315–330, 1998.

- [Ope06] OpenMCL. *The Ephemeral GC*, April 2006.
<http://openmcl.closure.com/Doc/The-Ephemeral-GC.html>.
- [Rag06] David L. Rager. Adding parallelism capabilities in ael2. 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.
- [RHH84] Jr. Robert H. Halstead. Implementation of multilisp: Lisp on a micro-processor. In *Conference on LISP and Functional Programming*, pages 9–17, 1984.
- [RHH89] Jr. Robert H. Halstead. New ideas in parallel lisp: Language design, implementation, and programming tools. In *Parallel Lisp: Languages and Systems*, pages 2–57, 1989.
- [Sch96] Johann Schumann. Sicotheo: Simple competitive parallel theorem provers. In *Conference on Automated Deduction*, pages 240–244, 1996.
- [SGG03] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley and Sons, Inc., sixth edition, 2003.
- [SL90] Johann Schumann and Reinhold Letz. PARTHEO: A high-performance parallel theorem prover. In *Conference on Automated Deduction*, pages 40–56, 1990.
- [Ste90] Guy L. Steele Jr. *Common Lisp the Language*. Digital Press, second edition, 1990.

- [WF97] Andreas Wolf and Marc Fuchs. Cooperative parallel automated theorem proving. Technical report, Munich University of Technology, 1997.
- [Wik07] Wikipedia. Closure (computer science) — Wikipedia, the free encyclopedia, 2007. [Online; access 25-November-2007].

Vita

David Lawrence Rager was born in Wimbledon, England, the eldest child of Brent and Diane Rager. Afterwards he moved to Houston, Texas, followed a year and a half later by a stay of similar length in Singapore. After enjoying the foods of Punggol Point, he returned to Houston, to earn a high school degree from Memorial High School. He subsequently enrolled in the Business Honors program at The University of Texas at Austin. After completing his Management Information Systems courses early in the program, he was able to toy around in other subjects, including Actuarial Science and Computer Science. Having his interest in computational science piqued, he continued his education by pursuing a Masters in Computer Science, working as a Teaching Assistant, Research Assistant at the university, and as an Engineering Intern at Google, Inc. His lengthy pursuit of the Masters degree ends in 2008, as he continues his pursuit of his next degree, the Ph.D..

Permanent Address: 562 Lanecrest

Houston, TX 77024

This thesis was typeset with L^AT_EX 2_ε¹ by the author.

¹L^AT_EX 2_ε is an extension of L^AT_EX. L^AT_EX is a collection of macros for T_EX. T_EX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.