Parallelism in Lisp

Michael van Biema
Columbia University
Dept. of Computer Science
New York, N.Y. 10027
Tel: (212)280-2736
MICHAEL@CS.COLUMBIA.EDU

Abstract

This paper examines Lisp from the point of view of parallel computation. It attempts to identify exactly where the potential for parallel execution really exists in LISP and what constructs are useful in realizing that potential. Case studies of three attempts at augmenting Lisp with parallel constructs are examined and critiqued.

1. Parallelism in Lisp

There are two main approaches to executing Lisp in parallel. One is to use existing code and clever compiling methods to parallelize the execution of the code [9, 14, 11]. This approach is very attractive because it allows the use of already existing code without modification, and it relieves the programmer from the significant conceptual overhead of parallelism. This approach, known as the "dusty deck" approach, suffers from a simple problem: it is very hard to do. This is particularly true in a language such as Lisp that shows a much less well defined flow of control than languages such as FORTRAN where such techniques have been applied relatively successfully [13]. A result of this is that, given the current compiler techniques, the amount of parallelism that can be achieved is limited.

The other approach to the problem is the addition of so-called parallel constructs to Lisp. The idea is to allow the programmer to help the compiler out, by specifying the parallelism using special, added language constructs. This, depending on the constructs used, places a significant additional conceptual burden on the programmer. The degree of the burden depends directly on the level or the elegance and simplicity of the constructs used. The higher the level of the constructs, the lighter conceptual burden on the programmer. To put it another way, the more the constructs are able to hide and protect the programmer from the problems inherent in the parallel execution of a language with side effects such as Lisp, the better. This approach suffers from the additional problem that existing code may not be executed as is, but rather must be rewritten using these added constructs in order to take advantage of any parallelism. Finally, there is the problem of defining a set of constructs that fulfills the goals of placing the minimal conceptual overhead on the programmer while providing a complete set, in the sense that all the parallelism in any given problem may be suitably expressed using only this set.

In this paper, we focus on the second of the two approaches, because it is in this area that most recent progress has been made. We study in depth three attempts to define a useful set of parallel constructs for Lisp and discuss exactly where the opportunities for parallelism in Lisp really seem to lie. The three attempts are very interesting, in that two arc very similar in their approach but very different in the level of their constructs, and the third takes a very different approach. We do not study the so called "pure Lisp" approaches to parallelizing Lisp since these are applicative approaches and do not present many of the more complex problems presented by a Lisp with side-effects [4, 3].

The first two attempts concentrate on what we call control parallelism. Control parallelism is viewed here as a medium- or course-grained parallelism on the order of a function call in Lisp or a procedure call in a traditional, procedure-oriented language. A good example of this type of parallelism is the parallel evaluation of all the arguments to a function in Lisp, or the remote procedure call or fork of a process in some procedural language. Notice that within each parallel block of computation, there may be encapsulated a significant amount of sequential computation.

The third attempt exploits what we call data parallelism. Data parallelism corresponds closely to the vector parallelism in numerical programming. The basic idea is that, given a large set of data objects on which to apply a given function, that function may be applied to all the data objects in parallel as long as there are no dependencies between the data. Here, rather than distributing a number of tasks or function calls between a set of processors, one distributes a set of data and then invokes the same function in all processors.

These two forms of parallelism have been described as the MIMD (Multiple Instruction stream, Multiple Data stream) and SMD (Single Instruction stream, Multiple Data stream) approaches [2], These terms are generally used in classifying parallel architectures based on the type of computation for which they are particularly suited. Generally, it is felt that the finer the grain of an architecture, i.e. the simpler and the more numerous the processors, the more SIMD in nature, and conversely, the larger the grain, i.e. the larger and fewer the processors, the more MIMD the architecture. The terms are also frequently used to distinguish between distributed- and shared-memory machines, but it is important to remember that they actually refer to particular models of computation rather than any given particular architectural characteristics.

The case studies described in this paper deal exclusively with one or the other of these two forms of parallelism. Interestingly, the compiler, or dusty deck approaches that we have seen [9, 14], also seem to deal exclusively with one or the other of the two forms of parallelism. Some work has been done in the functional programming area in combining the two forms of parallelism [11] and the need to do so has been recognized by parallel Lisp designers [15], but to date very little work has been done in this area. This is surprising given that the distinction between the two forms is, in reality, quite weak. This is especially true in a language such as Lisp, where there is a continuum between what might be considered code and what might be considered data. To see this more clearly, let us take the case of the parallel evaluation of a function's arguments. We have seen this is generally considered a form of control parallelism. Assuming each argument involves the evaluation of a function, for example:

```
(foo (bar a) (car b) (dar c))
```

We generally view this as each argument being bundled up with its environment and sent off to some processor to be evaluated. What about the case where the same function is applied to each argument?

```
(foo (f a) (f b) (f c))
```

This is generally viewed as an occasion for exploiting data parallelism. The arguments (the data in this case) are distributed to a number of processors and then the common function is called on each. In the case of moving from a data to a control point of view, consider a vector, each element of which resides in its own processor. Invoking a given function on each member of the vector involves first distributing one member of the vector to each processor (this step is often ignored in descriptions of SIMD execution), and then, broadcasting the code for that function to each processor, or if the code is already stored in the processor, broadcasting the function pointer and a command to begin execution. In the control model, instead of broadcasting the function to all the processors at one time, one must distribute the function along with a particular element of the vector to each processor and immediately begin execution. This could just as well be viewed as first distributing the code and the corresponding data element to each processor and then broadcasting the instruction to eval that form in each processor. The synchronization is different in the two models, but the actual steps in the execution may be viewed as being the same.

2. Case Studies

We will return to the subject of the distinction between control and data parallelism later, when we discuss where the opportunities for parallelism in Lisp actually lie. First, we present three case studies of the approaches already mentioned. The current efforts concentrate on adding some additional constructs to a dialect of Lisp. They are, therefore, extensions to Lisp rather than completely new parallel languages based on Lisp. In the text we refer to them as languages, but the meaning should be taken as: Lisp and the additional constructs used to express parallelism. In the case studies new constructs are indicated by italics and normal Lisp constructs are indicated by a `typewriter` font

2.1. Multilisp

The first extended Lisp language we present is Multilisp [7, 8] which is based on Scheme [1], a dialect of Lisp which treats functions as first class objects, unlike Common Lisp [16], but is lexically scoped, like Common Lisp. In this paper, we do not distinguish Lisp based languages by the particular dialect of Lisp on which they are based, but rather by the constructs that have been added to the language and the effect, if any, that the base dialect has on these constructs.

Multilisp is notable both for the elegance and the economy of the constructs through which it introduces parallelism. In fact, a single construct, the future, is used for the expression of all parallelism in the language. A future is basically a promise or an IOU for a particular computation. In creating a future, the programmer is implicitly stating two things: One is that it is okay to proceed with the computation before the value of the future is calculated, but that the value will have been calculated or will be calculated at the time it is needed. The other is that the computation of the future is independent of all other computation occurring between the time of creating and before its use, and thus may be carried out at any time, in no particular order, with the rest of the computation or other futures that may have been created. The danger here, of course, is any side effects caused by the future must not depend on their ordering in relation to the rest of these computations. It is the responsibility of the programmer to ensure that these side effects do not cause any "unfortunate interactions". It is this responsibility that places additional conceptual overhead on the programmer. The overhead is reduced by having only one basic construct to deal with, but should not be underestimated. This overhead may be further reduced by what Halstead describes as a data abstraction and modular programming discipline. An example of the use of a future is:

```
(setq cc (cons (future (foo a))
               (future (bar b))))
```

Here we build a cons cell, both the car and cdr of which are futures, representing respectively the future or promised evaluation of `(foo a)` and `(bar b)`. This cons will return immediately and be assigned to the atom `cc`. If we later pass `cc` to the function `printcons` below,

```
(defun printcons (conscell)
  (print (car conscell))
  (print (cdr conscell)))
```

there are four possibilities:

1. both futures will have already been evaluated, in which case the futures will have been coerced into their actual values and the computation will proceed just as if it was dealing with a regular cons cell,
2. `(foo a)` has not yet been evaluated in which case `printcons` will have to wait for it to be evaluated before proceeding.
3. `(bar b)`has not yet been evaluated in which case `printcons` will have to wait for it to be evaluated before proceeding.
4. both `(foo a)` and `(bar b)` have not yet been evaluated in which case `printcons` must wait for the evaluation of the futures before continuing.

Multilisp has an additional construct known as a delay or delayed future that corresponds to a regular future, except that the evaluation of the future is delayed until the value is required. This additional construct is necessary in order to represent infinite data structures and nonterminating computation. For example, suppose we wished to represent the fist of all prime numbers greater than some prime number n. We could do this using a delay as follows:

```
(defun primes(n)
  (cons n (delay (primes (next-prime n)))))
```

Multilisp allows the computation of non-strict functions (functions whose arguments may not terminate) through the use of both the delay and of the future. However, in the case of futures, significant amounts of resources may be lost in every such computation, and ultimately storage will be exhausted for any non-finite data structure. By using delays, one only computes what is needed. AW futures may be removed from a program without changing the resources used by the program, but the same is not true for delays, since removing a delay may cause the computation of infinite data structures.

Delays thus represent a form of lazy evaluation, whereas the future represents a reduction in the ordering constraints of a computation. Eager evaluation strategies are also possible, in which computations are begun that are not necessarily needed at all, but are computed anyway, just in case they are needed. Multilisp does not provide such eager constructs or any construct that allows a computation to be halted prematurely. Constructs, which allow computations to begin, but are later able to halt them, are useful in freeing the computational resources of an eager computation, once it has been determined that its result is not needed. An alternate technique is to allow such a process to run until it can be determined mat the result being returned is no longer needed at which time it may be garbage collected by the system.

Multilisp includes one additional construct which is the pcall. Pcall provides for the simultaneous evaluation of the arguments to a function, but does not continue the evaluation of the function itself until all the arguments have fmished evaluating. Notice how this differs from a function call in which all the arguments are futures. Pcall thus provides a much more limited form of parallelism than futures, but is useful a midway point between the completely undefined flow of control between futures and the complete order of sequential execution. Pcall may of course be simulated by a function call, all of whose arguments are futures, provided the first act of the function is to access the values of all its arguments. Multilisp provides a primitive identity operator touch which causes a future to be evaluated and which is in fact used to implement the pcall construct.

The implementation of Multilisp calls for a shared-memory multiprocessor, and two implementations are underway [8,17]. Each processor maintains its own queue of pending futures, and a processor that has no current task may access another processor pending queue to find a future to execute. An unfair scheduling algorithm is necessary to ensure that constant computational progress is made and the system does not deadlock. The scheduling strategy has been chosen so that a saturated system behaves like a group of processors executing sequentially, i.e. as if all future calls had been removed from the code. Once a future has been evaluated, it is coerced to its return value by changing a flag bit stored among the tag bits of the Lisp item that represents it. This causes one extra level of indirection (one pointer traversal) in references to values that are the result of future calculations. These indirect references are removed by the garbage collector.

2.2. QLISP

The additional constructs in Qlisp [5] are quite similar in semantics to those of Multilisp, but very different in form. There are a much larger number of constructs in Qlisp, although the increase in the expressive power of the language is not great. There are two primary constructs that allow the expression of parallelism. They are qlet and qlambda.

Qlet does much what one might expect It performs the bindings of a let in parallel. Qlet takes an extra predicate as an argument along with its usual binding pairs. When this extra predicate evaluates to nil, qlet behaves exactly like let. When the predicate evaluates to the atom eager, the qlet spawns a process to evaluate each of its arguments and continues the execution of the following computation. Finally, if the predicate evaluates to neither nil nor eager the qlet spawns processes for each of its arguments as before, but waits for all of them to finish evaluating before continuing with the subsequent computation. These last semantics for qlet closely resemble those of pcall in Multilisp and may be easily used to mimic its semantics exactly (by placing the function call within the qlet that assigns the value of the functions arguments to temporaries). Further, a qlet where the predicate evaluates to eager may be used to simulate a function call where all the arguments were passed as futures:

```
(qlet 'eager
    ((x (foo a)) (y (bar b)) (z (car c)))
  (f x y 2))
```

Qlambda takes the same additional predicate as qlet and forms a closure in the same way as its namesake lambda. If the predicate evaluates to nil, then qlambda behaves exactly as a lambda does. If the predicate evaluates to eager, the process representing the closure is spawned as soon as the closure is created. If the predicate evaluates to something other than nil or eager, the closure is run as a separate process when it is applied. When a process closure defined by a qlambda is applied in a non value requiring position, such as in the middle rather than at the end of a prog, it is spawned, and the subsequent computation continues. Upon return of the spawned process, its return value is discarded. If a process closure is spawned in a value requiring position the spawning process waits for the return value. In addition, two operators are supplied to alter this behavior. They are wait and no-wait with the obvious semantics. The constructs that have been defined up to this point give the language the same semantic power as Multilisp's/wrwre mechanism.

Qlisp deals with an issue not handled in Multilisp, which is what happens if a spawned process throws itself out, that is, what happens if a spawned process throws to a catch outside its scope? When a catch returns a value in Qlisp, all the processes that were spawned in the scope that catch are immediately killed. The additional construct qcatch behaves slightly differently. If the qcatch returns normally (i.e. it is not thrown to), it waits for all the processes spawned below it to complete before returning its value. Only if it is thrown to does it kill its subprocesses. In addition, Qlisp defines the semantics of an unwind-protect form over spawned processes which ensures the evaluation of a cleanup form upon a non-local exit These additional constructs allow the programmer the power to begin and later kill processes, and therefore give him the power to perform the type of eager evaluations not available in Multilisp.

Qlisp has more of a process-oriented flavor to it than Multilisp, and, although its constructs have similar power to those of Multilisp they appear to be on a much lower level. A similar statement may be made for the C-lisp language [18].

In Qlisp, processes are scheduled on the least busy processor at the time of their creation. Unlike Multilisp, more than one process is run on a single processor and processes are time swapped in a round robin fashion. The predicates of qlet and qlambda allow for dynamic tuning of the number of processes created at run-time. This is another feature that Multilisp does not have, but one that is also better left to the runtime system rather than the programmer.

2.3. Connection Machine Lisp

Connection Machine Lisp, unlike the previous two languages, introduces parallelism in the form of data rather than control. The basic parallel data structure in Connection Machine Lisp is a xapping (a distributed mapping). A xapping is a set of ordered pairs. The first element of each pair is a domain element of the map, and the second is the corresponding range element. The mapping representing the square of the integers 1, 2, 3 would be denoted in Connection Machine Lisp as:

```
{ 1->1 2->4 3->9 }
```

If the domain and range elements of all the pairs of the xapping are the same (i.e. an identity map), this is represented as:

```
{ 1 2 3 } - { 1->1 2->2 3->3 }
```

and is called a xet. Finally, a xapping in which all of the domain elements are successive integers is known as a xector and is represented as:

```
[ John tom andy ] =
  { l->john 2->tom 3->andy }
```

Connection Machine Lisp also allows the definition of infinite xappings. There are three ways to define an infinite xapping. Constant xapping takes all domain elements (or indices as they are also called in Connection Machine Lisp) and maps them into a constant. This is denoted:

```
{ ->v )
```

Where v is the value all indices are mapped into. The universal xapping may also be defined. It is written { -> } and maps all Lisp objects into themselves. Finally, there is the concept of lazy xappings which yield their values only on demand. For example the xapping that maps any number to its squarte root may be defined by:

```
{ - sqrt }
```

and (xref {. sqrt} 100) would return 10. Notice that xappings are a type of Common Lisp sequence and many of the common sequence operators are available and may be meaningfully applied to them.

Connection Machine Lisp defines two main operators that can be applied to xappings. The a operator is the apply-to-all elements of a xapping operator. It takes a function and applies it to all of the elements of the xapping in parallel. If the function to be applied is n-ary, it takes n xappings as arguments and is applied in parallel to the n elements of each xapping sharing a common index. If the correct number of arguments to the function are not available, that index is omitted from the result element. For example:

```
(acona {a->l b->2 c->3 d->4 e->5}
                 {b->6 c->4 e->5)) ->
(b-><2.6) c->(3.4) e->(5.5)}
```

Notice that the domain of the result is the intersection of the domains of the function and argument xappings. The a operator is the main source of parallelism in Common Machine Lisp.

The other main operator is the P or reduction operator. It takes a xapping and a binary function and reduces the xapping to a value by applying the binary function to all the elements of the xapping in some order. Since the order in which the function is applied to the xapping is undefined, the functions used are generally limited to being associative and commutative. For example:

```
(P+ (l 2 3})
```

always returns 6, but

```
(p- {1 2 3})
```

may return 0,-2, 2, 4 or -4. A non-commutative or non-associative function may be useful on occasion however; for example, P applied to the function (lambda (x y) y) will return some arbitrary element of the xapping to which it is applied. This particular function has been found to be so useful in Connection Machine Lisp that it has been made into a regular operator called *choice*. The P operator has a second form in which it may serve as a generalized communication operator. When the p operator is passed a binary function and two xappings, the semantics are that the operator returns a new xapping whose indices are specified by the value of its first argument and whose values are specified by the values of the second argument. If more than one pair with the same index would appear in the resulting xapping (which is, of course, not allowed), the xector of values or these pairs is combined using the binary function supplied with the p operator. For example:

```
(pmax
 '{john->old tom~>young phil->dead
                          joe->young al->22}
 '{john->66 tom->12 phil->120 joe->ll}) ->
{old->66 young->12 dead->120}
```

In this example, we are using a database of xappings about people to generate some new knowledge. Given a qualitative measure of some people's age (old, young, and dead) and their actual age, we generate a value for the qualitative measures. Notice that when two indices collide, the results are combined by max, our heuristic being that it is best to represent an age group by its oldest member (not a terribly good heuristic in the general case). The interprocessor communication aspect of the p operator becomes clearer if one considers that all the information for one person (one index) is stored in one processor. In order to generate our new xapping, we transfer all the information about each age group to a new processor and do the necessary calculation there. In the above example, the information from Philip and George is transferred to the processor with label "young" and combined with the max operator there.

The parallelism in Connection Machine Lisp may be compared with the parallelism of the *pcall* construct of Multilisp. As pointed out by Guy Steele, the distinction between the two is due to the MIMD nature of *pcall* and the SIMD nature of the a operator in Connection Machine Lisp. To be more specific, although in the *pcall* all the arguments are evaluated in parallel, their synchronous execution is not assured. In fact, in both Multilisp and Qlisp the proposed implementations would almost guarantee that the evaluation would occur asynchronously. In Connection Machine Lisp, when a function is applied to a xapping, the function is executed synchronously by all processors. In the case of Connection Machine Lisp, the control has been centralized, whereas in the case of Multilisp, it is distributed. This centralization of control definitely reduces the conceptual overhead placed on the programmer, as well as reducing the computational overhead by requiring only a single call to eval rather than many calls occurring in different processors. The price for this reduced overhead is that the opportunity for exploiting the control parallelism that exists in many problems is lost Steele comments on this, suggesting that some of the lost control parallelism may be reintroduced by allowing the application of xectors of functions. For example, the following construct in Connection Machine Lisp:

```
(Otfuncall '[sin cos tan) [x y z])
```

is equivalent to the Multilisp construct:

```
(pcall #'xector (sin x) (cos y) (tan z))
```

As of yet, this aspect of the language has not been developed.

3. Discussion

In the languages, presented above we have seen two different methods of providing parallelism in Lisp. In one case, there is a process style of parallelism where code and data are bundled together and sent off to a processor to be executed. In the other, there is a distributed data structure to which sections of code are sent to be executed synchronously. The major question that remains is whether these two methods of exploiting parallelism can be merged in some useful way, or is there a different model that can encompass both methods. Before exploring this question further it is interesting to examine Lisp itself in order to see where the opportunities for parallelism actually lie.

The obvious place to apply control parallelism in Lisp is in its binding constructs. We have seen examples of this both in the *pcall* of Multilisp and the *qlet* of Qlisp. In addition to this form of parallelism, any function application may be executed in parallel with any other provided that there are no "unfortunate interactions" between them. We have seen this in *the futures* of Multilisp and the *qlambda* of Qlisp. A different approach often taken in traditional block-structured languages is to have a parallel block, or a parallel prog in the case of Lisp, in which all function calls may be evaluated in parallel. An interesting approach that has been taken along these lines is to treat each function application as a nested transaction and attempt to execute all applications in parallel, redoing the application when a conflict is detected [10]. A hardware version of this is also being investigated [12]. Yet another very interesting approach to control parallelism is that of making environments first class objects which may be created and evaluted in in parallel [6].

Conditionals may also be executed in parallel. In particular, and and or are natural places for expressing eager evaluation. In a parallel and or or, one might imagine all of the arguments being spawned in parallel and then killed as soon as one of them returns with a false or a true value respectively. This, of course, results in very different semantics for these special forms, which must be made clear to the programmer. Conditionals and case statements may, of course, also be executed in parallel in an eager fashion. In addition to evaluating their antecedents in parallel, if more parallelism is desired, the evaluation of the consequents may be commenced before the evaluation of the predicates has terminated, and it has been determined which value will actually be used. These eager evaluation methods bring with them a number of problems. Computations that used to halt may no longer do so, side effects may have to be undone, and the scheduling mechanism must ensure that some infinite computation does not use up all of the resources.

In order to understand the potentials for data parallelism in Lisp, we must look at both the data structures themselves and the control structures used to traverse them. The data structures that provide the potential for parallelism fall under the type known as sequences in Common Lisp. They are lists, vectors and sets. Sets are implemented as lists in Common Lisp, but need not be in a parallel implementation. The control structures that operate on these data structures in a way that may be exploited are iterative constructs, mapping constructs and recursion. The parallelism available from sequences and iterative constructs is much the same as the parallelism that has been exploited in numerical processing [13]. The flow of control in Lisp, as has already been mentioned, is generally more complex then that in numerical programs, complicating the compile time analysis. Mapping functions, on the other hand, are easily parallelized. Since a mapping construct applies the function passed to it to every element of a list, it can be modeled after the a construct in Connection Machine Lisp.

Recursion in the simplest case reduces to iteration (tail recursion) and the same comments that were made above apply. Notice also that in the general case the structure of recursion is very similar to that of iteration. There is the body of the recursion and the the recursion step, just as in iteration there is the body of the iteration and the iteration step. This similarity is taken advantage of by some compilers [9]. Recursion is also frequently used in the same way as the mapping constructs to apply a function to a sequence. What distinguishes recursion is that it may also be applied to more complex data structures, such as tree structured data. In traversing these more complex data structures, the parallelism available is often dependent on the actual structure of the data. For example, much more parallelism is available in a balanced rather than an unbalanced tree [5]. This type of parallelism is generally exploited as control rather than data parallelism, but mere is no reason that this must be so. The only thing that is necessary to enable a data point of view is the distribution of the tree structured data. Such distribution of tree structures may, in fact, be accomplished through the use of nested xappings in Connection Machine Lisp. Finally, there are some problems that are recursive in nature and do not lend themselves to any iterative or parallel solution; the Tower of Hanoi is the classic example.

By examining Lisp itself, we have seen exactly where the opportunities for parallelism are and we can judge the extent to which each of the languages studied is successful in allowing its expression. One thing that is immediately clear is that none of the languages allows for the expression of control and data parallelism within a single coherent model of parallel execution. It is quite possible that no single such model exists, but it should be the goal of future efforts to provide at least for the convenient expression of both forms of parallelism within a single language.

References

[1] Abelson, H., Sussman, G. J.
Structure and Interpretation of Computer Programs.
Massachusetts Institute Technology Press, 1985.

[2] FlynnM.J.
Some Computer Organizations and Their Effectiveness.
The Institute of Electrical and Electronic Engineers Transactions on Computers v-21, September, 1972.

[3] Friedman, D., Wise, D.
CONS should not evaluate its arguments.
In Michaelson, S., Milner, R. (editors), Automata, Languages and Programming, pages 257-284. Edinburgh University Press, Edinburgh, 1976.

[4] Friedman, D. P.
Aspects of Applicative Programming for Parallel Processing.
IEEE Transactions on Computers c-27(4):289-296, April, 1978.

[5] Gabriel, R. P., McCarthy, J.
Queue-based Multi-processing Lisp.
In Symposium on Lisp and Function Programming, pages 25-44. ACM, 1984.

[6] Gelemter, D., Jagannathan, S., London, .T,.
Environments as First Class Objects.
In Proceedings of the ACM Symposium on the Principles of Programming Languages. ACM, Jan, 1987.

[7] Halstread, R. H.
Implementation of Multilisp: Lisp on a Multiprocessor.
In Symposium on Lisp and Function Programming, pages 9-18. ACM, 1984.

[8] Halstead, R. H.
Multilisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages and Systems 7(4):501-538, October, 1985.

[9] Harrison, W. L.
Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor.
PhD thesis, University of Illinois, 1986.

[10] Katz,M.J.
A Transparent Transaction Based Runtime Mechanism for the Parallel Execution of Scheme.
May, 1986
Master Thesis.

[11] Keller, R.
Rediflow Multiprocessing.
In IEEE COMPCON, pages 410-417. IEEE Compcon, Feb, 1984.

[12] Knight, Tom.
An Architecture for Mostly Functional Languages.
In Conference on Lisp and Functional Programming, pages 105-112. ACM, 1986.

[13] Kuck, D., Muraoka, Y., Chen, S.
On the number of operations executable simultaneously in Fortran like programs and their resulting speedup.
IEEE Transactions on Computing C-21(12):1293-1310, December, 1972.

[14] Marti, J., Fitch, J.
The Bath Concurrent Lisp Machine.
In EUROCAM '83, pages 78-90. EUROCAM, Springer Verlag, 1983.

[15] Steele, G. L., Hiliis W. D.
Connection Machine lisp.
In Conference on Lisp and Functional Programming, pages 279-297. ACM, 1986.

[16] Guy L. Steele.
Common Lisp: The Language.
Digital Press, Burlington, M.A., 1984.

[17] Steinberg, S. A., et al.
The Butterfly Lisp System.
In AAAI-86 , pages 730-734. AAAI, August, 1986.

[18] Sugimoto, S., Agusa, K., Ohno, Y.
A Multi-Microprocessor System for Concurrent LISP.
In International Conference on Parallel Processing, pages 135-143. IEEE, June, 1983.