Queue-based Multi-processing Lisp

Richard P. Gabriel
John McCarthy

Stanford University

1. Introduction

As the need for high-speed computers increases, the need for multi-processors will be become more apparent. One of the major stumbling blocks to the development of useful multi-processors has been the lack of a good multi-processing language—one which is both powerful and understandable to programmers.

Among the most compute-intensive programs are artificial intelligence (AI) programs, and researchers hope that the potential degree of parallelism in AI programs is higher than in many other applications. In this paper we propose multi-processing extensions to Lisp. Unlike other proposed multi-processing Lisps, this one provides only a few very powerful and intuitive primitives rather than a number of parallel variants of familiar constructs.

Support for this research was provided by the Defense Advanced Research Projects Agency under Contract DARPA/N00039-82-C-0250

2. Design Goals

1. Because Lisp manipulates pointers, this Lisp dialect will run in a *shared-memory* architecture;

2. Because any real multi-processor will have only a finite number of CPU's, and because the cost of maintaining a process along with its communications channels will not be zero, there must be a means to limit the degree of multi-processing at runtime;

3. Only minimal extensions to Lisp should be made to help programmers use the new constructs;

4. Ordinary Lisp constructs should take on new meanings in the multi-processing setting, where appropriate, rather than proliferating new constructs.

5. The constructs should all work in a uni-processing setting (for example, it should be possible to set the degree of multi-processing to 1 as outlined in point 2); and

3. This Paper

This paper presents the added and re-interpreted Lisp constructs, and examples of how to use them are shown. A simulator for the language has been written and used to obtain performance estimates on sample problems. This simulator and some of the problems are be briefly presented.

4. QLET

The obvious choice for a multi-processing primitive for Lisp is one which evaluates arguments to a lambda-form in parallel. **QLET** serves this purpose. Its form is:

$$\begin{array}{l}(\textbf{QLET}\ \mathit{pred}\ ((x_1\, \mathit{arg}_1) \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\quad (x_n\ \mathit{arg}_n)) \\ \qquad\qquad .\ \mathit{body})\end{array}$$

Pred is a predicate that is evaluated before any other action regarding this form is taken; it is assumed to evaluate to one of: (), **EAGER**, or something else.

If *pred* evaluates to (), then the **QLET** acts exactly as a **LET**. That is, the arguments $\mathit{arg}_1 \ldots \mathit{arg}_n$ are evaluated as usual and their values bound to $x_1 \ldots x_n$, respectively.

If *pred* evaluates to non-(), then the **QLET** will cause some multi-processing to happen. Assume *pred* returns something other than () or **EAGER**. Then processes are spawned, one for each arg_i. The process evaluating the **QLET** goes into a wait state: When all of the values $arg_1 \ldots arg_n$ are available, their values are bound to $x_1 \ldots x_n$, respectively, and each form in the list of forms, *body*, is evaluated.

Assume *pred* returns **EAGER**. Then **QLET** acts exactly as above, except that the process evaluating the **QLET** does not wait: It proceeds to evaluate the forms in *body*. But if in evaluating the forms in *body* the value of one of the arguments is required, arg_i, the process evaluating the **QLET** waits. If that value has been supplied already, it is simply used.

To implement **EAGER** binding, the value of the **EAGER** variables could be set to an 'empty' value, which could either be an empty memory location, like that supported by the Denelcor HEP [Smith 1978], or a Lisp object with a tag field indicating an empty or pending object. At worst, every use of a value would have to check for a full pointer.

We will refer to this style of parallelism as **QLET** *application*.

4.1 *Queue-based*

The Lisp is described as 'queue-based' because the model of computation is that whenever a process is spawned, it is placed on a global queue of processes. A scheduler then assigns that process to some processor. Each processor is assumed to be able to run any number of processes, much as a timesharing system does, so that regardless of the number of processes spawned, progress will be made. We will call a process running on a processor a *job*.

The ideal situation is that the number of processes active at any one time will be roughly equal to the number of physical processors available.[1]

The idea behind *pred*, then, is that at runtime it is desirable to control the number of processes spawned. Simulations show a marked dropoff in total performance as the

[1] Strictly speaking this isn't true. Simulations show that the ideal situation depends on the length of time it takes to create a process and the amount of waiting the average process needs to do. If the creation time is short, but realistic, and if there is a lot of waiting for values, then it is better to use some of the waiting time creating active processes, so that no processor will be idle. The ideal situation has no physical processor idle.

number of processes running on each processor increases, *assuming that process creation time is non-zero.*

4.2 *Example* **QLET**

Here is a simple example of the use of **QLET**. The point of this piece of code is to apply the function **CRUNCH** to the n_1^{th} element of the list L_1, the n_2^{th} element of the list L_2, and the n_3^{th} element of the list L_3.

```
(QLET T ((X
          (DO ((L L1 (CDR L))
               (I 1 (1+ I))
               ((= I N1) (CAR L)))))
         (Y
          (DO ((L L2 (CDR L))
               (I 1 (1+ I))
               ((= I N2) (CAR L)))))
         (Z
          (DO ((L L3 (CDR L))
               (I 1 (1+ I))
               ((= I N3) (CAR L))))))
        (CRUNCH X Y Z))
```

4.3 *Functions*

You might ask: Can a function, like **CRUNCH**, be defined to be 'parallel' so that expressions like the **QLET** above don't appear in code? The answer is no.

The reasons are complex, but the primary reason is lexicality. Suppose it were possible to define a function so that a call to that function would cause the arguments to it to be evaluated in parallel. That is, a form like (f $a_1 \ldots a_n$) would cause each argument, a_i, to be evaluated concurrently with the evaluation of the others. In this case, to be safe, one would only be able to invoke f on arguments whose evaluations were independent of each other. Because the definition of a function can be, textually, far away from some of its invocations, the programmer would not know on seeing an invocation of a function whether the arguments would be evaluated in parallel.

Using our formulation, one could define a macro, **PCALL**, such that:

(**PCALL** $f\, a_1 \ldots a_n$)

would accomplish parallel argument evaluation. Of course, this is just a macro for a **QLET** application.

4.4 *A Real Example*

This is an example of a simple, but real, Lisp function. It performs the function of the traditional Lisp function, **SUBST**, but in parallel:

```
(DEFUN QSUBST (X Y Z)
       (COND ((EQ Y Z) X)
             ((ATOM Z) Z)
             (T
              (QLET T ((Q (QSUBST X Y (CAR Z)))
                       (R (QSUBST X Y (CDR Z))))
                    (CONS Q R)))))
```

5. QLAMBDA Closures

In some Lisps (Common Lisp, for example) it is possible to create *closures*: function-like objects that capture their definition-time environment. When a closure is applied, that environment is re-established.

QLET application, as we saw above, is a good means for expressing parallelism that has the regularity of, for example, an underlying data structure. Because a closure is already a lot like a separate process, it could be used as a means for expressing less regular parallel computations.

(**QLAMBDA** *pred* (*lambda-list*) . *body*)

creates a closure. *Pred* is a predicate that is evaluated before any other action regarding this form is taken. It is assumed to evaluate to either (), **EAGER**, or something else. If *pred* evaluates to (), then the **QLAMBDA** acts exactly as a **LAMBDA**. That is, a closure is created; applying this closure is exactly the same as applying a normal closure.

If *pred* evaluates to something other than **EAGER**, the **QLAMBDA** creates a closure that, when applied, is run as a separate process. Creating the closure by evaluating the **QLAMBDA** expression is called *spawning*; the process that evaluates the **QLAMBDA** is called the *spawning process*; and the process that is created by the **QLAMBDA** is called the *spawned process*. When a closure running as a separate process is applied, the separate process is started, the arguments are evaluated by the spawning process, and a message is sent to the spawned process containing the evaluated arguments and a return address. The spawned process does the appropriate lambda-binding, evaluates its body, and finally returns the results to the spawning process. We call a closure that will run or is running in its own process a *process closure*. In short, the expression (**QLAMBDA** non-() ...) returns a process closure as its value.

If *pred* evaluates to **EAGER**, then a closure is created which is immediately spawned. It lambda-binds empty binding cells as described earlier, and evaluation of its body starts immediately. When an argument is needed, the process either has had it supplied or it blocks. Similarly, if the process completes before the return address has been supplied, the process blocks.

This curious method of evaluation will be used surprisingly to write a parallel **Y** function!

5.1 *Value-Requiring Situations*

Suppose there are no further rules for the timing of evaluations than those given, along with their obvious implications; have we defined a useful set of primitives?

No. Consider the situation:

(**PROGN** (**F** X) (**G** Y))

If **F** happens to be bound to a process closure, then the process evaluating the **PROGN** will spawn off the process to evaluate (**F** X), wait for the result, and then move on to evaluate (**G** Y), throwing away the value **F** returned. If this is the case, it is plain that there is not much of a reason to have process closures.

Therefore we make the following behavioral requirement: If a process closure is called in a value-requiring context, the calling process waits; and if a process closure is called in

a value-ignoring situation, the caller does not wait for the result, and the callee is given a void return address.

For example, given the following code:

```
(LET ((F (QLAMBDA T (Y)(PRINT (* Y Y)))))
     (F 7)
     (PRINT (* 6 6)))
```

there is no *a priori* way to know whether you will see 49 printed before or after 36.[2]

To increase the readability of code we introduce two forms, which could be defined as macros, to guarantee a form will appear in a value-requiring or in a value-ignoring position.

(**WAIT** *form*)

will evaluate *form* and wait for the result;

(**NO-WAIT** *form*)

will evaluate *form* and not wait for the result.

For example,

```
(PROGN
 (WAIT form1)
 form2)
```

will wait for $form_1$ to complete.

[2] We can assume that there is a single print routine that guarantees that when something is printed, no other print request interferes with it. Thus, we will not see 43 and then 96 printed in this example.

5.2 *Applying a Process Closure*

Process closures can be passed as arguments and returned as values. Therefore, a process closure can be in the middle of evaluating its body given a set of arguments when it is applied by another process. Similarly, a process can apply a process closure in a value-ignoring position and then immediately apply the same process closure with a different set of arguments.

Each process closure has a queue for arguments and return addresses. When a process closure is applied, the new set of arguments and the return address is placed on this queue. The body of the process closure is evaluated to completion before the set of arguments at the head of the queue is processed.

We will call this property *integrity*, because a process closure is not copied or disrupted from evaluating its body with a set of arguments: Multiple applications of the same process closure will not create multiple copies of it.

6. CATCH and QCATCH

So far we have discussed methods for spawning processes and communicating results. Are there any ways to kill processes? Yes, there is one basic method, and it is based on an intuitively similar, already-existing mechanism in many Lisps.

CATCH and **THROW** are a way to do non-local, dynamic exits within Lisp. The idea is that if a computation is surrounded by a **CATCH**, then a **THROW** will force return from that **CATCH** with a specified value, terminating any intermediate computations.

(**CATCH** *tag form*)

will evaluate *form*. If *form* returns with a value, the value of the **CATCH** expression is the value of the *form*. If the evaluation of *form* causes the form

(**THROW** *tag value*)

to be evaluated, then **CATCH** is exited immediately with the value *value*. **THROW** causes all special bindings done between the **CATCH** and the **THROW** to revert. If

there are several **CATCH**'s, the **THROW** returns from the **CATCH** dynamically closest with a tag **EQ** to the **THROW** tag.

6.1 CATCH

In a multi-processing setting, when a **CATCH** returns a value, all processes that were spawned as part of the evaluation of the **CATCH** are killed at that time.

Consider:

```
(CATCH 'QUIT
       (QLET T ((X
                 (DO ((L L1 (CDR L)))
                     ((NULL L) 'NEITHER)
                     (COND ((P (CAR L))
                            (THROW 'QUIT L1)))))
                (Y
                 (DO ((L L2 (CDR L)))
                     ((NULL L) 'NEITHER)
                     (COND ((P (CAR L))
                            (THROW 'QUIT L2))))))
          X))
```

This piece of code will scan down L_1 and L_2 looking for an element that satisfies **P**. When such an element is found, the list that contains that element is returned, and the other process is killed, because the **THROW** causes the **CATCH** to exit with a value. If both lists terminate without such an element being found, the atom NEITHER is returned.

Note that if L_1 and L_2 are both circular lists, but one of them is guaranteed to contain an element satisfying **P**, the entire process terminates.

If a process closure was spawned beneath a **CATCH** and if that **CATCH** returns while that process closure is running, that process closure will be killed when the **CATCH** returns.

6.2 QCATCH

(**QCATCH** *tag form*)

QCATCH is similar to **CATCH**, but if the *form* returns with a value (no **THROW** occurs) and there are other processes still active, **QCATCH** will wait until they all finish. The value of the **QCATCH** is the value of *form*. For there to be any processes active when *form* returns, each one had to have been applied in a value-ignoring setting, and therefore all of the values of the outstanding processes will be duly ignored.

If a **THROW** causes the **QCATCH** to exit with a value, the **QCATCH** kills all processes spawned beneath it.

We will define another macro to simplify code. Suppose we want to spawn the evaluation of some form as a separate process. Here is one way to do that:

```
((LAMBDA (F)
         (F) T)
 (QLAMBDA T () form))
```

A second way is:

```
(FUNCALL (QLAMBDA T () form))
```

We will chose the latter as the definition of:

```
(SPAWN form)
```

Notice that **SPAWN** combines spawning and application.

Here are a pair of functions which work together to define a parallel **EQUAL** function on binary trees:

```
(DEFUN EQUAL (X Y)
       (QCATCH 'EQUAL
               (EQUAL-1 X Y)))
```

EQUAL uses an auxiliary function, **EQUAL-1**:

```
(DEFUN EQUAL-1 (X Y)
       (COND ((EQ X Y))
             ((OR (ATOM X)
                  (ATOM Y))
              (THROW 'EQUAL ()))
             (T
              (SPAWN (EQUAL-1 (CAR X)(CAR Y)))
              (SPAWN (EQUAL-1 (CDR X)(CDR Y)))
              T)))
```

The idea is to spawn off processes that examine parts of the trees independently. If the trees are not equal, a **THROW** will return a () and kill the computation. If the trees are equal, no **THROW** will ever occur. In this case, the main process will return T to the **QCATCH** in **EQUAL**. This **QCATCH** will then wait until all of the other processes die off; finally it will return this T.

6.3 THROW

THROW will throw a value to the **CATCH** above it, and processes will be killed where applicable. The question is, when a **THROW** is seen, exactly which **CATCH** is thrown to and exactly which processes will be killed?

The processes that will be killed are precisely those processes spawned beneath the **CATCH** that receives the **THROW** and those spawned by processes spawned beneath those, and so on.

The question boils down to which **CATCH** is thrown to. To determine that **CATCH**, find the process in which the **THROW** is evaluated and look up the process-creation chain to find the first matching tag.

If you see a code fragment like:

(**QLAMBDA** T () (**THROW** *tag value*))

the **THROW** is evaluated within the **QLAMBDA** process closure, so look at the process in which the **QLAMBDA** is created to start searching for the proper **CATCH**. Thus, if you apply a process closure with a **THROW** in it, the **THROW** will be to the first

CATCH with a matching tag *in the process chain that the* **QLAMBDA** *was created in*, not in the current process chain.

Thus we say that **THROW** throws dynamically by creation.

7. UNWIND-PROTECT

When **THROW** is used to terminate a computation, there may be other actions that need to be performed before the context is destroyed. For instance, suppose that some files have been opened and their streams lambda-bound. If the bindings are lost, the files will remain open until the next garbage collection. There must be a way to gracefully close these files when a **THROW** occurs. The construct to do that is **UNWIND-PROTECT**.

(**UNWIND-PROTECT** *form cleanup*)

will evaluate *form*. When *form* returns, *cleanup* is evaluated. If *form* causes a **THROW** to be evaluated, *cleanup* will be performed anyway. Here is a typical use:

```
(LET ((F (OPEN "FOO.BAR")))
     (UNWIND-PROTECT (READ-SOME-STUFF) (CLOSE F)))
```

In a multi-processing setting, when a cleanup form needs to be evaluated because a **THROW** occurred, the process that contains the **UNWIND-PROTECT** is retained to evaluate all of the cleanup forms for that process before it is killed. The process is placed in an un-killable state, and if a further **THROW** occurs, it has no effect until the current cleanup forms have been completed,.

Thus, if control ever enters an **UNWIND-PROTECT**, it is guaranteed that the cleanup form will be evaluated. Dynamically nested **UNWIND-PROTECT**'s will have their cleanup forms evaluated from the inside-out, even if a **THROW** has occurred.

To be more explicit, recall that the **CATCH** that receives the value thrown by a **THROW** performs the kill operations. The **UNWIND-PROTECT** cleanup forms are evaluated in un-killable states by the appropriate **CATCH** *before* any kill operations are performed. This means that the process structure below that **CATCH** is left in tact until the **UNWIND-PROTECT** cleanup forms have completed.

7.1 *Other Primitives*

One pair of primitives is useful for controlling the operation of the processes as they are running; they are **SUSPEND-PROCESS** and **RESUME-PROCESS**. The former takes a process closure and puts it in a wait state. This state cannot be interrupted, except by a **RESUME-PROCESS**, which will resume this process. This is useful if some controlling process wishes to pause some processes in order to favor some process more likely to succeed than these.

A use for **SUSPEND-PROCESS** is to implement a general locking mechanism, which will be described later.

7.2 *An Unacceptable Alternative*

There is another approach that could have been taken to the semantics of:

(**QLAMBDA** *pred* (*lambda-list*) . *body*)

Namely, we could have stated that the arguments to a process closure could trickle in, some from one source and some from another. Because a process closure could then need to wait for arguments from several sources, we could use this behavior as a means to achieve the effects of **SUSPEND-PROCESS**. That is, we could apply a process closure which requires one argument to no arguments; the process closure would then need to wait for an argument to be supplied. Because we would not supply that argument until we wanted the process to continue, supplying the argument would achieve **RESUME-PROCESS**.

This would be quite elegant, but for the fact that process closures would then be able to get arguments from anywhere chaotically. We would have to abandon the ability to know the order of variable-value pairing in the lambda-binding that occurs in process closures. For instance, if we had a process closure that took two arguments, one a number and the other a list, and if one argument were to be supplied by one process and the second by another, there would be no way to cause one argument to arrive at the process closure before the other, and hence one would not be sure that the number paired with the variable that was intended to have a numeric value.

One could use keyword arguments [Steele 1984] in this case, but that would not solve all the problems with this scheme. How could &**REST** arguments be handled? There would be no way to know when all of the arguments to the process closure had been

supplied. Suppose that a process wanted to send 5 values to a process closure that needed exactly 5 arguments; if some other process had sent 2 to that process closure already, how could one require that the first 3 of the 5 sent would not be bundled with the 2 already sent to supply the process closure with random arguments?

In short, this alternative is unacceptable.

8. The Rest of the Paper

This completes the definition of the extensions to Lisp. Although these primitives form a complete set—any concurrent algorithm can be programmed with only these primitives along with the underlying Lisp—a real implementation of these extensions would supply further convenient functions, such as an efficient locking mechanism.

The remainder of this paper will describe some of the tricky things that can be done in this language, and it will present some performance studies done with a simple simulator.

9. Resource Management

We've mentioned that we assume a shared-memory Lisp, which implies that many processes can be accessing and updating a single data structure at the same time. In this section we show how to protect these data structures with critical sections to allow consistent updates and accesses.

The key is closures. We spawn a process closure which is to be used as the sole manager of a given resource, and we conduct all transactions through that closure. We illustrate the method with an example.

Suppose we have an application where we will need to know for very many n whether $\exists\ i$ s.t. $n = \mathrm{Fib}(i)$, where **Fib** is the Fibonacci function. We will call this predicate *Fib-p*. Suppose further that we want to keep a global table of all of the Fibonacci argument/value pairs known, so that *Fib-p* will be a table lookup whenever possible. We can use a variable, $*\mathbf{V}*$, which has a pair—a cons cell—as its value with the **CAR** being i and the **CDR** being n, and $n = \mathbf{Fib}(i)$, such that this is the largest i in the table. We imagine filling up this table as needed, using it as a cache, but the variable $*\mathbf{V}*$ is used in a quick test to decide whether to use the table rather than Fibonacci function to decide *Fib-p*.

We will ignore the details of the table manipulation and discuss only the variable $*\mathbf{V}*$. When a process wants to find out the highest Fibonacci number in the table, it simply will

do (**CDR** ∗**V**∗). If a process wants to find out the pair (i . Fib(i)), it had better do this indivisibly because some other processes might updating ∗**V**∗ concurrently.

We assume that we do not want to **CONS** another pair to update ∗**V**∗—we will destructively update the pair. Thus, we do not want to say:

```
...
(SETQ *V* (CONS arg val))
...
```

Here is some code to set up the ∗**V**∗ handler:

```
(SETQ *V-HANDLER* (QLAMBDA T (CODE) (CODE *V*)))
```

The idea is to pass this process closure a second closure which will perform the desired operations on its lone argument; the ∗**V**∗ handler passes ∗**V**∗ to the supplied closure.

Here is a code fragment to set up two variables, I and J, which will receive the values of the components of ∗**V**∗, along with the code to get those values:

```
(LET ((I ())(J ()))
     (*V-HANDLER* (LAMBDA (V)
                          (SETQ I (CAR V))
                          (SETQ J (CDR V))))
     ...)
```

Because the process closure will evaluate its body without creating any other copies of itself, and because all updates to ∗**V**∗ will go through ∗**V-HANDLER**∗, I and J will be such that J = Fib(I).

The code to update the value of ∗**V**∗ would be:

```
...
(*V-HANDLER* (LAMBDA (V)
                     (SETF (CAR V) arg)
                     (SETF (CDR V) val)))
...
```

If the process updating ∗**V**∗ does not need to wait for the update, this call can be put in a value-ignoring position.

9.1 *Fine Points*

If the process closure that controls a resource is created outside of any **CATCH** or **QCATCH** that might be used to terminate subordinate process closures, then once the process closure has been invoked, it will be completed. If this process closure is busy when it is invoked by some process, then even if the invoking process is killed, the invocation will proceed. Thus requests on a resource controlled by this process closure are always completed. Another way to guarantee that a request happens is to put it inside of an **UNWIND-PROTECT**.

10. Locks

When we discussed **SUSPEND-PROCESS** and **RESUME-PROCESS** we mentioned that a general locking mechanism could be implemented using **SUSPEND-PROCESS**. Here is the code for this example:

```
(DEFMACRO GET-LOCK ()
          '(CATCH 'FOO
                  (PROGN
                   (LOCK
                    (QLAMBDA T (RES)(THROW 'FOO RES)))
                   (SUSPEND-PROCESS))))
```

When **SUSPEND-PROCESS** is called with no arguments, it puts the currently running job (itself) into a wait state.

```
1 (LET ((LOCK
2        (QLAMBDA T (RETURNER)
3         (CATCH LOCKTAG
4                (LET ((RES (QLAMBDA T () (THROW 'LOCKTAG T))))
5                 (RETURNER RES)
6                 (SUSPEND-PROCESS))))))
```

```
 7        (QLET T ((X
 8                  (LET ((OWNED-LOCK (GET-LOCK)))
 9                       (DO ((I 10 (1- I)))
10                           ((= I 0)
11                            (OWNED-LOCK) 7))))
12                  (Y
13                   (LET ((OWNED-LOCK (GET-LOCK)))
14                        (DO ((I 10 (1- I)))
15                            ((= I 0)
16                             (OWNED-LOCK) 8))))))
17         (LIST X Y))
```

The idea is to evaluate a GET-LOCK form, which in this case is a macro, that will return when the lock is available; at that point, the process that called the GET-LOCK form will have control of the lock and, hence, the resource in question. GET-LOCK returns a function that is invoked to release the lock.

Lines 7–17 are the test of the locking mechanism: The **QLET** on line 7 spawns two processes; the first is the **LET** on lines 8–11; the second is the **LET** on lines 13–16. Each process will attempt to grab the lock, and when a process has that lock, it will count down from 10, release the lock, and return a number—either 7 or 8. The two numbers are put into a list that is the return value for the test program.

As we mentioned earlier, when a process closure is evaluating its body given a set of arguments, it cannot be disrupted—no other call to that process closure can occur until the previous calls are complete. To implement a lock, then, we must produce a process closure that will return an unlocking function, but which will not actually return!

GET-LOCK sets up a **CATCH** and calls the LOCK function with a process closure that will return from this **CATCH**. The value that the process closure throws will be the function we use to return the lock. We call LOCK in a value-ignoring position so that when the lock is finally released, LOCK will not try to return a value to the process evaluating the GET-LOCK form. The **SUSPEND-PROCESS** application will cause the process evaluating the GET-LOCK form to wait for the **THROW** that will happen when LOCK sends back the unlocking function.

LOCK takes a function, the RETURNER function, that will return the unlocking

function. LOCK binds RES to a process closure that throws to the **CATCH** on line 3. This process closure is the function that we will apply to return the lock. The RETURNER function is applied to RES, which throws RES to the catch frame with tag FOO. Because (RETURNER RES) appears in a value-ignoring position, this process closure is applied with no intent to return a value. Evaluation in LOCK proceeds with the call to **SUSPEND-PROCESS**.

The effect is that the process closure that will throw to LOCKTAG—and which will eventually cause LOCK to complete—is thrown back to the caller of GET-LOCK, but LOCK does not complete. No other call to LOCK will begin to execute until the **THROW** to LOCKTAG occurs—that is, when the function, OWNED-LOCK, is applied.

Hence, exactly one process at a time will execute with this lock.

The key to understanding this code is to see that when a **THROW** occurs, it searches up the process-creation chain that reflects dynamically scoped **CATCH**'s. Because we spawned the process closure in GET-LOCK beneath the **CATCH** there, the **THROW** in the process closure bound to RETURNER will throw to that **CATCH**, ignoring the one in LOCK. Similarly, the **THROW** that RES performs was created underneath the **CATCH** in LOCK, and so the process closure that throws to LOCKTAG returns from the **CATCH** in LOCK.

10.1 *Reality.*

As mentioned earlier, a real implementation of this language would supply an efficient locking mechanism. We have tried to keep the number of primitives down to see what would constitute a minimum language.

11. Killing Processes

We've seen that a process can commit suicide, but is there any way to kill another process? Yes; the idea is to force a process to commit suicide. Naturally, everything must be set up correctly.

We'll show a simple example of this 'bomb' technique.

Here is the entire code for this example:

```
1  (DEFUN TEST ()
2   (LET ((BOMBS ()))
```

```
 3  (LET ((BOMB-HANDLER
 4         (QLAMBDA T (TYPE ID MESSAGE)
 5                  (COND ((EQ TYPE 'BOMB)
 6                         (PRINT '(BOMB FOR ,ID))
 7                         (PUSH '(,ID . ,MESSAGE) BOMBS))
 8                        ((EQ TYPE 'KILL)
 9                         (PRINT '(KILL FOR ,ID))
10                         (FUNCALL
11                          (CDR (ASSQ ID BOMBS)))
12                         T)))))
13   (QLET 'EAGER ((X
14                  (CATCH 'QUIT (TESTER BOMB-HANDLER 'A)))
15                 (Y
16                  (CATCH 'QUIT (TESTER BOMB-HANDLER 'B))))
17         (SPAWN
18          (PROGN (DO ((I 10. (1- I)))
19                     ((= I 0)
20                      (PRINT '(KILLING A))
21                      (BOMB-HANDLER 'KILL 'A ()))
22                     (PRINT '(COUNTDOWN A ,I)))
23                 (DO ((I 10. (1- I)))
24                     ((= I 0)
25                      (PRINT '(KILLING B))
26                      (BOMB-HANDLER 'KILL 'B ()))
27                     (PRINT '(COUNTDOWN B ,I)))))
28         (LIST X Y)))))

29  (DEFUN TESTER (BOMB-HANDLER LETTER)
30         (BOMB-HANDLER 'BOMB LETTER
31                       (QLAMBDA T () (THROW 'QUIT LETTER)))
32         (DO ()(()) (PRINT LETTER)))
```

First we set up a process closure which will collect bombs and explode them. Line 2 defines the variable that will hold the bombs. A bomb is an ID and a piece of code. Lines 3–12 define the bomb handler. It is a piece of code that takes a message type, an

ID, and a message It looks at the type; if the type is BOMB, then the message is a piece of code. The ID/code pair is placed on the list, BOMBS. If the type is KILL, then the ID is used to find the proper bomb and explode it.

Lines 13–28 demonstrate the use of the bomb-handler. Lines 14 and 16 are **CATCH**'s that the bombs will kill back to. Two processes are created, each running TESTER. TESTER sends a bomb to **BOMB-HANDLER**, which is a process closure that will throw back to the appropriate **CATCH**. Because the process closure is created under one of two **CATCH**'s, the **THROW** will kill the intermediate processes. The main body of TESTER is an infinite loop that prints the second argument, which will either be the letter **A** or the letter **B**.

The **QLET** on line 13 is eager. Unless something kills the two processes spawned as argument calculation processes, neither X nor Y will ever receive values. But because the **QLET** is eager, the **SPAWN** on line 17 will be evaluated. This **SPAWN** creates a process closure that will kill the two argument processes.

The result of TEST is (**LIST** X Y), which will block while waiting for values until the argument processes are killed.

The killing process (lines 17–27) counts down from 10, kills the first argument process, counts down from 10 again, and finally kills the second argument process.

To kill the argument process, the **BOMB-HANDLER** is called with the message type KILL and the name of the process as the ID. The **BOMB-HANDLER** kills a process by searching the list, BOMBS, for the right bomb (which is a piece of code) and then **FUNCALL**ing that bomb.

Because a process closure is created for each call to TESTER (line 31), and because one is spawned dynamically beneath the **CATCH** on line 14 and the other beneath the **CATCH** on line 16, the **BOMB-HANDLER** will not be killed by the **THROW**. When the process that is printing **A** is killed, the corresponding **THROW** throws **A**. Similarly for the process printing **B**.

The value of TEST is (A B). Of course there is a problem with the code, which is that the **BOMB-HANDLER** is not killed when TEST exits.

12. Eager Process Closures

We saw that EAGER is a useful value for the predicate in **QLET** applications, that is, in constructions of this form:

(**QLET** *pred* ((x_1 arg_1)
⋮
(x_n arg_2))
. *body*)

But it may not be certain what use it has in the context of a process closure.

When a process closure of the form:

(**QLAMBDA** 'EAGER (*lambda-list*) . *body*)

is spawned, it is immediately run. And if it needs arguments or a return address to be supplied, it waits.

Suppose we have a program with two distinct parts: The first part takes some time to complete and the second part takes some large fraction of that time to initialize, at which point it requires the result of the first part. The easiest way to accomplish this is to start a eager process closure, which will immediately start running its initialization. When the first part is ready to hand its result to the process closure, it simply applies the process closure.

Here is an example of this overlapping of a lengthy initialization with a lengthy computation of an argument:

```
(LET ((F (QLAMBDA 'EAGER (X)
                  [Lengthy Initialization]
                  (OPERATE-ON X))))
     (F [Lengthy computation of X]))
```

There are other ways to accomplish this effect in this language, but this is the most flexible technique.

12.1 *A Curious Example.*

A curious example of this arises when the **Y** function is being defined in this language.

The **Y** function is the applicative version of the **Y** combinator, which can be used to define **LABELS** (see Scheme [Steele 1978], [Sussman 1975]). We will briefly review the problem that **Y** solves.

Suppose you write the code:

(**LET** ((CONS (**LAMBDA** (X Y) (*CONS* Y X)))) ...),

will *CONS* refer to the CONS being defined or to the built-in **CONS**? The answer is that it will refer to the built-in **CONS**, and this is not a non-terminating definition. It defines a constructor that builds lists in the **CAR** rather than the traditional **CDR** direction.

The idea is that the **LAMBDA** creates a closure that captures the environment at the time of the closure creation, but this environment does not contain the binding for CONS because the process has not gotten that far yet—it is evaluating the form that will be the value to place in the binding for CONS that it is about to make.

Suppose, though, that you want to define factorial in this style. You cannot write:

```
(LET ((FACT
       (LAMBDA (N)
        (COND ((ZEROP N) 1)
              (T (* N (FACT (1- N))))))))
 ...)
```

because the recursive call to **FACT** refers to some global definition, which presumably does not exist. Traditionally there is a **LAMBDA**-like form, called **LABELS** which gets around this by creating a special environment and then re-stuffing the bindings appropriately, but there is a way to avoid introducing **LABELS**, at the expense of speed.

There is a function, called the **Y** function, that allows one to define a recursive function given an abstraction of the 'normal' definition of the recursive function. Here is an example of the abstract version of FACT that we would need:

```
F = (LAMBDA (G)
      (LAMBDA (N)
       (COND ((ZEROP N) 1)
             (T (* N (G (1- N)))))))
```

The key property of **Y** is: $\forall f, \mathbf{Y}(f) = f(\mathbf{Y}(f))$.

If we were to pass **F** the mathematical function *fact*, then $\mathbf{F}(fact) = fact$ in the mathematical sense. That is, *fact* is a fixed point for **F**. If we define FACT to be **Y**(F), FACT is also a fixed point for **F**, using the property given above. Actually, **Y** produces the least fixed point of **F**, but you can read about that in a textbook.

The definition of **Y** in Lisp is:

```
(DEFUN Y (F)
 (LET ((H (LAMBDA (G)
            (F (LAMBDA (X)
                 (FUNCALL (G G) X))))))
   (LAMBDA (X) (FUNCALL (H H) X))))
```

We can trace through the operation of **Y** briefly. **Y**(F) returns a function that looks like:

```
(LAMBDA (X) (FUNCALL (H H) X))
```

with H that looks like:

```
(LAMBDA (G)
 (F (LAMBDA (X)
      (FUNCALL (G G) X))))
```

and F is bound to **F** above. What does (H H) return? Well, it is F applied to some function, so it returns the inner **LAMBDA** closure—(**LAMBDA** (N) ...)—in **F** above, which will be applied to X: a good sign.

H takes a function—itself in this case—and binds it to the variable G. We can substitute H for G throughout to get F being applied to:

(**LAMBDA** (X) (**FUNCALL** (H H) X))

But **F** simply makes up a closure that binds the variable G to the above closure and returns it. So if evaluation ever applies G to something, it simply applies the above closure to its argument. G would be applied to something in the event that a recursive call was to be made. H is still bound as before, and the F within the H closure is bound to the code for **F**. Thus we end up in the same situation as we were at the outset (that is what the property $\forall f, \mathbf{Y}(f) = f(\mathbf{Y}(f))$ means!).

The machinations of this construction have the effect of providing another *approximation* to the function *fact* as it is needed, by carefullly packaging up new closures that will present the code for **F** over and over as needed.

This is pretty expensive in time and space. It turns out that we can define **QY** as follows:

```
(DEFUN QY (F)
 (LET ((H (LAMBDA (G)
            (F (QLAMBDA 'EAGER (X)
                 (FUNCALL (G G) X))))))
      (QLAMBDA 'EAGER (X)
       (CATCH (NCONS ()) (FUNCALL (H H) X)))))
```

QY is just like **Y**, except that the major closures are eager process closures, and there is a **CATCH** at the toplevel. The eager process closure at the toplevel will run until it blocks, which means until it needs the value of X. So (H H) will begin to be evaluated immediately. Likewise, subsequent applications of **F** will start off some pre-processing. Essentially, **QY** will start spawning off processes that will be pre-computing the approximations spontaneously. They block when they need return addresses, but they are ready to go when the main process, the one calculating factorial, gets around to them.

The **CATCH** stops the spawning process when we get to the end.

The performance of **QY** is between that of **Y** and **LABELS**, because **QY** *pipelines* the creation of the approximations.

13. Performance

The whole point of this language is to provide high performance for Lisp programs. Because the speed of light and the size of objects needed to build circuits limits the expected speed of a single-processor computer, we need multi-processors to achieve higher speeds than these limits imply.

If the language provided cannot achieve speedups that improve as we increase the number of processors in a multi-processor configuration, then there is no point in using that language or in pursuing its implementation.

Because there are few true multi-processors on the market and because it is difficult to vary the parameters of the performance of hardware to study the effects of the variations, we have chosen to write a rudimentary simulator for this language. With this simulator we have performed a small number of experiments. In this section we will briefly describe that simulator and review some of the results.

13.1 *The Simulator*

The simulator simulates a multi-processor with a shared memory and a variable number of processors. So far we have simulated configurations with 1 to 50 processors. The simulator is an interpreter for the language described above. Processes are scheduled on the least busy processor as they are invoked, but no other load-balancing is performed. The scheduling is round-robin, but the effect of queues of jobs within a processor is modelled—so that a process in a wait state does not impact the reported performance of the multi-processor (much).

Two important aspects of the expected performance are modelled carefully: the time that it takes to create a process and the time that it takes for the same memory location to be accessed simultaneously by several processors. In creating processes to achieve **QLET** application, closures must be created to capture the environment that the argument forms must evaluate within. This can be a significant overhead in a real implementation. Aspects that are not well-modelled are the communications overhead (sending and receiving messages), simultaneous access to the same memory region by two processors, and scheduling overhead. The various overheads associated with processes can be modelled, to some extent, by increased process creation times.

Lisp functions in the underlying Lisp take one unit of time; if one wishes to be more exact in the simulator these functions must be re-written in the interpreted multi-processing Lisp. Function calls take around 3 units of time and assignments 1 unit, for example.

The simulator comprises 60,000 characters of Lisp code and runs in PDP-10 MacLisp and in Symbolics 3600 ZetaLisp.

13.2 *Fibonacci*

The first simulation is one that shows that in a real multi-processor, with a small number of processors and realistic process creation times, the runtime tuning provided by the predicates in the **QLET**'s is important. The Figure 1 shows the performance of the Fibonacci function written as a parallel function on a multi-processor with 5 processors.

Here is the code:

```
(DEFUN FIB (N DEPTH)
       (COND ((= N 0) 1)
             ((= N 1) 1)
             (T
              (QLET (< DEPTH CUTOFF)
                    ((X
                      (FIB (- N 1) (1+ DEPTH))
                     (Y
                      (FIB (- N 2) (1+ DEPTH)))))
               (+ X Y)))))
```

Although this is not the best way to write Fibonacci it serves to demonstrate some of the performance aspects of a doubly recursive function.

The x-axis is the value of CUTOFF, which varies from 0–20; the y-axis is the runtime in simulator units. The curves are the plots of runs where the process creation time is set to 0, 20, 40, and 100, where 3 such units is the time for a function call.

As can be seen, for nearly all positive process creation times, the program can be tuned to the configuration; and for high process creation times, this is extremely important. The curves all flatten out because only 177 processes are required by the problem, and beyond a certain cutoff, which is the depth of the recursion, all of these processes have been spawned.

13.3 *Adding Up Leaves*

The performance of a parallel algorithm can depend on the structure of the data. Not only can a process be serialized by requiring a single shared resource—such as a data

structure—but it can be serialized by the shape of an unshared data structure. Consider adding up the leaves of a tree. We assume that the leaves of a tree—in this case a Lisp binary tree—can either be () or a number. If a leaf is () it is assumed to have value 0.

Here is a simple program to do that:

```
(DEFUN ADD-UP (L)
       (COND ((NULL L) 0)
             ((NUMBERP L) L)
             (T (QLET T ((N (ADD-UP (CAR L)))
                         (M (ADD-UP (CDR L))))
                (+ N M)))))
```

The curves in Figure 2 show the speedup graphs for this program on a full binary tree and on a CDR tree. A full binary tree is one in which every node is either a leaf or its left and right subtrees are the same height. A CDR tree is one in which every node is either a leaf or the left son is a leaf and the right son is a CDR tree.

These 'speedup' graphs have the number of processors on the x-axis and the ratio of the speed of one processor to the speed of n processors on the y-axis. Theoretically with n processors one cannot perform any task faster than n times faster than with one, so the best possible curve would be a 45° line.

Note that a full binary tree shows good speedup because the two processes sprouted at each node do the same amount of work, so that the load between these processes are balanced: If one process were to do the entire task, it would have to do the work of one of the processes and then the work of the other, where the amount of work for each is the same. With a CDR tree, the process that processes the **CAR** immediately finds a leaf and it terminates. If a single process were to do the entire task, it would only need to do a little extra work to process the **CAR** over what it did to process the **CDR**.

From this we see that the structure of the data can serialize a process.

Let's look at another way to write this program, which will demonstrate the serialization of a shared resource:

```
(DEFUN ADD-UP (L)
        (LET ((SUM 0))
         (LET ((ADDER
                 (QLAMBDA T (X)
                           (SETQ SUM (+ SUM X)))))
               (QCATCH 'END
                       (NO-WAIT (SPAWN (ADD-ALL ADDER L))))
               SUM)))


      (DEFUN ADD-ALL (ADDER X)
             (COND ((NULL X) T)
                   ((NUMBERP X)
                    (WAIT (ADDER X)))
                   (T (SPAWN (ADD-ALL ADDER (CAR X)))
                      (ADD-ALL F (CDR X))))))
```

This program works by creating a process closure (called ADDER in ADD-UP) that will perform all of the additions. SUM is the variable that will hold the sum.

ADD-ALL searches the tree for leaves. When it finds one, if the leaf is (), the process returns and terminates; if the leaf is a number, the number is sent in a message to ADDER. Then the process returns and terminates. We **WAIT** for ADDER in ADD-ALL so that SUM cannot be returned from ADD-UP before all additions have been completed.

If the node is an internal node, a process is spawned which explores the **CAR** part, while the current process goes on to the **CDR** part. The performance of this program is not as good as the other because ADDER serializes the additions, which form a significant proportion of the total computation. If the search for the leaves were more complex, then this serialization might not make as much difference. The curves in Figure 3 show the speedup for this program on the same full binary and CDR trees as in Figure 2.

13.4 *Data Structures*

As we have just seen, the shape of a data structure can influence the achieved degree of parallelism in an algorithm. Because most modern Lisps support arrays and some support

vectors, we recommend using arrays and vectors over lists and even trees—these random-access data structures do not introduce access-time penalties that could adversely affect parallelism. To assign a number of processes to subparts of a vector only requires passing a pointer to the vector and a pair of indices indicating the range within which the process is to operate.

13.5 *Traveling Salesman*

The performance of an algorithm can depend drastically on the details of the data and on the distribution of the processes among the processors. A variant of the traveling salesman illustrates these points.

The variant of the traveling salesman problem is as follows: Given n cities represented as nodes in a graph where the arcs between the nodes are labelled with a cost, find the path with the lowest total cost that visits each of the cities, starting and ending with a given city. That is, we want to produce the shortest circuit.

The solution we adopt, for illustrative purposes, is exhaustive search. We will sprout a process that takes as arguments a control process, a node, a path cost so far, and a path. This process will check several things: First it sees whether the path cost so far is less than the path cost of the best circuit known. If not, the process dies. Next the process checks whether the node is the start node. If so, and if the path is a complete circuit—if it visits every node—then the control process is sent a progress report which states the path cost and the path.

Failing these, the process spawns a process for each outgoing arc from the node, including the arc just traversed to get to this node—it is possible that a node is isolated with only one arc connecting it to the remainder of the graph.

The control program simply keeps track of the best path and its cost. It maintains these as two separate global variables, and its purpose is to keep them from being updated inconsistently.

The key is that it may take a long time to find the first circuit—initially the best path cost is ∞. Once a circuit is found many search processes can be eliminated. If there are relatively few processors and many active processes, then the load on each processor can interfere with finding the first circuit, and many processes can be spawned, increasing the load. We have intentionally kept the problem general and not subject to any heuristics

that would help find a circuit rapidly, in order to explore the performance of various multi-processors on the problem.

Figure 4 shows the speedup graph. Two things are of note. One is that it wildly fluctuates as the number of processors increases. Drawing a smooth curve through the points results in a pleasing performance improvement, but the fluctuations are great, and in particular 21 processors does much better than 31, even though 36 processors is better than either of those configurations.

In the round-robin scheduler, the positioning of the process—relative to the other processes—that will find the first complete circuit is critical, especially when those processes are sprouting other processes at a high rate.

The graph, by the way, is for a problem with only 5 cities.

The second thing to note is that the graph goes above the curve of the theoretically best performance—the 45° line. This is because the 1 processor case is sprouting processes, and is thrashing very much worse than a non-multi-processed version of the algorithm would. In other words, all such speedup graphs need to be normalized to the highest point of curve, not the 1 processor case.

13.6 *Browse*

Browse is a benchmark used as one of a series of benchmarks for evaluating the performance of Lisp systems. [Gabriel 1982] It essentially builds a data base of atoms and their property lists. Each property list contains some 'information' in the form of a list of tree structures. The list of atoms is randomized and a sequence of patterns is matched, one at a time, against each of the tree structures on the property list of each of the atoms. In this pattern matcher EQL objects match, '?' variables match anything, and '$*$' variables match a list of 0 or more things. A variable of the form '$*$-*atom*' must match the same list with each occurrence.

The pattern matcher and the control of the exhaustive matching have been written as parallel code. This sort of 'searching' and matching in data bases of this form is typical of many artificial intelligence programs, especially expert systems. The performance of multi-processors on this benchmark is remarkable.

Two curves are shown in Figure 5: One shows the speedup with the process creation time set to 10 (where a function call is 3), and the other is with the process creation time set to 30. In both cases there is near linear improvement as the number of processors

increases. Approximately 1000 processes are sprouted in this benchmark, although at most 187 processes are alive at any given time, averaging 117 with a standard deviation of .25.

14. Conclusions

We have presented a new language for multi-processing. A variant of Lisp, this language features a unique and powerful diction for parallel programs. Parallel constructs are expressed elegantly, and the language extensions are entirely within the spirit of Lisp.

The problem of runtime tuning of a program is addressed and adequately solved. The performance of programs written in this language as a function of the size of the multi-processor is explored.

Multi-processors that support shared memory among processors is important, and even some or all of the nodes in a distributed system should be multi-processors of this style. To achieve maximum performance we will need to pull every trick in the book, from coarse-grained down to fine-grained parallelism. This language is a step in the direction of achieving that goal by allowing programmers to easily express parallel algorithms.

15. Acknowledgments

We would like to thank Jeff Ullman whose questions and comments provided precise direction to some of this work.

References

[**Gabriel 1982**] Gabriel, R. P., Masinter, L. M. *Performance of Lisp Systems*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Smith 1978**] Smith, Burton J., *A Pipelined, Shared Resource MIMD Computer* in **Proceedings of the International Conference on Parallel Processors**, 1978.

[**Steele 1978**] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. *The Revised Report on SCHEME: A Dialect of LISP*, AI Memo 452, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, January, 1978.

[**Steele 1984**] Steele, Guy Lewis Jr. et. al. **Common Lisp Reference Manual**, Digital Press, 1984.

§

[**Sussman 1975**] Sussman, Gerald Jay, and Steele, Guy Lewis Jr. *SCHEME: An Interpreter for Extended Lambda Calculus*, Technical Report 349, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, December, 1975.