

Qlisp: Parallel Processing in Lisp

Ron Goldman and Richard P. Gabriel
Lucid, Inc.

Abstract

One of the major problems in writing programs to take advantage of parallel processing has been the lack of good multiprocessing languages—one which is both powerful and understandable to programmers. In this paper we describe multiprocessing extensions to Common Lisp designed to be suitable for studying styles of parallel programming at the medium-grain level in a shared-memory architecture. The resulting language is called Qlisp.

A problem with parallel programming is the degree to which the programmer must explicitly address synchronization problems. Two new approaches to this problem look promising: the first is the concept of heavyweight futures, and the second is a new type of function called a partially, multiply invoked function.

1. Introduction

The quest for higher-speed computers continues, and as the physical limitations on uniprocessor speed inhibit continued improvements, the need for parallel computers becomes more pressing. However, a computer that cannot be programmed is worthless, and so it makes sense to turn our attention to programming languages that can express parallel computations.

We have decided to focus our attention on medium-grained parallelism within the confines of artificial intelligence and symbolic computing. Our interest in researching language design is to study how to write parallel programs. To that end we are not initially concerned about introducing the minimal number of new constructs or in the fine details of syntax. Rather, we are trying to create a rich blend of language constructs that will allow programmers to describe parallel algorithms in a variety of styles—simplification will follow. In doing so we are hoping to help artificial intelligence programming in the future.

This research was supported by DARPA under contract N00039-84-C-0211.

Therefore, we are investigating parallel extensions to the programming language Common Lisp [7]. The resulting language is called Qlisp. Other research into extending Lisp to support parallel programming is described in [4], [6] and [8].

This paper describes the Qlisp language, giving examples of its use. We also discuss additional extensions based on our experience programming in Qlisp. Performance results of the initial implementation of Qlisp are reported in [3].

1.1 *History*

Qlisp was initially designed by John McCarthy and Richard Gabriel [2] while they were affiliated with the Lawrence Livermore National Laboratory's S1 Project. The S1 was to have been a 16-processor multiprocessor, with each uniprocessor being a Cray-class supercomputer. Until 1987 the only implementations of Qlisp were interpreter-based simulators. Since late 1987 we have been engaged in implementing Qlisp on an Alliant FX/8 parallel computer; this implementation is based on Lucid Common Lisp, a commercial Common Lisp system.

The Qlisp project supports an exploratory programming component which is researching the effectiveness of Qlisp for symbolic mathematics. The inclusion of an application component of the Qlisp research has been important for gaining critical experience before design and implementation decisions are frozen. Experience gained during the implementation process has resulted in a number of changes to the original design of Qlisp.

1.2 *Design Goals*

The design of Qlisp was aimed at satisfying the following goals:

- The language will support medium-grained parallelism. Medium-grained parallelism matches well the intuitions programmers have about how to parallelize programs. Fine-grained parallelism often requires special hardware support, which is unlikely to be found in stock hardware except for vector processors. Most vector processors are designed for numeric computation. A commercial multiprocessor will typically be designed to support multiple users, and medium-grained parallelism is the best one can do on such hardware. Coarse-grained parallelism can often be achieved with simple message-passing techniques.
- The language will support the explicit expression of parallelism. There will be little or no support for implicit parallelism such as that provided by vectorizing or parallelizing

compilers. Medium-grained parallelism often involves dealing with side-effects, and reasoning about when it is safe to parallelize in the presence of side-effects usually requires domain-specific knowledge.

- The target computer will support a shared address space. It is not important whether the shared address space is implemented using shared memory, except for performance requirements. Symbolic computation typically involves manipulating large shared data structures, which are best handled in a shared address spaces. We do not wish the programmer to worry about access to data structures in non-uniform memory.
- The language will support a variable number of processors. The number of processors a parallel computation requires may depend on the data. In this case it would be important to be able to adapt the number of processes used to solve a problem to the number of processors available.
- The language will provide mechanisms for limiting the number of processes. The cost of creating and maintaining a process can be high. If a process cannot be immediately run, that cost may overshadow the potential gains from spawning it. Also, the number of instructions to spawn the process may be larger than the number of instructions needed for the computation it performs. Therefore, limiting the parallelism can often improve performance.

2. The Qlisp Language

The approach used for Qlisp is queue-based multiprocessing. The programmer must explicitly indicate in the program when parallelism is possible by using the special parallel constructs described below. When a running program executes a statement specifying parallelism, it then adds a collection of new tasks to a queue for subsequent evaluation. When a processor completes a task it goes to this queue for its next task. Basing parallelism on runtime queues means that a program is not written or compiled for a specific number of processors. The number available could even change during the course of a computation. Tasks need not be of similar length, since a processor finishing a short task merely takes another from the queue.

2.1 Futures

Whenever a new process is created to perform some computation, the process will have associated with it a special datatype called a future [1]. This future is a promise to eventually deliver the value that is being computed by the process. Initially the future has no value and is unrealized. The future is realized when the process associated with it finishes its computation. If some other process needs to know the value of an unrealized future in order to perform some operation (such as addition), then it must block and wait until the future has been realized. However many operations, such as **cons**, assignment, or parameter passing, only require a pointer to the future and do not need to wait for it to be realized.

To explicitly wait for a future to be realized, the construct (**realize-future** *form*) can be used. When called, **realize-future** will evaluate *form*, and then, if its value is a future, wait for it to be realized; it then returns the future's value.

2.2 SPAWN

The simplest way to introduce parallelism into a Qlisp program is to use the construct (**spawn** *prop form*) to create a new process to evaluate *form*. The form *prop* is a propositional parameter that is evaluated first. If its value is **nil** (i.e. false) then no new process is created; the process originally executing the **spawn** will proceed to execute *form* and **spawn** will return the resulting value. If *prop* is any non-**nil** value (i.e. true), then a new process is created to evaluate *form* and **spawn** will return a future which will eventually be realized with an actual value when the new process finishes computing *form*.

All of the constructs in Qlisp that can be used to create new processes make use of a similar propositional parameter to give the programmer a way to limit the degree of parallelism during program execution.

The following computes a list that contains a series of values of the function **fun**:

```
(defun function-list (fun start next-arg spawn-p &optional (count 10))
  (let ((initial-list (list nil))
        (arg start)
        val)
    (let ((point initial-list))
      (labels
        ((next-value (&optional (count 1))
          (dotimes (i count)
            (let ((tmp arg))
              (setf val (spawn spawn-p (funcall fun tmp))))
              (setf arg (funcall next-arg arg))
              (setf (car point) val)
              (setf (cdr point) (list #'next-value))
              (setf point (cdr point))))
          val))
        (next-value count)
        initial-list))))
```

Note that **labels** is a Common Lisp construct used to define locally named, mutually recursive functions, in this case one called **next-value**. Also notice that the definition of **next-value** includes the definition-time environment, in this case the local variables **fun**, **next-arg**, **spawn-p**, **arg**, **point**, and **val**. This combination of code plus environment is called a *closure*. When a closure is invoked, the definition-time environment is reestablished. When a new process is created by **spawn** or any other Qlisp construct, a closure is created to evaluate the spawned forms. This ensures that the new task will share the environment that existed when it was created. Part of this environment is the values of any special (dynamic) variables that are currently bound.¹ Note that even if the parent process goes away, the spawned process can still access and modify the values of variables captured in the closure.

The above example is also interesting because the function will produce a list of length **count**, and the list ends with a continuation function which when invoked extends the list. The list will be of the following form:

$$((f x_1) \dots (f x_n) \dots C)$$

where $f = \mathbf{fun}$, $x_1 = \mathbf{start}$, and $x_{i+1} = (\mathbf{next-arg} x_i)$. C is the continuation function and takes an optional argument, which is the number of elements by which to extend the list.

The following function is useful when traversing a list produced by **function-list**:

¹ Qlisp uses a deep-binding scheme: a stack of variable name/value pairs.

```
(defun force (object)
  (realize-future
   (if (typep object 'function)
       (funcall object)
       object)))
```

2.3 QLET

The primary means of introducing parallelism into a Qlisp program is the **qlet** construct, which is used to evaluate a number of arguments to a let-form in parallel. Its form is:

```
(qlet prop ((x1 arg1)... (xn argn)) . body)
```

The form *prop* is again a propositional parameter that is evaluated first. If its value is **nil**, then the **qlet** behaves like an ordinary **let** in Common Lisp: The arguments *arg₁ ... arg_n* are evaluated, their values bound to *x₁ ... x_n*, and the statements in *body* are evaluated.

If *prop* evaluates to any non-**nil** value, then the **qlet** will spawn a number of new processes, one for each *arg_i*, and add them to the queue of processes waiting to run. If the value of *prop* is not the special keyword **:eager** then the process evaluating the **qlet** will wait until all of its newly created child processes have finished. When the values for *arg₁ ... arg_n* are available, the parent process will be awakened, the values bound to *x₁ ... x_n*, and the statements in *body* evaluated.

The following is an example of one way to write parallel factorial using **qlet**:

```
(defun pfact (n depth)
  (labels
   ((prod (m n depth)
        (if (= m n)
            m
            (let ((h (floor (+ m n) 2)))
              (qlet (> depth 0)
                ((x (prod m h (1- depth)))
                 (y (prod (+ h 1) n (1- depth))))
                (* x y))))))
    (prod 1 n depth)))
```

The internal function **prod** computes the product of integers from *m* to *n* inclusive. It does this by dividing the interval *m–n* into two approximately equal parts, recursively computing the products of the integers in those two intervals, and then multiplying the two results.

The cutoff **depth** is used to control the number of processes created. Because two are created for every recursive call in **prod**, at most $2^{\text{depth}+1} - 2$ processes will be spawned. Notice that the propositional parameter to **qlet** simply looks at the value of **depth**.

In the case that *prop* evaluates to the special keyword **:eager** then the process evaluating the **qlet** will not wait for the processes it has just spawned to complete the evaluation of the arguments $arg_1 \dots arg_n$. Instead, it will bind each **qlet** variable, $x_1 \dots x_n$, to a future and then proceed to evaluate the forms in *body*. If in evaluating *body* the value of one of the **qlet** variables x_i is required, the process evaluating the **qlet** will wait for the spawned process computing arg_i to finish. If the value has already been computed, no waiting is necessary.

The following is an example of the eager form of **qlet**. Suppose we need to compute very many values of several computationally expensive functions, but suppose we store selected values in a table on secondary storage. Suppose that the lookup procedure, **lookup-stored-fun**, takes a function and its arguments, and returns a location descriptor which may contain the value of the function and which can be used to store the value. Then we might wish to optimize the use of such functions as follows:

```
(defun function-cache (fun &rest arguments)
  (qlet :eager ((value (apply fun arguments)))
    (let ((loc-desc (lookup-stored-fun fun arguments)))
      (if (value-stored-p loc-desc)
          (progn
            (kill-process value)
            (setf value (loc-desc-value loc-desc)))
          (setf (loc-desc-value loc-desc)
                (realize-future value))))
    value))
```

The predicate **value-stored-p** is assumed to indicate whether the desired value of the function already exists in the cache. The **setf** method for **loc-desc-value** is assumed to cause the supplied value to be stored in secondary storage. The primitive **kill-process** kills the process computing the value of the future **value** if it is not needed.

The function **function-cache** overlaps the computation of the expensive function **fun** with the possibly lengthy search in secondary storage for the pre-computed value of **fun**.

2.4 Excessive Parallelism

Because Lisp programs (and symbolic computations in general) are highly recursive, they can very easily generate a large number of parallel tasks—the opportunities leap out. Because any real multiprocessor will have only a finite number of processors, and because the cost of creating and maintaining a new process is non-zero, the use of *prop* during runtime to limit the degree of multiprocessing is quite important. We need only enough

parallelism to keep all the available processors busy. The **qlet** propositional parameter *prop* is a direct consequence of our design goal of limiting the number of processes created.

The following function is a frequent target of parallel benchmarking:²

```
(defun fibonacci (n)
  (if (< n 2)
      1
      (qlet t ((x (fibonacci (- n 1)))
                (y (fibonacci (- n 2))))
            (+ x y))))
```

Even though there are vastly better ways to write this function, it is illustrative in several ways. First, it is pointless to use mindless parallelism as is shown above because the cost to create and maintain a process for small values of *n* is much greater than the computation of (`fib n`). It is better to use a depth cutoff as follows:

```
(defun fibonacci (n depth)
  (if (< n 2)
      1
      (qlet (> depth 0)
            ((x (fibonacci (- n 1) (1- depth)))
              (y (fibonacci (- n 2) (1- depth))))
            (+ x y))))
```

Here processes to compute the recursive calls are spawned down to some depth. The number of processes created is at most $2^{\text{depth}+1} - 2$. The use of such depth cutoffs is typical in Qlisp programming: a recursively defined function spawns processes to compute recursive calls down to some predefined depth.

One important aspect of the treatment of parallelism is the amount of computation required per task as compared with the amount of work needed to create a task and combine its results with those of other tasks. If the **qlet** propositional parameter is true then the parent `fibonacci` process will not have much work to do, though for large arguments, the addition can be a significant amount of work because Common Lisp supports arbitrary precision integer arithmetic. The **qlet** propositional parameter serves to increase the amount of computation per task.

If the above code was to be run on a system with four processors then one might expect that the best performance could be achieved by using a depth of two. However this is not the case as the amount of work required by each process will vary considerably.

² The discussion here is based on various experiments [3] done to test different ways to limit parallelism. Running the various Qlisp programs was quite educational as the results often did not correspond to our intuitions.

Three of the processes will finish well before the last, and those three processors will then be idle for the rest of the computation. By increasing the depth of the tree of processes created, the granularity of the computations done by the final nodes becomes smaller, and when a processor finishes one task, it can get another one. The result of this is to balance the work load more evenly among all of the processors, keeping them all busy and finishing the entire computation sooner. As the depth is increased beyond the optimum point, then the additional time needed to create more processes starts to slow the computation down as expected.

A better measure of when to spawn processes then is the amount of computation that each process has to do. After several runs, a programmer may have determined typical usage patterns and might be able to estimate the amount of computation for each process. Another approximation to the amount of work for each process can be based on the size of the data structures on which the processes will operate. When operating on tree structures, this approximation is essentially the number of nodes below a certain point in the tree. Note that a depth cutoff can depend only on the size above a certain point.

For a function like `fibonacci` the amount of computation is directly related to the argument `n`. This can be used to spawn additional processes whenever `n` is greater than some predetermined cutoff value. Beneath that cutoff no additional parallelism would occur.

Another predicate to use for the propositional parameter is `qemptyp`. This returns true if there are no tasks in the queue. Therefore, if progress is good as measured by this predicate, it usually is reasonable to spawn an extra task:

```
(defun fibonacci (n)
  (if (< n 2)
      1
      (qlet (qemptyp)
            ((x (fibonacci (- n 1)))
             (y (fibonacci (- n 2))))
            (+ x y))))
```

Using a predicate like `qemptyp` results in behavior that is quite different than that of using a cutoff, since it depends very strongly on the interactions of all the running processes. This can sometimes be quite desirable, but for a function like `fibonacci` it is a disaster—since most of the calls to `fibonacci` are for very small values of `n`, it will be these trivial calculations that will first detect when the run queue becomes empty, and so they will spawn most of the new processes. For `fibonacci`, combining `qemptyp` with a cutoff based on the argument value eliminates this problem.

Many recursive programs share the property that each recursive call requires less computation than its parent. From our point of view this has serious implications when the amount of computation at a level is comparable to the amount of overhead required to create and maintain processes. If process creation is eliminated below the point at which process overhead dominates, the effectiveness of a parallel program depends on how closely the actual scheduling of processes to processors approximates ideal scheduling. Note that the actual scheduling can depend on which processors suffer page faults and when. The use of a fixed depth as a control on process spawning approximates the ideal of not spawning too-small tasks only for a small range of argument values. The use of a cutoff based on argument values directly implements the ideal, but the knowledge of the cutoff is not adaptively obtained, and the exact value to use as a cutoff can vary depending on the details of scheduling. Therefore, the adaptive policies of checking for idle processors or an empty run queue coupled with a cutoff is probably a close approximation to the ideal, assuming that the cost of running the policy is not too large.

2.5 *AND/OR-parallelism*

The construct **qlet** is an example of AND-parallelism—where there is a set of tasks to do and all of them must be completed. We also need a way to specify OR-parallelism—where there is again a set of tasks, but now when the first task is successfully completed, the other tasks can be abandoned. The initial design of Qlisp proposed to do OR-parallelism by combining **qlet** with the explicit killing of processes. This will work, but the resulting code often seems unnecessarily awkward and unclear. We now feel that providing Qlisp constructs to directly express AND/OR-parallelism will result in higher quality Qlisp programs that will be easier to write and will more clearly communicate the programmer’s intent.

To do this we generalize the notion of a future to allow several processes to be associated with it, along with a combining function. As each process finishes, it calls the combining function with the value of the form it has just finished computing. When all of the processes have completed, the future will be realized. For example if the combining function is $+$, then the sum of all the values computed by the associated processes will be the value of the future; if it is **max**, then the maximum value returned by the processes will be the future’s value. OR-parallelism is accomplished by also associating an end test predicate with the future: When the value computed by a process satisfies this end test, then the future will be realized immediately, and any processes associated with the future

that have not yet finished will be killed. We distinguish between a simple *lightweight* future whose value is computed by one process, and the more complex *heavyweight* future where several processes are involved in computing the value of the future.

Heavyweight futures are created by using an extended definition of **spawn**, which accepts arguments to specify a combining function, an end test, and multiple forms to be evaluated. Additional processes can be added to a heavyweight future by passing the same future to several calls to **spawn**. The following illustrates this. The problem is to find the minimum for a function of one real variable within a given interval. The strategy is to break up the interval into n equal subintervals and to have each process search its subinterval for a local minimum. The mesh should not be any finer than the value supplied by the parameter `delta`.

```
(defun minimum-function (f lower upper delta n)
  (let ((f-min (spawn t :combine
                    #'(lambda (report1 report2)
                        (if (< (min-value report1)
                              (min-value report2))
                            report1
                            report2))))))
    (let ((dx (/ (- upper lower) n)))
      (dotimes (i n)
        (let ((subinterval lower))
          (spawn t :future f-min
                 (minf f subinterval (+ subinterval dx) delta)))
          (incf lower dx))
        (realize-future f-min))))))
```

The function `minf` does the actual search within a subinterval. It returns a data structure that includes the minimum value found for `f` within the interval and the argument for which that minimum is attained. The form `(min-value x)` extracts the minimum value from the data structure.

A new heavyweight future, `f-min`, is created that will find the data structure which represents the overall minimum. Initially no processes are associated with the future. Then n processes are added to the future using **spawn**. The future is realized to return the data structure representing the location of the minimum for the function `f`.

2.6 QLAMBDA

The parallel constructs described above are primarily intended to create a new process that will perform a specific task and then go away when the task is completed. We also need to provide for another class of parallel operations: where a task is repeated many

times, usually at the request of other processes. Monitors are an example of this class. The characteristics are (1) that the process can be shared by many other processes, (2) that the requests to the process are sent via messages and stored in a queue, and (3) that the process fully completes the work requested of it by one process before starting on the next request. The way to do this in Qlisp is with the **qlambda** construct:

```
(qlambda prop (lambda-list) . body)
```

which is used to create a closure for the code in *body* similarly to an ordinary **lambda**. The form *prop* is again evaluated first, and if its value is **nil** then no new process is created; when the **qlambda** is subsequently invoked it is treated much like a normal function call. The difference in this case between **qlambda** and a regular function defined with **lambda** is that if two processes call the same **qlambda** function, the first call to it will be completed before the second call is commenced—the second process calling it must wait for the first call to complete. The *body* of the **qlambda** constitutes a critical region. We will use the term *integrity* to refer to this property of **qlambda**. Process closures can thus be used to restrict access to various system resources and data structures.

If *prop* evaluates to non-**nil**, then a new process is created and associated with the closure. When the closure is later invoked, the calling process will evaluate the arguments and send them in a message to the process closure. A future will be returned to the calling process as the value of the call on the **qlambda**. The process associated with the closure will then do the appropriate lambda-binding, evaluate *body*, and then return the result to the calling process by realizing the future. If the evaluation of the **qlambda** body makes any use of special (dynamic) variables, these variables are looked up in the environment of the calling process, rather than the environment where the **qlambda** was defined. This is in keeping with the function calling nature of **qlambda**. The process closure has a queue of requests associated with it, and when it is invoked the arguments and calling process are added to the end of this queue. The body of the process closure is fully evaluated before the next set of arguments at the head of the queue is processed. Multiple invocations of the same process closure will not create multiple copies of it.

If *prop* evaluates to **:eager**, the new process closure will immediately begin the evaluation of its body. Any arguments are bound to unrealized futures and, if one is needed, the process will block unless the future has been realized by a call on the **qlambda**. Similarly, if the evaluation of *body* completes before the **qlambda** has been called, the process again needs to block.

Note that a call to **qlambda** returns a closure as its value. This closure can then be passed to functions as an argument, returned from functions as a value, or stored in a data

structure. The closure can then be invoked to execute the body of the **qlambda**. This allows us to treat processes as first-class data objects. Also note that because a closure is created, **qlambda** captures the local environment in effect when it was defined. This can be used to create variables that can only be referenced in the body of the **qlambda**.

Locally named functions can be defined in Common Lisp with **flet** and **labels**. Qlisp extends these constructs to define local process closures with **qflet** and **qlabels**.

Here is an example of the use of **qlambda** using **qflet**:

```
(defun print-leaves (trans tree stream)
  (qflet t ((print-leaf (string)
                     (dotimes (i (length string))
                               (output (funcall trans (elt string i)) stream))))
    (labels
      ((worker (tree)
               (cond ((null tree) nil)
                     ((atom tree) (print-leaf (coerce tree 'string)))
                     (t (spawn t (worker (car tree)))
                        (worker (cdr tree))))))
      (worker tree))))
```

This function is used to traverse a tree in some order, outputting all of the leaves to a stream. A transformation function, **trans**, is passed as an argument and is used to map characters in the strings associated with each leaf; these transformed characters are output to the stream **stream**. The **qlambda** is created by the form **qflet**. We have used a **qlambda** for two reasons. One is to guarantee that the transformed characters from the strings aren't mixed up. The other reason, which is not apparent from the code, is that we want to get the traversal over with so that the tree can be modified while the process that is outputting the strings moves ahead at its own pace. The integrity property of **qlambda** accomplishes the first goal, and the use of a separate process accomplishes the second.

The original design of Qlisp had calls on a process closure explicitly wait for the body of the **qlambda** to process the calling arguments. To get the full benefit of process closures, the additional constructs, **wait** and **no-wait**, had to be added. By changing the design such that when a process closure is called a future is immediately returned, these additional constructs are no longer necessary.

2.7 Locks

Another way to interlock critical code sections is to explicitly use a lock. Qlisp provides basic functions to create, acquire, release, and test locks. When created, a lock can be specified to be either a spin or a sleep lock. When a process waits on a spin lock, it will

busy wait, continually checking the lock until it becomes free. A process waiting on a sleep lock will block and not consume computing resources while it waits. When the lock becomes available the process will be given ownership of the lock and added to the queue of runnable processes. Spin locks are intended for use by critical regions that need to be locked for only a very small amount of time, for example to safely update a counter or to get the next element of a queue. Sleep locks require more overhead and are intended for use when the code that is being interlocked will take an arbitrary amount of time to execute.

2.8 Process Synchronization

A construct to simplify process synchronization is the event. Qlisp provides basic functions to create, test, wait for, signal, and reset events. When waiting for an event the default is to wait for the event to be signaled once. If the event is signaled before the call to wait on it, then no actual waiting takes place. Otherwise the process is put to sleep until another process signals the event. When the event is signaled all of the processes waiting on it are awakened. After an event has been signaled, it must be reset before any process will need to wait on it again. It is also possible to request that a process wait until the event has been signaled a specified number of times.

With futures and process closures it becomes possible to spawn a large number of tasks, and not be able to easily determine when they have all completed. The construct (`qwait form`) will cause *form* to be evaluated and return its value. If the evaluation of *form* causes any new processes to be created or makes any calls to process closures, then **qwait** will wait for them to finish before it returns the value of *form*. This can be useful for process synchronization and for guaranteeing that returned data structures contain only realized futures.

In the original Qlisp design a variation of this functionality was provided by **qcatch**. We now feel that the ability to wait for processes to finish is important enough to warrant a separate construct, hence the addition of **qwait**. This also allows us to reserve **qcatch** for a use that is more like the standard Common Lisp construct **catch**.

2.9 *Killing Processes*

So far we have described a number of constructs to create processes, but we have not yet said much about how to get rid of these processes when they are no longer useful. Each process we create consumes system resources and for efficiency we would like to eliminate a process as soon as it is no longer contributing to the overall computation. However determining when a process is superfluous is non-trivial.

The traditional way that Lisp reclaims resources that are no longer being used is via garbage collection. When it can be determined that no pointers exist to an object in memory, then that memory can be reclaimed for later use. Similarly when no pointers exist to a future, then the value of that future is no longer accessible and there is no point in continuing to work on computing it, so any processes associated with the future can be killed. A process closure can likewise be killed when there are no longer any pointers to it, provided that it has completed all of the previous calls to it. There are several problems with relying on garbage collection to kill no longer needed processes. First, garbage collection does not occur frequently (one hopes), so the interval can be quite long between when a future is no longer pointed to and when the process computing it is actually killed so that it is no longer using system resources. Second, if a task was spawned for effect no pointer to the associated future may ever be retained, as the value will never be used.³

Qlisp provides two explicit ways to kill a process. The simplest way is to call the Qlisp construct **kill-process** which takes as its argument a future or a pointer to a process closure. This future is used as a handle to refer to the process associated with the future. That process is then killed. If there are several processes associated with the future then they are all killed. If the future is associated with an invocation of a **qlambda** process closure, then that set of arguments is removed from the process closure's queue, or, if they were currently being processed by the process closure, then it will abort the computation and proceed to the next set of arguments. Only the specific invocation of the process closure that is associated with the future is aborted; the process closure itself is not killed. Attempting to get the value of a future whose associated process has been killed is an error.

³ We are investigating ways of allowing the programmer to specify the dynamic extent of a process so that it is not necessary to maintain lists of all those processes performing useful work, but that will not be returning a value.

The other way to explicitly kill a process is to do a non-local exit from the process. In Common Lisp if a computation is surrounded by a **catch**, then a **throw** to that **catch** will force a return with the specified value, terminating any intermediate computations. In Qlisp **throw** can be used to kill other processes. For example, here is a function to determine if two binary trees are equivalent:

```
(defun tree-equal (x y)
  (labels
    ((equal-aux (x y)
      (cond ((eq x y) 't)
            ((or (atom x) (atom y))
             (return-from tree-equal 'nil))
            (t
             (qprogn t
                    (equal-aux (car x) (car y))
                    (equal-aux (cdr x) (cdr y)))
             ))))
    (equal-aux x y)))
```

Processes are spawned to compare corresponding branches of the two trees. If a process finds that two leaves are different, then the **return-from** returns **nil** and causes all of the other processes examining the tree to be killed. Note that **return-from** is equivalent to **throw**, except that block names are lexically scoped while **catch** tags are dynamically scoped. If the trees are equal then no **throw** will be done, and **tree-equal** will return **t** after all of the spawned processes have finished. The construct **qprogn** spawns a new process for each form in its body, evaluating them in parallel.

Normally when a process is killed, any processes that it spawned will not be affected. In cases like the example above where the parent process has spawned child processes via a **qprogn** or **qlet** and is waiting for them to finish, if the parent is killed or if a **throw** causes control to leave the parallel construct, we can safely kill all of the child processes.

When a process is created it inherits the chain of catch frames being used by its parent. During the execution of the child process, a **throw** to a catch frame defined by the parent will result in the child process being killed and the parent process continuing the processing of the **throw**, interrupting whatever it had been doing. If the parent process exits the scope of a given catch frame, then it is no longer possible for any child process to throw to that catch frame. Also any catch frames established after a child process has been spawned are not part of the child process's chain of catch frames. If the body of a **qlambda** process closure does a **throw**, the catch frames of the process that called the **qlambda** are searched rather than those in the process that created the **qlambda**.

The various constructs for AND/OR-parallelism provide additional opportunities to kill processes because of a **throw**. If one of the processes associated with a heavyweight future is killed by a throw, then all of the other processes associated with the future may also be killed. One of the reasons for adding heavyweight futures to Qlisp is that they define a set of processes having similar lifetimes.

As an aside, sometimes it is useful to place a process in a suspended state, so that it does not compete for computing resources, and possibly resume it later. Qlisp does this with the primitives **suspend-process** and **resume-process** which also use a future or a process closure to point to the processes to suspend or resume. When suspending a process care must be taken that the process does not currently own some system resource such as a lock. A deadlock situation can occur if a process tries to acquire a lock that is owned by a suspended process. A similar problem arises if a process needs the value of a future that is associated with a suspended process. A simple solution to avoid these two types of deadlock that we are considering is to automatically resume the suspended process in such a case.⁴

3. Need for Higher-Level Constructs

One important aspect of the initial experience with Qlisp has been doing a preliminary analysis of how well Qlisp constructs model the program structures needed for parallel programming. In many ways the **qlet** construct models quite well an important class of programs.

It was thought that **catch/throw** and **qlambda** would be important in modeling two other important classes of programs: one class involving killing processes and the other class involving autonomous agents. However, we have seen that there is some inadequacy here, even though our experience has been limited. One aspect of the inadequacy can be seen when we look at one particular problem which would seem to be well-suited to the **qlambda** approach.

The problem is, given three processes where two are supplying arguments to the third, how are the arguments to be correctly paired without excessive synchronization code?

⁴ This also allows us to do demand-driven evaluation in a manner similar to the **delay** primitive in Multilisp [4].

3.1 *Partially, Multiply Invoked Functions*

We have been exploring a solution to this class of problems. Our solution is termed *partially, multiply invoked functions* or *PMI* functions. The basic idea is to separate the process of coordinating the arrival of arguments from the actual processing of arguments by the function. A related approach can be found in [5]. We prefer a functional approach rather than a stream-based approach in order to minimize the number of large paradigms a programmer must keep in mind.

In Common Lisp parameters can specify how arguments are to be passed and whether they are required. Required arguments must be passed by position, and optional arguments may be passed by position or by name. If an optional argument is not passed, a default value is supplied. In all cases all supplied arguments must come from the same source.

We are experimenting with a technique in which all arguments to a function are passed by position to the function by an *interface* to the function. The interface accepts only named arguments, provides for all defaulting, and coordinates the arrival of arguments from multiple sources for the function.

Here is a simple example of the technique:

```
(pmi-defun add-up (x y) (:summand :summand) (+ x y))
```

This function adds up a pair of arguments, called `x` and `y`. The interface to this function names both of these arguments `:summand`, because there is no particular need to have different names for them.

The expression:

```
(add-up :summand 1 :summand 2)
```

simply produces the answer 3. However, one can partially invoke the function as follows:

```
(add-up :summand 1) → future
```

In this case the interface remembers the supplied argument and returns a future. A second call will complete the invocation and supply a value to the future:

```
(add-up :summand 2) → future = 3
```

This technique, then, is not unlike currying functions, but because all arguments to the interface are named, one does not need to curry in any particular order. All calls to a PMI function that supply arguments to the same invocation receive the same future as their value. A future is returned whenever some required arguments to the function have not been supplied to the interface by a function call. If a particular invocation of a function has returned a future, the value returned when all required arguments have been supplied is a realized future. This is to preserve **eq**-ness of all values returned for a particular invocation.

When the names of the arguments to a PMI function are different, it is possible to stream arguments to it from different sources. For example, we can produce a list of the sums from two streams supplied by two processes as follows:

```
(let ((answer (make-queue)))
  (pmi-qflet
    ((add-stream (x y) (:summand1 :summand2)
      (add-queue (+ x y) answer))))
  (qprogn t
    (loop ...
      (add-stream :summand1 computation)...)
    (loop ...
      (add-stream :summand2 computation)...)
    answer))
```

The **pmi-qflet** expression creates a local PMI function in a process closure. The details of queue management are elided.

Sometimes the elements of a stream of arguments will get out of order. In this case we can exploit a further wrinkle on PMI functions. What we would like to do is associate a secondary tag with each argument, so that arguments with matching secondary tags are paired. This would correspond to the tagged token architectures used by dataflow. For our experiments we have substituted the concept of *colored* arguments, where the color of a set of arguments is explicitly passed as follows:

```
(add-up :color 1 :summand 1) → future1
(add-up :color 2 :summand 2) → future2
(add-up :color 2 :summand 3) → future2 = 5
(add-up :color 1 :summand 5) → future1 = 6
```

All PMI functions accept colored arguments. When a color is not supplied, a default, private color is used for that invocation.

Another reason for colored arguments is to insulate invocations from separate parts of the overall computation from each other.

At present we are not sure how well the mechanisms of partially, multiply invoked functions help programmers to understand and program in parallel effectively. But it seems clear that constructs that allow a programmer to ignore the details of synchronization are essential.

4. Summary

Qlisp has been proposed as a language for programming multiprocessors. An initial implementation of it has been done, and various experiments performed. Results to date indicate that the performance of Qlisp programs is good. The more interesting set of

results concern how well Qlisp captures the intuition programmers have about parallel programs. Here we have found that our original conception of Qlisp requires modification. Some constructs, such as **qlambda** and **throw**, might be too low level to be easily used. To address this concern, our current strategy is to examine problems with natural parallel solutions and to find parallel constructs that express those solutions well.

References

- [1] Henry G. Baker, Jr. and Carl Hewitt, *The Incremental Garbage Collection of Processes*, Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages, August 1977.
- [2] Richard P. Gabriel and John McCarthy, *Qlisp* in **Parallel Computation and Computers for Artificial Intelligence** edited by Janusz S. Kowalik, Kluwer Academic Publishers, 1988.
- [3] Ron Goldman and Richard P. Gabriel, *Preliminary Results with the Initial Implementation of Qlisp*, Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, July 1988.
- [4] Robert H. Halstead, Jr., *Multilisp: A Language for Concurrent Symbolic Computation*, ACM Transactions on Programming Languages and Systems, Vol 7, No. 4, October 1985, pp 501-538.
- [5] John Lamping, *A Unified System of Parameterization for Programming Languages*, Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, July 1988.
- [6] James S. Miller, **MultiScheme: A Parallel Processing System Based on MIT Scheme**, PhD thesis, MIT, August 1987.
- [7] Guy L. Steele Jr. et. al. **Common Lisp Reference Manual**, Digital Press, 1984.
- [8] Mark R. Swanson, Robert R. Kessler, and Gary Lindstrom, *An Implementation of Portable Standard Lisp on the BBN Butterfly*, Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, July 1988.