Qlisp

Richard P. Gabriel
Lucid, Inc.

John McCarthy
Stanford University

Abstract

Multiprocessing programming languages must support the styles of programming that are natural to the programmer, and they must be able to express all of the forms of concurrency that exist. Qlisp, a dialect of Lisp, is proposed as a multiprocessing programming language which is suitable for studying the styles of parallel programming.

There are two sorts of parallelism in Qlisp: 1) the true parallelism that derives from the parallel evaluation of arguments, and 2) the unstructured concurrency of process invocation in which a number of processes are created and messages are passed among them, causing some pattern of concurrent activity.

1. Introduction

As the need for high-speed computers increases, the need for multiprocessors becomes more apparent. One of the major stumbling blocks to the development of useful multiprocessors has been the lack of a good multiprocessing 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 multiprocessing extensions to Lisp.

Unlike other proposed multiprocessing Lisps, this one provides only a few very powerful and intuitive primitives rather than a number of parallel variants of familiar constructs.

This language is called Qlisp.

2. Design Goals

1. Because Lisp manipulates pointers, this Lisp dialect will run in a *shared-memory* architecture;
2. Because any real multiprocessor 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 multiprocessing 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 multiprocessing setting, where appropriate, rather than proliferating new constructs; and
5. The constructs should all work in a uni-processing setting (for example, it should be possible to set the degree of multiprocessing to 1 as outlined in point 2).

3. QLET

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

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

Prop is a propositional parameter 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 *prop* 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 *prop* evaluates to non-(), then the **QLET** will cause some multiprocessing to happen. Assume *prop* 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 *prop* 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*.

3.1 *Queue-based*

The Lisp is described as 'queue-based' because each spawned process is placed on a global queue of processes. Processes are assigned to processors. 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 purpose of *prop*, then, is to control the number of processes spawned. Simulations show a marked dropoff in total performance as the number of processes running on each processor increases, *assuming that process creation time is non-zero.*

[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.

3.2 *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))
```

3.3 *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)))))
```

4. Excessive Parallelism

The most straightforward possibilities for achieving effective parallelism occur when only *AND-parallelism* is required. AND-parallelism occurs when there are some number

of tasks, each of which must be performed, with minimal interdependence between them. However, since Lisp programming is recursive, AND-parallelism alone is not always possible, and there is still the problem of avoiding too much parallelism. For example **QSUBST** will generate parallelism in computing the arguments of a single **CONS**. It is straightforward to avoid excessive parallelism if the program can determine how big a computation will be. If it is going to be too small, the **QLET** propositional parameter can be set to be (), or control can be given to a version of the program that has no parallelism. In general, efficiency will not be very sensitive to how the parallelism is arranged to occur provided that amount of bookkeeping to determine when to introduce parallelism is small, and this will be true if parallelism is avoided for small computations.

There are at least two ways of doing this.

First, it may be trivial to determine how big the computation is. Consider computing $n!$. In order to do the computation recursively, we generalize it to computing the product of the numbers from m to n:

```
(DEFUN FACT (N)
 (LABELS
  ((MULT
    (LAMBDA (N M)
     (COND ((= N M) M)
           (T
            (LET ((H (FLOOR (+ N M) 2)))
             (* (MULT N H) (MULT (+ H 1) M)))))))
  (MULT 1 N)))
```

If $|n-m|$ is small enough, we can call a function that multiplies them non-recursively. Otherwise, we split the range in half and compute the products of the halves in parallel. This immediately generalizes to computing any commutative operation applied to elements indexed by the numbers from m to n, because $|n-m|$ provides an estimate of the size of the problem, which can be used to determine whether parallelism is worthwhile.

In this example it might be better to put the parallelism in the bignum multiplication code rather than in the overall structure of the program. The point is that it is immediately decided whether the computation is too small to do in parallel.

Second, perhap the size of the computation can be computed by a linear scan of the arguments of the function. When the time required to compute the function is much larger than the time taken by the scan, this approach can be used to decide whether to compute the argument values in parallel.

From this point of view, **QSUBST** presents the most difficult case: The time required to determine the size of the computation is as large as the time required to do the computation. Here one might as well gamble on parallelism if the program is really going to have to do substitutions with no advance estimate of the size of the expressions.

However, if an operation of substitution or something of similar characteristics is going to occur frequently in a program, it may be worthwhile to include a size estimate as part of the data structures involved. An extreme would be to use a modified **CONS** that includes a size as well as the two pointers. We suspect that this is rarely worthwhile and that it will be more common to use data structures that contain size estimates sparingly.

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 invoked, 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** *prop* (*lambda-list*) . *body*)

creates a closure. *Prop* is a propositional parameter that is evaluated before any other action regarding this form is taken. It is assumed to evaluate to either (), **EAGER**, or something else. If *prop* 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 *prop* evaluates to something other than **EAGER**, the **QLAMBDA** creates a closure that, when invoked, 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 invoked, 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 *prop* 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.

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 start the process evaluating (**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 invoked in a value-requiring context, the calling process waits; and if a process closure is invoked 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** $form_1$)
$form_2$)

will wait for $form_1$ to complete.

5.2 *Invoking 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 invoked by another process. Similarly, a process can invoke a process closure in a value-ignoring position and then immediately invoke 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 invoked, 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.

[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.

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 invocations 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 multiprocessing 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 invoked 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 invocation.

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.

In 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** was created to start searching for the proper **CATCH**. Thus, if a process closure is invoked with a **THROW** in it, the **THROW** will be to the first

CATCH with a matching tag *in the process chain in which the* **QLAMBDA** *was created*, 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 let-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 multiprocessing 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.

8. The Rest of the Article

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 article will describe some of the tricks that can be done in this language.

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 ∗**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 ∗**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 Qlisp would supply an efficient locking mechanism, and the details of a realistic locking protocol will be discussed later. 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:

```
(DEFUN TEST ()
 (LET ((BOMBS ()))
  (LET ((BOMB-HANDLER
         (QLAMBDA T (TYPE ID MESSAGE)
                  (COND ((EQ TYPE 'BOMB)
                         (PRINT '(BOMB FOR ,ID))
                         (PUSH '(,ID . ,MESSAGE) BOMBS))
                        ((EQ TYPE 'KILL)
                         (PRINT '(KILL FOR ,ID))
                         (FUNCALL
                          (CDR (ASSQ ID BOMBS)))
                         T)))))
   (QLET 'EAGER ((X
                  (CATCH 'QUIT (TESTER BOMB-HANDLER 'A)))
                 (Y
                  (CATCH 'QUIT (TESTER BOMB-HANDLER 'B))))
         (SPAWN
          (PROGN (DO ((I 10. (1- I)))
                     ((= I 0)
                      (PRINT '(KILLING A))
                      (BOMB-HANDLER 'KILL 'A ()))
                     (PRINT '(COUNTDOWN A ,I)))
                 (DO ((I 10. (1- I)))
                     ((= I 0)
                      (PRINT '(KILLING B))
                      (BOMB-HANDLER 'KILL 'B ()))
                     (PRINT '(COUNTDOWN B ,I)))))
         (LIST X Y)))))

(DEFUN TESTER (BOMB-HANDLER LETTER)
         (BOMB-HANDLER 'BOMB LETTER
                       (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 18–27) 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 propositional parameter in **QLET** applications, that is, in constructions of this form:

(**QLET** *prop* ((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 an 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 invokes 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.

13. Software Pipelining

Multiprocessing programming languages must support the styles of programming that are natural to the programmer, and they must be able to express all of the forms of concurrency that exist. In this section we will introduce a style of programming, called *software pipelining*, which can produce speedups in sequential code under certain circumstances. Moreover, sequential code can be easily transformed into pipelined code.

13.1 *Pipelining*

Pipelining is a technique used in computer architectures to increase the execution speed of programs. The idea is that while one instruction is being executed the next instruction in the instruction stream can be decoded. Thus, the execution of one instruction can overlap the execution of another. This is possible because the execution of an instruction can be broken up into independent stages. These stages are implemented as separate pieces of hardware that perform the steps in each stage.

There are several complications with this scheme as far as the hardware is concerned. First, if the second instruction requires a result from the first instruction, and if the result is not ready for use from the first instruction when the second instruction wants to use it, then the pipeline 'blocks.' This delay can cause the throughput of the pipeline to decrease, and sometimes this decrease can be significant.

Second, if the first instruction is a conditional jump, then perhaps the address to which control will pass might not be known when the second instruction is being fetched. This will result in a pipeline blockage also. This type of complication is similar to the first—simply consider the address from which the the next instruction is to be fetched as a result of the current instruction. Normally this address—the program counter or PC—is implicitly known to be the next sequential PC after that of the current instruction.

The effect of pipelining is that programs that are not inherently parallel can enjoy some degree of parallelism and, hence, can run faster on a pipelined architecture than on the same architecture without pipelining. There is a window of instructions such that if one instruction within that window depends on the result of a second instruction within that window, then we might expect the pipeline to block. This window is certainly no larger than the length of the pipeline, and this window slides along the stream of executed

instructions. For most pipelined computers the window is smaller than the length of the pipeline.

The hope is that the window is small enough and the dependencies within that window are rare enough that there will not be a significant slowdown from the ideal situation, which is a program with no dependencies within any window.

13.2 *Software Pipelining*

Software pipelining is the adaptation of the idea of hardware pipelining to Qlisp. We will see that often programs can be pipelined when they cannot be made fully parallel, and we will also see that dependencies between stages of a software pipeline can be implemented using a locking mechanism with certain characteristics.

There are two sorts of parallelism in Qlisp: 1) the true parallelism that derives from the parallel evaluation of arguments in a **QLET**, and 2) the unstructured concurrency of process-closure invocation, particularly when the invocation is in a value-ignoring situation. In the latter case, a number of processes are created and messages are passed among them, causing some pattern of concurrent activity.

Software pipelining is a subspecies of the latter type of parallelism, but it is a structured subspecies. Suppose that there is a computation which is inherently sequential and which must be performed repeatedly. In fact, suppose that this computation takes a certain set of arguments and that we intend to stream sets of arguments to that computation at the fastest possible rate. If that computation is expressed as a process closure, then the second set of arguments cannot be processed until the first set has been completely processed. To apply software pipelining to that computation, we break up that computation into stages such that the amount of information that needs to passed from one stage to the next is manageable; each stage is a process closure. Because each stage will be smaller than the entire original computation, the second set of arguments can enter the computation sooner after the first set than in the original computation.

An example of an inherently sequential computation is one which performs some actions on a shared resource, such as a global data structure.

13.2.1 *Simple Example*

We will present a simple example of a computation along with a pipelined implementation of it. This example is designed to explain the concept of software pipelining; the example is not an example of a computation that requires software pipelining—much

better speedups can be achieved with true parallelism or even with a reformulation of the algorithm than with pipelining.

The example is polynomial evaluation: Given a polynomial, $P(x)$, and a value, v, for x, produce $P(v)$. Let

$$P(x) = 5x^4 + 4x^3 + 3x^2 + 2x + 1,$$

then using Horner's rule we have that,

$$P(x) = (((5x + 4)x + 3)x + 2)x + 1$$

.

We can define four functions—F_1, F_2, F_3, and F_4—such that

$$P(x) = F_4(x, F_3(x, F_2(x, F_1(x)))),$$

as follows:

```
(DEFUN F1 (X)
       (+ (* 5 X) 4))

(DEFUN F2 (X V)
       (+ (* V X) 3))

(DEFUN F3 (X V)
       (+ (* V X) 2))

(DEFUN F4 (X V)
       (+ (* V X) 1))
```

These functions define stages of the computation of P. Each stage is largely independent of the others aside from the values x and v which are passed as arguments, and each

stage does about the same amount of computation as the others. Given these definitions, the pipelined version of the original polynomial evaluation function can be written:

```
(DEFUN HORNER-STREAM ()
 (QCATCH 'HST
  (LABELS ((P1 (QLAMBDA T (X)
                (P2 (+ (* 5 X) 4) X)
                T))
           (P2 (QLAMBDA T (X V)
                (P3 X (+ (* V X) 3))
                T))
           (P3 (QLAMBDA T (X V)
                (P4 X (+ (* V X) 2))
                T))
           (P4 (QLAMBDA T (X V)
                (+ (* V X) 1))))
           (P1 x1)(P1 x2)(P1 x3)...)))
```

The **QCATCH** will kill all of the processes when HORNER-STREAM is exited, but only after each of the process closures, P1–P4, has finished its last computation. **LABELS** is a construct for defining mutually recursive functions, and each one of P1–P4 is bound to a process closure. P1 corresponds to F_1, P2 corresponds to F_2, P3 corresponds to F_3, and P4 corresponds to F_4. The next stage of each pipe is called in a value-ignoring position.

The ellipsis [...] is a sequence of invocations of P1 for various arguments. Each process, P1–P4, is presumed to be running on a different processor.

Imagine a stream of arguments to P1, $v_1 \ldots v_n$. Suppose v_1 is passed to P1 and then immediately v_2 is passed. P1 cannot process v_2 until v_1 has been processed; as soon as $5x + 4$ has been passed on to P2, P1 can accept v_2, and so on. When P4 is processing the final stage of the computation of $P(v_1)$, P3 can be processing the third stage of $P(v_2)$, P2 can be processing the second stage of $P(v_3)$, and P1 can be processing the first stage of $P(v_4)$.

The straightforward code for P is:

```
(DEFUN HORNER ()
 (LABELS ((P1 (LAMBDA (X)
                (+ 1 (* X (+ 2 (* X (+ 3 (* X (+ 4 (* 5 X))))))))))
          (P1 x1)(P1 x2)(P1 x3)...)))
```

Using a simple performance simulator for Qlisp [Gabriel 1984], when 40 requests are streamed through HORNER and HORNER-STREAM, HORNER-STREAM is approximately 3.8 times faster than HORNER, compared with the theoretical maximum of 4.0.

Of course, there are better ways to speed up this particular program, but the technique is the important point. The technique enables speedups in sequential code, and these speedups can achieved by applying the software pipelining technique rather than by discovering the, possibly, clever parallel algorithm.

13.2.2 *Discussion*

There are several things to note about this example and the technique. First, the amount of computation per process closure is quite small, and the performance of the pipeline will depend on the function-call overhead in HORNER-STREAM which is not present in HORNER. Note that because process-closures are invoked to move control from one stage of the pipe to the next, function-call overhead is, in reality, message-passing overhead.

Second, the software pipelining style of programming is very much like that used in continuation-passing style. With continuation passing, an additional function—the continuation—is passed as an argument to each function. When a function computes a value, it applies the continuation to that value in a tail-recursive position rather than returning to its caller. With software pipelining, the continuation for each stage of the pipe is the next stage of the pipe, if there is one.

The last stage of a pipeline can either invoke some other process, or else that stage could invoke a continuation that had been explicitly passed to it.

Third, this example does not contain any global or special variables. For this technique to be useful, it ought to be possible to use it in the presence of global variables. Of course, heavy use of globals will diminish the performance advantages, much as instruction dependencies limit the effectiveness of a pipeline in a computer.

13.2.3 *Global Variable Example*

In the simple example of polynomial evaluation, all of the information that is passed among the stages is passed from one stage to the next as arguments. Therefore, we can say that all of the information flow is in the forward direction—each stage can only receive information from the preceeding stages. With global variables it is possible for this information to be passed in the backward direction.

One use of global variables is for one invocation of a function to communicate with a later invocation. The first function can store some value in a variable whose extent is indefinite—a global variable. A later invocation of that function (or some other function) can read that value and act on it.

Global variables are used quite often to record a *state* for a function. In Lisps that do not support closures, a programmer will often construct a closure equivalent by packaging the code for a function with an explicit environment, which is simply a set of global variables. In a closure, maintaining an environment—a state—is simply an example of one function invocation communicating with a later one.

In the instruction stream in a computer, an early instruction can write some value into a memory location or a register, and a later instruction can read and use that value. When this happens within the pipeline window, as we discussed earlier, the pipeline may block, and performance may be lost.

Similarly, we can define a software pipeline in which later stages of the pipe can communicate with earlier stages—early arguments to the pipeline can influence the behavior of the pipeline on later arguments.

To handle global variables, we use vectors in place of variables and a locking mechanism to keep reads and writes of the global variables straight. We will demonstrate the technique with a variant of the polynomial example:

$$P(x) = 5x^4 + 4x^3 + 3x^2 + Qx + 1$$

.

Q is a global variable, and let us assume that after each evaluation of P, Q is incremented. Thus, the HORNER code should become:

```
(DEFUN HORNER ()
 (LABELS
  ((P1 (LAMBDA (X)
        (LET ((ANS (+ 1 (* X (+ Q (* X (+ 3 (* X (+ 4 (* 5 X))))))))))
             (SETQ Q (1+ Q))
             ANS))))
  ...))
```

In this new version of HORNER, an early invocation of the code will communicate with a later invocation by changing the value of Q.

For the sake of the example, we will stipulate that the global variable, Q, must be updated after the computation of the value of the polynomial has been completed, though there is no apparent reason for this in the code. This will allow us to see how a variable might be handled at various stages in the pipeline.

Because there are four stages in the pipe, we will substitute a 4-long vector for Q. We will call an index into this vector a 'slice of Q.' Each slice is associated with one complete flow of control through the pipeline. That is, each time a value is passed through the pipe in order to evaluate the polynomial once, a slice of Q is assigned to that computation.

To allocate the slice, we will create a lock for each slice of Q. In fact, there will be a 4-long vector that holds the locks. Earlier we showed that locks could be implemented with only those primitives described earlier. However, Qlisp directly supports a more streamlined locking mechanism.

In Qlisp locks are first-class citizens and can be passed around as any other Lisp object can. Locks can be created, gotten, and released. One interesting aspect of Qlisp locks is that they can be *named*. Most of the time anonymous locks are sufficient, and such locks are granted in the order in which requests are received. A lock is a Qlisp structure with several fields: the owner field, the request-queue field, and the *name* field. Locks are created and pointers to them are passed exactly as with other Lisp objects. In order to request a lock, a process must have a pointer to it; if a process requests a lock and it already has an owner, then that requestor is put in the queue for that lock. If there is no owner, then the name of the lock comes into play.

When a lock is owned by a certain process, that process might wish to pass the lock

to a particular process. To do this, the owner sets the name of the lock to some particular value.

A lock which has a name can only be granted to a process that asks for that lock by its name. If several processes request a named lock, then the first process that asks for that lock by its name gets it. If a request for a named lock is placed, and the lock has no name (at that time), then the request is granted regardless of what name was supplied by the requestor.

Named locks will solve a difficult problem in pipelines with global variables.

This is the new code for HORNER-STREAM:

```
1  (DEFUN HORNER-STREAM ()
2   (QCATCH 'HST
3    (LABELS ((P1 (LET ((N 0)
4                       (ID (NCONS ()))
5                       (NEXT-ID (NCONS ())))
6                      (QLAMBDA T (X)
7                       (LET ((LOCK (GET-LOCK (LOCK N))))
8                            (P2 X (+ (* 5 X) 4) LOCK N ID NEXT-ID)
9                            (SETQ ID NEXT-ID)
10                           (SETQ NEXT-ID (NCONS ()))
11                           (SETQ N (REMAINDER (1+ N) 4))
12                           T))
13             (P2 (QLAMBDA T (X V LOCK N ID NEXT-ID)
14              (P3 X (+ (* V X) 3) LOCK N ID NEXT-ID)
15              T))
16             (P3 (QLAMBDA T (X V LOCK N ID NEXT-ID)
17              (P4 X (+ (* V X) (GET-Q N ID)) LOCK N ID NEXT-ID)
18              T))
19             (P4 (QLAMBDA T (X V LOCK N ID NEXT-ID)
20                 (LET ((ANS
21                          (+ (* V X) 1)))
22                       (SETF (ASET *QAR* N) (1+ (AREF *QAR* N)))
23                       (SET-LOCK-NAME LOCK NEXT-ID)
24                       (RELEASE-LOCK LOCK)
```

```
25                          ANS))))))
26              ...)))
```

where GET-Q is defined as:

```
27 (DEFUN GET-Q (N ID)
28         (LET ((LOCK (GET-NAMED-LOCK ID (PREVIOUS-LOCK N))))
29              (LET ((Q (AREF *QAR* (PREVIOUS-INDEX N))))
30                   (SETF (AREF *QAR* N) Q)
31                   (SET-LOCK-NAME LOCK ())
32                   (RELEASE-LOCK LOCK)
33                   Q)))
```

where *QAR* holds the slices of Q.

N is a variable that is local to process P1 and whose values circulate among 0-1-2-3-0.... As each new argument arrives at P1, the proper slice of Q and the correct lock are selected by this N. When control is passed onto P2, the second pipe stage, N is incremented (mod 4).

ID and NEXT-ID are variables that are local to process P1; ID is the name to be used for the current lock, and NEXT-ID is the name for the next lock. These two variables are initialized on lines 4 and 5. **NCONS** takes one argument and returns a list containing that argument as its sole element. The idiom (**NCONS** ()) is frequently used to create a unique pointer for use as a name.

There are 4 locks used to control access to the four slices of Q. Each of the 4 locks—one for each stage—is initially given a null name.

The lock for the N^{th} slice of Q is grabbed at line 7, so that no other process has access to that slice of Q until it is released with **RELEASE-LOCK**. Each time a lock is grabbed at line 7, it has no name. The lock, along with the values of N, ID, and NEXT-ID, are passed from stage to stage.

A process executing in stage P4 wants the next process in line to get a hold of the value of Q it just wrote—no other process should be able to get to it first. Therefore, we want to name the lock in such a way that this is inevitable. Consider the environments of

the processes in question. From within stage P4, ID is the current name and NEXT-ID is the name of the next process in line behind the one currently in P4 (lines 9 and 10). Thus, the value of NEXT-ID in stage P4 is EQ to the value of ID in stage P3.

At line 23 the name of the lock for the N^{th} slice of Q is set to NEXT-ID. Because the process in stage P4 has had control of this lock since that process started in stage P1, no other process can have control of it, and now that the lock's name has been set, there is exactly one process that can get a hold of it—the process that asks for that name, which is the process right behind the one in stage P4.

The process in stage P3 asks for the value of Q, using GET-Q, at line 17. In GET-Q at line 28, the lock with ID as its name is requested. This will result in the process in stage P3 getting the lock right after the process in stage P4 is through with it. In GET-Q, after the value for Q written into the N^{th} slice of Q has been copied into the $(N+1)^{\text{st}}(mod\ 4)$ slice of Q (line 30), the name of the lock is set to () (line 31), and the lock is released. Now any other process can grab this lock.

The worst case is that the reference to Q in P3 will need to wait until stage P4 of the previous computation has released that slice; this will happen quite frequently. Also, the amount of computation that goes into each stage is quite low compared with the amount of computation that goes into maintaining the locks. The result is that HORNER-STREAM is slower than HORNER on 40 computed values by a factor of approximately 1.7.

There are two factors which could improve the performance of pipelines that communicate among stages with global variables: 1) The frequency of interaction between pipe stages using global variables could be decreased, and 2) the amount of computation per pipe stage could be increased.

The frequency of global variable use as a determiner of the performance of a pipeline is obvious. The amount of computation per stage as compared with the amount of computation needed to maintain the locks can be shown to be a determiner of performance by modifying HORNER and HORNER-STREAM so as to increase the amount of computation per pipe stage.

```
(DEFUN HORNER-STREAM ()
 (QCATCH 'HST
  (LABELS ((P1 (LET ((N 0)
                     (ID (NCONS ()))
```

```
                    (NEXT-ID (NCONS ())))
                   (QLAMBDA T (X)
                    (LET ((LOCK (GET-LOCK (LOCK N))))
                         (P2 X (+ (* (DELAY 5) X) (DELAY 4))
                              LOCK N ID NEXT-ID)
                         (SETQ ID NEXT-ID)
                         (SETQ NEXT-ID (NCONS ()))
                         (SETQ N (REMAINDER (1+ N) 4))
                         T))
               (P2 (QLAMBDA T (X V LOCK N ID NEXT-ID)
                (P3 X (+ (* V X) (DELAY 3)) LOCK N ID NEXT-ID)
                T))
               (P3 (QLAMBDA T (X V LOCK N ID NEXT-ID)
                (P4 X (+ (* V X) (DELAY (GET-Q N ID))) LOCK N ID NEXT-ID)
                T))
               (P4 (QLAMBDA T (X V LOCK N ID NEXT-ID)
                   (LET ((ANS
                          (+ (* V X) (DELAY 1))))
                   (INCR-GLOBAL N)
                   (SET-LOCK-NAME LOCK NEXT-ID)
                   (RELEASE-LOCK LOCK)
                   ANS))))))
              ...)))
```

where DELAY is a macro defined as:

```
(DEFMACRO DELAY (FORM)
          `(DO ((I *DELAY* (1- I)))
               ((ZEROP I) ,FORM)))
```

We can similarly modify HORNER. With *DELAY* set to 20, HORNER-STREAM is faster than HORNER over 20 polynmial evaluations by a factor of approximately 2.2.

13.3 *Comparison with Other Techniques*

Qlisp is essentially an asynchronous language—we assume several CPU's attached to a single address space, where each CPU can run several jobs (or processes) at once in a timeshared mode. We do not assume that there is any explicit control of the scheduling of processes outside of those implied by the language (locks and flow of data and control).

Because of this, Qlisp supports a style of parallelism that is quite different from both systolic arrays and systolic-array-like mechanisms like the Bagel [Shapiro 1983]. There is no network or grid of processors, and data/control is not shunted from one processor to the next. Therefore, the use of locks is necessary to control access to globals.

Moreover, within each stage of a pipeline it is possible to use the full power of Qlisp to achieve local speedups—each stage can exploit a high degree of parallelism within it. This is not readily performed with any of the systolic-array-like techniques.

The examples we have shown have been coded using the underlying Qlisp mechanisms without using any macros or other structuring. The code in each stage of the pipes presented, along with the actions to deal with global variables within those stages, is so stylized that macros are easily written to support software pipelining. This would shorten and simplify the unreadable code we have been using for expository purposes.

The Qlisp programming environment supports a suite of such macros, and here is how the second version of HORNER-STREAM is actually written:

```
(DEFUN HORNER-STREAM ()
       (PIPELINE FOO ((Q 0))
                 ((STAGE (X) X (+ (* 5 X) 4))
                  (STAGE (X V) X (+ (* V X) 3))
                  (STAGE (X V) X (+ (* V X) (GLOBAL-REF Q)))
                  (STAGE (X V) (+ (* V X) 1)
                                 (SETF (GLOBAL-REF Q)
                                       (1+ (GLOBAL-REF Q)))))
                 ...))
```

In this formulation, each stage simply invokes the next; the form for a stage in the pipe is:

(**STAGE** <*formal arguments*> . <*arguments to next stage*>)

There is also a formulation in which pipe stages are explicitly named and can be invoked by name. This allows one to write a pipeline which is a directed graph.

Software pipelining is also similar to stream processing, in which one process supplies a stream of values to another. Software pipelining is different in that it is useful for introducing concurrency to an existing serial program by breaking it up into a stream with several stages. Thus, it is not only a technique that is useful for thinking about programs as processes which produce or consume a sequence of values, but it is also useful for thinking about increasing the running speed of a program.

This viewpoint allowed us to introduce software pipelining into programs which use global variables to enable early invocations of a program to communicate with later invocations.

14. Geometric Control Structures

In this section we will introduce a style of programming called *geometric control structuring.*

Systolic arrays have been used extensively in numeric analysis computing. The idea is that a network of computers organized as an array can be programmed to perform operations—typically on arrays—by streaming data from several computers in the array to a single computer. That single computer performs some operation on the data and passes along its value to some other computer.

The advantage of this style of programming is that geometric intuition about the structure of a computation can be brought to bear by the programmer to produce an effective and clear program. If there is a multiprocessor whose interconnection structure corresponds to the program structure, then there may be a performance advantage as well.

14.1 *Motivation for Geometric Control Structures*

The key observation about systolic arrays is that the control structure corresponds very closely to the geometry (or topology) of the problem. Process closures were used to define software pipelining; process closures can also be used to define any hierarchical or heterarchical control structure.

14.2 *Data-Structure-resident Closures*

The key idea is to allocate processes within a data structure. If this data structure is global, then each process within the data structure is able to access other processes—that is, any process can invoke any other process that it is able to access through the data structure.

An example might be a 2-dimensional array, which could correspond to some physical aspect of the problem the programmer wishes to solve. If the solution to the problem requires that each element in the array have an associated process, which performs some computation, then the programmer can store a process closure in each element in the array. These process closures can incorporate communications capabilities to other elements in the array, perhaps limited to nearby neighbors.

Once the problem is solved in its own terms, attention can be turned towards laying the data structure of process closures down on the physical structure of the multiprocessor. Perhaps the multiprocessor is itself a rectangular array and the mapping is simple. Perhaps not. The key is to solve the problem without much concern for the geometry of the underlying hardware, leaving the matching of the software architecture to the hardware architecture until later.

Software pipelining can be seen as a simple example of this idea, with the underlying data structure being a simple list. Systolic arrays can be viewed this way with some sort of grid as the underlying data structure.

15. Conclusions

We have presented a new language for multiprocessing. 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.

Multiprocessors that support shared memory among processors are important, and even some or all of the nodes in a distributed system should be multiprocessors 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.

§

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.

[**Gabriel 1984**] Gabriel, R. P., McCarthy, J. M., *Queue-based Multiprocessor Lisp*, Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984.

[**Gabriel 1985**] Gabriel, R. P., **Performance and Evaluation of Lisp Systems**, The MIT Press, Cambridge, Massachusetts, 1985

[**Halstead 1984**] Halstead, Robert, *MultiLisp*, Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984.

[**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.