Parallelism in Lisp

Guy L. Steele Jr.
Sun Microsystems Laboratories*
2 Elizabeth Drive
Chelmsford, Massachusetts 01824
Guy.Steele@East.Sun.com

Maybe not as hot a topic in computer architecture as it used to be, but still of considerable interest, is parallelism. How do you make a faster computer? Just strap 20 or 200 or 2000 processors together? As we have learned, the architectural and hardware difficulties are immense (How do you connect them? A shared bus? A network? Is there a single system clock or many clocks?), and after these have been solved there remains the matter of programming.

You could just add parallelism to an existing sequential language such as C, Pascal, Fortran, or Lisp by adding a few subroutines: one to spawn a new process, one to terminate a process, and perhaps *send-message* and *wait-for-message*; or instead of message-passing, one could rely on shared memory and a *test-and-set* procedure. This approach amounts to making you code parallelism at the conceptual level of assembly language, disguised by the fact that the arithmetic expressions, loops, and conditionals are written in a higher-level language.

More thoughtfully designed parallel languages begin with an *idea* about how to use parallelism in a conceptually convenient and abstract manner; the language is then organized to support and complement the idea. But if the rest of the language design is simply borrowed wholesale from an existing language, semantic clash may result.

The "garbage-can" approach to language design is seldom successful. You can't always simply add another feature. The whole may be *less* than the sum of its parts, for, as we will see, the introduction of a new feature can diminish the expressive power of existing features or lay traps for the unwary.

How to Evaluate a Parallel Language Feature?

There have been many ideas about how to exploit parallelism in a programming language, some more successful than others. Many of these ideas have been tried out in Lisp, again with varying degrees of success. But the measure of success has been largely subjective. How can we evaluate different features beyond a judgement of "I find this one easier to use" or "That's gross!"?

One can analyze a programming language by dividing its features into three categories: primitive notions, combination, and abstraction. Combining features allow large ideas to be built up be aggregating smaller pieces. Abstraction features allow these combinations to be packaged up and treated as if they were primitive.

The design of a programming language can then be evaluated on the generality and completeness of features in each category. For example, in Pascal [11] numerical constants and names are primitive. Arithmetic operators can be regarded as combining features; one can build up such expressions as `b*b-4.0*a*c`. Parentheses (and implicit parentheses introduced by the rules of

*Most of this article was written while the author was at Thinking Machines Corporation. The article was originally prepared in 1987 for *BYTE* magazine, whose editors accepted the article for publication as part of a special issue and then decided, at the last minute, not to use this one article for reasons of space. *BYTE* cordially returned the rights to the article so that it could be published elsewhere. I have further revised the article slightly with *Lisp Pointers* and the progress of the last seven years in mind.

precedence) are an abstraction feature; an arithmetic expression, once packaged up in parentheses, can be used almost anywhere a numerical constant or name can. However, the abstraction of parenthesization is not completely general. One may use the numerical constant 5 in a declaration such as

```
const Philosophers = 5;
```

but one may not use an equivalent expression:

```
const Stooges = 3;
const Philosophers = (Stooges + 2);  (* Illegal *)
```

and this is an objective basis on which to criticize the design of Pascal. (This design defect was later corrected in Modula-2 [17].)

Here we shall examine a number of different ideas for introducing parallelism into Lisp. The goal is not merely to enumerate a variety of approaches, but to judge how well the approach to parallelism fits into the rest of the language. Parallelism is a method of combination: it arranges for a number of program fragments to execute concurrently. Our yardstick then is this: how well does the resulting language support abstraction? Does the particular approach to introducing parallelism enhance the abstraction mechanisms or violate them?

Lisp as a Language Laboratory

The Lisp language provides an effective and efficient framework for experimenting with and evaluating new language ideas. This is a result of two important characteristics of Lisp.

First, there is a standard representation of Lisp programs as Lisp data structures, and a standard way of writing an interpreter for Lisp programs in the Lisp language itself. This means that you can invent a new dialect of Lisp simply by taking an existing Lisp interpreter, changing it a bit, and then executing that interpreter within an existing Lisp system. The interpreter can be written so that the new dialect has access to most of the facilities of the existing dialect. This means that only a small effort is required to try out a small change to the language. It is not necessary to write a new parser, or new mathematical functions, or new runtime I/O procedures.

Second, the number of absolutely essential concepts in the language is small. A production-quality dialect of Lisp, such as Common Lisp [13], may contain hundreds of operations. But a dialect is still recognizably Lisp even if it contains only the data types `symbol` and `list`; the programming primitives `lambda`, `if`, quotation, function call, variable reference, and perhaps `set!` for assigning to variables; and the primitive functions `cons`, `car`, `cdr`, `atom`, and `eq`. (The Scheme dialect of Lisp [15, 10] started out almost this small; but, indeed, even this short list of features is not absolutely minimal.) An interpreter for such a tiny dialect requires only one page of code, and yet addresses most of the interesting issues of how to execute Lisp programs. This means that a proposed new feature for Lisp can often be judged by its interaction with only a dozen constructs.

We are interested in how parallelism interacts with abstraction in Lisp. If we restrict our attention to our tiny dialect, we see that there are two means of abstraction. First, anywhere a variable reference can occur, an arbitrary expression may appear instead. (This corresponds to the parenthetical abstraction in Pascal that we discussed above.) Second, `lambda` may be used to package up an arbitrarily complex expression as a primitive procedure. If we believe (as many Lisp experts do) that our tiny dialect does indeed capture most of the important aspects of Lisp, then we need consider only these two abstraction mechanisms when judging features for parallelism. (We note in passing that it is possible to make do with only one means of abstraction; the arbitrary

nesting of expressions may be avoided by writing Lisp code in the so-called *continuation-passing style*. However, we are interested in the customary style of expression rather than the theoretical minimum. Because nested expressions are convenient and customary, we do wish to analyze their interaction with parallel features.)

One note of caution is in order. Some approaches to parallelism rely on using side effects, such as assignment to variables, to communicate among processes. In these cases we must recognize that side effects themselves can violate both forms of abstraction, and then determine whether the particular approach to parallelism ameliorates or exacerbates the problem of side effects.

Idea 1: Completely Independent Processes

The most obvious way to add parallelism to Lisp is simply to organize a system as a set of completely independent sequential Lisp processes augmented with access to some sort of interprocess communication facility. By "independent" we mean not only that execution within each process proceeds asynchronously with respect to other processes, but that the data structures belonging to one process are not directly accessible to other processes. In other words, different processes have disjoint address spaces.

Suitable means of interprocess communication include message-passing (either synchronized or buffered), I/O streams (such as UNIX pipes), and remote procedure calls. If the data to be transmitted between processes has a simple structure, such as numbers or character strings, this approach is not so bad: communicating with another process and having an effect on it is not particularly much worse than the side effect produced by an assignment statement. However, Lisp supports a rich set of data structures that cannot be transmitted to an independent address space without destroying certain properties, such as object identity, that are important to Lisp programming style. This prevents certain procedures that must operate on such data structures from making effective use of parallelism.

For this reason most approaches to parallelism in Lisp have supported the illusion of a single address space. In such a model every process has access to the same data structures. (It may be, however, that one process can access a given data structure more efficiently than another—that is, data may be "belong" to a process and be considered "local" to it.)

Idea 2: Processes in a Shared Address Space

For concreteness, consider a version of Scheme augmented with a primitive function `start-process` that takes a *thunk* (a piece of code encapsulated as a function of no arguments) and spawns a new process to invoke the thunk. The old process (that called `start-process`) continues execution; it sees the value `#t` returned from the call to `start-process`. The new process proceeds to execute the code in the thunk concurrently. If the thunk ever completes execution, then the value it returned is discarded and its process effectively ceases to exist.

By itself `start-process` is not very useful. It allows new processes to be created, but without a way for them to communicate. The new processes can share data structures, but rules must be established for simultaneous access. An obvious rule to establish is that references to any given variable, and assignments to that variable using `set!`, are in effect serialized; the system behaves as if each process is made to wait its turn. (Although this may be an obvious solution, it is of utmost importance to state it explicitly, because it may affect the underlying implementation. Imagine an implementation of Lisp on a 16-bit microprocessor with a 24-bit address space. Two processes A

```
(define (sieve integers)
  (begin (sieve-loop integers (sqrt (last integers)))
         (remove #f integers)))

(define (sieve-loop integers limit)
  (if (first integers)
      (if (not (> (first integers) limit))
          (begin (sieve-step (first integers) (rest integers))
                 (sieve-loop (rest integers) limit)))))

(define (sieve-step candidate integers)
  (if (not (empty? integers))
      (begin (if (first integers)
                 (if (= (mod (first integers) candidate) 0)
                     (set-first! integers #f)))
             (sieve-step candidate (rest integers)))))
```

Figure 1: Sieve of Eratosthenes Using Side Effects with One Process

and B attempting to store the address of a Lisp data structure into the doubleword at location V might proceed as follows:

```
process A stores into location V
process B stores into location V
process B stores into location V+1
process A stores into location V+1
```

The net result would be garbled data in the doubleword at location V.)

We will assume that variable references and assignments are serialized, and therefore may be regarded as atomic actions. We will furthermore assume that references to elements of lists, and alterations of such elements using the procedure `set-car!` (which alters a list to have a different first element), are also serialized. Then we can write a parallel version of the Sieve of Eratosthenes.

It is easiest to explain the parallel algorithm by first discussing a serial version of the algorithm (see Figure 1). The procedure `sieve` takes a list of integers of the form `(2 3 4 5 6 ...` n`)`. It modifies the list by changing every non-prime element to `#f`, and then produces a list of primes by removing the `#f` elements from the modified list.

The procedure `sieve-loop` is essentially a loop that calls `sieve-step` once for every candidate value between 2 and $\sqrt{n}$. The job of `sieve-step` is to remove all multiples of the candidate from the list of integers by overwriting such multiples with `#f`. The code of `sieve-loop` is complicated by the fact that the candidate value may have been a multiple of a previous candidate, and so may already have been set to `#f`. The same complication arises in `sieve-step`, where the next potential multiple may already have been overwritten because it was a multiple of a previous candidate.

As an example, if the original argument list is

```
(2 3 4 5 6 7 8 9 10 11 12 13 14 15)
```

then the first call to `sieve-step` from `sieve-loop` modifies the argument list to be

```
(2 3 #f 5 #f 7 #f 9 #f 11 #f 13 #f 15)
```

and the next call modifies it to be

```
(2 3 #f 5 #f 7 #f #f #f 11 #f 13 #f #f)
```

The next candidate, 5, is larger than $\sqrt{15}$, and so the iteration ends and all occurrences of `#f` are removed to produce

```
(2 3 5 7 11 13)
```

which is returned as the result.

Two stylistic oddities arise from the use of side effects (that is, the use of `set-first!` to alter the list of integers). The first oddity is that `begin` is used in order to perform first one expression and then another, discarding the value of the first. (If the value of an expression is discarded, then its execution cannot affect the overall computation unless it causes a side effect.) The second oddity is that a form of `if` is used that has no "else" part: `(if E0 E1)` evaluates E0, and if the result is true then E1 is executed and its value is discarded. Whether or not E1 is executed, the value of the `if` expression is unspecified. This kind of `if` expression is useful only when executing E1 may produce a side effect.

A parallel version of this algorithm may be produced by running each call to `sieve-step` in a separate process (see Figure 2). This is achieved roughly by replacing the call

```
(sieve-step ...)
```

with the code

```
(start-process (lambda () (sieve-step ...)))
```

However, there are two complications. The first stems from having many processes performing side effects on the same list. To see this, consider the procedure `sieve-loop`. As shown in Figure 1, it examines the first element of the list of integers three times: once to see whether it is false, once to see whether it is greater than $\sqrt{n}$ (the `limit`), and once to pass it to `sieve-step`. This works in the serial case, but with parallel processes it can fail miserably if, for example, some other process sets that element of the list to `#f` after the test for falsehood but before the comparison to $\sqrt{n}$; the effect would be to apply the `>` operation to the value `#f`. To avoid this error we must rewrite the code as shown in Figure 2 so as to read the first element of the list only once, not three times, saving the value in a temporary variable (here named `candidate`). The same trick must also be employed in `sieve-step`.

We pause here to observe that expression abstraction has been sadly violated. We cannot write `(first integers)` wherever we please; we *must* write it exactly once and subsequently use a variable in its place. This violation arises from the use of side effects, but the problem is greatly exacerbated by the parallelism. It is not merely that a side effect can occur—we *want* it to occur—but that we cannot predict *when* it will occur, and that is the fault of the particular approach to parallelism.

The second complication is that, having spawned many processes to strike out non-primes in the list, we must somehow wait for all these processes to terminate before attempting to remove `#f` values from the list to produce the final result. Figure 2 shows one way to do this, by introducing additional code into `sieve-loop`. For every process spawned there is a new binding of a variable named `done`; this variable is initialized to false, and is set to true when the spawned process has completed its work. The procedure `wait` is used to busy-wait until a thunk passed to it yields a

```
(define (sieve integers)
  (begin (sieve-loop integers (sqrt (last integers)))
         (remove #f integers)))

(define (sieve-loop integers limit)
  (let ((candidate (first integers))
        (done #f))
    (if candidate
        (if (not (> candidate limit))
            (begin (start-process
                    (lambda ()
                      (begin (sieve-step candidate
                                         (rest integers))
                             (set! done #t))))
                   (begin (sieve-loop (rest integers) limit)
                          (wait (lambda () done))))))))

(define (sieve-step candidate integers)
  (if (not (empty? integers))
      (begin (let ((next (first integers)))
               (if next
                   (if (= (mod next candidate) 0)
                       (set-first! integers #f))))
             (sieve-step candidate (rest integers)))))

(define wait
  (lambda (thunk) (if (not (thunk)) (wait thunk))))
```

Figure 2: Sieve of Eratosthenes Using Side Effects with Asynchronous Processes

true value. Note that the recursive code of `sieve-loop` is arranged so that all the processes are spawned before waiting for any of them to be completed.

This may be regarded as a violation of procedural abstraction. One of the expected properties of a procedure created by *lambda* is that once you invoke it, the procedure will have finished doing its duty by the time you resume execution. The use of `start-process` violates this notion. It can cause a procedure no longer to appear to be primitive; its caller can perceive that the procedure is made up of specific smaller parts.

While Scheme augmented with `start-process` and a rule of serialization makes it easy to execute many processes in parallel, it facilitates neither process synchronization nor interprocess communication. Communication must be expressed in terms of side effects, and synchronizing process termination is particularly clumsy. (Most Lisp implementations, such as Zetalisp [16], that provide parallel processes of this kind also provide more powerful synchronization primitives, such as *test-and-set* or *compare-and-swap*.)

Particularly annoying is the fact that the value of an expression executed in another process is discarded (another violation of procedural abstraction). This makes it impossible to write programs

```
(define (sieve integers)
  (sieve-loop integers (sqrt (last integers))))

(define (sieve-loop integers limit)
  (if (> (first integers) limit)
      integers
      (cons (first integers)
            (sieve-loop (sieve-step (first integers)
                                    (rest integers))
                        limit))))

(define (sieve-step candidate integers)
  (if (empty? integers)
      '()
      (let ((x (sieve-step candidate (rest integers))))
        (if (= (mod (first integers) candidate) 0)
            x
            (cons (first integers) x)))))
```

Figure 3: Purely Functional Sieve of Eratosthenes with One Process

that use `start-process` in a style that is truly Lisp-like, that is, functional.

Idea 3: Futures

A different primitive that may be added to Scheme for parallel processing is `make-future`, which takes a thunk and spawns a process in which to invoke it. Thus far it is exactly like `start-process`. However, instead of returning `#t`, `make-future` returns to its caller a *future*, a data object constituting a promise to deliver the value of the thunk "someday." The Multilisp dialect [7], which has been implemented on more than one hardware multiprocessor, relies on the use of futures (with a slightly different syntax).

A future value can be treated as the real thing for certain purposes, such as passing it as an argument, returning it as a value, or making it an element of a list. Think of it as a dry-cleaning ticket, and better than the usual kind because the cleaner will replace it with the actual suit (when it has been cleaned) wherever the ticket happens to be at the time. If you want to give your suit to someone else, you can just hand him the ticket; he can hand it back to you; and you can even hang it in your closet in place of a suit. The only problem comes when you want to wear it. You can't wear a ticket; you have to wait for it to become an actual suit. Same thing for a future. A future number can be put into a list, but if you try to take its square root you have to wait for it to become an actual number, which means waiting for the process computing the number to finish.

In one stroke the future solves two problems: it avoids discarding the value computed by another process, and it provides a simple way to synchronize process termination. Eliminating these two problems allows parallel programs to be written in a functional style, without side effects.

Figure 3 contains a version of the `sieve` program that is sequential and uses no side effects. Instead of modifying its argument, it repeatedly makes copies of the list, eliminating all the multiples

```
(define (sieve integers)
  (sieve-loop integers (sqrt (last integers))))

(define (sieve-loop integers limit)
  (if (> (first integers) limit)
      integers
      (cons (first integers)
            (sieve-loop (sieve-step (first integers)
                                    (rest integers))
                        limit))))

(define (sieve-step candidate integers)
  (make-future
   (lambda ()
     (if (empty? integers)
         '()
         (let ((x (sieve-step candidate (rest integers))))
           (if (= (mod (first integers) candidate) 0)
               x
               (cons (first integers) x)))))))
```

Figure 4: Purely Functional Sieve of Eratosthenes using Futures

of some prime during each copy operation. (We call such a selective copying operation a *filter.*) For example, if the original argument list is

```
(2 3 4 5 6 7 8 9 10 11 12 13 14 15)
```

then the first call to `sieve-step` from `sieve-loop` returns the value

```
(2 3 5 7 9 11 13 15)
```

and the second call returns the value

```
(2 3 5 7 11 13)
```

which is then returned from `sieve`.

Figure 4 is shows a parallel version of this algorithm that uses futures. Figure 4 is identical to Figure 3 except that a call to `make-future` has been introduced into `sieve-step`. This means that a call to `sieve-step` will return immediately with a promise to deliver a filtered list later. This means that `sieve-loop` does not need to wait for `sieve-step` to finish making its filtered copy of the argument list before going on to the next iteration. This allows all of the filtering steps to begin their work in parallel. It also allows `sieve-step` to make each element of its result available immediately, without waiting for the rest of the elements to be computed. Therefore the filtering operations may proceed in parallel, each hot on the heels of its predecessor, rather than requiring one filtering operation to finish before the next begins.

We can conclude that although the `make-future` construct may be superficially similar to the `start-process` construct, it is sufficiently different to permit a strikingly different pattern of use.

Because `make-future` does not discard the value of its argument thunk, side effects are not needed to deliver results from the spawned process.

Indeed, it is not difficult to show that futures may be introduced anywhere in a functional (side-effect-free) Lisp program without affecting its correctness. Only its speed will be affected (possibly adversely, because the use of futures involves some overhead). But what if side effects are used anyway in conjunction with futures? Ah, then we still have a distressing violation of procedural abstraction. The process created for a future may survive long after the return of control from the procedure that created the future. If the process for the future performs a side effect, there is still no way to predict when it will occur; it may happen arbitrarily far in the future (no pun intended). We may summarize the problem thus: while a future allows a spawned process to deliver a value, it places no constraint on how long it can take to do so.

One way to solve this is to introduce a primitive `touch` that simply returns its argument after making sure that it has completed if it is a future. It is usually awkward, however, to introduce calls to `touch` in all the necessary places in a program. And yet, one common pattern of using `touch` in conjunction with futures suggests another, more constrained, approach to functional parallelism.

Idea 4: Parallel Argument Evaluation

The construct `pcall` performs a function call by first performing all the argument computations in parallel and then passing them to the called function only when all the arguments are available. One may describe the action of (`pcall` *f a1 a2 ... an*) as

```
(let ((v1 (future a1))
      (v2 (future a2))
      ...
      (vn (future an)))
  (touch a1)
  (touch a2)
  ...
  (touch an)
  (f v1 v2 ... vn))
```

The parallelism afforded by `pcall` is limited in duration; it guarantees that spawned processes will complete before certain other actions (the call to the function and all subsequent actions) are begun. This limitation is sufficiently stringent, however, that it is difficult to render the sieving method of Figure 4 using only `pcall`.

An alternative approach is shown in Figure 5. The idea is not to limit the filtering processes to eliminating primes, because that implies that a new filtering step cannot begin until a new prime is found (by the preceding filtering step). Instead we have a filtering process for every value in the original argument list (up to the square root limit), whether prime or composite. These filtering processes can proceed completely independently; the results are then merged by another series of parallel processes. The function `merge2` simply takes two lists of numbers, each in ascending order, and produces a list of numbers appearing in *both* input lists. The function `sieve-merge` takes a list of many lists and organizes them into a binary tree with a `merge2` computation at each parent node. It does this by grouping the lists into pairs and performing `merge2` on each pair in parallel, producing half as many result lists; the pairing process is then iterated until only one list remains. (This pairing method allows much more parallelism than the more obvious approach of starting with the first list and successively merging in the others.)

```
(define (sieve integers)
  (first (sieve-merge (sieve-loop integers
                                  (sqrt (last integers))))))

(define (sieve-loop integers limit)
  (if (> (first integers) limit)
      (list integers)
      (pcall cons
             (sieve-step (first integers) (rest integers))
             (sieve-loop (rest integers) limit))))

(define (sieve-step candidate integers)
  (if (empty? integers)
      '()
      (let ((x (sieve-step candidate (rest integers))))
        (if (= (mod (first integers) candidate) 0)
            x
            (cons (first integers) x)))))

(define (sieve-merge lists)
  (if (or (empty? lists)
          (empty? (rest lists)))
      lists
      (sieve-merge
       (pcall cons
              (merge2 (first lists)
                      (first (rest lists)))
              (sieve-merge (rest (rest lists)))))))

(define (merge2 list1 list2)
  (cond ((empty? list1) list2)
        ((empty? list2) list1)
        ((<? (first list1) (first list2))
         (merge2 (rest list1) list2))
        ((>? (first list1) (first list2))
         (merge2 list1 (rest list2)))
        (else (cons (first list1)
                    (merge2 (rest list1) (rest list2))))))
```

Figure 5: Functional Sieve with Independent Filtering Processes

This is a much more brute force approach than previous algorithms. The idea is to gain *speed* at the expense of *efficiency*. By increasing the parallelism and throwing a much larger amount of computation at the problem, knowing that some of it is redundant and therefore wasted, we can hope to get the answer with less net delay. The tradeoff between speed and efficiency is one of the puzzling problems in the design of parallel algorithms.

The `pcall` primitive, like `make-future`, can be introduced anywhere in place of an ordinary function call without affecting the correctness of a program that has no side effects. (It has frequently been suggested that Lisp, or some other language, be parallelized simply by making *every* function call behave as a `pcall`. Unfortunately, this can lead to a combinatorial explosion of processes whose overhead swamps the speed saving gained from the parallelism. Finer control by the programmer is needed.)

If spawned processes to compute the arguments do produce side effects, the time interval within which the side effects take place is limited; they will occur before the function is called. This means that a procedure abstraction built from `pcall` will properly contain (in time) any side effects of procedures it invokes. We can fairly say, then, that `pcall` does less violence to procedural abstraction than `make-future` or `start-process`.

Idea 5: The Data-Parallel Approach

The patterns in which `pcall` is used in Figure 5 suggest an even higher level of parallel abstractions. Suppose that certain primitive procedures were defined to operate on all elements of a list in parallel. Obvious candidates for such definitions are operations that take functions as arguments, such as `map`, `reduce`, and `remove-if`. Define `(map f x)` to apply the function `f` to every element of the list `x`, spawning a new process for each application, and return a list of the results. Define `(reduce g x)` to use a function `g` of two arguments to combine elements of the list `x`. For example, the value of `(reduce + '(1 4 3 2))` is 10. (This can be done in parallel in the manner of the function `merge2` of Figure 5.) Define `(remove-if f x)` to apply the test function `f` to every element of `x` in parallel and return a copy of `x` in which those elements have been eliminated for which `f` did not return the false value `#f`. For example, the value of `(remove-if even? '(1 4 3 2))` is `(1 3)`.

With such primitives lists can be treated as if they were arrays whose elements can be processed in parallel. (One might actually use arrays in the actual programs instead of lists. Alternatively, there are ways to process linked lists in time logarithmic in the length of the list if one has a processor for every cell of the list [8].) Figure 6 shows a version of the sieve program written in this manner.

Idea 6: The SIMD Approach

Observe that operations such as `map` and `reduce` apply *the same operation* to a number of data items. The amount of parallelism is therefore governed more by the structure of the data in the program than by the structure of the code; the more data, the more parallelism. If the same operation is to be performed on many pieces of data, it may be wasteful to reinterpret the same piece of program text over and over again, once for each data item. In lower level terms, memory bandwidth may be wasted fetching the same instruction for each data item. A *SIMD* (Single Instruction Multiple Data) architecture may be more effective for executing programs organized in this way.

In such an architecture a single instruction, once fetched, is applied to many pieces of data before the next instruction is fetched. The advantage of such an architecture is that instruction

```
(define (sieve integers)
  (reduce merge2
          (map (lambda (candidate)
                 (sieve-step candidate integers))
               (remove-if (lambda (x)
                            (> x (sqrt (last integers))))
                          integers))))

(define (sieve-step candidate integers)
  (remove-if (lambda (integer)
               (= (mod integer candidate) 0))
             integers))

;The function merge2 is as in figure 5.
```

Figure 6: Functional Sieve with Independent Filtering Processes

fetch and decode hardware need not be replicated, and so for the same cost more hardware can be devoted to the processing of data; and less memory bandwidth is consumed by instruction fetching, and so more memory bandwidth can be devoted to the fetching of data.

The disadvantage of a SIMD architecture is that it is not possible to execute the *then* part of an `if` expression for some data and concurrently execute the *else* part for other data. Because there is only one instruction decoder, one must process first one kind of data and then the other. This may cause some data processors to idle while others are executing the "other branch" of a conditional, resulting is wasted resources. If conditional expressions are nested then the problem is compounded exponentially.

The program of Figure 6 may be regarded as a SIMD program without change; the only differences will be in its execution characteristics and the cost of the hardware. Another approach is represented by Connection Machine Lisp [14], which introduces special data structures and operators for expressing SIMD-style parallelism.

A programming language organized around a SIMD model of computation (whether or not it is actually implemented on hardware of SIMD architecture) has an important software engineering advantage. It is easier to reason about the behavior of such programs—to prove them correct, for example, or to predict their behavior when debugging—for the very simple reason that there is a single thread of control. In other words, control is always at exactly one place in the program text at a time. This means that it is not necessary to think about interactions that can arise from control being in more than one place in the program simultaneously.

The SIMD model still permits the use of side effects, but their behavior is even more constrained. If several processes perform a side effect, then they all perform their side effects at the same time, and all the side effects are of the same type.

Idea 7: Parallelism with No Side Effects

As the programs of Figures 3 through 6 indicate, if a programming language is designed with sufficient care it is possible to exploit parallelism in various ways without any use of side effects.

One might then propose to design a language for parallel programming so that there are no side effects, or so that any side effects in the language cannot have bad interactions with parallelism. The best existing examples of such language design are not Lisp dialects per se, but may be regarded respectively as first and second cousins to Lisp: the functional programming languages FP [1] and Haskell [9] and the array-oriented language APL [5]. Such a language makes the use of parallelism completely safe and avoids any violation of the abstraction mechanisms of the language. The parallelism is so completely packaged as to be inaccessible to the user. The net effect is a language that is amenable to speedier execution by parallel hardware, but is to all intents and purposes a sequential language as seen by the programmer.

Conclusions

We have by no means examined all the ways that have been tried to introduce parallelism into the Lisp framework, nor have we mentioned all the variants of ideas we have touched upon. (In particular, we have not already mentioned QLISP [4, 6], Paralation Lisp [12], and NESL [3, 2], which are worth looking up and studying.) But we can see from our few examples that the tradeoffs are not unlike those in the all too famous debate about the use of the `goto` statement. There are certainly many useful patterns of control that are easily expressed explicitly using `goto` that cannot be expressed easily using only sequencing, conditionals, and `while` loops. On the other hand, renouncing the use of `goto` makes every use of `if` or `while` implicitly more expressive, because one can rely on the notion that if control flows into the top of such a statement then control will flow out the bottom, not jump out from the middle.

We have seen that what destroys abstraction is not parallelism *per se* but side effects. To the extent that a paradigm for parallel programming requires or encourages the use of side effects, it will violate abstraction mechanisms, making programs harder to write, understand, and debug. There is a tradeoff between abstraction and flexibility: the more one can say explicitly with side effects, the less one can imply through the use of abstraction. The language designer must weigh the usefulness of explicit expressiveness against the more subtle advantages of such implicit expressiveness.

References

[1] Backus, John. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*, 8 (August 1978), 613–641. 1977 ACM Turing Award Lecture.

[2] Blelloch, Guy E. *NESL: A Nested Data-Parallel Language.* Technical Report CMU-CSD-92-103. School of Computer Science, Carnegie-Mellon University (Pittsburgh, January 1992).

[3] Blelloch, Guy E. *Vector Models for Data-Parallel Computing.* MIT Press (Cambridge, Massachusetts, 1990).

[4] Gabriel, Richard P., and McCarthy, John. Queue-based multiprocessing Lisp. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming.* ACM SIGPLAN/SIGACT/SIGART (Austin, Texas, August 1984), 25–44.

[5] Gilman, Leonard, and Rose, Allen J. *APL: An Interactive Approach*, second edition. Wiley (New York, 1976).

[6] Goldman, Ron, and Gabriel, Richard P. Preliminary results with the initial implementation of Qlisp. In *Proc. 1988 ACM Conference on Lisp and Functional Programming.* ACM SIGPLAN/SIGACT/SIGART (Snowbird, Utah, July 1988), 143–152.

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

[8] Hillis, W. Daniel, and Steele, Guy L., Jr. Data parallel algorithms. *Communications of the ACM 29*, 12 (December 1986), 1170–1183.

[9] Hudak, Paul, Peyton Jones, Simon, and Wadler, Philip, editors. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language (Version 1.1).* Technical Report. Yale University and Glasgow University (New Haven and Glasgow (respectively), August 1991).

[10] *IEEE Standard for the Scheme Programming Language*, ieee std 1178-1990 edition. IEEE Computer Society (New York, 1991).

[11] Jensen, Kathleen, and Wirth, Niklaus. *Pascal User Manual and Report.* Springer-Verlag (New York, 1974).

[12] Sabot, Gary W. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press (Cambridge, Massachusetts, 1988).

[13] Steele, Guy L., Jr., Fahlman, Scott E., Gabriel, Richard P., Moon, David A., and Weinreb, Daniel L. *Common Lisp: The Language.* Digital Press (Burlington, Massachusetts, 1984).

[14] Steele, Guy L., Jr., and Hillis, W. Daniel. Connection Machine Lisp: Fine-grained parallel symbolic processing. In *Proc. 1986 ACM Conference on Lisp and Functional Programming.* ACM SIGPLAN/SIGACT/SIGART (Cambridge, Massachusetts, August 1986), 279–297.

[15] Sussman, Gerald Jay, and Steele, Guy Lewis, Jr. *SCHEME: An Interpreter for Extended Lambda Calculus.* AI Memo 349. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, December 1975).

[16] Weinreb, Daniel, and Moon, David. *LISP Machine Manual, Fourth Edition.* Symbolics, Inc. (Cambridge, Massachusetts, July 1981).

[17] Wirth, Niklaus. *Programming in Modula-2.* Springer-Verlag (Berlin, 1982).