May 1988 Report No. STAN-CS-884206

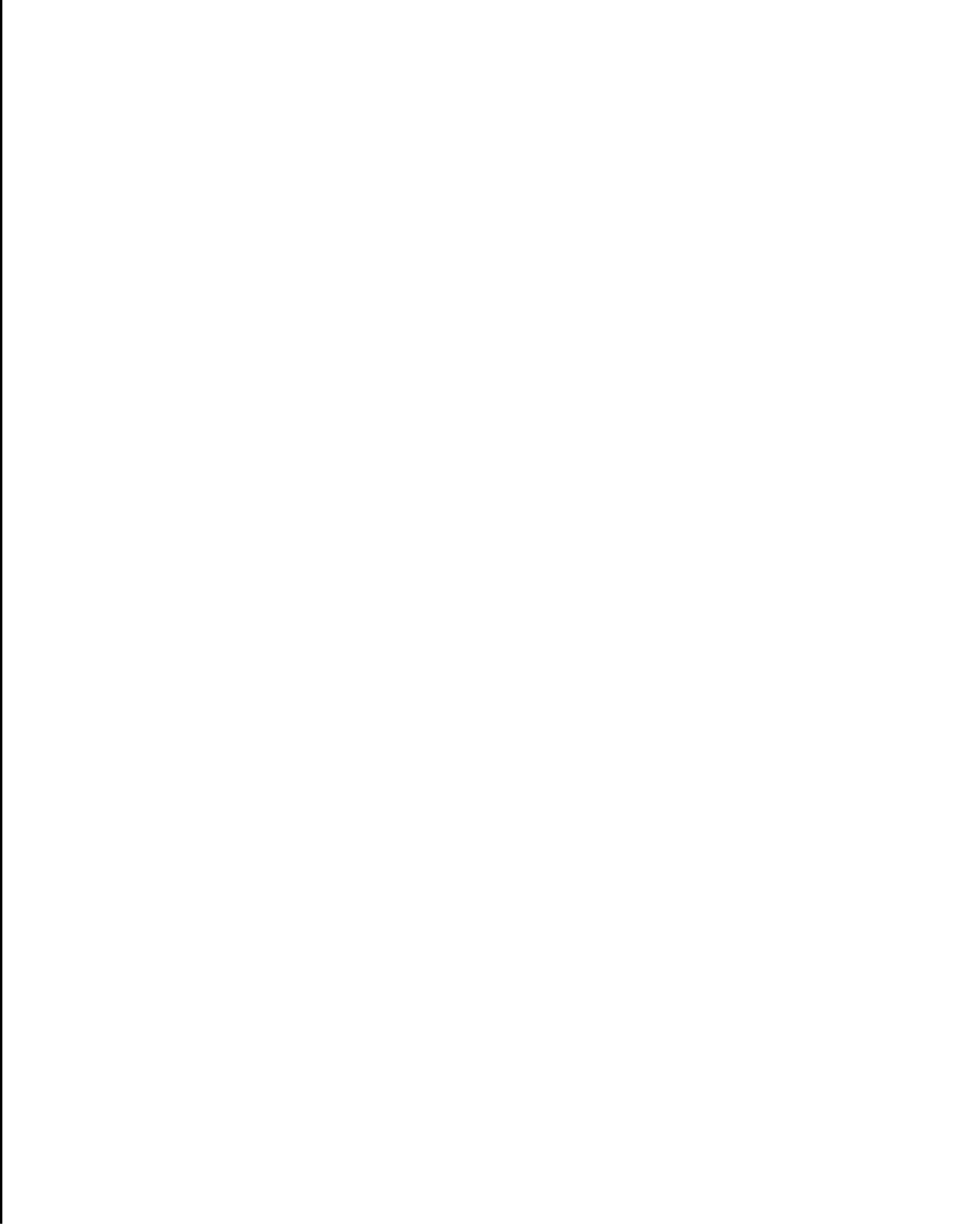
A Parallel Lisp Simulator

by

Joseph S. Weening

Department of Computer Science

Stanford University
Stanford, California 94305



A Parallel Lisp Simulator

Joseph S. Weening*

Abstract

CSIM is a simulator for parallel Lisp, based on a continuation passing interpreter. It models a shared-memory multiprocessor executing programs written in Common Lisp, extended with several primitives for creating and controlling processes. This paper describes the structure of the simulator, measures its performance, and gives an example of its use with a parallel Lisp program.

*Research supported by Defense Advanced Research Projects Agency contract N00039-84-C-0211 and a fellowship from the Fannie and John Hertz Foundation. The views and conclusions presented in this document are those of the author and sould not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1 Introduction

This paper describes a simulator for parallel Lisp, called CSIM. The "C" stands for continuation passing, which is the basic programming technique that the simulator uses to model multiprocessing. CSIM is written in Common Lisp and runs on several systems. It provides the following facilities:

- In the absence of an actual multiprocessing system, CSIM can be used as a testbed for parallel Lisp programs.
- The user can investigate the effects of varying parameters in a parallel environment, such as number of processors, cost of process creation, and contention for resources. Using CSIM, one can modify these parameters beyond the ranges in currently available hardware.
- CSIM allows metering and performance debugging of programs without modifying them or changing their execution environment. This is easier to do with a simulator than on a real machine.

CSIM was used extensively by the Qlisp project at Stanford until an initial implementation of Qlisp became available, and continues to be a valuable tool in our study of parallel Lisp programming.

2 The parallel machine model

Our machine model is a MIMD (multiple-instruction, multiple-data) computer with identical processors and *uniform access shared memory*. In such a system all data objects are stored in a single address space, and access to any location in the address space by any processor takes roughly the same amount of time. These features combine to ensure that there is no benefit or penalty for storing data in any particular location or running code on any particular processor.

Although it is currently difficult to build a computer that is faithful to this model and has mode than a few dozen processors, there is an emerging consensus that shared memory is an appropriate model given the current state of the programming art. Development of shared-memory multiprocessors therefore continues, and our results should be applicable to such machines with large numbers of processors if they emerge, as well as the currently available class of small- and medium-scale multiprocessors.

Another major assumption we make is that control of parallelism is *explicit*: programmers can indicate what computations are to be performed in parallel, while the default is sequential execution. This is not to preclude program-transformation tools that may detect parallelism in ordinary (sequential) programs and produce parallel programs; our parallel Lisp can serve as the target language for such tools.

In our programming model, processes are created at run time, and the decision to create a process may be conditional. We will not say much here about how such decisions may be usefully made; this paper is mainly a description of the implementation of our simulator.

Several proposals for changes to Lisp to accomodate such a programming model have been made, such as Qlisp [3], Multilisp [5], and MultiScheme [7]. While these dialects of Lisp differ in syntax and to some extent in the semantic power of the features they provide,

they all agree with the shared-memory philosophy that we have described. In CSIM we provide support for several of the Qlisp and Multilisp forms that we have found useful in writing parallel programs.

The goal of the Qlisp project at Stanford is to produce a compiler and run-time system for parallel Lisp, that will be used on an actual multiprocessor for serious applications. CSIM, however, is an interpreter running on a single processor and modeling a multiprocessor. In this approach there is an inherent tradeoff between the speed of the simulation and its degree of realism. We have chosen a middle ground that we believe will accurately model the issues in parallel programming that we want to investigate.

3 A continuation passing Lisp interpreter

As an introduction to the style in which CSIM is written, we describe here a simple continuation passing interpreter for a subset of Common Lisp. Readers familiar with the continuation passing style of programming may wish to skip this section.

Writing a Lisp interpreter in Lisp is easier than the equivalent task in most other languages, for several reasons. First, the representation of Lisp programs as Lisp data greatly simplifies syntactic analysis. More importantly, the interpreter can be "metacircular," using parts of the environment in which it runs to simulate the same constructs in the language being interpreted. This lets us focus on the parts of the evaluation process that are of interest. (See [1] for a discussion of metacircular interpreters in Scheme, a simple dialect of Lisp. Our examples will all be based on Common Lisp.)

The main function of the interpreter is `eval`, which takes a form (a Lisp expression representing a program) and an environment (a data structure representing the values of variables), and returns the value of the form in the environment. It usually looks something like this:

```
(defun eval (form env)
  (cond ((symbolp form)
         (lookup-variable form env))
        ((atom form)
         form)
        ((special-form-p form)
         ...)
        (t (apply (first form)
                  (eval-list (rest form))))))

(defun eval-list (formlist env)
  (if (null formlist)
      nil
      (cons (eval (first form) env)
            (eval-list (rest form) env))))
```

This program is not yet complete. In place of the '...' must be inserted code to handle all of Lisp's special forms. We also need a definition of the representation of environments, and we need to define the functions `lookup-variable` and `apply`. These involve details that are unimportant at this point.

The above interpreter is a *functional* program, and its runtime behavior follows the pattern of function calls and returns in the program being interpreted. For a subset of Lisp restricted to functional constructs, such an interpreter is fine. However, it becomes increasingly hard to maintain the simple structure of the interpreter as we add Common Lisp's special forms for sequencing (`progn`), iteration (`tagbody/go` or `do`), and non-local return (`catch/throw` and `block/return-from`), as well as the parallel constructs that we will introduce.

Using *continuations* allows us to expand the range of constructs that the interpreter can handle with a manageable increase in the complexity of the program. Continuations, which were originally invented to define the semantics of sequential programming constructs (see [4] and [10]), were shown in [9] and related papers to be a very convenient programming tool as well.

A continuation is a function that represents "the rest of the program" as the interpreter progresses. The interpreter's job changes from "evaluate a form in an environment and return the result" to "evaluate a form in an environment and call a continuation with the result." Using continuation passing style,[1] our example becomes:

```
(defun eval (form env cont)
  (cond ((symbolp form)
          (funcall cont (lookup-variable form env)))
         ((atom form)
          (funcall cont form))
         ((special-form-p form)
          ...)
         (t (eval-list (rest form) env
              #'(lambda (args)
                  (apply (first form) args cont))))))

(defun eval-list (formlist env cont)
  (if (null formlist)
      (funcall cont nil)
      (eval (first form) env
        #' (lambda (first-value)
             (eval-list (rest formlist) env
               #' (lambda (rest-values)
                    (funcall cont (cons first-value
                                        rest-values))))))>>
```

It is important to notice that the functions defined above by `lambda` expressions are closures; they contain free references to variables that are lexically bound outside the `lambda` expressions.

In a continuation passing program such as this one, each function that is called with a continuation as an argument ends by calling another function, passing it a new continuation. If the interpreter is run using an ordinary stack-based Lisp system, the stack will grow quite

[1]The reader familiar with continuation passing style will notice that some parts of this code do not pass continuations: for instance the lookup-variable function. We do this to improve the performance of the interpreter by creating fewer unnecessary closures.

large, and any program doing a non-trivial amount of work will cause the system to run out of memory. To avoid this, the Lisp system in which the interpreter is run must detect *tail recursion* and cause stack space to be reused whenever such a call is encountered. While coding the interpreter, the programmer must ensure that all functions called with continuations are tail-recursive.

Let us go through a simple example to illustrate how the continuation passing interpreter works. Suppose we want to evaluate the expression (+ x 3) and print the result. Previously, we would have said

```
(print (eval '(+ x 3) *top-level-env*))
```

where `*top-level-env*` is used to hold the "top-level" environment of values assigned to global variables. Let us assume that it associates x with the value 4. With the continuation passing interpreter, we say

```
(eval '(+ x 3) *top-level-env* #'print)
```

This call to `eval` examines the form `(+ x 3)`. It is not an atom or a special form, so it results in a call to

```
(eval-list '(x 3) *top-level-env*
  #'(lambda (args) (apply #'+ args #'print>>)
```

The quoted expressions in the above call and the rest of this example are used to represent the values that will actually be passed. The original continuation `#'print` has become part of a new continuation (the lambda expression above). Eval-list now calls

```
(eval 'x *top-level-env*
  #'(lambda  (first-value)
       (eval-list '(3) *top-level-env*
         #'(lambda (rest-values)
             (funcall #'(lambda (args)
                           (apply #'+ args #'print))
                      (cons first-value
                            rest-values))))))
```

which has constructed a new continuation that contains the old one buried inside two levels of closures! But now we have called `eval` with an atom, and it calls

```
(lookup-value 'x *top-level-env*)
```

to find the value associated with `x` in `*top-level-env*`. This will return 4. Then `eval` will call

```
(funcall #'(lambda (first-value)
             (eval-list '(3) *top-level-env*
               #'(lambda (rest-values)
                   (funcall #'(lambda (args)
                                 (apply #'+ args #'print)>
                            (cons first-value rest-values)))))
         4)
```

This becomes

```
(eval-list '(3) *top-level-env*
  #'(lambda (rest-values)
      (funcall #'(lambda (args) (apply #'+ args #'print>)
                 (cons 4 rest-values))))
```

so we are making some progress. After several more steps similar to those above, the interpreter will call

```
(funcall #'(lambda (args) (apply #'+ args #'print))
         (cons 4 '(3)))
```

and finally

```
(apply #'+ '(4 3) #'print)
```

The continuation passing version of apply (which we haven't yet defined) will call the continuation `#'print` with the result of applying the function `#' +` to the argument list `'(4 3)`, so we will finally call `(print 7)` and see our answer.

4 An interpreter for Common Lisp

We now extend the simple continuation passing interpreter to one that accepts almost all of Common Lisp. This will be the basis of our parallel Lisp simulator. To avoid discussing various unimportant details, the code described in the next few sections is often a simplification of what actually appears in CSIM.

4.1 Environments

Symbols in Common Lisp programs refer to values based on the rules of *scope* and *extent* as described in [8], ch. 5. While it would be possible to pass in a single `env` variable all of the information needed to resolve any symbol reference, CSIM divides the kinds of references into two classes, *lexical* and *dynamic*, and uses variables `lex-env` and `dyn-env` to store different parts of the environment. The pragmatic reason for this separation is that a call to a new function defined at "top level," which is a frequent occurrence, uses none of the lexical information present in its calling environment, but retains all of the dynamic environment.

Lexical environments are represented by structures with four components:

- variables that are lexically bound, for example as function parameters or by `let`. What is actually stored is an association list (*alist*) of `(`*variable* `.` *value*`)` pairs. Since lexical binding is the default in Common Lisp, most variable references will be found here.
- functions defined by `flet` or `labels`. This slot contains an alist that associates each name with a lexical closure (see definition below), since lexically bound functions can have free variable references.

- **blocks** defined by **block**. Also contain an alist, which is described in more detail in section 4.4.
- **tagbodies** defined by **tagbody** (or implicitly by **prog**, **do**, etc.) This slot contains a list each of whose members is the entire body of a **tagbody** form.

Lexical closures are represented by structures with two components:

- **function**, represented by a **lambda** expression.
- **environment**, a lexical environment.

Dynamic environments are represented by structures with three components:

- **variables** that are "special," and hence dynamically bound (an alist).
- **catches**, information about **catch** forms that have been entered and not yet exited.
- **unwinds**, representing **unwind-protect** forms that are pending.

A new environment is created whenever there is a new piece of information to add to an existing environment. For example, to interpret a `let` form that binds lexical variables, we create a new lexical environment structure, copy the slots that have not changed from the existing environment (**functions**, **blocks**, **tagbodies**), and store in the **variables** slot an alist that begins with the variables being bound and eventually shares the list structure of the variables in the original environment. We create a new environment, rather than change the slots in the existing environment structure, because the extent of each binding in Lisp is finite and the binding must at some point be "undone;" the best way to do this is to preserve the environment existing before the binding.

Sometimes we modify the data structures contained in an environment without changing the environment itself. For example, to interpret `setq` we find a (*variable* . *value*) pair in an environment and destructively modify the value part of this cons-cell.

4.2 The global environment

CSIM does not use the environment structures just described to implement Common Lisp's "global environment," consisting of values and functions assigned to unbound special variables (`symbol-value` and `symbol-function`). When simulating a reference or assignment to an unbound symbol's value, we use **symbol-value**, which lets the simulated program share the global environment of the simulator.

This makes using CSIM more convenient, because assignments to global variables can be made in the ordinary Lisp environment and then be seen by simulated code, or vice versa. Doing this for function definitions would cause difficulties, however (since CSIM provides interpreted definitions for many of the predefined Common Lisp functions), so these are stored on the symbol's property list.

4.3 Function application

Let us now look further into CSIM's `apply` function, which has been mentioned several times but not yet defined. The role of `apply` is to take a function object, a list of argument

values, a lexical and dynamic environment, and a continuation, and to call the continuation with the result of the function applied to the arguments.

The function objects that `apply` allows as its first argument fall into the following classes:

1. Symbols naming primitive Common Lisp functions. These functions are called directly by the simulator.[2]

2. Symbols naming Common Lisp functions that must be treated specially. For example, an instance of `eval` in code being simulated should result in a call to CSIM's `eval`, not the `eval` in the underlying Common Lisp.

3. Symbols naming functions whose definitions should be interpreted. CSIM finds the definition for such a function on the symbol's property list, where it will have been stored as a `lambda` expression by CSIM's version of `defun`, and applies it in a null lexical environment and the current dynamic environment.

4. Explicit `lambda` expressions. These are applied in the current lexical and dynamic environment.

5. Closures. These are represented by structures containing both a `lambda` expression and a lexical environment in which to apply it. The current dynamic environment is used.

Applying a `lambda` expression is fairly straightforward. We create new environments to contain the bindings of the `lambda` expression's variables. (Since some of them may be special variables we may create both a new lexical environment and a new dynamic environment.) In the new environments, we associate the variables with the corresponding values taken from the argument list to `apply`. Finally, we evaluate the body of the `lambda` expression in the new environments. Since its value should be passed to the continuation that was given to `apply`, we use this continuation in the call to `eval` for the body. A skeleton of the code for this is:

```
(let ((new-lex-env ...)
      (new-dyn-env ...))
  (eval <body-of-lambda-expr> new-lex-env new-dyn-env cont))
```

In the actual simulator, the application of `lambda` expressions is more complicated because we interpret Common Lisp's `&optional`, `&rest` and `&aux` parameters, and avoid creating new lexical or dynamic environments when not necessary.

4.4 Special forms

As an example of how continuations simplify the simulation of Common Lisp special forms, let us look at the implementation of `block` and `return-from`. In a program such as

```
(block b1
  (foo (block b 2
         (if p (return-from b1 7) 3))))
```

[2] The dynamic environment in which these calls take place will not correspond to the simulated dynamic environment. The user of CSIM must expect dynamic binding of variables to affect only references that are interpreted by the simulator.

if the value of p is nil, the inner block will return 3 and the outer block will compute (f oo 3). But if p is not nil, the return-from form will cause 7 to be immediately returned from the outer block and foo will not be called. The symbol b1 in the return-from matches the name of the outer block because it is lexically contained within that block, but if the inner block were also named b1 then the return-from would match the inner block's name.

Each lexical environment includes a blocks slot. To interpret a block form, we create a new lexical environment; in the blocks slot of this environment we put a list whose first element represents the block we are interpreting; the rest of the list is the blocks slot from the previous environment. With the block name we associate the continuation for the block, because this represents what we want to do with the value returned by the block, whether it comes from the last form in the block or is supplied by a return-from.

The code to interpret a block form is therefore

```
(defun eval-block (form lex-env dyn-env cont)
  (let ((new-lex-env (copy-lex-env lex-env)))
    (push (cons (block-name form) cont)
          (lex-env-blocks new-lex-env))
    (eval (block-body form) new-lex-env dyn-env cont)))
```

and the code to interpret a return-from form is[3]

```
(defun eval-return-from (form lex-env-dyn-env cont)
  (let ((find-block (assoc (return-block-name form)
                           (lex-env-blocks lex-env))))
    (if find-block
        (eval (return-expr form) lex-env dyn-env
              (cdr find-block))
        (error "No block for ~S" form))))
```

When return-from is seen, the interpreter looks through the list of blocks in the current lexical environment, which will have the innermost blocks listed first. It examines block names (using assoc) until it finds one matching the name in the return-from. The continuation that is associated with this block name is the one to which we want to pass the return value. Therefore we end with a (tail-recursive) call to eval using this continuation. Note that the cont argument to eval-return-from is ignored. This is because return-from never returns a value to its caller; it always passes a value to some other continuation.

If no return-from is encountered in the course of evaluating the body of a block, then the evaluation of (block-body block) will eventually call cont with a value, as expected.

Catch and throw are simulated in a very similar way. Catch saves its tag and continuation in a new dynamic environment, and throw looks for the appropriate continuation by matching its tag to those saved in its dynamic environment.

Unwind-protect is not hard to handle, although it must be coded quite carefully. The main idea is that every time an unwind-protect form is evaluated, a new dynamic environment is created; its unwinds slot contains a list with the cleanup forms and the lexical and dynamic environments in which they must be executed. Upon normal return through

[3]The code as shown here is incomplete, because it doesn't handle unwind-protect forms that may have to be evaluated as a result of a return-from. CSIM does handle this case.

an `unwind-protect` these forms are evaluated in a straightforward way. A non-local exit (caused by `throw`, `go` or `return-from`) causes a change from the current dynamic environment to a previous dynamic environment. When this happens, we evaluate all of the cleanup forms associated with environments between the one we are leaving and the one we are returning to, in the proper order.

Another important special form is `setq`. For the moment, the following code will suffice to simulate (`setq` *var value*) :

```
(defun eval-setq (form lex-env dyn-env cont)
  (eval (third form) lex-env dyn-env
        #'(lambda (value)
            (modify-binding (second form) value lex-env dyn-env)
            (funcall cont value))))
```

`Modify-binding` finds the association-list pair for the variable in the appropriate environment and changes the value. In section 6 we will make some additions to this code.

Most of the remaining special forms of Common Lisp perform various operations on environments; they are straightforward to implement so we omit them from the description here.

4.5 Multiple values

Common Lisp's multiple values are supported by CSIM. We have previously defined a continuation to be a function of one variable, and simulated returning a value from a function call by calling a continuation with the value that is returned. To allow multiple values to be returned, we let a continuation be called with any number of arguments.

Instead of a function with one parameter, we let each continuation be a function with a `&rest` parameter. When the continuation is called, the `&rest` parameter variable is bound to a list of the argument values. It is then straightforward either to use just the first element of this list when only one value is expected, or to use the whole list in the places that allow multiple values.

The initial implementation of CSIM was done without supporting multiple values. When it came time to add this feature (because some programs that we wanted to simulate used multiple values), it took very little effort to do so.

We will not mention multiple values in the remainder of this paper since in general they are not relevant to issues of parallelism.

4.6 Timing statistics

Up to now, we have not made CSIM do anything more than the Lisp system that it is built on. The first feature that we will add is the ability to measure and record "simulated" execution time. This meets one of our initial goals, which is to reflect the timing of computation on an actual or hypothetical machine.

We use a global variable `*time*`, which is initialized to 0 at the start of each "top-level" call to the interpreter. Whenever CSIM performs an operation that reflects work in the simulated machine, it adds an appropriate amount to `*time*`. (Section 8.1 explains how the basic timings are chosen.) When the computation is done, we can see how much work our simulation corresponded to.

A benefit of the simulator is that we can gather some statistics that would be hard to obtain in a real machine without affecting the timings. For example, we keep track of how much work is spent in each function, in addition to the total work done. This cannot generally be done on standard hardware without, for instance, having the compiler generate additional code at each function entry and exit; this extra code will affect the statistics. Worse, from our point of view, it will affect the relative timing of activity in a parallel processor and possibly change the amount of speedup for the program.

CSIM keeps track of time spent in functions in three different ways. The first is the time spent in each function exclusive of the functions that it calls. These timings will add up to the total time spent in the program.

A more useful statistic is obtained by counting all of the computation done in a function, including functions that it calls. When a function recursively calls itself (either directly or with calls to other functions intervening), we must decide whether to charge it only once, or once for each call. CSIM actually does both, because a different useful measure is obtained each way. These are the second and third sets of function timings.

The information needed to compute these timings is stored in extra slots in each dynamic environment structure. One slot contains the name of the current function being simulated; it is used to charge time to just that function. The second slot contains a list of all function calls currently in progress. The third slot contains such a list, but with each function appearing only once.

When a basic operation is simulated, CSIM adds its simulated time to the time for the current function, and the times for functions in the two lists. For each of the three statistics there is a hash table indexed by the function's name and containing its accumulated time.

After a top-level form has been simulated, CSIM optionally prints the timings. The timings for functions in which recursive calls are counted more than once are not useful by themselves (some may be more than 100% of the total simulated time), but when divided by the number of calls to the function, they give the average time spent in that function.

For example, suppose we have the sequence of calls

$$\texttt{FOO} \rightarrow \texttt{FOO} \rightarrow \texttt{FOO} \rightarrow \texttt{BAR},$$

in which each call to FOO takes 10 steps before calling the next FOO or calling BAR, and the call to BAR takes 40 steps. Thus the total computation takes 70 steps. The first statistic would show 30 steps spent in FOO and 40 steps spent in BAR.

The second statistic would show 70 steps spent in FOO (since all the work is done within the toplevel call to FOO), and 40 steps spent in BAR. The third statistic would show an average of 60 steps spent in FOO, since 70 steps are charged to the first call, 60 to the second, and 50 to the third. It would also show 40 steps spent in BAR.

Section 9 will describe how we use these statistics.

5 Parallel Lisp constructs

Parallel Lisp programs are executed on a shared memory multiprocessor by means of a *process queue*, a data structure containing *processes*, which represent computations that may be performed in parallel by the processors in the system. (Although we refer to it as a "queue," another data structure may prove to be a better choice.)

Processors that are idle will remove processes from the queue and execute them; a running process may generate new processes and put them on the queue. When a process finishes, the processor running it becomes idle again and will look for another process to run. A process may also wait for an event, causing it to be suspended and making its processor idle.

There have been several proposed extensions to Lisp to support this model of computation. CSIM provides the following constructs:

1. Qlisp's `qlet` (both regular and `eager` forms) and `qlambda`, described in [3].

2. Multilisp's `future`, `dfuture` and `touch`, defined in [5].

3. Simple test-and-set locks (busy waiting).

We do not yet support the extensions to `catch` and `throw` defined by Qlisp. (Their meaning is currently being revised.)

Multilisp's `future` and `dfuture`, and the `eager` form of Qlisp's `qlet`, use a special kind of data object called a "future" (or sometimes a "promise" or "placeholder"), which represents the undetermined value of an expression that is being computed in another process. Lisp operations that do not depend on the values of their operands (in a well-defined sense) treat a future just as any other data object. A future can be passed as an argument to a function, returned as a value, assigned to a variable, or stored in a data structure; none of these operations depend on its value.

Most of the primitive operations of Common Lisp do depend on the values of their operands, however, so whenever one of these primitives is called the future is said to be *touched*. This causes one of two things to happen: either the process computing the future's value has finished, in which case the value is available to the process touching the future, or the process has not finished; then the touching process will be suspended, and when the value is available it may be resumed.

The use of futures is not without some cost, especially on processor architectures not designed to support them. The primitives that need to touch their arguments must all perform additional work even when those arguments are not futures (just to check whether they are), and every reference to a future costs more, often even after its value has been determined. CSIM, by assigning varying costs to these operations, can indicate how much of a performance penalty this is.

Locks are provided as a low-level synchronization primitive for two reasons: first, they are better suited for certain parallel algorithms than futures (particularly for "in-place" algorithms that destructively modify data structures); and they are needed to write the scheduler, as described in section 6.2.

5.1 Scope and extent issues

The definitions of scope and extent for variables and other objects in Common Lisp require some reinterpretation in parallel Lisp. This was forseen in [8, p. 38], where Steele writes:

> *Behind the assertion that dynamic extents nest properly is the assumption that there is only one program or process. Common Lisp does not address the problems of multiprogramming (timesharing) or multiprocessing (more than one active processor) within a single Lisp environment.*

We have chosen the following policies:

- Lexical variable references behave the same as in Common Lisp, even if the binding of a variable is in a different process from the reference. Thus, in

```
(qlet t ((x (let ((v 5)) (foo v)))
         (y (let ((v 4)) (bar v))))
  ...)
```

  there is no relation between the binding of v in the two processes created by qlet, while in

```
(let ((v 5))
  (qlet t ((x (foo v))
           (y (bar v)))
    ...))
```

  the two references to v are both to the binding established by the let. If one of the processes used setq to change the value of v, the new value would be seen in the other process (and in the body of the qlet).

  If the parameter t in qlet is changed to 'eager, then the process computing the body may return from the qlet even though the processes computing the bindings are still running. In this case, the variable v must remain accessible to these processes. (The same situation can occur if future is used.)

  CSIM has no problem implementing this, because it uses list structure to store lexical environments and never explicitly deallocates them. (They are garbage collected once they are no longer needed.) An efficient parallel Lisp implementation might avoid allocating environments when possible, but will have to use a lexical closure to allow the passing of bindings from a parent process to a child in this manner.

- The dynamic environment of a process cannot be changed by other processes, even when a binding is undone in a process. If we change our first example to

```
(defvar v)
(qlet t ((x (let ((v 5)) (foo v)))
         (y (let ((v 4)) (bar v))))
  ...)
```

  then the two bindings of v are independent, even though they may occur concurrently. The "shallow binding" technique used by many Lisp implementations does not do the right thing in this case; each process would try to store its new value for v into a shared global value cell. Deep binding, on the other hand, does work correctly if each process is provided with its own stack for bindings, and inherits the bindings of its parent process. However, in the case

```
(defvar v)
(let ((v 5))
  (qlet 'eager ((x (foo v))
                (y (bar v)))
    ...))
```

we want the binding of v to be accessible to the processes created by the qlet even after the qlet returns. This is a problem, since the process that established the binding now will undo it. In a stack-based implementation of dynamic binding, even with deep binding, this will not work. CSIM uses list structure to implement its dynamic environments, just as with lexical environments, and hence does what we want.

6 Simulating the parallel machine

Our main concern in simulating a multiprocessor is that we accurately model the order of reads and writes to the shared memory. Although parallel programs that share data generally use synchronization constructs such as futures or locks, we want to produce realistic results for programs that make unsynchronized memory references. (Among other benefits, this will help us find bugs in programs that do not use correct synchronization.)

In sections 3 and 4 we described how our single-processor interpreter keeps track of its progress using continuations. This takes the place of the "control stack" in an ordinary interpreter, and consequently it is very easy to capture the interpreter's state. This design lets us have an interpreter for each processor in the simulated machine, and switch between them whenever we want.

We do this by introducing a new kind of continuation, which we call a *process continuation*. Process continuations are closures with no parameters; their purpose is solely to capture the lexical environment of the interpreter at a point where we wish to switch the simulation to a new processor, so that we can later resume the current processor's simulation. (In [11], continuations created by catch in the then-current version of Scheme were used for much the same purposes as our process continuations.)

For example, the code to handle setq that was presented in section 4.4 is modified in the parallel simulator to

```
(defun eval-setq (form lex-env dyn-env cont)
  (eval (third form) lex-env dyn-env
        #'(lambda (value)
            (switch-processors
              #'(lambda ()
                  (modify-binding (second form) value
                                  lex-env dyn-env)
                  (funcall cont value)>>)))
```

where switch-processors is a. function that does what we have been describing. Its argument is a process continuation that captures the necessary parts of the simulator's state in its free variables. Calling the process continuation will resume the interpretation of the

`setq` form, but the `switch-processors` function can defer this call until the appropriate time to do so.

Process waiting is also simulated using process continuations. When a process needs to wait for an event (such as a future's value being determined, or to call a `qlambda` process closure), the simulator stores a process continuation representing the work to be done after that event happens, in a data structure associated with the waiting process. Calling the process continuation resumes the suspended process.

6.1 Processors

The variable `*number-of-processors*` is used at the beginning of each top-level evaluation to determine how many processors to simulate. Each processor is always running a process, possibly an "idle" process. A processor is represented by a data structure containing its current process and its current simulated time.

The simulated times are the key to deciding when to switch the simulation from one processor to another. As long as CSIM performs operations that can have no effect on processors other than the current one, it continues to simulate the same processor, incrementing that processor's time.[4] The only operations by which one processor can affect others are those that read or write data in shared memory. To make sure that these operations are done in the correct order, CSIM enforces the following rule:

> *Any operation that can affect other processors must be done when the current processor's time is the lowest of any in the system.*

To see why this works, consider two processors, P_1 and P_2, that perform shared-memory operations at times t_1 and t_2, with $t_1 \leq t_2$. Without following the rule above, we might run the simulation of P_2 beyond time t_2 before we have simulated P_1 at time t_1. This would be wrong: for instance, if P_1's operation is a write and P_2's is a read of the same memory location, then we would not read the correct value. (We call this a write/read conflict. Read/write or write/write conflicts cause similar problems.) However, because of the above rule this cannot happen. When we see that P_2 is about to perform a memory operation at time t_2, we stop its simulation. We do not restart it until has the lowest simulation time of any processor (or is equal to others with the same time). At that point, P_1 must have been simulated past time t_1, because if it hasn't been, then its time is less than t_1 and $t_1 \leq t_2$, so P_2's time isn't the lowest.

What this does is *serialize* all of the shared-memory operations that can cause one processor to affect another. We do this for unsynchronized memory operations (i.e., ordinary reads and writes) as well as synchronous operations such as acquiring locks. This ensures that our simulation corresponds to the order of operations that would occur in a real multiprocessor. However, we do not place any restrictions on shared-memory operations performed at the exact same time by two processors. The results of these are unpredictable.

Serialization is implemented by means of a priority queue (called the "run queue") that holds the structures representing processors, sorted in increasing order of simulation time. When the interpreter is about to perform a shared memory operation (for instance, at the call to `switch-processors` above), it updates the data structures for the current process

---

[4] Actually, it increments the global variable *time*, and will store its value back into the processor's structure before switching to a new processor.

and processor and inserts the processor into the run queue. Then, the processor with the lowest simulation time is removed from the run queue and its simulation is resumed.

CSIM's serialization method was chosen because it is easily to implement and prove correct. Since CSIM does not itself attempt to do work in parallel, this is a reasonable choice. Serialization would become a bottleneck if we were to try to speed up CSIM by having it simulate several processors at the same time, and we would probably need to use a more sophisticated mechanism, such as the "time warp" system described in [6].

6.2 The scheduler

As described at the beginning of section 5, we assume there is a queue or some other data structure to hold processes that are ready to run. We call the code that maintains this data structure the *scheduler*, since it decides in what order the processes will run.

Scheduling algorithms are one of our objects of study, and we do not want to build one into the design of our simulator. Instead, we want to make it possible for a user of the simulator to write a scheduler in ordinary Lisp code (not in continuation passing style). CSIM models the execution of the scheduler by simulating it in the same way as other Lisp code.

The scheduler consists of two functions:

- `add-process` is called whenever a new process is created. It is given a process as its argument, and inserts it into whatever queue or other data structure is being used to schedule processes.

- `get-process` is called whenever a processor is idle. It finds a process to run and returns it.

When a processor becomes idle, the simulator creates a temporary "idle" process in which the call to `get-process` takes place. (Since this call is interpreted, there must be a process for it to run in.) Upon return from `get-process`, the new process replaces the idle process.

Currently, CSIM gives the user a choice of two schedulers: FIFO and LIFO. These both organize the runnable processes into a single list; their difference is in which process is chosen by `get-process`. The FIFO ("first-in, first-out") scheduler takes the process that has been in the queue for the longest time, while the LIFO ("last in, first out") scheduler takes the most recent process.

LIFO scheduling, while perhaps counterintuitive at first, has been found to often perform better than FIFO scheduling. Halstead [5] discusses this in some detail, and argues for an "unfair" scheduling policy as a way to reduce memory usage.

LIFO scheduling also allows some optimizations in process management.

1. When a process is about to create a child process and immediately wait for its result, as in the `(qlet t ... )` construct of Qlisp, it can perform an ordinary function call instead, since there is no reason to put a process on the queue, make the processor become idle, and have it then remove the same process right away.

2. When a process finishes and has a list of waiting processes to wake up, its processor can put all but one of them on the process queue and run the last one itself, since it otherwise would become idle and immediately choose the last process that it added.

CSIM has a flag that is turned on by the LIFO scheduler, and turned off by the FIFO scheduler, which enables these optimizations. This interaction between the scheduler and the simulator is needed because creation and termination of processes are simulated directly, not interpreted as the scheduler is.

The FIFO and LIFO schedulers just described both suffer from potential contention for the various locks on the process queue needed to ensure correct operation. We are therefore looking into the use of more sophisticated schedulers that distribute the runnable processes among several queues. In many cases a single-queue scheduler is sufficient, since our goal is to create processes of a large enough granularity so that scheduling does not happen very often. As the number of processors we simulate increases, however, the process size must also increase to avoid contention, and this may reduce the potential speedup of a program.

6.3 Processes

A process is represented by a structure containing its current process continuation, a flag to indicate whether it has terminated, and a list of other processes that are waiting for it (if it has not yet terminated). When interpreting a form that creates a process, such as `qlet` or `future`, the simulator calls a function `create-process` defined as follows:

```
(defun create-process (form lex-env dyn-env new-cont cont)
  (let ((new-proc (make-proc
                   :pcont #'(lambda ()
                              (eval form lex-env dyn-env
                                    new-cont)))))
    (call-user-function 'add-process (list new-proc)
     #' (lambda (v)
          (funcall cont new-proc)))))
```

The `form` argument is what the new process will evaluate, using `lex-env` and `dyn-env` as its initial environment. `New-cont` is the continuation that the new process will call with the value of `form`. `Cont` is the continuation for the parent process. The call to call-user-function tells CSIM to interpret the definition of `add-process` with the new process as an argument, and pass the result to the specified continuation. This continuation returns the new process to the parent, which may have a need to refer to it. (For instance, `qlet` may wait for the new process to finish.)

The continuation `new-cont` called by the new process is always written to end with a call to the function `finish-process`, which wakes up any processes that have decided to wait for the given process to terminate. It does this by calling `add-process` on each of these. After this, the process is done. Its processor becomes idle and will try to find a new process. If we are using the LIFO scheduler described in the previous section, then if there were waiting processes we switch directly to one of them, avoiding a call to both add-process and get-process.

6.4 Process closures

Qlisp defines a new type of object called a *process closure*, which provides both concurrency and synchronization. A call to a process closure may proceed without the caller waiting for the result (but only when the call is in a position where the result value is ignored). Calls

to each process closure are serialized; if one happens while a previous call is still in progress, it is put on a queue.

At present, CSIM implements only the synchronization features of process closures. To do this, we represent a process closure by a structure containing a (first-in, first-out) queue of waiting processes and a closure. When a process closure is called, the calling process is added to the queue. If it is the only one there, it proceeds by calling the closure. Otherwise, its processor becomes idle and calls `get-process` as described above.

When the call to the closure returns, the simulator removes the current process from the process closure's queue. If there are now other processes waiting on the queue, it calls `add-process` to resume the first one.

7 Miscellaneous details

In the previous sections, we omitted certain details in order to simplify the presentation. This section explains several of them.

7.1 Use of symbols

We find it convenient to have the interpreter share symbols with its underlying Common Lisp environment. As mentioned in section 4.2, the values of unbound special variables are shared between the simulator and the program being simulated. Other information about symbols is kept on their property lists, using the following property names:

- `cexpr` is the `lambda` expression for an interpreted function definition. CSIM's version of `defun` sets the value of this property.

- `csubr` is a function to handle a special form. It is called by CSIM's `eval` to handle such a form, with the form, the current lexical and dynamic environments, and the current continuation as arguments. The `def csubr` macro defines such a function. This makes the code more modular, since we do not need to enumerate all of the special forms inside `eval`.

- `esubr` is a function to handle a primitive Lisp function that cannot be called directly, such as `apply`, because its operation needs to be simulated. The arguments to an `esubr` are evaluated normally before the function is called.

- `cinfo` is a list whose `car` is the number of time units that the simulator should charge to interpret the function or special form named by this symbol, and whose cdr indicates which arguments to the function may be passed without being touched. These are both meant mainly for functions that are interpreted by calling the Common Lisp functions directly. Functions that are simulated (using either `cexpr` or `csubr` definitions) do not make use of the `cdr` part of this property. For interpreted `cexpr` definitions, if this property is present it overrides the normal time charged for a function call.

7.2 Preprocessing of definitions

CSIM has its own version of defun. which stores the function definition of a symbol as its `cexpr` property. Before doing so, it preprocesses the function definition to perform the following transformations.

1. Macros are expanded wherever they are recognized. This avoids having to expand them in the interpreter.

2. Parallel Lisp constructs (`qlet`, `future`, etc.) are converted to a form that assigns a unique tag to each process creation point. For example, if a function `f oo` contains several calls to future and the second one is (future expr) then it is converted to (future-tagged f oo2 expr). When future-tagged is interpreted, it is treated just like `future` except that the tag is assigned to the new process that is created. CSIM keeps track of how much time is spent in each call to a process with a given tag. With this information the user can decide whether the processes created at each point in the program are of a reasonable size.

It is possible for a macro to be encountered in interpreted code even after preprocessing; if this occurs, CSIM expands it and then destructively replaces the original form by the expansion. This avoids the overhead of expanding the same expression each time it is encountered.

Many of the basic Common Lisp forms described in [8] are macros. Unfortunately, different implementations of Common Lisp expand these forms in different ways, causing noticeable changes in the times charged by the simulator. Even more of a problem is that the expansions may use implementation-specific functions. Because of this, CSIM must include timing information for functions that are not part of standard Common Lisp, particularly those resulting from expansions of setf.

CSIM also has a special version of `def struct`, so that it can perform preprocessing to define timi ng information for the accessor, constructor, copier and predicate functions of the structure being defined.

7.3 Interpreted primitives

Many Common Lisp functions may be simulated by calling them directly with the values of their argument expressions. CSIM must be careful not to pass futures to these functions, because they are not part of Common Lisp. Therefore in most cases it "touches" arguments before calling a Common Lisp function. This would have to be done anyway in a Lisp system that uses futures, except for functions such as cons that do not depend on the values of their arguments; for those cases we have included a mechanism (the `cinf o` property described above) to avoid unnecessary touches.

Some functions cannot be called directly, however, because they reference objects other than their direct arguments, and these may be futures. Consider, for instance, the `cddr` function (cdr applied twice). Even if we ensure that the argument to cddr is not a future, it may be a cons cell whose `cdr` is a future, so calling Common Lisp's `cddr` would result in an error. CSIM uses an interpreted definition of `cddr` and most other such functions for this reason.

Another class of functions that cannot be called directly is those whose running time depends on the size (or some other properties) of their input. The `equal` and `length` functions are examples of this. CSIM uses interpreted definitions of these functions also.

7.4 Top level

To interact with the user, CSIM provides a read-eval-print loop, but the top-level evaluator does a number of special things. It begins by initializing the processor data structures and clearing all of the statistics counters. Then it creates an initial process whose process continuation is set to call CSIM's `eval` with the form to be evaluated. This process is passed to initialization code for the scheduler, which sets it up as ready to run, with no other processes in the system. One of the processors is then chosen to begin the simulation.

While running, CSIM keeps track of the number of running processes. A process is said to be running between the time it is created and the time it terminates, except when it is suspended to wait for an event (such as a future being determined). If the number of running processes drops to zero, we halt the simulation and return to top level. Usually this happens when the top-level process returns a value (which is then printed) and terminates. But there may still be other running processes at this point, because of futures that have not yet been determined, or for other reasons. In this case, we continue simulation until the number of running processes is zero.

7.5 Memory allocation and garbage collection

CSIM calls the underlying Common Lisp functions (`cons` etc.) to simulate memory allocation by the program we are interpreting, and assumes that each such call takes a constant amount of time in a parallel machine. A real parallel Lisp can achieve this by giving each processor a private pool of free cells to allocate from, so this is realistic.

CSIM does not model garbage collection at all, except to estimate its eventual cost and include this in the simulated times for `cons` and other allocation functions. It assumes that the garbage collector will achieve the same amount of parallelism as the rest of the program.

Parallel garbage collection is an important problem and there are many approaches currently under investigation. However, we view this area of research as orthogonal to our main interests, which are modelling the execution of processes and investigating partitioning and scheduling algorithms.

8 Accuracy and performance

To produce meaningful results, CSIM's timings must approximate those of an actual machine. And to be usable, the simulator must be fairly fast. This section describes the results of some experiments done to see how well it meets these goals.

8.1 Accuracy of simulated times

To derive timings for basic Lisp operations, we compiled and ran a set of small test programs. Each consisted of a loop performing a primitive Lisp operation; one of these was a "no-op" to measure the overhead of the loop code. Subtracting the time for the "no-op" test from the time of each other test, and dividing by the number of iterations of the loop, indicated

how much time was spent in each function being tested. These tests were performed on a single processor of an Alliant FX/8 running Lucid Common Lisp[5] and scaled to a set of small integer values. Here are some of these values:

Lexical var. ref.	1
CDR	1
+	2
EQ	3
Function call	4
Special var. ref.	5
CONS	15
*	17

We then ran several of the Gabriel benchmarks [2], first as ordinary compiled programs and then using CSIM with the timings derived from the test programs. The table below shows, for each program, the compiled time in seconds, the simulated time in units of 10^6 steps, and the ratio of simulated time to compiled time. The compiled times are the average of five runs of each program.

	Compiled Time	Simulated Time	Ratio
boyer	22.06	27.74	1.3
browse	19.63	41.02	2.1
ctak	1.56	3.26	2.1
dderiv	6.99	7.72	1.1
deriv	5.96	7.15	1.2
destructive	2.18	7.89	3.6
div test-1	2.63	4.22	1.6
div test-2	3.44	3.62	1.1
stak	6.09	2.39	0.4
tak	0.53	1.11	2.1
takl	2.04	8.39	4.1
takr	0.72	1.11	1.5

The accuracy of our simulator is reflected by how close the ratios are to each other. They are not as close as we might like, but they are all of the same general order of magnitude. To account for the differences, we can provide several explanations:

- CSIM's interpreter sometimes performs different operations than the compiled code. For example, CSIM does not optimize the evaluation of common subexpressions, and charges for each reference to a variable, whereas in compiled code some of these might be eliminated. The most extreme ratios each have an explanation of this sort:
  - *destructive* contains do loops that the compiler can optimize, while CSIM treats them as ordinary loops performing index computation and conditional branches.

[5]Actually, we used a version of the Qlisp system in development on the Alliant, running on a single processor. In this Lisp, memory allocation and special variable references are somewhat slower than a Lisp designed for only one processor would be.

- *stak* uses special variables, which are quite slow on the version of Lisp that we used. The compiled code uses deep binding, which takes a varying amount of time per reference, while CSIM charges a constant amount of time.
- *takl* does a lot of tail-recursive function calling, which is optimized by the compiler.

- CSIM pretends that garbage collection time is a constant multiple of the time spent in allocation functions, by including it in the cost of these functions. This is not accurate; a copying garbage collector takes time proportional to the amount of memory in use when it is called, which may be large or small depending on the program being tested.

8.2 Speed of the simulator

Next we will compare the speed of CSIM itself with the speed of compiled code that it is simulating. The computation of function timing statistics (see section 4.6) was disabled during these tests; turning it on slows CSIM by an extra factor of 2 or more. We also ran the programs through the Lucid Common Lisp interpreter for comparison.

The times in the table below are all in seconds. The CSIM runtimes are the average of three runs, except for *boyer* and *browse* which were only run once. The runtimes for interpreted and compiled code are the average of five runs.

	CSIM	Interpreted		Compiled	
	Runtime	Time	Ratio	Time	Ratio
boyer	9441.32	1313.54	7.2	22.06	428
browse	12125.03	1142.68	10.6	19.63	618
ctak	488.61	94.65	5.2	1.56	313
dderiv	1105.40	103.66	10.7	6.99	158
deriv	1192.60	116.65	10.2	5.96	200
destructive	2315.31	244.35	9.5	2.18	1062
div test-1	1406.86	207.71	6.8	2.63	535
div test-2	999.59	118.05	8.5	3.44	291
stak	475.61	107.76	4.4	6.09	78
tak	433.87	65.67	6.6	0.53	818
takl	3230.16	582.13	5.5	2.04	1583
takr	456.09	66.67	6.8	0.72	633

During these tests there was some variation in running conditions. Running time on the Alliant generally increases when several programs are executing simultaneously. This is probably due to contention for the cache, which is shared between its processors. This factor makes as much as a 10% difference, so the figures above should be taken as rough approximations.

In some of the tests there was a significant amount of garbage collection. Enough memory was allocated to limit the garbage collection to once every few seconds, but not so much as to cause paging of the Lisp process.

CSIM generally took 300 to 1000 times as long as the compiled version of the code it was interpreting, i.e., 5 to 15 minutes to simulate a second of compiled code.

The comparison with the Lucid interpreter shows much less variation in the speed ratio, reflecting the fact that CSIM and an ordinary interpreter do similar things with a program.

In general, CSIM is about 5 to 10 times slower than the interpreter, which is a reasonable price to pay for the extra work that CSIM does to handle parallel programs.

Both CSIM and the Lucid interpreter spend a lot of time doing storage allocation and garbage collection. The Lucid interpreter dynamically allocates lexical environments just as CSIM does [13], but it uses a stack for dynamic binding and function calls. CSIM spends much of its time creating lexical closures for use as continuations. It runs best when given a lot of free storage, since this decreases the frequency of garbage collection. But the physical memory of the machine provides a limit to the amount of useful storage we can allocate; once this is exceeded and we start paging, performance drops tremendously.

Although these tests simulated only one processor, they are indicative of the times that we get simulating parallel programs, since none of the code to manage concurrency has been removed. The time to simulate a parallel program is roughly proportional to the product of the number of processors we are simulating and the parallel runtime, with the same ratio as above, as long as most of the processors are doing useful work. Simulating idle processors turns out to be more expensive than simulating processors running ordinary code, because they are generally in a loop referencing memory (checking a queue for work to do), and each such reference must be serialized as described in section 6.1. We could probably modify CSIM to avoid this source of inefficiency, but the difference does not seem worth the effort it would require.

9 A Parallel example

As an example of how CSIM is used, we will try to apply parallelism to the *boyer* benchmark. *Boyer* [2, pp. 116–135] is a simple theorem prover that works by rewriting a formula into a canonical form (a structure of nested `if`-expressions), and then applying a tautology checker to the result.

Converting *boyer* to a parallel program is mainly an exercise; it is unlikely that anyone will want to use the result. This is because there are better algorithms to do what *boyer* does, so it would pay to start from scratch and write a good parallel theorem prover. Still, the case of parallelizing an existing sequential program is an important one, and we expect to see it come up fairly often.

We begin with little knowledge of where the program spends its time. The first step, therefore, is to simulate it running as a sequential program on one processor and look at the function timing statistics (section 4.6). Unfortunately, the benchmark as given takes too much time and memory for easy experimentation; a single run through CSIM with statistics gathering turned on takes about 20 hours and causes a large amount of paging. Therefore we will modify it to create a faster test.

The top level of the program is a function called `test`, which first constructs a term by calling

```
(apply-subst
  (quote ((x f (plus (plus a b)
                     (plus c (zero))))
          (y f (times (times a b)
                      (plus c d)))
          (z f (reverse (append (append a b)
```

```
                                    (nil))))
             (u equal (plus a b)
                (difference x y))
             (w lessp (remainder a b)
                (member a (length b)))))
    (quote (implies (and (implies x y)
                         (and (implies y z)
                              (and (implies z u)
                                   (implies u w))))
                    (implies x w))))
```

and then calls **tautp**, the main function of the theorem prover, with this term. Our simplified test case uses instead the term

```
  (apply-subst
    (quote ((x f (plus (plus a b)
                       (plus c (zero))))
            (y f (times (times a b)
                        (plus c d)))
            (z f (reverse (append (append a b)
                                  (nil))))))
    (quote (implies (and (implies x y)
                         (implies y z))
                    (implies x z))))
```

Running this test through CSIM, we get three sets of function timing statistics. First, for each function we have the amount of time spent just in that function:

```
ONE-WAY-UNIFY1                358379  36.5%
REWRITE-WITH-LEMMAS           145404  14.8%
ONE-WAY-UNIFY1-LST            127357  13.0%
REWRITE                       115362  11.7%
REWRITE-ARGS                   91899   9.3%
ONE-WAY-UNIFY                  66324   6.7%
ASSQ                           49480   5.0%
APPLY-SUBST-LST                12504   1.3%
APPLY-SUBST                    10442   1.1%
...
```

Next, we have the time spent in each function including other functions that it calls:

```
TEST                          983065 100.0%
TAUTP                         982079  99.9%
REWRITE                       976180  99.3%
REWRITE-WITH-LEMMAS           973337  99.0%
REWRITE-ARGS                  972961  99.0%
ONE-WAY-UNIFY                 635750  64.7%
ONE-WAY-UNIFY1                608115  61.9%
ONE-WAY-UNIFY1-LST            420197  42.7%
```

ASSQ	179506	18.3%
APPLY-SUBST	38393	3.9%
APPLY-SUBST-LST	37718	3.8%
TAUTOLOGYP	5899	0.6%
...		

Finally, we have the average time per call to each function.

TEST	983065.0 (1 call)
TAUTP	982079.0 (1 call)
REWRITE	4083.7 (2595 calls)
TAUTOLOGYP	3038.5 (13 calls)
REWRITE-ARGS	2486.4 (4626 calls)
REWRITE-WITH-LEMMAS	907.2 (7559 calls)
APPLY-SUBST-LST	280.3 (494 calls)
APPLY-SUBST	231.2 (394 calls)
TRUEP	158.4 (24 calls)
ONE-WAY-UNIFY	115.0 (5527 calls)
FALSEP	110.4 (19 calls)
ASSQ	83.5 (2798 calls)
ONE-WAY-UNIFY1	73.0 (12951 calls)
ONE-WAY-UNIFYl-LST	72.0 (7513 calls)
...	

From these statistics, we see that most of the time is spent in `rewrite` and `one-way-unify` and their subsidiary functions. But the calls to `one-way-unify` are, on the average, much smaller than calls to `rewrite`. This suggests that we should try to parallelize calls to `rewrite`, since this will create processes of larger size and thus reduce the process creation overhead. If this does not achieve enough speedup, we will look at calls to `one-way-unify`.

Notice that the first set of statistics, which is what ordinary "profiling" of the program would produce, tells us that `one-way-unif y1` accounts for a large portion of the execution time, but it does not tell us that calls to this function are parts of higher-level tasks. Thus, it does not tell us as much about the places to look for effective use of parallelism as the second set of statistics does.

Before we investigate `rewrite`, we must notice that *boyer* contains some uses of global (special) variables that would cause improper sharing of data in parallel processes. One of these is easy to fix: the variable `temp-temp`, declared special with a defvar at the beginning of the program, is used only as a local temporary variable in the functions `apply-subst` and one-way-unifyl. By removing the defvar and adding `&aux temp-temp` to the parameter lists of these functions, we avoid the use of the global variable.

The other global variable, `unify-subst`, is a bit more difficult to deal with. It is used in the following way:

```
(defun rewrite-with-lemmas (term 1st)
  (cond ((null 1st)
         term)
        ((one-way-unify term (cadr (car 1st)))
         (rewrite (apply-subst unify-subst (caddr (car 1st)))))
        (t (rewrite-with-lemmas term (cdr 1st)))))
```

Each call to one-way-unify sets unif y-subst to NIL, and then incrementally modifies it (in one-way-unif yl). When one-way-unify returns, unif y-subst contains a list which is referenced by the code shown above, and then there are no further references. Since we are going to parallelize calls to `rewrite`, several processes may be running `rewrite-with-lemmas` at the same time and they should not share the same global variable.

CSIM's definition of dynamic binding (see section 5.1) makes it possible to establish a separate instance of `unify-subst` for each call to `one-way-unify`, as follows:

```
(defun rewrite-with-lemmas (term 1st)
  (let ((unif y-subst nil) )
    (cond ((null 1st)
           term)
          ((one-way-unify term (cadr (car lst)))
           (rewrite (apply-subst unify-subst (caddr (car lst)))))
          (t (rewrite-with-lemmas term (cdr 1st))))))
```

Recall that `unify-subst` is a special variable because of the `defvar` at the beginning of the program. If rewrite-with-lemmas is called concurrently in different processes, they will each perform a dynamic binding of `unify-subst`, which will be invisible to other processes because each establishes a new dynamic environment. Thus, the references to `unif y-subst` in `one-way-unify` will not interfere with each other.

Having made these changes, we now proceed to examine `rewrite` and `rewrite-args`.

```
(defun rewrite (term)
  (cond ((atom term)
         term)
        (t (rewrite-with-lemmas (cons (car term)
                                      (rewrite-args (cdr term)))
                                (get (car term)
                                     (quote lemmas))))))

(defun rewrite-args (1st)
  (cond ((null 1st)
         nil)
        (t (cons (rewrite (car lst))
                 (rewrite-args (cdr lst))))))
```

The main potential for parallelism here is in `rewrite-args`, which performs independent computations on each member of the list given as its argument. We can create a separate process for each one of these. We can also use futures to return a value from `rewrite-args` before these processes finish, which may add some more parallelism.

A single change to the function accomplishes this:

```
(defun rewrite-args (1st)
  (cond ((null 1st)
         nil)
        (t (cons (future (rewrite (car lst)))
                 (rewrite-args (cdr lst))))))
```

When we run the resulting program through CSIM, we get the following results:[6]

# of Proc.	Running Time	Speedup vs. 1 proc.	Speedup vs. serial	Useful Work	Idle Overhead	Other Overhead
1	1367478	1.00	0.72	0.72	0.13	0.15
2	704366	1.94	1.40	0.70	0.15	0.15
3	491812	2.78	2.00	0.67	0.19	0.15
4	396999	3.44	2.48	0.62	0.22	0.16
5	346768	3.94	2.84	0.57	0.27	0.17
10	310896	4.40	3.16	0.32	0.50	0.18
15	315602	4.33	3.12	0.21	0.63	0.17
20	299398	4.57	3.28	0.16	0.68	0.15

CSIM provides the "running time" and "idle overhead" data, and we have computed the other numbers in the table from these. Speedup versus one processor is the time for the parallel program on one processor divided by the time on n processors. Speedup versus the serial program is a more meaningful measure, since it accounts for overhead in the parallel program that we must try to avoid. The serial program's time is 983065 steps, as computed in the earlier simulator run.

The last three columns show the fractions of processor time spent doing useful work and overhead of various sorts. Useful work is the speedup vs. the serial time, divided by the number of processors. This number stays well below 1.00 because of overhead in the parallel program. For each future, the parallel program does extra work to create the future, to add a process to the queue, to remove it when a processor becomes idle, and to reference data indirectly through the future. The costs of future creation and adding processes are part of the "other overhead" above. The costs of finding processes in the queue and removing them are counted in "idle overhead." Idle overhead also counts time spent waiting for the lock on the queue, and time when there is no work for a processor to do.

Beyond about 10 processors, there is simply not enough work to keep all the processors busy, and the idle overhead begins to climb rapidly, while the "other overhead" fraction drops because the idle processors are not doing the operations that are charged to that category.

This is just a first step; with further work on the program, we can try to minimize the overhead of the parallelism we have added, and also find more parallel work as the number of processors increases. The results presented in [12] continue this investigation.

10 Acknowledgements

The original Maclisp simulator for Qlisp [3], written by Dick Gabriel, provided a number of ideas used in CSIM. Ramin Zabih suggested using a continuation passing interpreter as the basis for CSIM. This considerably simplified its previous design.

Hiroshi Okuno, Dan Pehoushek, Arkady Rabinov and Igor Rivin served as guinea pigs and users of CSIM and have contributed many useful ideas and improvements. Carolyn

[6]CSIM's default LIFO scheduler was used for these tests. Different results would be obtained using the FIFO scheduler, or if we changed FUTURE to DFUTURE. This is mainly because a reference to an undetermined future causes extra overhead, and the order in which processes are scheduled decides whether the values of futures are computed by the time they are referenced.

Talcott provided comments on several drafts of this paper, as did some of the people named above.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems. Computer Systems Series,* MIT Press, Cambridge, Massachusetts, 1985.

[3] Richard P. Gabriel and John McCarthy. Queue-based multiprocessing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* pages 25–44, Austin, Texas, August 1984.

[4] Michael J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, New York, 1979.

[5] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems,* 7(4):501–538, October 1985.

[6] David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems,* 7(3), July 1985.

[7] James S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1987.

[8] Guy L. Steele Jr. *Common Lisp; The Language.* Digital Press, Burlington, Massachusetts, 1984.

[9] Guy L. Steele Jr. and Gerald J. Sussman. *LAMBDA: The Ultimate Imperative.* AI Memo 353, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1976.

[10] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[11] Mitchell Wand. Continuation based multiprocessing. In *Conference Record of the 1980 LISP Conference,* Stanford, California, August 1980.

[12] Joseph S. Weening. *Parallel Execution of Lisp Programs.* PhD thesis, Stanford University, Stanford, California. In preparation.

[13] Jon L. White. Personal communication.