A Survey of Parallel Programming Constructs

Michael van Biema

Columbia University

Dept. of Computer Science

New York, N.Y. 10027

Tel: (212)280-2736

MICHAEL@CS.COLUMBIA.EDU

CUCS-CS-312-87

Table of Contents

1. Introduction 1
2. Parallelism in Functional Programming Languages 1
    2.1. Introduction 1
    2.2. Discussion 6
3. Parallelism in Lisp 8
    3.1. Introduction 8
    3.2. Case Studies 11
        3.2.1. Multilisp 11
        3.2.2. QLISP 14
        3.2.3. Connection Machine Lisp 16
    3.3. Discussion 19
4. Parallelism in Object-Oriented Programming Languages 22
    4.1. Introduction 22
    4.2. Encapsulation, Sharing, and Polymorphism 24
    4.3. Parallelism 25
    4.4. Discussion 30
5. Conclusion 32

1. Introduction

This paper surveys the types of parallelism found in Functional, Lisp and Object-Oriented languages. In particular, it concentrates on the addition of high level parallel constructs to these types of languages. The traditional area of the automatic extraction of parallelism by a compiler [39] is ignored here in favor of the addition of new constructs, because the long history of such automatic techniques has shown that they are not sufficient to allow the massive parallelism promised from modern computer architectures [26, 58].

The problem then, simply stated, is given that it is now possible for us to build massively parallel machines and given that our current compilers seem incapable of generating sufficient parallelism automatically, what should the language designer do? A reasonable answer seems to be to add constructs to languages that allow the expression of additional parallelism in a natural way. Indeed that is what the designers of the the Functional, Lisp, and Object-Oriented languages described below have attempted to do.

The three particular programming formalisms were picked because most of the initial ideas seem to have been generated by the designers of the functional languages and most of the current activity seems to be in the Lisp and Objected-Oriented domains. There is also a great deal of activity in the Logic programming area, but this activity is more in the area of executing the existing constructs in parallel as opposed to adding constructs specifically designed to increase parallelism.

2. Parallelism in Functional Programming Languages

2.1. Introduction

In his now classic paper "Can programming be Liberated from the von Neuman Style", [6] Backus argues strongly in favor of the use of functional languages (known also as applicative or reduction languages) as opposed to traditional or imperative languages. Now, almost ten years later, the debate still rages on, and although the ultimate use of functional languages is still in question, there is no doubt that functional languages have been the origin of many original and elegant language constructs.

The basic concept behind functional languages is quite simple: To allow a language to consist only of expressions and function applications and to disallow all forms of assignment and side effects. This can be accomplished either by disallowing assignment entirely - as in languages such as FP [6] - or by allowing only so called single assignments as in VAL [49]. In other words, either assignment is disallowed entirely or, once a variable is bound to a value, it keeps that value for the remainder of its life time. Single assignment actually represents no more than a notational convenience for the programmer.

In a sense, functional languages have served as the starting point for the design of parallel languages. Most of the parallel constructs that are used by the Lisp and Object-Oriented languages that we examine later were either inspired or directly taken from earlier work done on functional languages. It is also fair to say that the addition of parallel constructs to functional languages is easier than the addition to non-functional ones. The functional language designer is free from the major concern of side effects that constrains and complicates the life of the designer of non-functional languages. The absence of side effects in functional languages makes them inherently parallel since their order of evaluation is constrained only by the actual data dependencies of the program. This is known as the Church-Rosser property [17]. Thus the implementor of a functional language is free to reorder the evaluation of a program in any way as long as he maintains its data dependencies. Quite a large number of methods of evaluation of functional languages have been proposed: data flow, demand driven, graph reduction, string reduction, and combinators [68]. The majority of these methods can be classified into one of two camps: data flow or reduction. The two approaches differ in the way they treat the program graph. In the case of the data flow approach the program graph is mapped onto the hardware and the data of the program flows through it. Functions (nodes of the graph) are evaluated when the data necessary for the execution has arrived at the node. In the case of reduction, the entire program graph is treated as the object of computation and pieces of it are reduced by a given set of reduction rules. The process is continued until the entire graph is reduced to an atomic value. This piecewise reduction may of course be performed in parallel. The actual translation of the program text into the program graph determines to a large extent the degree of parallelism possible.

The actual method of evaluation of a functional program may be either sequential, data driven, or demand driven [68]. In the demand driven case the amount of parallelism may be reduced due to the fact that functions are only evaluated when actually needed to continue the computation, but the total amount of computation may be reduced as well. It must be kept in mind that the goal of the parallel language designer is to reduce the overall computation time which is not necessarily the same as creating the maximum amount of parallelism nor is it the same as reducing the total amount of computation. Demand driven evaluation corresponds to lazy evaluation where a function is never evaluated until its value is actually required to complete the computation. Lazy evaluation brings with it two major advantages. The first is that the total amount of computation may be significantly reduced. The second is that it makes possible the representation of infinite data structures. It has also been pointed out [14] that lazy evaluations can take the place of the co-routine or iterators seen in earlier imperative languages, by producing data only when it is needed to be consumed. Hybrid schemes of evaluation combining the data flow and demand driven models have been proposed as well [5].

Rather than discuss the pros and cons of these various implementation techniques here, we will discuss those additional constructs that have been added to Functional Languages in order to increase the available concurrency. The techniques that have been suggested to increase the available parallelism include: For Alls, Reduction Operators, Lazy Evaluation, Eager Evaluation, argument binding by Pattern Matching, and Distributed Data structures. One can see how these various features have been combined by the languages studied in fig. 2-1.

As can be seen, most of the languages have a forall type of operator that always comes along with a set of reduction operators. These forall and reductions operators operate on either lists or vectors or in the case of FLucid a explicitly distributed data structure know as a Ferd. A Ferd is a data structure that can be thought of as being equivalent to a stream, but distributed in space as opposed to time. Space is defined here as being a set of processing elements. Ferds bear a close resemblance to the *Xappings* of Connection Machine Lisp [58] that we will examine later.

The forall and reduction operators take somewhat different syntactic forms among the various languages, but the semantics are almost identical. Some of the languages allow the programmer

	ForAll	Reduct	Pattern	Lazy	Eager	DistData
1) FP	X	X				
2) SASL			X	X		
3) VAL	X	X				X
4) HOPE	X	X	X	X		
5) SISAL	X	X				
6) SAL	X	X			X	
7) ML	X	X	X			
8) FLucid	X	X				X

Figure 2-1: Functional Languages and their Parallel Constructs

to specify the particular order in which the reduction is to be done [7, 50]. The choice is generally between left, right, and tree order which may be performed in log as opposed to linear time on a parallel machine with tree connections. Some languages limit the operators that may be used for reduction to a predefined set of associative operators. The set generally includes: plus, times, max, min, and, or [16, 49]. Other languages allow the use of any function along with the reduction operator. If the operator is not associative it is up to the programer in ensure the results returned are meaningful.

Pattern matching is a method of binding arguments to functions that allows for more parallelism and better compilation than the traditional method of passing arguments and then dispatching on their value within the function using either a case or an if statement. For example the function to reverse a list would be written as:

```
    reverse(nil)  = nil
    reverse(a:l)  = reverse(l)  :  [a]
```

using pattern matching (: is the cons operator) as opposed to its usual form of:

```
reverse(l) =
  if null(l) then nil
    else (reverse(cdr l) : [a])
```

The delayed evaluation of the constructor function **cons** allows for the creation of arbitrary amounts of potential parallelism and the possibility for reduced overall computation [19]. By delaying the evaluation of the car and cdr parts of the cons until they are actually needed, a great deal of computation may be saved. In addition such "lazy" evaluation allows the definition of infinite data structures since only the parts that are actually accessed will ever be created. For example:

```
zeros = 0 : zeros
```

represents an infinite string of zeros. If additional processing power is available it may be used to evaluate these delayed constructs and replace them with their actual values.

It is also possible to pre or eagerly evaluate expressions. The semantics of SAL, for example, allow for the eager evaluation of the various branches of an if or case statement. Notice that such eager evaluation is "safe" in a functional language due to the absence of side effects, whereas the side effects might have to be undone in a traditional imperative language.

There are also some more subtle design decisions that affect the available parallelism in a language. Within a let or a where clause, for example, the restrictions placed on the ordering of the binding affects the degree of sequentiality necessary in their evaluation. So, for example, in a language such as SASL [53] where no ordering is specified the binding may be carried out in parallel, whereas this is not the case when some specific ordering dependencies are introduced [16]. Perhaps the most interesting addition to the SASL language is the use of Zermelo-Frankel (ZF) [67] set abstraction to express list formation. Syntactically, ZF-expression has the form:

```
[expression; qualifier; ... ;qualifier]
```

A qualifier is either a boolean expression or a generator. A generator has the form:

```
<name> <- <list expression>
```

<name> is a local variable whose scope is that of the expression. So, for example, a map

function might be defined as:

```
map f x = [f a; a<-x]
```

where a takes successive values from the list x. Another example is the definition of a function that generates the permutations of a list which uses both ZF-notation and pattern matching:

```
perms [] = [[]]
perms x = [a:p; a<-x; p<-perms(x--[a])]
```

(The -- operator represents list difference) ZF-notation presents interesting possibilities for parallel compilation.

There are several other places one can exploit parallelism in an applicative environment, The most obvious is the collateral evaluation of arguments to a function call. Recursion with more than one internal recursive call leads to a tree of calls, all the leaves of which may be evaluated in parallel. We have already seen the application of the same function to a large number of arguments (i.e. ForAll or mapping [15]), but it is also possible to perform the application of a large number of functions to the same argument (i.e. construction [7]) or the application of a large number of functions to a large number of arguments (i.e. combination [20]).

2.2. Discussion

From the above language descriptions it should be clear that the basic parallel constructs that have been used in functional languages do not vary greatly from one language to another except in syntax.

Although most of the current interest in functional languages seems to have concentrated on their type systems [13, 14, 15], experimentation has been done with adding notations to functional programs for both specifying which processor a function should execute on and for specifying which expressions should be evaluated eagerly. As an example, the ParAlfl base language for Para-Functional Programming [28] has been extended with mapping annotations that specify the processor on which a function should be executed. Programs written in this way can still be executed on a uniprocessor by ignoring the annotations, and the results will be equivalent due to the Church-Rosser property. ParAlfl also allows the programmer to annotate functions that he wishes to be executed in an eager fashion.

It is important to point out that it is also quite possible to have too much parallelism for a particular architecture to execute efficiently. Ideally, there should be just enough tasks spawned to keep all of the processors busy all of the time. If more than this "ideal" number of tasks are spawned, some tasks will need to queued and this will waste space and processing resources. This has been seen on BBN Butterfly machine, and as a result BBN is developing a semi-applicative language which allows annotations such as precedence that limit the degree of parallelism [57]. The other interesting feature of this language is annotations which indicate that particular results should be cached as opposed to being recomputed.

Another addition that has been made to applicative languages with some success is the use of logical unification for parameter binding. Unification is useful because it provides an implicit form of synchronization (the unification of the variable) as well as a mechanism to pass information both into and out of a function invocation (the unification can occur either before or during the function's evaluation) [43, 57].

Others have experimented with the addition of imperative sections of code to an applicative language. This has been found to yield great gains in efficiency and when done in a structured way does not destroy the semantics of the applicative part of the language. In Rediflow [37], for example, the exchange of data between applicative and imperative sections of code is done by mapping a stream data structure to a von Neuman stream of tokens and performing the inverse mapping when passing the processed data tokens back. Keller also introduces the interesting notion of using pressure as a analog to load. By allowing each processor to keep some measure of its internal and external pressures (loads), tasks are allowed to flow along pressure gradients to new less-loaded sites of execution.

Interesting work has also been done in the graphical programming of functional languages [46]. Such work is useful in that the programmer can get an idea of the amount of parallelism by looking directly at the dataflow graph itself.

The basic form of functional languages varies significantly, but the parallel constructs are remarkably similar. The basic notions of lazy evaluation, eager evaluation, "for all" constructs and reduction operators appear in only slightly different forms in most of the languages studied. As we will see in the second and third chapters, these constructs, in different forms and combinations, form much of the basis for the parallel constructs in imperative languages as well.

3. Parallelism in Lisp

3.1. Introduction

As with functional languages 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 [26, 37, 47]. 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 [39]. 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 explicit 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 current 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 are 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 [19, 20].

The first two attempts [21, 24] 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 [58] 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 SIMD (Single Instruction stream, Multiple Data stream) approaches [18]. 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 [26, 47], 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 [37] and the need to do so has been recognized by parallel Lisp designers [58], 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.

3.2. 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.

3.2.1. Multilisp

The first extended Lisp language we present is Multilisp [24, 25] which is based on Scheme [1], a dialect of Lisp which treats functions as first class objects, unlike Common Lisp [61], 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 *future*s 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 [24] 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 *future*s, 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 *future*s will have already been evaluated, in which case the *future*s 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 *future*s 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 expressly delayed until the value is required. This additional construct allows the expression of infinite data structures and nonterminating computation. For example, suppose we wished to represent the list 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 tied up or lost if the results are not used, and ultimately storage will be exhausted for any non-finite data structure. By using *delays*, one only computes what is needed. All *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, since the recursion or iteration does not pause after each cycle. *Delays* thus represent a form of lazy evaluation, whereas the *future* represents a reduction in the ordering constraints of a computation.

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 finished evaluating. Notice how this differs from a function call in which all the arguments are *futures*. *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. *Pcall* thus provides a much more limited form of parallelism than *futures*, but is useful as midway point between the completely undefined flow of control between *futures* and the complete order of sequential execution.

The implementation of Multilisp calls for a shared-memory multiprocessor, and two implementations are underway [25, 62]. 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 since the future at the begining of the queue will not be the same as the next process to be executed if the program were being executed sequentially. 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 when a collection occurs after evaluation of the future has terminated.

3.2.2. QLISP

The additional constructs in Qlisp [21] 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 except for the ability to express eager evaluation. 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 [2]. An alternate technique is to allow such a process to run until it can be determined that the result being returned is no longer needed at which time it may be garbage collected by the system.

There are two primary constructs that allow the expression of parallelism in Qlisp. 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 z))
```

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. The *wait* construct ensures that the computation will wait for the spawned process to complete before continuing even if it appears in a non-value requiring postion and the process *no-wait* tells the computation to proceed without waiting for the return value. The constructs that have been defined up to this point give the language the same semantic power as Multilisp's *future* 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 of 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 [64].

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 shared in a round robin fashion. The predicates of *qlet* and *qlambda* allow for dynamic tuning of the number of processes created at run-time. For example, the predicate might be set to eager if some measure of the system wide load is less than a certain threshold. There is no method of performing this type of dynamic load balancing in Multilisp.

3.2.3. Connection Machine Lisp

Connection Machine Lisp [58], 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 ] =
  { 1->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 square root may be defined by:

```
{ . sqrt }
```

and (xref {. sqrt} 100) would return 10. Notice that xappings may be thought of as 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 α 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:

```
(αcons {a->1 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 α operator is the main source of parallelism in Connection Machine Lisp.

The other main operator is the β 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:

```
(β+ {1 2 3})
```

always returns 6, but

```
(β- {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, β 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 β operator has a second form in which it may serve as a generalized communication operator. When the β 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 of these pairs is combined using the binary function supplied with the β operator. For example:

```
(βmax
 '{john->old tom->young phil->dead
                          joe->young al->22}
 '{john->66 tom->12 phil->120 joe->11}) =>

{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 β 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 Tom and Joe 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 [58], the distinction between the two is due to the MIMD nature of *pcall* and the SIMD nature of the α 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 or synchronous execution 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 paid for this reduced overhead is that the opportunity for exploiting the control parallelism that exists in many problems is lost. Steele comments on this [58], 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:

```
(αfuncall '[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 defined.

3.3. 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.

An 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 [36]. A hardware version of this is also being investigated [38]. Yet another very interesting approach to control parallelism is that of making environments first class objects which may be created and evaluated in parallel [22].

Logical connectives 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 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 once halted 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 potential 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 elements of the sequence may be of any data type. 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 [39]. The flow of control in Lisp, as has already been mentioned, is generally more complex than 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 α 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 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 [26]. 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 [21]. This type of parallelism is generally exploited as control rather than data parallelism, but there 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 to a set of processors in such a way that its previous structure may be inferred. 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 (although an iterative solution does exist it is much less "natural").

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.

4. Parallelism in Object-Oriented Programming Languages

4.1. Introduction

A number of definitions have been given for Object-Oriented programming. Rather than attempting yet another, we set forth the minimal characteristics that an Object-Oriented programming language must exhibit in order to be true to its name. The primary characteristic of an Object-Oriented programming language is that of encapsulation or information hiding. An object provides a framework by which the programmer may keep certain parts of his implementation private, presenting only a certain interface to the user. Yet there are a number of languages that provide such information hiding, but cannot be considered true Object-Oriented programming languages. Examples of such languages are Ada [29], Modula [69], and CLU [45]. In order to properly be considered an Object-Oriented programming language a language must have two additional characteristics besides information hiding. The first such property is the ability to share parts of the implementation of one object with another. This is most frequently implemented as some form of inheritance. It is important to note that inheritance is by no means the only way to implement such sharing, but is the most commonly used method in current Object-Oriented programming languages. Other methods are mentioned later in the text. The final characteristic is the polymorphism of functions. Functional polymorphism means that a function's invocation is determined by the type or types of its arguments not only by its name. This polymorphism is usually implemented within a message passing framework, but as with inheritance, this is not essential. In some languages, such as Ada [29], it is implemented as operator overloading.

To make these characteristics more concrete let us examine the form they take in two specific, but very different, Object-Oriented programming languages: Smalltalk and Actors.

Smalltalk [23] is the language that for many is the representative Object-Oriented programming language. This is not without foundation since Smalltalk is one of the earliest and most sustained efforts at implementing and popularizing the Object-Oriented approach. In Smalltalk all objects are instances or instantiations of some abstraction or class. A class is a specification of the local storage and the behavior of the objects that are its members. The behavior of the members of the class is defined by the the class' methods (procedures). Methods may be declared to be either private to the instances of the class or public and therefore accessible by objects outside the class. The public methods of a class present the external view of an object to the rest of the world. The private methods hide the internal workings of the object. In Smalltalk, classes are structured into class hierarchies and one class may be defined as a subclass of another. By default a subclass inherits all the attributes of its parent. If the subclass does not wish to inherit a particular attribute, as is, from its parent it may explicitly redefine that attribute. In Smalltalk everything is an object, including the classes themselves, which are instances of their meta-class. All meta-classes are instances of the special class *meta-class* which ends this recursion. If a message is sent to an object and that object's class has no method that implements the message, the message is looked up in object's parent class to see if the message is implemented there. This passing on of messages is continued until the top of the inheritance chain is reached. Since every class is defined to be a subclass of the predefined class *Object*, all unresolved messages will end up there, where they are either handled or an error is signaled. The class *Object* thus serves as the location for all default system methods such as printing and error handling.

Although the base definition of Smalltalk does not allow multiple inheritance (the inheritance of attributes from more than one parent class) [10] most Smalltalk systems implement some form of multiple inheritance. These schemes differ mainly in the way conflicts are resolved. A conflict arises when more than one parent defines the same attribute or method. Typical schemes for resolving such conflicts are to place some ordering on the inheritance lattice and to resolve the conflict on the basis of that ordering. Other techniques combine the attributes in some way specified either by the system or the user. Still others merely signal the conflict and leave its resolution up to the user [9, 52] .

In contrast to Smalltalk, we examine the Actor [2, 40, 42, 66] model of Object-Oriented programming. In the Actor formalism there is no class-instance hierarchy. Uniformity is maintained by viewing everything as an actor. Sharing is implemented by viewing actors as "prototypes". For example the actor representing a bicycle may serve as a prototype for a unicycle except that the number of wheels attribute (acquaintance, see below) would be changed.

Actors maintain their local state by naming a set of acquaintance actors: other actors of which they are composed. An actor also has a script (list of methods) which defines its behavior in response to messages. An actor may pass a message on to another actor at will. This allows for dynamic sharing as opposed to the static sharing provided by inheritance. When a message is sent, no reply is expected. Rather, a continuation (as implemented in [59]) may be sent along as part of the message if a reply is desired. The actor sending the message does not wait for a reply, but rather continues its computation. It is the responsibility of the receiver to return a response to the continuation if a response is required. As we will discuss in more depth later, these are very different semantics from the usual procedure-return semantics seen in traditional languages and closely modeled by the message passing semantics of Smalltalk.

4.2. Encapsulation, Sharing, and Polymorphism

As we have just seen, the model of computation presented by Smalltalk and Actors is quite different. Yet both may be considered true Object-Oriented programming languages as they both possess the characteristics of encapsulation, sharing and functional polymorphism. A large number of other Object-Oriented programming languages have also been designed [8, 9, 31, 32, 52, 54, 56, 55, 70, 71]. It is interesting to examine how they vary along these three defining dimensions.

Sharing is the dimension along which the greatest variance is seen. Some languages provide a static inheritance lattice while others provide for a more dynamic form of sharing [42]. The way in which conflicts are resolved also varies widely among languages, ranging from complex automatic resolution schemes to those in which the user must resolve all conflicts himself. The degree of encapsulation also varies a great deal mainly in the method used to implement sharing. In an inheritance-based system, for example, the means by which a subclass inherits from its parent class can have a major effect on the encapsulation and therefore on the modifiability of the system. Systems that permit direct access to inherited instance variables, or to the local storage of a parent object, tie subclasses to the implementations of their super classes. By permitting such access only through message passing, a system is not only more consistent, but provides encapsulation between sub- and super-classes [42, 56].

Functional polymorphism in message passing is usually implemented based only on the type of the object to which the message is passed. It is also possible to make the function determination based on the types of any or all of the arguments. This scheme closely resembles generic function calls, but differs in that the code describing a function is distributed as a set of methods, rather than being centralized in a single generic function which dispatches on type [9, 51]. Although this effect can be easily simulated in languages providing only dispatching on the first argument by embedding the rest of the dispatches within additional methods [30] there are advantages to making it a part of the language definition, as in New Flavors [52] and CommonLoops [9].

Another factor which affects the polymorphism of the language is the degree to which it is type-checked at compile time. One finds Object-Oriented programming languages occupying most points along the type checking dimension. They range from Smalltalk, which is not checked at compile time, to a number of strictly type checked languages [54, 63] and languages which allow, but do not require, the user to provide type information (this group includes all of the Common Lisp based languages) [9, 52, 55] as well as other mixed schemes [51]. Although, these features of an Object-Oriented language do not have a direct effect on the amount of parallelism exhibited at the language level, which we examine next, they do play an important role in the parallelism at the implementation level.

4.3. Parallelism

The fourth dimension along which Object-Oriented languages vary is the way they deal with concurrent computation. Again, as with the characteristics discussed above, a number of points along this dimension are occupied. There are systems which allow for no parallelism within their model, systems that provide only the most primitive facilities, and those in which parallelism is an integral part of their specification. Smalltalk, for example, provides two classes - the *process* and the *semaphore* - which allow the programmer to express parallelism at the most basic level. Unlike the Actor model described above, all message passing in Smalltalk follows the procedure return model where the object that initiates the message waits for a reply. Smalltalk provides no special parallel constructs and its basic execution model is sequential.

Thus Smalltalk provides a significant contrast to Actors where all message passing is asynchronous, and a number of powerful parallel constructs are provided.

The Object-Oriented approach would at first seem to lend itself very naturally to the expression of parallelism. Objects being well encapsulated and communicating only through some form of message passing would seem to be easily modeled as units of computation. Two models are possible. In the first objects are passive as they are in current sequential systems and only "wake up" when they are sent a message. In the second an object may be associated with a process or a number of processes which are always active. Hence an object is always involved in some computation from which it must be distracted when it receives a message. The problems caused by concurrency in both these models are largely the same, however. The difficulties come, as with all parallel models, in maintaining consistency while handling multiple simultaneous messages.

An obvious solution is to provide each object with a queue and process messages from it in a FIFO manner. This simple approach gives an object much the same semantics as a monitor [12] and has the drawback that it may significantly limit the amount of parallelism. A more flexible approach is to provide some type of priority system so that messages of higher priority may interrupt lower priority messages. A third approach is for all incoming messages to suspend the current computation as long as it is not protected by a critical section. More complex mixes of these traditional strategies are possible as well, as we will see below. Early distributed object systems were based on little more than the concept of the remote procedure call [65]. More recent systems are significantly more sophisticated, both in the number and types of their parallel features. It should also be noted here that we have chosen to deal with the question of parallelism in Object-Oriented languages on a rather high level. There is ample opportunity for the exploitation of parallelism in Object-Oriented languages on lower levels as well, for example, such things as method lookup and instance variable access, but this lower level form of parallelism is not discussed further here.

Parallelism is implicit in the Actor model. All requests are executed in parallel except in the case when an actor is explicitly specified as being a *serializer*. A *serializer* behaves like a regular actor except that it handles its messages in a FIFO fashion and stores messages on a queue while they are waiting to be processed. In addition, all actors (including serializers) assume no explicit ordering in the arrival of messages. This certainly increases the available parallelism, but may place a large conceptual overhead on the programmer or force him to include ordering information in his messages. Finally, as we have already seen, actors pass continuations rather than providing the usual call-return semantics to message passing. If a return value is needed it must be explicitly returned to the continuation by the receiver of a message.

Emerald [8], which evolved out of Eden [3], is a strongly typed language which allows its objects to be always active by having their own subprocesses. Protection of internal state is accomplished by means of monitors, and objects have two additional parallel properties: mutability and location. Objects that are immutable may be duplicated freely around the system. The location of an object is provided so the user may explicitly fix object locations or perform relocation for efficiency reasons. Finally, the compiler assigns to each object one of three types that vary in their semantics and efficiency. Global objects may be moved freely around the system. They are accessed via a location table and forwarding pointers. Local objects are objects that are always local to other objects and move with them. They are heap allocated. Direct objects are generally like Local objects, but may be allocated directly along with their enclosing object at compile time. These are generally built in types and record structures whose storage may be deduced at compile time. Eden also provides a call-by-move primitive. This retains the usual call-by-reference semantics, and adds the additional semantics that the parameter objects are relocated to the same site as the object on which the method is called. The effectiveness of this type of location optimization is not yet known and is probably very system dependent.

In the language ABCL/1 [71] the *select* construct allows arbitrary guards [44] or constraints to be placed on sections of code involved in the receiving of messages, and messages are kept in a queue until these guards are satisfied. The queue discipline is FIFO, but all messages are first checked against the guards and executed if a match is found. The messages in the queue are then rechecked against the guards in a FIFO manner. This technique allows a message somewhere in the middle of the queue to be chosen for execution. ABCL/1 also provides a priority scheme based on two modes of messages: *ordinary* and *express*. *Express* mode messages may interrupt *ordinary* mode messages unless they are in a critical section as described below. In addition there are 3 types of message passing: *past* type message passing where the sender continues without waiting for a response, *now* type message passing which has the usual call-return semantics, and *future* [19] message passing in which the sender continues without waiting for a return value, but the return value may be accessed later through a special variable or *future*. The concept of the *future* is discussed further below. ABCL/1 also provides critical sections through its *atomic* construct which causes the enclosed region to be uninterruptable.

In the Yokote and Tokoro paper on Concurrent Smalltalk [70], they identify four ways of increasing the parallelism of traditional message passing semantics:

1. to let the receiver object continue execution after it returns a reply
2. to initiate many message sends at one time and wait for all the reply messages
3. to send a message and proceed with its execution without waiting for a reply message
4. to send many messages at one time without waiting for a reply

In Concurrent Smalltalk they have chosen to use methods 1 and 3 after finding that using methods 1 and 2 provided only limited concurrency in an earlier version of the language. A construct known as a *Cbox* is also introduced that may be coupled with the third mode of message passing to implement a *future*. The fourth form of message passing allows a method to continue execution after sending a reply. Two types of objects are provided by Concurrent Smalltalk as well. Normal objects have the same LIFO behavior as Smalltalk objects: a new context being created every time a message is received. This type of object is provided for compatibility with Smalltalk. The other type of object provided by Concurrent Smalltalk is the Atomic object. Atomic objects are implemented using FIFO queues in the same way as the objects in ABCL/1.

Recently, a number of mixed formalism Object-Logic languages have been defined [32]. In these languages consistent variable instantiation under unification is generally used to provide synchronization. The language Oil [4] being designed in conjunction with the FAIM-1 computer borrows some notions from dataflow programming as well. In Oil an object has a number of *ports* rather than methods and these *ports* may be grouped together into *entries*. An *entry* may only fire when all its *ports* have messages pending. The execution of an *entry* is called a behavior and is atomic and mutually exclusive. Objects may alter their behavior by closing down certain *ports*. An object may be given either a procedural or a logical behavior. Procedural objects are written in a Lisp-like language and logical objects in a declarative Prolog-like language. Finally, Oil allows the programmer to add *pragmas* to his code to assert the probability that a certain branch will be taken, estimate the size of dynamic data structures, and provide hints about allocation decisions. These pragmas are intended to help the compiler make intelligent allocation decisions for the various objects. Orient84/K [31] is another Object-Logic based language that provides much the same synchronous and asynchronous message passing as Concurrent Smalltalk. Additional *Or-wait* and *And-wait* constructs are provided for synchronization allowing the programmer to choose between waiting for the first reply or all the replies to a group of concurrent requests. Orient84/K also provides a standard priority scheme which may be used to define critical sections (by giving them the highest priority). It also provides mutual exclusion by allowing the *activation* and *deactivation* of methods and access control to methods through a permission mechanism.

Several Object-Oriented languages provide additional constructs to the user in order to increase the potential parallelism and simplify its use. The concept of a *future* is commonly used to increase potential parallelism. Instead of a message to another object returning a value, it returns a special object known as a *future* which represents the promise to compute the actual value needed. The message sender or client may then continue its computation carrying along the *future* instead of the actual value, and if resources are available, the value or the *future* may be computed in parallel. If all goes well, the value will be computed by the time the on-going computation actually needs it. If the on-going computation needs the value before it has been computed, it will be necessary to suspend the computation and wait for the value. Once the value is calculated the *future* is generally replaced with the actual value transparently by the system.

Additional constructs may be built upon that of the future. For example, in ACT1 [41] a *race* construct is a list forming construct that returns the results of its arguments in the list in the time order in which their values are computed. This construct in turn is then used to implement a parallel *or* which returns as soon as one of its arguments returns a positive truth value. In the Actor language it is interesting to note that there is no explicit method for halting a computation. A computation is garbage collected when it is determined that the value it is computing is no longer required. It is not yet known what effects on efficiency and resource utilization this will have, but its benefits to the programmer are probably much the same as those of garbage collecting storage. In Oil there is the addition of a parallel condition statement whose semantics are that the first temporally satisfied condition is chosen for execution. A recent Object-Oriented language, MELD [33, 34, 35] which is based on some ideas developed for constraint languages [10, 11, 60] and specification languages [13], introduces the notion of programming with constraints as opposed to statements. A constraint comes with an implicit assertion about the system which must be true in order for the constraint to be satisfied. Thus the computation performed by the constraint is exactly that computation necessary to maintain the truth of the assertion. There is no explicit ordering between the constraints of a system; they are ordered only by their implicit data dependencies. This allows for maximal amounts of parallelism in the execution of these constraints; however, it is not yet known if an efficient implementation can be achieved.

4.4. Discussion

As we have seen, despite the fact that a community of concurrently processing objects interacting through message passing appears to be a natural environment for the exploitation of parallelism, major unresolved difficulties remain. These difficulties center around how an object can receive multiple simultaneous messages and process those messages without compromising its internal state, while still maintaining maximal concurrency. A number of the approaches, discussed above, are being implemented, but it is too early to give any concrete evidence for the success of any of these methods.

Since it is hoped that a collection of objects will provide a way of modeling the real world and make some kind of joint progress against its entropy, it seems reasonable to propose a model of communication based on a real system that demonstrates such behavior. A corporate entity (more commonly known as a company) might serve as such a model. Let us briefly explore the types of interactions found in a typical (hopefully successful) company and see how they may be used as a model for an Object-Oriented programming language.

We are not interested in the hierarchical structure of the company, but rather the modes of communication embedded in it. We use the term workers here for all objects both animate and inanimate that perform tasks in this corporate setting. So, for example, both the CEO of the company and its timeclock are considered workers. Certain workers have changeable internal state. Others may not. Those that do not may be freely replicated and may handle multiple simultaneous messages as is done in the Actor model. It is, therefore, those with changeable state that concern us here. Let us take a typical worker. He is generally busy with some task. He may be interrupted from that task by a boss or a subordinate or someone on the same level. In the case of a boss, he is out of luck and must stop whatever he is doing and listen to the boss's request. In the case of the subordinate, he has the option of listening to the request then and there or telling the subordinate to wait. In the final case of someone on the same level he can either listen to the request or tell the person to come back later. If the worker is working on something very important, he might put up a "do not disturb" sign on his door that would prevent workers of a lower level or the same level from interrupting him at all. Finally, if more complex protocols are necessary, a worker may delegate to a secretary or receptionist the responsibility of handling his interactions for him. When a worker is interrupted by a peer or a subordinate he should have control over the interruption. He should be able to decide if the interruption is sufficiently important for him to stop what he was doing and deal with the interruption or to put off the interruption to some other time. The concept of a meeting is also very useful. It is efficient for a worker to be able to call together a group of subordinates or peers and give them orders all at one time (a broadcast). A number of the ideas presented above necessitate some form of negotiation between the objects. Negotiation in this setting and some ideas similar to the ones above are currently under investigation by Carl Hewitt and his group at MIT [27].

It would seem that this model offers reasonably complete and natural semantics in a message passing environment. Some of these modes of interaction are already encompassed in the languages we have examined; others are not. In addition, higher level language constructs such as the *future*, parallel conditionals, parallel OR, parallel AND, FOR ALL, and others perhaps not yet known are necessary to aid the programmer in the expression of parallelism within the Object-Oriented environment. It should be the goal of current parallel language designers to attempt to provide both a complete and convenient set of these constructs. The model of communication given above should provide a basis for this search.

5. Conclusion

Having examined the types of parallel constructs that have been added to three very different programming formalisms, it is fair to say that one is struck more by their similarity than by their differences. The notions of futures, eager evaluation, foralls, reduction operators, and distributed data structures reappear in slightly different forms in both Functional and Lisp languages. This is not that surprising since Lisp is a derivative of pure Lisp which was defined as a functional language [48]. This is less true for Object-Oriented languages where the main parallel construct seen is the future, and the remainder of most parallel implementations has to do with controlling parallelism and the safe access to objects.

What is perhaps the most startling result of this study is that there does not, in fact, appear to be a large and diverse set of parallel constructs, but rather a relatively small and well defined set. Whether this set is complete is not at all clear, but it does seem to represent a set of parallel constructs that programmers find natural for expressing parallelism. Whether other or better parallel constructs exist remains an open question. What is clear, is that a well chosen set of parallel constructs can greatly increase the amount of parallelism that can be extracted from a program and can significantly improve its ease of expression.

A consistent set of constructs that allow the expression of both control and data parallelism should be the current goal of the parallel language designer.

References

[1] Abelson, H., Sussman, G. J.
Structure and Interpretation of Computer Programs.
MIT Press, Cambridge, Mass., 1985.

[2] Agha, G.
An Overview of Actor Languages.
In *SIGPLAN Notices*, pages 58-67. ACM, October, 1986.

[3] Almes, G. T., Black, A. P., Lazowska, E.D., Noe, J. D.
The Eden System: A technical Review.
IEEE Transactions on Software Engineering SE-11(1):43-59, January, 1985.

[4] Anderson, J. M., Davis, A. L., Hon, R. W., Robinson, I. N., Robison, S. V., Stevens, K. S.
The Architecture of the FAIM-1.
Computer 20(1):55-66, January, 1987.

[5] Ashcroft, E.A.
Ferds -- Massive Parallelism in Lucid.
In *Phoenix Conference on Computation and Communication.* ACM, Phoenix, AZ, March, 1985.

[6] Backus, J.
Can Programmming Languages be Liberated from the von Neuman style?
Communications Of The ACM 21:613-641, August, 1978.

[7] Baden, Scott.
Berkeley FP User's Manual
4.1 edition, University of Calif at Berkeley, Berkeley, CA, 1984.

[8] Black, A., Hutchinson, N., Jul, E., Levy, H. .
Object Structure in the Emerald System.
In *ACM Conference on Object-Oriented Programming Systems* . ACM, Portland, Oregon, October, 1986.

[9] Bobrow, D. et al.
CommonLoops:Merging Lisp and Object Oriented Programming.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 17-29. ACM, Portland, Oregon, September, 1986.

[10] Borning, Alan.
The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory.
ACM Transactions on Programming Languages and Systems 3(4):353-387, October, 1981.

[11] Borning, Alan.
Constraints and Functional Programming.
Technical Report 85-09-05, University of Washington, Seattle, Wash, September, 1985.

[12] Brinch Hansen, P.
Operating System Principles.
Prentice-Hall, Englewood Cliffs, N.J., 1973.

[13] Burstall, R. M., Goguen, J. A.
Putting Theories Together to Make Specifications.
In *5th International Joint Conference on Artificial Intelligence.* ACM, Cambridge, MA, 1077.

[14] Burstall, R.M., MacQueen, D.B., Sannella, D.T.
HOPE: An Experimental Applicative Language.
In *Conference Record of the 1980 Lisp Conference*, pages 136-143. ACM, Palo Alto, CA, August, 1980.

[15] Cardelli, Luca.
ML under Unix
Bell Laboratories, Murray Hill, NJ, 1983.

[16] Celoni, J.R. and Hennessy J.L.
SAL: A Single-Assignment Language.
Technical Report, Stanford University, Stanford, CA, Septembe, 1983.

[17] Church, A.
A Calculi of Lambda Conversion.
Princeton University Press, Princeton, NJ, 1941.

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

[19] 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.

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

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

[22] Gelernter, 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.

[23] Goldberg, A., Robson, D.
Smalltalk-80.
Addison-Wesley, Reading, Mass., 1983.

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

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

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

[27] Hewitt, Carl.
Personal Communication.
1987

[28] Hudak, Paul.
Para-Functional Programming.
Technical Report, Yale University, New Haven, CT, April, 1986.

[29] Ichbiah et al.
Ada Programming Lnaguage Reference.
Technical Report MIL-STD-1815, Department of Defense, December, 1980.

[30] Ingalls, Daniel.
A Simple Technique for Handling Multiple Polymorphism.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 347-349. ACM, Portland, Oregon, September, 1986.

[31] Ishiwkawa, Y., Tokoro, M.
A Concurrent Object-Oriented Knowledge Representation Language.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 232-241. ACM, Portland, Oregon, September, 1986.

[32] Kahn, K., Tribble, E., Miller, M.,
A Concurrent Object-Oriented Knowledge Representation Language.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 232-241. ACM, Portland, Oregon, September, 1986.

[33] Kaiser, G. E., Garlan, D.
MELDing Data Flow and Object-Oriented Programming.
In *OOPSLA '87 Proceedings*, pages 254-267. ACM, Orlando, FL, October, 1987.

[34] Kaiser, G. E., Feiler, P. H.
An Architecture for Intelligent Assistance in Software Development.
In *Proceedings of the Ninth International Conference on Software Engineering*, pages 180-188. IEEE, Monterey, CA, March, 1987.

[35] Kaiser, G. E., Garlan, D.
MELD: A Declarative Language for Writing Methods.
In *6th Annual Phoenix Conference on Computers and Communications*. 6th Annual Phoenix Conference on Computers and Communications, Scottsdale, AZ, Febuary, 1987.

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

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

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

[39] 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.

[40] Liberman, Henry.
A Preview of Act 1.
Technical Report 625, MIT, June, 1981.

[41] Liberman, Henry.
Thinking About Lots Of Things At Once Without Getting Confused.
Technical Report 626, MIT, May, 1981.

[42] Liberman, Henry.
Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 214-223. ACM, Portland, Oregon, September, 1986.

[43] Lindstrom, Gary.
Functional Programming and the Logical Variable.
In *12th ACM Symp. on Principles of Programming Languages*, pages 266-280. ACM, New Orleans, LA, January, 1985.

[44] Liskcov, B.
Overview of the Argus Language System.
Programming Methodology Group 40, Massachusetts Institute Technology for Computer Science, Febuary, 1984.

[45] Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[46] Maguire, G. Q., Jr.
A Graphical Workstation and Programming Environment for Data Driven Computation.
PhD thesis, University of Utah, March, 1983.

[47] Marti, J., Fitch, J.
The Bath Concurrent Lisp Machine.
In *EUROCAM '83*, pages 78-90. EUROCAM, Springer Verlag, New York, N.Y., 1983.

[48] McCarthy, J.
Recursive Functions of Symbolic Expressions.
Communications of the ACM 3(4):184-195, April, 1960.

[49] McGraw, J.R.
The VAL Language: Description and Analysis.
ACM Transactions on Programming Languages and Systems 4(1):44-82, January, 1982.

[50] McGraw, James.
SISAL - Streams and Iteration in a Single Assignment Language
1.1 edition, Lawrence Livermore National Laboratory, Davis, CA, 1983.

[51] Meyer, Bertrand.
Genericity vs Inheritance.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 391-405. ACM, Portland, Oregon, September, 1986.

[52] Moon, David.
Object-Oriented Programming with Flavors.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 1-8. ACM, ACM, Portland, Oregon, September, 1986.

[53] Richards, H.
An Overview oF ARC SASL.
SIGPLAN Notices 19(10):40-45, October, 1984.

[54] Schaffert, C. et al.
An Introduction to Trellis/Owl.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 9-16. ACM, ACM, Portland, Oregon, September, 1986.

[55] Snyder, Alan.
CommonObjects: An Overview.
In *SIGPLAN Notices*, pages 19-28. ACM, October, 1986.

[56] Snyder, Alan.
Encapsulation and Inheritance in Object Oriented Programming Languages.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 38-45. ACM, Portland, Oregon, September, 1986.

[57] Sridharan,N.S.
Semi-Applicative Programming: ExamplAs of Context Free Recognizers.
Technical Report 6135, BBN Laboratories, Cambridge, MA, January, 1986.

[58] Steele, G. L., Hillis W. D.
Connection Machine Lisp.
In *Conference on Lisp and Functional Programming*, pages 279-297. ACM, Cambridge, Mass., 1986.

[59] Steele, G., Sussman, G.
Revised Report on Scheme.
Technical Report 472, MIT, 1978.

[60] Steele, G.L.
The Definition and Implementation of A Computer Programming Language Based on Constraints.
PhD thesis, Massachusetts Institute Technology, August, 1980.

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

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

[63] Stroustrup, B.
An Overview of C++.
In *SIGPLAN Notices*, pages 7-18. ACM, October, 1986.

[64] 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.

[65] Teitelman, W.
A Tour through CEDAR.
In *Proceeding of the 7th International Conference on Software Engineering*. ACM, Orlando, FL, March, 1984.

[66] Theriault, Daniel.
Issues in the Design and Implementation of Act2.
PhD thesis, MIT, June, 1983.
Tech Report 728.

[67] Turner, D. A.
The Semantic Elegance of Applicative Languages.
In *Functional Programming Languages and Computer Architectures*, pages 85-92. ACM, Portsmouth, NH, 1981.

[68] Vegdahl, S. R.
A Survey of Proposed Architectures for the Execution of Functional Languges.
IEEE Transactions on Computers c-33(12):1050-1071, December, 1984.

[69] Wirth, Niklaus.
Programming in Modula-2.
Springer-Verlag, New York, N.Y., 1983.

[70] Yokote, Y., Tokoro, M.
The Design and Implementation of Concurrent Smalltalk.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 331-340. ACM, Portland, Oregon, September, 1986.

[71] Yonezawa, Akinori, Briot, Jean-Pierre, Shibayama, Etsuya.
Object-Oriented Concurrent Programming in ABCL/1.
In Meyrowitz, Norman (editor), *OOPSLA'86*, pages 258-268. ACM, Portland, Oregon, September, 1986.