

# Implementing a Parallelism Library for a Functional Subset of LISP

David L. Rager

The University of Texas at Austin  
ragerdl@cs.utexas.edu

Warren A. Hunt Jr.

The University of Texas at Austin  
hunt@cs.utexas.edu

## Abstract

This paper discusses four primitives supporting parallel evaluation for a functional subset of LISP, specifically that subset supported by the ACL2 theorem prover. These primitives can be used to provide parallel execution for functions free from side effects without considering race conditions, deadlocks, and other common parallelism pitfalls.

We (1) introduce logical definitions for these primitives, (2) explain three features that improve the performance of these primitives, (3) give a brief explanation of the implementation, and (4) use the parallelism primitives in examples to show improvement in evaluation time.

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

**Keywords** functional language, parallel, plet, pargs, pand, por, granularity, LISP, ACL2

## 1. Introduction

This work is about adding parallelism capabilities to a functional subset of LISP, specifically the ACL2 theorem prover (ACL 2008). 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 (Brock et al. 1996). Completion of these projects can depend critically on ACL2’s ability to execute industrial-sized models (Greve et al. 2002). As multi-core CPUs become commonplace, ACL2 users would like to take advantage of the underlying available hardware resources (Kaufmann and Moore 2004, section 4.5). Since ACL2 is a functional language and one can introduce parallelism into functional languages without worrying about safety (Agha 1986, section

1.4), it is possible to create a parallelism library that fits naturally into this functional subset of LISP.

We introduce four semantically side-effect free primitives that enable automatic parallel evaluation: `pargs`, `plet`, `pand`, and `por`. `Pargs` (short for *parallelize-args*) is logically the identity macro, but when executed, it enables parallel evaluation of a function’s arguments. `Plet` allows parallel evaluation of variable bindings. `Pand` and `por` are similar to the LISP 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 of three features of the parallelism library. First, with recursive use of the parallelism primitives, computation can adapt to the data so that a function’s computation does not become completely serial when encountering asymmetric data. Second, the parallelism library provides a means to specify a criterion for determining the granularity of potential evaluations. This helps the system determine when arguments to a function are complex enough to warrant parallel evaluation. Third, when issuing a `pand` or `por`, the library recognizes opportunities for early termination and returns from evaluation when such an opportunity occurs.

An explanation of the implementation follows next. Included in this are discussions of (1) the LISP-level threading interface (section 5.1), (2) how parallelism work is produced and under what conditions (section 5.2), (3) how work is consumed and evaluated (section 5.3), and (4) some optimizations (section 5.4).

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.

## 2. Related Work

The parallelism library derives inspiration 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 work relevant to LISP development, only a sampling of (1) is provided below.

## 2.1 Parallelism in Functional Languages

Some of the work on parallelism began in the 1980s and includes ideas such as  *futures*  and primitives like  *pcall*  (Halstead 1984; Gabriel and McCarthy 1984). More recent work includes an MPI library for GCL (Cooperman 1995) and Hunt and Moore’s partial ACL2 parallelism library implementation (private communication).

Multilisp was created in the early 1980s as an extended version of scheme (Halstead 1984). It implemented the  *future*  operator, which is often thought of as a promise for a form’s evaluation result (Halstead 1989, section 4). The notion of a  *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 that needed the result was said to  *touch*  (Halstead 1989, 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.

While our parallelism library does not implement the  *future*  operator, this work inspired exploration of several issues like garbage collection, granularity, and the ordering of parallel evaluation.

Further analysis of the ordering of parallel evaluation is discussed in a book of Peyton Jones (Jones 1987). 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 provides better performance than parallelizing computations with smaller granularity. As a result, it can be concluded that it is likely 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 (Rager 2008), are added to the work queue in a FIFO manner. Halstead also supports a FIFO ordering on memory-rich machines (Halstead 1989, section 6.2).

In somewhat more recent times, Cooperman provides bindings to MPI for GCL (Cooperman 1995). 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` . Such a distributed approach lays the foundation for a future multi-machine version of our parallelism library.

## 3. Parallelism Primitives

Our design of the parallelism primitives is driven by two goals. First, users need a way to parallelize computation efficiently. Second, the use of parallelism primitives should be as transparent to the theorem prover as possible. Thus, each ACL2 function has two definitions: (1) the logical version, used to prove theorems and (2) the raw LISP version, used for efficient evaluation. The logical version avoids complicating reasoning with the complexity of threads.

The examples below illustrate an ACL2-specific feature, the use of  *guards* . Once the guard of a function is satisfied, the more efficient raw LISP version of the function will execute.

### 3.1 Pargs

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

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

In this example, the raw LISP version of the function can evaluate arguments  `(pfib (- x 1))`  and  `(pfib (- x 2))`  in parallel, and then the function  `binary+`  will be applied to the list formed from their evaluation. It is desirable to apply  `pargs`  to function calls whose argument evaluations require a large amount of time, in order to obtain speedup by minimizing parallelism overhead. For experimental results that measure parallelism overhead, see section 6.1. A  *granularity form*  (see section 4.2) may be used to avoid parallelizing computations that take shorter amounts of time.

It is an error to apply  `pargs`  to macro calls because they do not evaluate their arguments (and we do not compensate by calling  `macroexpand` ). However, macro calls can often be handled using another primitive,  `plet` , which we describe next.

### 3.2 Plet

The second primitive,  `plet` , is logically equivalent to the macro  `let` . But 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 ((<= x 0) 0)
        ((= x 1) 1)
        (t (plet
             (declare (granularity (> x 30)))
             ((fibx-1 (pfib (- x 1)))
              (fibx-2 (pfib (- x 2))))
             (+ fibx-1 fibx-2))))))
```

As with `pargs`, the evaluations for the bindings of `fibx-1` and `fibx-2` occur in parallel, and then the closure containing the call of the macro `+` is applied.

### 3.3 Pand

The third primitive, `pand`, evaluates its (zero or more) arguments in parallel and returns their conjunction, a Boolean result. This evaluation differs from the evaluation of a corresponding call of `and` in two ways. First, `pand` returns a Boolean result. This makes it consistent with `por`, which is described later. The second difference is that `pand` is not lazy: the second argument can be evaluated even if the first argument evaluates to `nil`. Why does this matter? Consider the following call:

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

With `pand` replaced by `and`, the falsity of `(consp x)` prevents the evaluation of `(car x)`. This is different from `pand`, where both `(consp x)` and `(equal (car x) 'foo)` can evaluate in parallel.

However, `pand` calls need not evaluate all their arguments, because of an *early termination* feature. For further explanation of this feature, see section 4.3.

### 3.4 Por

`Por` evaluates its arguments in parallel, evaluates their disjunction, and returns a Boolean result. Since the evaluation order of parallel computation is nondeterministic, return of a Boolean value is important in order to avoid different results for the same `por` call. To avoid always having to evaluate the left-most non-`nil` argument result, the result is simply turned into a Boolean. Analogous to `pand`, the truth of an earlier argument to `por` does not necessarily prevent evaluation of later arguments, but early termination can terminate evaluation of arguments once a non-`nil` argument value is found.

## 4. User-level Parallelism Features

The parallelism library provides three features anticipated to enhance its usefulness. 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 re-

sources 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. Consider for example the following function that counts the leaves of a tree:

```
(defun atom-count (x)
  (declare (xargs :guards t))
  (if (atom x)
      1
      (binary-+ (atom-count (car x))
                (atom-count (cdr x)))))
```

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

If this function is given a degenerate tree shaped like a list, parallelizing the computation at only the top level 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. 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 how the system determines whether resources are available.

### 4.2 Granularity Form

When computing many functions, for example the naive Fibonacci, some inputs are large enough to warrant evaluating the arguments in parallel, while other inputs are too small to be worth the parallelism overhead. For example, consider the definition of the Fibonacci function found in section 3.1. Experiments on an eight-core machine at the University of Texas (see `lhug-7` in appendix A.1 for further details), indicate that whenever `x` is less than 30, 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 ((<= x 0) 0)
        ((= x 1) 1)
        (t (binary-+ (fib (- x 1))
                    (fib (- x 2))))))
```

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((<= 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. Instead, the user can provide a *granularity form* with the use of 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 an example definition of the Fibonacci function with a granularity form.

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((<= 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. Consider the `mergesort` below, which uses a depth argument to help determine when to parallelize computation. In this example, the depth is incremented in both recursive calls.

```
(defun mergesort (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 could analyze the data itself for structural properties. For example, one could define an auxiliary function that tests whether both the `car` and `cdr` of a particular input are cons pairs.

### 4.3 Early Termination

When computing a call of `and`, due to lazy evaluation, some of the arguments to the `and` may never be evaluated. Furthermore, if a user employs `pand`, even more computation 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 difference that the early termination condition is when an argument evaluates to non-`nil`.

## 5. Implementation

A sample of the implementation details that make up the parallelism library's evaluation strategy is provided below. This implementation is included with the ACL2 distribution.

There are three main library components that implement the parallelism strategy:

1. In order to avoid proliferation of code specific to individual LISP implementations, the library provides a LISP threading interface; this interface provides mechanisms for mutual exclusion, signaling, and controlling threads. An overview of this interface is included in section 5.1.
2. The library implements parallelism work producers (parallelism primitives) and work consumers (threads for evaluation). A discussion of the strategy and implementation of these producers and consumers follows after the interface's explanation in section 5.2.
3. There are optimizations to increase the library's 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 library 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 manner, the interface only supports SBCL and CCL.

### 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 calling the CCL and SBCL equivalents. While CCL has provided semaphores for some years, SBCL has only recently implemented them. As such, our implementation of semaphores for SBCL use a data structure that contains a counter, a lock, and a condition variable. Our semaphore 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 the cleanup form of a `without-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 library 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 completes, the cleanup form evaluates. In the event that an error occurs, the cleanup form will always run. Furthermore, in our library, interrupts are disabled while evaluating the cleanup form. These properties prove useful for implementing many functions in the parallelism library.

While using semaphores and condition variables is almost always more efficient than busy waiting (Halstead 1989, section 5.2.1), the threading interface also provides the function `thread-wait`, which allows a thread to busy 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 an arbitrary amount of time (currently 50 milliseconds) and then re-applies the given function to the appropriate arguments. 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 library implementation of these functions only serves as a wrapper for the underlying CCL and SBCL primitives.

## 5.2 Producing Work

Any thread that encounters a parallelism primitive has the potential to become a parallelism work producer. Three conditions must be true if a producer 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. Resources must be available. See section 5.2.1 for a discussion of how this property is determined.
2. The potential parallel evaluations must be lengthy enough in time to warrant parallel computation. As previously described in section 4.2, the programmer may use a *granularity form* to specify this requirement. If the granularity form is omitted, this condition is assumed to be true.
3. 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 queue. These pieces of work will be called *child work* throughout the remainder of this paper. After adding the pieces of work to the work queue, the thread that parallelizes evaluation signals consumer threads to evaluate the work and waits (on a signaling mechanism) until the parallel evaluations are finished. After the parallel evaluations finish, the producer returns the result.

### 5.2.1 Estimating Resource Availability

Before explaining how we estimate resource availability, it is necessary to explain how pieces of parallelism work are classified. Pieces of parallelism work are organized conceptually into four states: *unassigned* (U), *started* (S), *pending* (P), and *resumed* (R). The first classification refers to *unassigned* work not yet acquired by a work consumer. Until acquired by a consumer, these pieces of work are stored in the global *work queue*. The second state, *started*, describes the pieces of work that have been removed from the work queue and are being evaluated by an active thread. These pieces of work have not parallelized computation. The third section, *pending*, refers to work that itself encountered a parallelism primitive and decided to parallelize evaluation. After this piece of 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 classified as *active* or *inactive*. It is considered *active* if it is in either the *started* or *resumed* state and *inactive* when 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 prevents the underlying and finite system resources from being exceeded. When appropriate, 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.

### 5.2.2 Examining the Work Producer Code

Once the system decides to parallelize a computation, 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. Once these pieces of work are assembled, they are ready to be placed in the work queue. For the sake of brevity, a discussion of these subroutines is omitted from this paper. However, if the reader is interested in their detailed implementation or the reader would like to read how the threading interface is used, the reader should consult Rager (2008).

As a substitute for an in-depth explanation, pseudo code for the work producer is included in Figure 1.

### 5.3 Consuming Work

Threads implement work consumers for two reasons. First, threads share memory, which is good for one of LISP's current target systems, the SMP desktop market. Second, threads are lighter weight than processes (Silberschatz et al. 2003, page 131), lending themselves to finer-granularity problems.

#### 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 may also become a producer if the piece of work it is evaluating encounters a parallelism primitive. Figure 4 shows pseudo code for the worker thread.

#### 5.3.2 Limiting the Number of Worker Threads

Since parallel evaluation consumes resources, the library 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 shown in Figure 3.

```
(if {parallelism resources are unavailable}
  {evaluate the primitive serially}
  (progn
    {create appropriate closures and pieces of
      parallelism work for the primitive}
    {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}
    ;; the child pieces of work are now finished
    ;; being evaluated
    {wait for parallelism resources in a
      'resumptive' manner}
    {finish evaluating the primitive}
    {return the result}))
```

**Figure 1.** Pseudo Code for Work Producers (parallelism primitives)

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 LISP thread on it. Likewise, a CPU core is considered *idle* when the OS does not assign it a LISP thread to execute. Since the library does not have access to the OS scheduler, it assumes that LISP 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 the number of busy CPU cores (in SBCL the number of CPU cores is assumed to be a constant specified at compile time). As such, the library need not interact with the operating system to make a thread runnable, but instead, just blocks threads that don't have permission to run on LISP-level signaling mechanisms.

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 or for work to enter the work queue. 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 2 illustrates the relationships between pieces of work and their possible states, CPU cores, and worker threads.

**Limiting the Number of Active Worker Threads** The number of active worker threads is limited to twice the number of CPU cores. Once a worker thread finishes a piece of work, if there is work in the unassigned section, it will immedi-

**Figure 2.** Life Cycle of a Piece of Work

Work State	U	S	P*	R*
Allocated Core	no	yes	no	yes
Worker Thread State	n/a	active	pending	active

\*the pending and resumed states are not always entered.

ately acquire another piece of work. Limiting the number of active threads in this way helps minimize context switching overhead (Jones 1989).

**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 but serial evaluations were performed because all CPU cores were busy, this idleness would be wasted time. To avoid this, the unassigned portion of the 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 3 shows the limits imposed for a system with  $p$  CPU cores and a limit  $l$  on the total number of pieces of work allowed to be in the parallelism system at once. The need for this limit is explained in the following paragraph.

**Figure 3.** Ideal Amounts of Work by Classification

Unassigned	Started + Resumed	Pending
— $\cong p$ —	— $p \leq \&\& \leq 2p$ —	-irrelevant-
————— $\leq l$ —————		

**Limiting Total Workload** Since the OS supports a limited number of worker threads, restrictions must be imposed to ensure application stability. It is insufficient simply to set a limit on the number of threads spawned, because: (1) the stack of parents waiting on a deeply recursive nest of children can only unroll itself when the nest of children finishes evaluating and (2) any piece of work allowed into the system must eventually be allocated to a worker thread (with a caveat for early termination). 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.<sup>1</sup> Therefore, before a parent can decide to add work, it must first determine whether 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. For

<sup>1</sup>In 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.

now, 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 increased.

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)
      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 sum of  $n$  and the current number of idle threads is equal to twice the number of CPU cores in the system.

When a worker thread begins executing, it first sets up some catch blocks for early termination. The purpose of these blocks is explained in depth in (Rager 2008), but the details are omitted in this paper. The worker thread then waits for two conditions to be true: (1) there exists work to consume and (2) at least one idle CPU core is available for processing. If either of these conditions is false, the thread waits on a condition variable. This condition variable is signaled after work is added to the work queue and also signaled whenever a CPU core becomes available. As a result, whenever either of these two conditions is true, a worker thread awakens and checks for the conjunction of these two conditions. Once the worker thread has a piece of parallelism work<sup>2</sup>, it evaluates the piece of work and saves the result in a shared data structure specified by the piece of work.

### 5.3.4 Cleaning Up After Work Evaluation

After the result is saved, if appropriate, the worker thread aborts its siblings' computations. It does this by first remov-

<sup>2</sup>See (Rager 2008) for a more in depth explanation of how a worker thread safely acquires a piece of parallelism work

ing every now-irrelevant parallelism piece from the work queue. It then interrupts every sibling thread with a function that throws a tag. By throw'ing and catch'ing this tag, the interrupted thread can abort evaluation of a piece of parallelism work in a controlled manner. This throw'ing effectively aborts the active evaluation of unnecessary siblings' pieces of parallelism work.

The worker thread next records the freeing of parallelism resources. 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.

Now that the worker thread has recorded the freeing of resources, it begins to signal different groups of threads. The worker first signals its piece of work's parent, indicating that it has finished evaluating one of the parent's children. It next signals two condition variables. The first condition variable tells worker threads waiting for the two conditions described in section 5.3.3 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 a second and separate condition variable allows a thread waiting on this set of *resumptive heuristics* to obtain an idle CPU core sooner. See section 5.4.3 for an in-depth explanation of this second set of heuristics.

The current thread next reenters the top of the loop and joins the group of idle worker threads waiting for parallelism work and resources.

## 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 library 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. When computing the (pfib 45), this optimization limits required semaphore allocations to the low thousands (as opposed to upwards

```
{setup catch block for when a thread is no
  longer needed
  {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 piece of 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 await
    for work by looping back to the
    beginning}}
```

**Figure 4.** Pseudo Code for Worker Thread

of 55,000 semaphore allocations required without this optimization).

### 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. There is overhead associated with setting up the data structures necessary to recycle threads. These costs can be compared to the time it takes the LISP to create a new thread.

To do this comparison in CCL, 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.1).

```
(defvar *x* 0)
(defvar *y* 0)
(defun parallelism-call ()
  (let* ((semaphore-to-signal (make-semaphore))
        (closure-1
         (lambda ()
           (prog1 (setf *x* 3)
                 (signal-semaphore
                  semaphore-to-signal))))))
```

```

(closure-2
 (lambda ()
  (prog1 (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)))
(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 emphasized 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, should they have a higher priority than the pieces of work still on the work queue? 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 other 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, that only the worker threads that have spawned

children wait upon. When a waiting thread receives this condition variable signal, it will claim an idle core in a more liberal fashion. Instead of waiting for the count of idle CPU cores to be positive, the thread will wait for it to be greater than or equal to the negation of the number of CPU cores in the system. 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 resuming 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.

## 6. Performance Results

The parallelism library is evaluated on three machines: BBH (see A.1), Megaera (see A.1), and LHug-7 (see A.1). BBH is a 64 bit PPC, and Megaera and LHug-7 are 64 bit x86 machines. CCL is used on all three machines, and SBCL is run on LHug-7. With this setup, “perfect parallelism” would compute parallelized LISP 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 mentioned 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 four 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 LISP garbage collectors.

The scripts and output from running these tests are available for download at: <http://www.cs.utexas.edu/users/ragerdl/lisp2009/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 ACL2’s raw LISP read-eval-print-loop.<sup>3</sup>

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:

```
(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.
```

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))))
```

<sup>3</sup>Raw LISP mode can be entered by typing `:q` from an ACL2 prompt.

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.
```

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. 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 to follow. Additionally, the Fibonacci computation demonstrates an ability to adapt to asymmetric data, as each time the Fibonacci function 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, the Fibonacci function 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 doubly 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 a relatively 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 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 of 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 (Bryant and O’Hallaron 2003, 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.

### 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

**Table 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

collector (Clo 2008), the best the parallelism library 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 performance results 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.

**Table 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

### 6.5 Results Summary

In short, Fibonacci is a problem that experiences near-linear speedup 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.

## 7. Conclusions and Future Work

The four parallelism primitives described in this paper are: `pargs`, `p1et`, `pand`, and `por`. These primitives automatically allow significant speedup of execution for function calls that generate small amounts of garbage and have large granularity. Functions of varying granularity 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 that are relatively accessible compared to the LISP machines of the past.

The remainder of this section focuses on (1) how the parallelism library itself can be improved, and (2) how the underlying LISPs could better support parallelism.

### 7.1 Parallelism Library Improvements

The parallelism library is ported to SBCL, but more of the threading interface functions are directly translated in CCL. With the potential 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 possibly be made more efficient.

The parallelism library and tests still manage to occasionally "break" CCL x86-64 and SBCL, usually in the form of halting the LISP. One contribution of this work is that CCL and SBCL were stressed, and as a result, the implementors have improved the multi-threading aspects of these LISPs. Future work includes further stressing of these multi-threading components and fixing the bugs that cause the system to halt, wherever they may be.

The ACL2 parallelism paper from 2006 (Rager 2006) 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.

### 7.2 LISP Improvements

The LISP community could focus on two main features to more effectively support the parallelization of LISP programs. First, the LISP community could create a standard for LISP multi-threading implementations. The parallelism library 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 (Lis 2006a, section 14) (Lis 2006b, section 11) and can probably be used in this application, but we have avoided it thus far. Allegro does support threading (Fra 2007a, section 8.1), but since the threads can not run concurrently in Linux (Fra 2007b), the library has not been implemented for Allegro. While GCL does not provide a threading library, it does provide a process-level fork-based `p1et`. This implementation of `p1et` may be useful for the future, but it is not currently used by this parallelism library.

Second, the mergesort tests suggest that performance gains would occur if LISP implementations provide a concurrent garbage collector, such as that in MultiScheme (Miller and Epstein 1989) or in JVM implementations like pSemispaces and pMarkcompact (Flood et al. 2001). 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.

## A. Appendix

### A.1 Test Machine Specifications

**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

**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

**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

## Acknowledgments

Gary Byers has provided Closure Common Lisp threading primitives in support of this project and has continually stepped up to the CCL implementation challenges this work has produced. Initially, Matt Kaufmann created an ACL2 compilation flag that made `mv` and `mv-1et` thread safe. Since then he has helped refine the parallelism extension design, implementation, and documentation. We thank J Strother

Moore, Greg Lavender, Robert Boyer, Sandip Ray, Jared Davis, and the ACL2 user group for their valuable discussions and technical advice.

This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591, and the National Science Foundation under Grant No. EIA-0303609. We also thank Rockwell Collins, Inc. for supporting this work.

## References

- ACL2 Documentation*. ACL2, December 2008. URL <http://www.cs.utexas.edu/moore/ac12/v3-4/-ac12-doc-index.html>.
- Gul Agha. An overview of actor languages. *SIG-PLAN Not.*, 21(10):58–67, 1986. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/323648.323743>.
- Bishop Brock, Matt Kaufmann, and J Strother Moore. Acl2 theorems about commercial microprocessors. In Mandayam K. Sriivas and Albert John Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 275–293. Springer-Verlag, 1996.
- Randal E. Bryant and David O'Hallaron. *Computer Systems, a Programmer's Perspective*. Prentice Hall, first edition, 2003.
- The Ephemeral GC*. Clozure, December 2008. <http://ccl.clozure.com/manual/chapter15.2.html>.
- 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. URL <http://www.ccs.neu.edu/home/gen/pargcl.html>.
- 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. URL [citeseer.ist.psu.edu/flood01parallel.html](http://citeseer.ist.psu.edu/flood01parallel.html).
- Allegro Common Lisp Documentation*. Franz Inc., July 2007a. URL <http://www.franz.com/support/documentation/8.1/-doc/index.htm>.
- Allegro Common Lisp FAQ*. Franz Inc., 2007b. URL <http://www.franz.com/support/faq/#s13q1>.
- Richard P. Gabriel and John McCarthy. Queue-based multiprocessing lisp. In *Conference on LISP and Functional Programming*, pages 25–44, 1984.
- 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.
- Robert H. Jr. Halstead. Implementation of multilisp: Lisp on a microprocessor. In *Conference on LISP and Functional Programming*, pages 9–17, 1984.
- Robert H. Jr. Halstead. New ideas in parallel lisp: Language design, implementation, and programming tools. In *Parallel Lisp: Languages and Systems*, pages 2–57, 1989.
- Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- Simon L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.
- 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.
- LispWorks User Guide*. LispWorks, July 2006a. URL <http://www.lispworks.com/documentation/-lw50/LWUG/html/lwuser.htm>.
- LispWorks Reference Manual*. LispWorks, July 2006b. URL <http://www.lispworks.com/documentation/-lw50/LWRM/html/lwref.htm>.
- James S. Miller and Barbara S. Epstein. Garbage collection in multischeme. In *Parallel Lisp: Languages and Systems*, pages 138–160, 1989.
- David L. Rager. Implementing a parallelism library for acl2 in modern day lisps. Master's thesis, The University of Texas at Austin, 2008.
- David L. Rager. Adding parallelism capabilities in acl2. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 90–94, New York, New York, USA, 2006. ACM. ISBN 0-9788493-0-2. doi: <http://doi.acm.org/10.1145/1217994>.
- Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley and Sons, Inc., sixth edition, 2003.