Multilisp: A Language for Concurrent Symbolic Computation

ROBERT H. HALSTEAD, JR.
Massachusetts Institute of Technology

Multilisp is a version of the Lisp dialect Scheme extended with constructs for parallel execution. Like Scheme, Multilisp is oriented toward symbolic computation. Unlike some parallel programming languages, Multilisp incorporates constructs for causing side effects and for explicitly introducing parallelism. The potential complexity of dealing with side effects in a parallel context is mitigated by the nature of the parallelism constructs and by support for abstract data types: a recommended Multilisp programming style is presented which, if followed, should lead to highly parallel, easily understandable programs.

Multilisp is being implemented on the 32-processor *Concert* multiprocessor; however, it is ultimately intended for use on larger multiprocessors. The current implementation, called *Concert Multilisp*, is complete enough to run the Multilisp compiler itself and has been run on Concert prototypes including up to eight processors. Concert Multilisp uses novel techniques for task scheduling and garbage collection. The task scheduler helps control excessive resource utilization by means of an unfair scheduling policy; the garbage collector uses a multiprocessor algorithm based on the incremental garbage collector of Baker.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*multiple-instruction-stream, multiple-data-stream processors* (*MIMD*); D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Lisp*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*multiprocessing/multiprogramming*; D.4.2 [**Operating Systems**]: Storage Management—*allocation/deallocation strategies*

General Terms: Design, Languages

Additional Key Words and Phrases: Garbage collection, process scheduling

1. INTRODUCTION

Multilisp is an extended version of the programming language Scheme [1]. Like Scheme, Multilisp supports and encourages a largely functional style of programming, but also includes constructs for causing side effects. Like Scheme, Multilisp is a pragmatic compromise, whose design recognizes both the many advantages

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract N00014-83-K-0125.

Author's address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

of functional programming and the expressive problems that purely functional languages still have in some situations.[1]

Unlike Scheme, Multilisp is intended as a vehicle for expressing concurrency in programs. Concurrency is expressed using a small set of constructs that have been added to Scheme to yield Multilisp. Using these constructs, the programmer can explicitly call for concurrency that might not be found by automated analysis at compile time or run time. Such analysis is rendered difficult in Multilisp by the inclusion of side effects.

The objects manipulated by Multilisp tasks all exist in one shared name space, making it easy for a task to use data structures created by other tasks. This is effectively a "shared memory" model and is well suited to a shared memory parallel computer, such as the Concert multiprocessor testbed [3] or more powerful machines such as the Denelcor HEP-1 [51] or the NYU Ultracomputer [21, 22, 48]; however, this shared name space can also be implemented in more distributed ways, as by the Rediflow [36, 37] or "myriaprocessor" [24, 25] designs.

Multilisp's principal construct for both creating tasks and synchronizing among them is the *future*. The construct `(future` X`)` immediately returns a future for the value of the expression X and concurrently begins evaluating X. When the evaluation of X yields a value, that value replaces the future. The future is said to be initially *undetermined*; it becomes *determined* when its value has been computed. An operation (such as addition) that needs to know the value of an undetermined future will be suspended until the future becomes determined, but many operations, such as assignment and parameter passing, do not need to know anything about the values of their operands and may be performed quite comfortably on undetermined futures. The use of futures often exposes surprisingly large amounts of parallelism in a program, as illustrated by a Quicksort program given in Figure 1.

Multilisp inherits from Scheme the ability to use procedures as a rather powerful modularity construct. Object-oriented programming using objects with hidden state information can be accomplished quite easily using a procedure to represent each object. If the hidden state is mutable, the Multilisp programmer can build monitor-like [33] structures to synchronize access to it. Multilisp provides the necessary low-level tools for this encapsulation and synchronization, but no distinctive style or set of higher level abstractions for building mutable objects in Multilisp has yet emerged. Experiments to date have focused mainly on the basic language design and implementation, and existing Multilisp programs make only minor use of side effects.

Experience has shown that Multilisp can be an effective way to expose significant amounts of parallelism in programs of realistic size and that futures expose considerably more parallelism than conventional fork/join constructs. Along with these results, the Multilisp project has generated interesting algorithms for task scheduling and parallel garbage collection. This paper is divided into three sections: Section 2 discusses goals for parallel programming languages and compares some contrasting approaches to parallel programming; Section 3

[1] The reader is referred to [1, ch. 3] for an excellent discussion of the advantages and disadvantages of including side effects in a language.

```
(defun qsort (l) (qs l nil))

(defun qs (l rest)
  (if (null l)
      rest
      (let ((parts (partition (car l) (cdr l))))
        (qs (left-part parts)
            (future (cons (car l) (qs (right-part parts) rest)))))))

(defun partition (elt lst)
  (if (null lst)
      (bundle-parts nil nil)
      (let ((cdrparts (future partition elt (cdr lst))))
        (if (> elt (car lst))
            (bundle-parts (cons (car lst)
                                (future (left-part cdrparts)))
                          (future (right-part cdrparts)))
            (bundle-parts (future (left-part cdrparts))
                          (cons (car lst)
                                (future (right-part cdrparts))))))))

(defun bundle-parts (x y) (cons x y))
(defun left-part (p) (car p))
(defun right-part (p) (cdr p))
```

Fig. 1. Algorithm 1: Quicksort program using futures.

describes Multilisp in detail, gives examples of its use, and compares it with some other Lisp-based languages for parallel computing; and Section 4 describes algorithms used in the Multilisp implementation on the Concert multiprocessor [3]. Finally, Section 5 offers some conclusions.

2. ISSUES IN PARALLEL LANGUAGE DESIGN

A rough taxonomy of programming languages can be constructed using the answers to these questions:

(1) Does the language explicitly include parallelism in its semantics?

(2) Does the language include side effects? The answer to this question may depend on the precise definition of "side effect," but most languages fall clearly on one side or the other of this distinction.

(3) Does the language assume a shared memory model? This question is even more open to interpretation than the previous one; let us say that a shared memory language is one that never requires explicit copying of data in order to make it accessible to some part of a program.

The answers to these questions divide programming languages into four major categories:

(1) Shared memory, side effects, and no explicit parallelism. We may call this the "sequential imperative" category. It includes most familiar uniprocessor languages, such as Fortran and Lisp [43, 55].

(2) Shared memory, no side effects, and no explicit parallelism. This category of "functional" languages includes SISAL [23, 44], VAL [2], ID [4], Pure Lisp [43], SASL [54], and FP [5].

(3) Explicit parallelism, side effects, and no shared memory. This is the category of "communicating sequential processes" (CSP) languages, and includes Communicating Sequential Processes [32] and various extensions of sequential imperative languages for programming nonshared-memory multiprocessor hardware [49].

(4) Explicit parallelism, side effects, and shared memory. These "parallel imperative" languages include Concurrent Prolog [50], Ada [34], and Multilisp.

Algorithms may have opportunities for parallelism at any of several levels of granularity, ranging from short sequences of primitive operations to large program modules. These opportunities are multiplicative: if the application of medium- or fine-grain parallelism within a module is sufficient to occupy *m* processors, and furthermore *n* of these modules can be executed in parallel, then *mn* processors will be able to be used efficiently to execute the program as a whole unless contention for shared resources imposes a smaller limit. Thus opportunities for parallelism should be exploited at all levels if execution on a highly parallel machine is desired.

The inclusion of side effects, shared memory, and explicit parallelism in Multilisp is motivated by the following syllogism:

(1) Side effects add to the expressive power of a programming language. Despite many interesting and powerful techniques for programming in side-effect-free languages, such as the use of streams [1, 56], functional languages still have difficulty efficiently expressing computations, such as database manipulation and constraint propagation, which are most naturally thought of in terms of objects with mutable state information.[2]

(2) Without shared memory, it is difficult to exploit parallelism at the medium and fine levels of granularity. The explicit copying of shared data from one domain to another, even if efficiently supported at the implementation level, is bothersome to the programmer and discourages the writing of programs where interaction is frequent. This point is elaborated in Section 2.3.

(3) With side effects, but no explicit parallelism (the "sequential imperative" family of languages), coarse-grain parallelism is hard to come by. If parallelism is not explicit, then *any* parallelism that is used must be uncovered by compile-time (or possibly run-time) analysis of a program [40]. Side effects make such analysis much more difficult, especially on the scale of large program modules. (This is not to say that compile-time analysis should not be used, only that it should not be relied upon as the sole way to discover parallelism.)

Thus if Multilisp is to enjoy the extra expressiveness of side effects, it must also include explicit parallelism, or forego opportunities for coarse-grain parallelism. Furthermore, if Multilisp adopts a compartmentalized, CSP-like style without shared memory, opportunities for medium- to fine-grain parallelism will be missed. The inclusion of side effects, shared memory, and explicit parallelism thus helps Multilisp be an expressive language capable of exposing parallelism at all levels of granularity.

[2] This point is further discussed in [1, ch. 3].

2.1 Software Engineering of Parallel Programs

Multilisp is an attempt to move the practice of parallel programming beyond the areas where it has heretofore enjoyed most of its success, namely, where either computations have a very regular structure, or where it is easy to decompose a program into virtually autonomous units that can execute using a relatively simple (and often low-bandwidth) style of interaction with each other. Tightly coupled algorithms of irregular structure have not been easy to program for parallel execution. A language model to attack this problem should be (1) reasonably independent of parameters of the actual target machine, such as topology and number of processing elements; and (2) easy enough to program so that it can serve as a basis for a reasonable software engineering discipline. Condition (1) is important if software is to be transported from one configuration to another, and if software is to have at least the degree of fault tolerance that will allow it to run on incomplete configurations (at some expense in execution time).

Condition (2) is vital if software systems of any size are to be developed. Existing sequential programming languages with parallelism constructs naively grafted onto them can be difficult to program in because the use of the parallelism constructs leads to excessively complex programs. Parallelism must be incorporated into a programming language in such a way that the cognitive load on the programmer remains within acceptable limits. In other words, parallelism must be integrated with the main structuring facilities of a programming langauge, and these structuring facilities must have the power to help the programmer cope with the extra complexity resulting from the introduction of parallelism. The `future` construct in Multilisp, for example, offers a way to introduce parallelism that fits very nicely with the principal program operation of Multilisp—expression evaluation. Also, no special care is required to use a value generated by `future`. Synchronization between the producer and the users of a future's value is implicit, freeing the programmer's mind from a possible source of concern.

Another way a programming language can help programmers cope with this extra complexity is by supplying useful modularity constructs and other support for structured program organization. Two such forms of support, particularly relevant to symbolic programming, are garbage collected heap storage and support for data abstractions. Data abstractions are a useful way to represent objects with state. Access to state variables within a data abstraction is restricted to procedures within the abstraction, which generally leads to more modular and understandable programs, especially in parallel programming, where the protocol for updating shared state variables is critical to program correctness.

Garbage collected heap storage allows the dynamic creation and deletion of objects, without imposing on the programmer the burden of tracking the use of each object so that it may be deallocated at the proper time. This assistance is valuable even in sequential programming (witness the widespread use of Lisp), but is even more important in parallel programming, where objects may be shared among several concurrent tasks and the task that created an object may not be the last to use it. Garbage collected heap storage simplifies programs and also strengthens modularity by not requiring different users of an object to understand the storage management protocol that applies to it.

2.2 Numerical Versus Symbolic Computation

Algorithms differ in the degree to which they emphasize numerical or symbolic computation. A favorite language for numerical programming has been Fortran, while heavily symbolic programs have more often been written in languages such as Lisp [55] or Smalltalk [20]. Each of these languages includes idioms for common program structures in its primary application area, such as matrix and vector processing (in the numerical case) and manipulation of tree-structured data (in the symbolic case).

Numerical computation emphasizes arithmetic: the principal function of a numerical program may be described as delivering numbers to an arithmetic unit to calculate a result. Numerical programs generally have a relatively data-independent flow of control:[3] within broad limits, the same sequence of calculations will be performed no matter what the operand values are. Matrices and vectors are common data structures in numerical programs, a fact exploited by SIMD techniques in such numerically oriented supercomputers as the Cray-1 [47] and Illiac IV [9].

In contrast, symbolic computation emphasizes rearrangement of data. The principal function of a symbolic program may be broadly stated as the reorganization of a set of data so that the relevant information in it is more useful or easier to extract. Examples of primarily symbolic algorithms include sorting, compiling, database management, symbolic algebra, expert systems, and other artificial intelligence applications. The sequence of operations in symbolic programs is often highly data dependent and less amenable to compile-time analysis than in the case of numerical computation. Moreover, there does not appear to be any simple operation style, comparable to vector operations in numerical programs, that can easily be exploited to increase performance with a SIMD type of architecture.[4] Accordingly, Multilisp adopts a MIMD approach and does not rely on compile-time analysis for extraction of parallelism.

2.3 Problems with the Communicating Sequential Processes Paradigm

The principal failings of the CSP paradigm, with its lack of shared memory, are

(1) It associates each protected domain of data in the program with a single sequential thread of execution.

(2) It leads to a nonuniform style of access to data: one style for accesses local to a process, another (message transmission) for accesses between processes. The burden of devising and using protocols for nonlocal accesses strongly discourages the programmer from carving his program into a large number of processes. This psychological disincentive operates whether or not there is any underlying difference in efficiency of nonlocal versus local accesses.

[3] This generalization, like most, has many exceptions. Inner loops of numerical programs may contain conditionals, and overall control of a program generally includes tests of convergence criteria and such. Still, most numerical programs have a relatively predictable control sequence, compared with the majority of symbolic programs.

[4] There are some operations, such as procedure calling, pointer following, and even tree search, that occur frequently in symbolic programs, but it is not obvious how SIMD parallelism can help very efficiently with these.

The combination of these two attributes encourages the programmer to create a few large processes, rather than a multitude of smaller ones; thus parallelism at small levels of granularity is discouraged. Large processes require large processors and thus lead inexorably toward systems with relatively small numbers of large processors and away from "myriaprocessors"—highly parallel machines with relatively small processing elements—that might exploit coming technology more effectively.

The information-hiding properties of processes in the CSP model superficially resemble those of data abstractions, but there are two differences. First, a data abstraction is not necessarily restricted to only one sequential thread of execution at a time, though some data abstractions might be written that way. Second, although the *state variables* of a data abstraction are private and hidden, their *values* may refer to data structures that are shared. The CSP model does not facilitate this kind of sharing between processes, forcing the programmer to implement the protocols necessary for sharing.

2.4 Determinacy

An important philosophical choice facing the designer of a language for parallel programming is whether or not to design the language so that all programs are guaranteed to be determinate (i.e., to produce the same output whenever applied to the same input). Determinacy is a property of ordinary sequential programming languages and is a powerful aid in debugging and validating programs. Indeed, it has been observed that programmer productivity decreases by a factor of three when working on operating system kernel code, a common context where nondeterminacy is encountered [10]. Unfortunately, the combination of side effects and explicit parallelism makes it easy to write nondeterminate programs.

In many applications, such as operating systems and databases, the response to one source of input may depend on inputs previously received from other sources. For example, an airline reservation system may respond differently to a query depending on whether or not a reservation was previously made from some other port. Such a system must impose some arbitrary time ordering on inputs that are not ordered by any causality relation in the system and is inherently nondeterminate. It is often argued, therefore, that no language that allows only determinate programs to be expressed can be powerful enough for a wide range of applications. This argument has led to the proposal of mechanisms such as the nondeterminate merge of two streams [17] to be appended to determinate languages, allowing nondeterminate programs to be built mostly out of determinate modules. In contrast, Multilisp can be viewed as a determinate language—Pure Lisp—plus a side effect mechanism. Debugging and validation of programs *may* be easier using streams, but there is extremely little evidence one way or the other.

Even for writing programs that are supposed to be determinate, the language restrictions required to *guarantee* determinacy limit the programmer's flexibility to specify certain resource use policies, such as the choice between updating a data structure in place and allocating a new area for an updated copy. A clever compiler or run-time system must then be relied upon to make reasonable resource allocation decisions. This is not too difficult in some cases, as when reference counts can be used to determine that a structure to be copied is about to be thrown away and therefore can instead be safely updated in place, but other situations are more complicated. Because nondeterminate languages such as Multilisp allow the programmer more flexibility, they must be used wisely, but they avoid relying on yet-to-be-developed compiler technology for efficient implementation.

2.5 Styles of Introducing Parallelism

The ways of introducing parallelism into the execution of an algorithm may be grouped into two categories. The more conservative way is to keep precisely the same set of operations, but (with the aid of suitable programming language constructs) relax the precedence constraints among operations, allowing some of them to be performed in parallel. The more aggressive way is to initiate additional computations, in "eager-beaver" fashion, on speculation that their values will prove useful. This speculative style would often be used in search problems, where many paths may be explored in parallel, but the finding of one solution renders all other explorations irrelevant.

These two approaches require very different kinds of support. In the first, more conservative, case, the scheduling of operations is a performance issue, not a semantic one: the same (presumably finite) set of operations will be performed, no matter what order is selected (ignoring interactions between tasks due to side effects). Thus questions of fairness in scheduling or resource allocation are not central to the correctness of programs. They are only matters of efficiency and can be left to a suitably clever run-time system.

The speculative approach, by contrast, involves the programmer more directly in scheduling and resource allocation. If expensive or even potentially nonterminating computations are started, in eager-beaver style, on speculation that their results may prove relevant, then the programmer will want to control the amount of system resources dedicated to their execution, so that the most important tasks are not neglected in favor of others. The programmer will possibly even want to suspend or terminate tasks because of information generated elsewhere in the execution of the program.

Some confusion has been generated in the design of languages for parallel computing by the failure to distinguish between these two sources of parallelism and appreciate their different requirements. For example, fairness of scheduling is necessary only when dealing with speculative parallelism: a language that does not promise fair scheduling in other cases may be able to be implemented more efficiently (this issue is discussed further in Section 4.3). To make the best use of parallel hardware, both approaches to parallelism should be available, but Multilisp currently supports the introduction of parallelism primarily by means of the first, more conservative, approach. The requirements of the second approach are more varied and less clear and will be further explored in the future.

3. MULTILISP

Multilisp is a member of the Scheme [1] family of Lisp dialects. Like all Lisp dialects, Multilisp allocates storage out of a garbage collected heap. In addition, Multilisp shares with Scheme three properties that distinguish them from the more common Lisp dialects:

(1) Exclusive reliance on lexical scoping. Lexical scoping (resolution of free variable references in the environment where the referencing procedure was created) decouples the choice of variable names in a procedure *P* from the choice of free variable names in other procedures that call or are called by *P* and thus promotes modularity more effectively than the traditional Lisp discipline of dynamic scoping (resolution of free variable references in the environment of the calling procedure). Furthermore, the usual optimized implementation of dynamic scoping by "shallow binding" [55] does not adapt gracefully to a multitask environment where various tasks running in the same address space may have different values for the same variable. Implementation of lexical binding by means of a static chain of environments continues to work well.

(2) "First-class citizenship" for procedure values. Procedures in Scheme and Multilisp may be freely passed as arguments, returned as values of other procedures, stored in data structures, and treated like any other kind of value. Correct implementation of this, in combination with lexical scoping, requires the use of garbage-collected heap storage for procedure environments. Once this expense is incurred, however, procedure values can be used as data abstractions. The nonlocal variables of a procedure, found in the lexically enclosing environment, can be considered as the underlying state variables of a data abstraction implemented by the procedure. Operations on this data abstraction can be performed by calls to the procedure. As with other implementations of abstract data types, the underlying state variables can be protected from access except through the channels provided by the abstraction [1].

(3) "Tail recursion." Multilisp, like Scheme, is tail recursive. This means that if a function *f* calls a function *g* in such a way that the value returned by *g* is also returned by *f*, with no further work required by *f* upon the return from *g*, then the call to *g* followed by the return from *f* are collapsed into effectively a jump to *g*. The return from *g* then returns directly to the caller of *f*. Tail recursion is analogous to the familiar machine-language optimization of converting a subroutine call immediately followed by a subroutine return into simply a jump instruction. This tactic saves a small amount of execution time by eliminating some subroutine return processing, but more importantly, it avoids unnecessary buildup of saved data on the stack, allowing a recursive subroutine that calls itself in a tail-recursive manner to be used in place of an iterative looping construct, without accumulating stack frames. Thus in Multilisp (as in Scheme) there is no need for a separate built-in looping construct. Instead, a recursive procedure, free of side effects and hence more amenable to parallel processing, can be used as a loop [1, 52].

3.1 The Multilisp Approach to Parallelism

Multilisp includes the usual Lisp side-effect-producing primitives for altering data structures and changing the values of variables. Therefore, control sequencing beyond that imposed by data dependencies may be required in order to assure determinate execution. In this respect, Multilisp parts company with some concurrent Lisp languages [11, 17, 36], which include only a side-effect-free subset of Lisp.

Although it does include side effects, Lisp is superior to most other common programming languges in that it includes a side-effect-free subset with substantial expressive power. This subset is part of Multilisp; thus it is possible to write significant bodies of Multilisp code in a completely side-effect-free way. Furthermore, where side effects are used, as in maintaining a changing database, they can be encapsulated within a data abstraction that synchronizes concurrent operations on the data. The data abstraction can ensure that the data are only accessed according to the proper protocol.

Multilisp thus supports a programming style in which most code is written without side effects, and data abstractions are used to encapsulate data on which side effects may be performed, to present a reasonable interface to the exterior. The programmer's aim in using this style should be to produce a program whose side effects are compartmentalized carefully enough that any module may safely be invoked in parallel with any other. If this style is followed, the difficulties caused by the presence of side effects will be isolated to small regions of the program and should therefore be reduced to manageable proportions.

Nevertheless, the inclusion of side effects in Multilisp requires constructs by which the sequence of execution can be controlled explicitly. A goal of the Multilisp design has been to identify the natural structures of Lisp programs and capitalize on them by introducing variants that perform similar functions but give rise to opportunities for parallelism. The intent is to minimize the extra cognitive load on the programmer that results from the introduction of parallelism by making the parallelism "flow" in the same direction as the rest of the language.

The default in Multilisp is sequential execution. This allows Lisp programs or subprograms written without attention to parallelism to run, albeit without using the potential concurrency of the target machine. To introduce parallelism, a programmer may use the constructs described below.

3.1.1 *PCALL.* A simple mechanism for introducing parallelism into a Multilisp program is the `pcall` construct. A Multilisp expression such as

`(pcall` *F A B C*`...)` (1)

is equivalent to the procedure call

`(`*F A B C*`...)` (2)

except that case (1) will result in concurrent evaluation of the expressions *F*, *A*, *B*, *C*, ..., whereas in case (2) the same expressions will be evaluated one after the other. More precisely, expression (1) results in concurrent evaluation of the expressions *F*, *A*, *B*, *C*, ... to their values *f*, *a*, *b*, *c*, ..., after which the value *f* (presumably a procedure) is applied to the argument values *a*, *b*, *c*, ... `pcall` thus embodies an implicit fork-join followed by a procedure call. Since procedure calls are a common construct in Lisp and Multilisp, `pcall` is useful in many situations. Even though the degree of parallelism created by any one `pcall` is limited by the number of arguments in that `pcall` expression, use of `pcall` within a recursive procedure such as a tree walk can result in exponential amounts of parallelism.[5]

The cognitive load imposed by `pcall` is low because it is closely integrated with the procedure-calling mechanism of Multilisp. Nevertheless, the programmer is left with the responsibility of determining whether the use of `pcall` in each instance is safe. In thoughtlessly written programs, this burden would often be severe; however, if the programmer adheres to the programming style recommended in Section 3.1, writing mostly side-effect-free code and encapsulating all side effects within suitable data abstractions, this load can be considerably reduced. In our experience thus far with programs of moderate size, we have found this approach to be quite tractable.

3.1.2 *Futures.* `pcall` allows concurrency between the evaluation of two or more expressions that are arguments to a function. Often an additional form of concurrency is desirable: concurrency between the computation of a value and the disposition or use of that value. For example, the expression

`(pcall cons` *A B*`)` (3)

will concurrently evaluate *A* and *B*, and then use the Lisp primitive `cons` to build a *conscell*, a data structure containing the two values. The actual values of *A* and *B* are not really needed when the data structure is built. Only tokens, or place-holders, for the eventual values of *A* and *B* are really needed. These are effectively "futures contracts" or "promises to deliver" the values of *A* and *B* when needed [6].[6] If the data structure built using these tokens is not accessed immediately, substantial concurrency may be possible between the evaluations of *A* and *B* *and* operations that follow construction of the data structure.

This concurrency can be captured using the `future` construct introduced in Section 1 by rewriting expression (3) above as

`(cons (future` *A*`) (future` *B*`))`. (4)

Note that this has more potential for parallelism than the similar expression

`(+ (future` *A*`) (future` *B*`))` (5)

because the + operator will immediately need to examine its argument values (so it can add them). Hence expression (5) yields essentially the same parallelism as

`(pcall +` *A B*`)`. (6)

`pcall` can be implemented in terms of futures. For example, `(pcall` *F A B*`)` can be implemented by creating futures for the values of *F*, *A*, and *B*, and then

[5] The notion of parallel evaluation of arguments to function calls in Lisp is hardly new, but the author does not know of any instances of its use on any serious scale on an actual parallel machine.
[6] These "futures" greatly resemble the "eventual values" of Hibbard's Algol 68 [39]. The principal difference is that eventual values are declared as a separate data type, distinct from the type of the value they take on, whereas futures cannot be distinguished from the type of the value they take on. This difference reflects the contrast between Algol's philosophy of type checking at compile time and the Lisp philosophy of run-time tagging of data.

"touching" all three to force their evaluation to be complete before F is called.[7] Thus `future` is a more fundamental construct than `pcall`. `pcall` is included along with `future` in Multilisp because there may well be situations in which a programmer feels confident that two expressions A and B can safely be evaluated in parallel with each other, but is less sure of the safety of evaluating A concurrently with arbitrary subsequent processing of the program. `pcall` is a more conservative approach to introducing parallelism, although it should be possible to use `future` with confidence if the programming style recommended in Section 3.1 is followed.

3.1.3 *Example of the Use of Futures.* Algorithm 1, shown in Figure 1, offers a more substantial example of the use of futures. The procedure `qs` sorts a list `l` of numbers using the Quicksort algorithm: the procedure `partition` uses the first element `elt` of `l` to divide the rest of `l` into two lists, one containing only elements less than `elt`, and the other containing only elements greater than or equal to `elt`. Each of these lists is recursively sorted using `qs`, and the results, along with the partitioning element `elt`, are appended in the proper order to form the result. To avoid the overhead of an explicit append operation, `qs` takes an additional argument `rest`, which is the list that should appear after the list of sorted elements of `l`. The top-level procedure `qsort` accordingly supplies `nil` as the initial value for `rest`.[8]

Algorithm 1 is free of side effects and was obtained from a side-effect-free sequential program simply by enclosing certain expressions inside a `future` operator. Where to put the `future` operators is not immediately obvious. Every use of `future` in Algorithm 1 gains a significant amount of parallelism. The one use of `future` in the procedure `qs` allows the sorting of the left and right sublists returned by `partition` to occur in parallel. Note that `pcall` could not easily be used in this case because the result of one of the sorts is an argument to the other sort. On the other hand, `future` works well because this argument will not be touched inside the sort—it will simply be built in at the end of the sorted list.

The uses of `future` in `partition` are more interesting, and distinguish this Quicksort program from those that have been studied on other parallel processing systems [13, 45]. `partition` is designed to get some results back to its caller

[7] The current Multilisp implementation of `pcall` uses this strategy. The "touching" can be done using the primitive `touch`, which acts as a strict identity operator whose execution will be suspended if its argument is an undetermined future.

[8] Algorithm 1 uses a number of common Lisp operators: `(car` X`)` returns the first element of the conscell X; thus `(car (cons` A B`))` always gives the value of A. `(cdr` X`)` returns the second element of the conscell X. `(null` X`)` returns true if X evaluates to the value `nil`, else `(null` X`)` returns `nil`. `(defun` F `(`*bvl*`)` *body*`)` defines a function named F, with formal parameters *bvl* and the designated *body*. `(if` A B C`)` returns the value of B if the value of A is not `nil`; else it returns the value of C. `(let ((`var_1 val_1`) (`var_2 val_2`) ⋯ )` *body*`)` evaluates *body* in an environment where each var_i has been bound to the value of the corresponding expression val_i (free variables in the `let`- expression are evaluated in the surrounding environment).

The input and output of Algorithm 1 are represented as *lists*. A list is a seqence of *elements* represented as a chain of conscells such that the `car` of the list contains the first element of the list, and the `cdr` is a list of the remaining elements. The empty list, that is, the list with no elements, is represented by `nil`.

as soon as possible, even while the partitioning is still continuing. Therefore, the recursive call to partition the cdr of the argument lst is enclosed within a future; otherwise, no value could be returned from partition until the recursive calls to partition had gone all the way to the end of lst and then returned.

The remaining uses of future in partition are needed to avoid throwing away the parallelism purchased with this first use. partition returns a "bundle" (represented as a conscell) containing the two sublists it produces. It builds this bundle using cdrparts, a future for the bundle obtained by recursively partitioning the cdr of the argument lst. The first element of lst is then consed onto the correct sublist of cdrparts, depending on whether this first element is greater than the discriminant element elt. However, since cdrparts starts its life as a future, even the simple operation of trying to select one of its component sublists may block, delaying the return from partition, largely negating the parallelism obtained by placing the recursive call to partition within a future. This problem is solved by also placing all expressions selecting the right or left sublists of cdrparts within futures. Thus partition can create and return a result bundle logically containing sublists that have not been fully (or even partially) computed yet.

Since partition can return a bundle of two sublists before the partitioning of its argument has completed, the sorting of the sublists can begin before the production of the sublists has ended. This sorting will, in general, involve further partitioning, followed by further sorting, etc. The net result is that a large amount of concurrent activity, at many levels of recursion, quickly comes into being. This contrasts with other parallel implementations of Quicksort, where the partitioning is a sequential process, and sorting of the sublists cannot begin until the partitioning has finished. Performance figures for Algorithm 1, which appear in Section 4.5, demonstrate the added parallelism of this program. In fairness, it should also be noted that this program, which copies its argument list during partitioning, is subtly different from the usual Quicksort program, which updates the list of argument values in place. Thus Algorithm 1 may incur some extra overhead in order to make extra parallelism available. Nevertheless, its structure shows some of the new ways in which futures can help expose parallelism.

An alternative to the explicit use of futures, in a side-effect-free dialect of Lisp, would be to simply cause every expression to return a future. This would ensure the availability of the maximum amount of parallelism, but at the cost of significant execution time overhead. Perhaps a compiler can do a good job of locating the most desirable places to put future operators in side-effect-free procedures; this idea is currently under investigation.

3.1.4 *Futures Versus Lazy Evaluation.* Futures are related to the concept of "lazy," or demand-driven, evaluation [18, 30]. Like lazy evaluation, futures can be used to compute nonstrict functions—functions that terminate despite the possible nontermination of the computation of one or more argument values. Such nonstrict functions are useful for creating and manipulating infinite, or merely large, objects such as the list of all the integers, the list of all the prime numbers, or the list of all leaf nodes of a tree, without having to calculate them fully in advance. Any finite computation using these objects would only examine a finite portion of them; by using lazy evaluation, it is possible to compute only those portions of an object that are needed to determine a desired result.

Futures allow a computation to proceed past the calculation of a value without waiting for that calculation to complete, and if the value is never used subsequently, the computation will never pause to wait for the calculation of the value to finish. Thus futures can be used to compute nonstrict functions, but the cost is high. Implicit in the concept of a Multilisp future is the devotion of some computational resources to evaluating the expression within the future, even before it is certain that the value of the future will ever be used. In this respect, Multilisp futures resemble eager-beaver evaluation [6, 7]. However, unlike many formulations of eager-beaver evaluation, Multilisp makes no provision for halting the evaluation of a future when it becomes clear that the associated value will never be used. Such a provision complicates the garbage collection mechanism of a language [7] and is not necessary for the intended use of futures in Multilisp, which (as discussed in Section 2.5) is for calculations that terminate and whose values the programmer is quite sure will be used.

In the case of a computation whose value might not be required, the programmer might desire to return a place-holder, like a future, for the value, but delay committing resources to the computation until the value is required (if this ever occurs). Multilisp provides for this by means of the `delay` primitive. `(delay` *X*`)` is identical to `(future` *X*`)`, except that in the case of `(delay` *X*`)` the evaluation of *X* will not begin until some other computation requires its value. Thus the Multilisp function `ints-from`, defined below, returns the list of all integers greater than or equal to the argument *n*, in ascending order, but only computes the integers as they are needed by other computations.

```
(defun ints-from (n)
  (cons n
        (delay (ints-from (+ n 1)))))
```

The semantics of `ints-from` would remain unchanged, in a theoretical sense, if the `delay` were replaced by a `future`. However, the version using `future` would spawn a nonterminating computation that would compete with other running processes. This computation would not only use up processing cycles that might be more usefully applied elsewhere, but, in our practical world of limited resources, would also fill up storage with list elements that might never be accessed.

There is really a large, multidimensional space of delayed evaluation constructs, in which `future` and `delay` are only two particular points. The major issues that distinguish points in this space are

(1) *Resource allocation.* How and when are computational and storage resources devoted to a computation?
(2) *Evaluation order.* What precedence constraints, if any, are enforced?

In a side-effect-free program, issue (2) is not a concern, since all evaluation orders that respect the explicit data dependencies of the program will produce the same result. In a hypothetical computer with infinite resources, issue (1) is not a concern, but when resources are finite, `delay` is a semantic extension of the language, since it permits the representation of infinite objects while *guaranteeing* that the storage used in representing an object is proportional to the size of the portion of the object that is actually explored. Thus all uses of `future` in a program could be removed with no effect on the ultimate resource use of the program, but removing uses of `delay` might cause unbounded resource usage (and hence nontermination) by a program whose resource usage was previously finite.

The programmer might specify `delay`, rather than `future`, just to indicate that the value of an expression may not be used, and therefore that machine resources should not be devoted to it until its value is needed. However, `delay` can also be used to indicate that interactions between the delayed expression and the rest of the program require semantically that the delayed expression not be evaluated until a later time, as in an expression of the form

```
(setq recursively-defined-stream
      (cons 1 (delay (f recursively-defined-stream))))
```

where the `delay` construct is essential. If it were left out, `recursively-defined-stream` would consist of a 1 consed onto the result of applying `f` to whatever list was previously bound to `recursively-defined-stream`. This style of definition may displease some programming disciplinarians, and it certainly is easy to make mistakes when using it; nevertheless, it has some expressive power [1] and is a case where the distinction between `delay` and `future` makes a profound semantic difference.

3.2 Automatic Versus Manual Resource Allocation

An important property of a programming langauge is the visibility of the target machine in the source language. If the target architecture is visible, the programmer may have the opportunity to choose a style of expression that is especially efficient for the target machine. Otherwise, the compiler and/or run-time system must be relied upon to perform such optimization, if necessary. The latter approach is clearly the more desirable, all other things being equal, since it leads to more machine-independent programs, but it requires the existence of reasonably good optimization algorithms for the target machine.

Parallel architectures have several new opportunities for optimization, in task scheduling and data placement, beyond those found in single-sequence architectures. We do not yet understand how to exploit these new opportunities well. Ideally, a language for programming parallel machines would be machine independent in the same sense as our best contemporary sequential langauges, but at the current state of the art it may not be feasible to completely hide the gross structure of the target machine. Scheduling and data placement can have a dramatic impact on the efficiency of execution of a parallel algorithm, and our insight into how to resolve these problems in the general case is still weak.

In Multilisp, data placement is handled automatically. Scheduling is specified in part by the programmer, who uses `pcall` and `future` to indicate task boundaries, and in part by the run-time system, which decides when and where to execute tasks. The goal of machine independence is compromised slightly, since the programmer is required to specify the granularity of tasks, a decision that presumably reflects the cost of task creation and the number of parallel processors in the target machine. Nevertheless, this is a much smaller degree of machine dependence than if the programmer were required, for example, to explicitly assign tasks to physical processors.

The automatic mechanisms for data placement and task scheduling work well enough that they do not cause any serious bottlenecks in the current Multilisp implementation, but may not work as well in future implementations where processor speed comes closer to challenging the bandwidth of the memory and communications systems. Thus it may eventually be necessary to give the programmer more control over these decisions, even at the price of increasing the machine dependence of Multilisp.

3.3 Other Multiprocessor Lisp Designs

Many researchers have been interested in using parallelism together with Lisp [11, 17, 19, 42, 53]. Of these projects, the one most similar to Multilisp in its goals and mechanism is the QLAMBDA language of Gabriel and McCarthy [19]. The QLET operator in this language can be used to duplicate the function of `pcall`. QLET also provides a capability similar to futures, by allowing the evaluation of an expression *E* to begin in parallel with the computation of some data that will eventually be used by *E*. This may be thought of as the "downward" capability of futures, in which an operand passed "down" to some computation *C* need not be fully calculated before *C* begins. However, Multilisp futures can also be used in an "upward" sense, allowing a computation *C* to return control to its caller even before the value returned by *C* is fully determined. The QLAMBDA langauge also offers this capability, but it is subject to some curious interactions with the CATCH and QCATCH operators of QLAMBDA. Exiting a CATCH causes all tasks created within its body to be killed; exiting a QCATCH via a THROW also has this result. This makes the upward use of futures somewhat hazardous. For example, the Quicksort program of Algorithm 1 may return a value containing not yet determined futures. If such a value were returned through a CATCH, the tasks calculating the values of those futures would be killed and the sort would never finish. Thus the QLAMBDA programmer needs to be careful with values that may contain futures, and not return such values through a CATCH until they are fully computed. This violates a principle of abstraction stating that the same operations should be legal on a value whether or not it was produced using futures.

On the other hand, QLAMBDA addresses a number of issues not yet confronted by Multilisp. By means of its QLAMBDA construct, the QLAMBDA langauge offers a synchronization construct similar to monitors [33]. Similar capabilities can be built in Multilisp (e.g., using closures and semaphores), but Multilisp does not currently offer all these capabilities packaged in a single construct. Also, the CATCH operator, despite the complaints discussed above, does provide some control over the execution of tasks, by making it possible for one process to kill another. Multilisp does not currently include a comparable facility.

C-LISP [53] is an extension of LISP 1.5 that allows explicit spawning of processes to evaluate designated forms. Concurrently executing processes may communicate with each other by means of shared variables. Primitives are provided for mutual exclusion, checking whether a process has terminated, and retrieving the value computed by a terminated process. These capabilities are all available in Multilisp. In particular, the processes of C-LISP can be emulated using futures; however, a C-LISP program must take explicit steps to retrieve a value produced by a process, whereas a Multilisp future can be used without knowledge that it is a future rather than an ordinary datum. This transparency simplifies programs and decouples the scheduling strategies used in producing objects from the protocols used in accessing them.

Friedman and Wise [17] describe a side-effect-free dialect of Lisp featuring a nonstrict `cons` operator whose operands are always evaluated as futures, allowing every call to `cons` to return quickly. Their langauge also features a nondeterministic constructor function `frons` that allows, among many other things, parallel execution of multiple algorithms for computing a desired result, where only the first result yielded by any of the algorithms is wanted. `frons` can also be used to effect the nondeterministic merge of two streams of values. Since side effects are not allowed in this dialect, partially completed computations whose values are not used can be quietly abandoned without the concern for clean termination that would arise in Multilisp or QLAMBDA.

Paralisp [11] is a side-effect-free langauge offering four evaluation modes—standard, parallel, eager, and lazy—which correspond roughly to sequential evaluation, `pcall`, `future`, and `delay` in Multilisp. However, in Paralisp, the evaluation mode is inherited by default from the dynamically containing expression, rather than being specified explicitly for each departure from sequential evaluation.

The Bath Concurrent Lisp Machine [42] relies on compile-time analysis to reveal the parallelism to be exploited during execution. There are no primitives by which the user may explicitly call for parallel execution; nevertheless, useful amounts of parallelism are found after interprocedural data flow analysis. This kind of analysis could also be applied to Multilisp programs of the sort studied in [42], with comparable results, but certain programming styles, notably data-directed programming and the use of functions as first-class data objects, present difficult problems for compile-time analysis, even where a programmer may see numerous opportunities for parallelism. A language with explicit constructs for introducing parallelism allows these opportunities to be exploited also.

4. IMPLEMENTATION OF MULTILISP

Multilisp is implemented on Concert [3], an experimental multiprocessor under construction in the author's laboratory. The goals of this implementation, called *Concert Multilisp,* are (1) to gain experience with the implementation algorithms described below and (2) to provide a genuinely parallel machine running Multilisp, for use in developing and measuring substantial applications. Activity (2) is very important to us. The design of future multiprocessors must be based on analysis of a representative body of software that might be expected to run on them. For a language as different from mainstream languages as Multilisp, this software does not exist. Therefore, we need a system on which we can develop software in Multilisp. Furthermore, we expect the development of this software to give us valuable feedback (as it already has) on the usefulness of the various Multilisp constructs.

An obvious alternative to implementation on a real multiprocessor is implementation via simulation. Indeed, this approach was used for various precursors of Multilisp [24, 25], but a danger lurks in the limited speed of simulated execution, which is often two or three orders of magnitude slower than that of an actual implementation. The programs for which we might like to use multiprocessors are time consuming; otherwise a single processor would suffice. A slow simulator discourages work on long programs in favor of short, small, "toy" programs. Short programs are often qualitatively different from long programs; they present different software engineering difficulties, different levels of opportunity for parallelism, and different patterns of access to data. It is therefore vital to have an experimental system powerful enough to allow work with substantial programs.

The Concert multiprocessor, when fully built, will comprise 32 MC68000 processors and a total of about 20 megabytes of memory. It can be described most concisely as a shared-memory multiprocessor, but the organization of Concert includes various local paths from processors to nearby memory modules, so Concert provides a higher overall bandwidth between processors and memory if most memory accesses are local.

As of this writing, the largest part of Concert on which it has been possible to test Multilisp is an 8-processor section of the eventual Concert machine. This machine consists of 8 processors, each connected to 1 port of a dual-ported memory. Each processor, and the remaining port of each memory, is connected to a shared Multibus [35]. Thus every processor can access any memory module in the system, but each processor has 1 tightly coupled (or *local*) memory module that it can access without using the Multibus. A 16-bit access by a processor to its local memory takes 500 nanoseconds; a 16-bit access using the Multibus takes about 1 microsecond, in the absence of contention for the Multibus. Performance of Concert Multilisp on this machine, given sufficient parallelism in the Multilisp program being executed, is essentially 8 times as fast as performance on a single processor; hence Concert Multilisp has no built-in bottlenecks that prevent full use of all eight processors. This is not an especially demanding performance test, but does give a certain empirical validation to the algorithms used.[9]

4.1 MCODE

A program to be executed by Concert Multilisp is first compiled into a machine-level language called MCODE. In the best recursive tradition, the compiler is itself written in Multilisp. MCODE is interpreted by a program written in the programming language C [38]. One copy of this 3000-line program is located in the local memory of, and executed by, each processor in the Concert machine. Each processor manipulates various per-processor data structures, such as top-of-stack caches, also located in the processor's local memory. Shared data structures (including MCODE programs) reside in a garbage collected heap that is distributed among the memory modules of Concert (the garbage collection

[9] Multilisp has also been implemented on a 128-processor Butterfly machine [8, 46]. It has exhibited speedups beyond those possible on the eight-processor Concert machine, but, due to various differences between Concert and the Butterfly, it needs further tuning. Therefore, it is not further discussed here.

algorithm is discussed in Section 4.4). Thus each processor makes mainly local accesses, punctuated by an occasional access to the shared heap for the next MCODE instruction or some other Multilisp object. When Multilisp is running on 8 processors, these heap accesses consume about 30 percent of the capacity of the shared Multibus. A large fraction of these accesses are to read-only objects such as code. Therefore, the number of global accesses can probably be reduced significantly through caching strategies. Such a reduction would be significant on a system with a lower ratio of communication bandwidth to processing speed than Concert; however, the exact magnitude of the savings obtainable through caching is not known at this time.

At the MCODE level, a program in execution may be viewed as a collection of *tasks* sharing pointers into a common garbage collected heap. The state of each task can be characterized by means of three pointers into the heap: the *program pointer*, the *stack pointer*, and the *environment pointer*. MCODE is a stack-oriented language: most operators obtain their operands by popping them off the current task's stack and push their results back onto the stack. Each task's stack is conceptually allocated in the heap, although for efficiency reasons the top several elements of each active task's stack are cached in the local memory of the processor executing that task. Nevertheless, a task's stack can be dumped completely into the heap when the task is suspended or transferred from one processor to another. This capability also makes possible the construction of full continuations [29] that could be used for the Lisp `catch/throw` constructs, but the interaction of full continuations with futures is troublesome. This issue is explored in [28], where an alternative way to implement `catch` and `throw` is given.

Tasks are created using the MCODE *FUTURE* instruction, which specifies the program pointer for the newly created task. The environment pointer is inherited from the parent task, leading to cactus-like environment structures in which variables may be shared between tasks. The stack of a new task always has just a single item on it—a reference to the future whose value the task has been created to compute.

Each datum manipulated by an MCODE instruction is a tagged datum with two principal fields: a *type* field and a *value* field.[10] We refer to such a tagged datum as a *Lisp value*. The type field indicates the type of the Lisp value. Many MCODE operators (e.g., addition) require their operands to have certain types.

There are two broad categories of types:

(1) *Pointer types*. The value field of a pointer-type Lisp value contains the address of the location in the heap of the sequence of Lisp values that compose the *contents* of the original Lisp value.[11]

[10] There are also a couple of auxiliary bits in each tagged datum. These are used by the garbage collector discussed in Section 4.4.2 and for implementing the synchronization operations discussed in Section 4.2; however, the existence of these bits is not explicitly visible in the semantics of MCODE.

[11] Concert Multilisp also includes a category of pointer types whose value field is interpreted as the address of a sequence of data that are not represented in the tagged-datum format. Character strings, for example, are members of this category.

(2) *Nonpointer types.* The value field of a nonpointer-type Lisp value is interpreted as a bit string in some suitable way. Integers are an example of a nonpointer type.

This "typed-pointer" style is fairly common among Lisp implementations.

The remainder of this paper details several of the implementation techniques that have been developed for Multilisp. They cover such areas as synchronization, task management, and garbage collection of a heap shared by many processors. These techniques have been sufficient to produce a working multiprocessor implementation of Multilisp powerful enough to run programs such as the Multilisp compiler itself.

4.2 Synchronization

The Multilisp programmer may wish to express synchronization requirements of two different types: *precedence* constraints and *mutual exclusion* constraints. Precedence constraints typically arise out of fork/join or producer/consumer situations, and may be enforced using the `pcall` and `future` mechanisms. The need for mutual exclusion arises from the presence of side effects in Multilisp: two threads of execution not ordered by any explicit data dependency may nevertheless share some common variable to which access must be controlled.

Tasks do not have Multilisp-accessible names and cannot send messages to each other. Instead, synchronization is accomplished via side effects to shared objects in the heap. MCODE includes two mechanisms for this. One mechanism provides special support for futures and is described in Section 4.2.3. The other is a pair of special atomic operations, `replace` and `replace-if-eq`, which can be applied to any location in the heap. `replace(`L`, `V`)` reads the current value of the location L and then replaces it with the value V, all in one atomic operation. The old value from location L is returned as the value of `replace`. `replace` is analogous to the conventional test-and-set operation, but returns a whole Lisp value, rather than just a single bit. `replace-if-eq(`L`, `V`, `X`)` is like `replace(`L`, `V`)` except that the replacement is only performed if the current value of the location is `eq` (in the Lisp sense) to X. `replace-if-eq` returns a non-`nil` value to indicate that the replacement *was* performed; a value of `nil` indicates that it was not. As discussed elsewhere [24, 25], `replace` by itself is sufficient to enforce both precedence and mutual exclusion constraints without any need for busy-waiting, but `replace-if-eq` allows more efficient and aesthetic implementations.[12]

Atomic operations, such as `replace` and `replace-if-eq`, that operate on only a single location, are more attractive building blocks than atomic operations that operate on two or more locations. In a system where different data may be stored at different processors, an atomic operation that involves more than one

[12] It is interesting to compare these operations to the atomic operations that have been proposed for the NYU Ultracomputer [21, 22, 48]. In Ultracomputer terminology, `replace` would be called "fetch and store." `replace-if-eq` has no direct equivalent in the Ultracomputer repertoire, but its inclusion in the Ultracomputer would be straightforward. `replace-if-eq` is also quite similar to the MOVCSF and MOVCSS instructions of the S-1 [12] and the `%store-conditional` subprimitive of the Symbolics 3600 Lisp machine [55]. `replace` is analogous to the S-1's RMW instruction.

storage location requires rather tight coordination between the affected processors, while an operation that involves only a single location can be handled by just a single processor. Of course, if there are multiple cached copies of the object on which the atomic operation is to be performed, then some coordination is still necessary, but it can be handled by removing all but one of the copies and then performing the atomic operation on the sole remaining copy.[13]

`replace` and `replace-if-eq` are not actually single primitives, but rather *families* of primitive operations that apply to different kinds of locations. For example, the `replace-if-eq` family includes `replace-car-eq` and `replace-cdr-eq`, which operate on the `car` and `cdr`, respectively, of the conscell supplied as their first argument. The members of the `replace` and `replace-if-eq` families are very low-level primitives that form part of the Multilisp implementation but need not be used by the average programmer. Higher level synchronization facilities, implemented in terms of `replace` and `replace-if-eq`, are made available for everyday use. The following sections show some useful synchronization constructs and their implementation using `replace` and `replace-if-eq`. These sections are intended to illustrate the implementation of synchronization constructs in Multilisp, not to suggest that everyday programming would be conducted at such a low level.

4.2.1 *Examples Using* `replace-if-eq`. A simple illustration of the power of the `replace-if-eq` primitive is afforded by the procedure `cons-onto-cdr` (below) to cons a value `val` onto the `cdr` of an argument `cell`. If `cell` is a list, `(cons-onto-cdr val cell)` inserts the value `val` between the first and second elements of the list. The `replace-cdr-eq` operator performs a `replace-if-eq` operation on the `cdr` of `cell`, ensuring, if there are many simultaneous calls to `cons-onto-cdr` specifying the same cell, that all the associated values will be inserted into the list in some order and none of the values will be lost.

```
(defun cons-onto-cdr (val cell)
  (let ((old-cdr (cdr cell)))
    (if (replace-cdr-eq cell (cons val old-cdr) old-cdr)
        cell
        (cons-onto-cdr val cell))))
```

A similar approach can be used in extending the implementation of the standard Lisp procedure `nconc` to work in a multiprocessor environment. `(nconc x y)` locates the last cell in the list `x` and modifies the `cdr` of that cell (which should contain `nil`) to contain `y`. It would be desirable to view `nconc` as atomic; thus the result of many concurrent `nconc` operations to the same list `x` should be a list that includes all lists that were `nconc`'ed to `x`, appended in some order. This goal is satisfied by the definition given in Algorithm 2 in Figure 2. The procedure `nconc1` does all the work; `nconc` is simply a wrapper ensuring that the value returned conforms to the standard Lisp definition. This procedure is used, among other places, in the maintenance of property lists in Multilisp.

[13] Concert Multilisp does not currently keep multiple cached copies of any object, but the "reference tree" protocol of [27] shows one way that such caching could be implemented.

```
(defun nconc (x y)
  (if (null x)
      y
      (progn (nconc1 x y)
             x)))

(defun nconc1 (x y)
  (if (replace-cdr-eq x y nil)
      nil
      (conc1 (cdr x) y)))
```

Fig. 2. Algorithm 2: Destructive append procedure.

Although any individual call to `cons-onto-cdr` or `nconc` may attempt its operation several times before succeeding, busy-waiting in the usual sense is not occurring. In `cons-onto-cdr`, repetition by a task T of the `replace-cdr-eq` only occurs if another task performs its replacement in between T's obtaining the `cdr` of `cell` and T's performing the replacement. If there are a finite number n of simultaneous calls, the number of attempts for any call will be at most n, and the total number of attempts will be at most $n(n + 1)/2$. This would be a bottleneck if an algorithm were heavily oriented around operations on a single list, but in our applications this has not occurred.[14]

In the case of `nconc`, no replacement is ever attempted twice; if a replacement fails, there must be more to the list `x`, and the algorithm proceeds to the `cdr` of `x`. Thus the amount of work for any individual call to `nconc` is proportional to the number of elements that appear before the first element of `y` in the result, just as in sequential dialects of Lisp.

4.2.2 *Mutual Exclusion.* A simple method of achieving mutual exclusion is by means of *semaphores* [14]. We restrict ourselves to "binary" semaphores; thus at any given time, a semaphore S may be *free*, indicating that no task is currently using the resource controlled by S, or *busy* with a particular task T, meaning that T is using the resource. Two operations may be performed on a semaphore S by a task T:

- `(wait` S`)` returns, having made S busy with T. This can occur immediately if S was free; else the return may be delayed.
- `(signal` S`)`, executed when S is busy with T, causes S to become free (which may in turn lead to its becoming busy with another task).

Algorithm 3, in Figure 3, gives an implementation of semaphores using `replace` and `replace-if-eq`.[15] Since the `wait` operation may need to suspend tasks that call it until the semaphore is free, Algorithm 3 uses a pair of

[14] If the frequency of operations on a given list is too high, both `cons-onto-cdr` and `nconc` may be subject to starvation, where an infinite stream of calls from one group of processes prevents another process from ever completing its own request.

[15] The use of `replace-if-eq` in Algorithm 3 is not logically necessary. In [25] an implementation of semaphores is given that does just as well as Algorithm 3 at being fair and avoiding busy-waiting and that uses `replace` as its only synchronization primitive. This implementation is, however, considerably more complex.

```
(defun make-semaphore ( )
  (cons '*semaphore* (cons '*free* nil)))

(defun wait (S)
  (suspend (lambda (suspension)
             (nconc S (cons suspension nil))
             (activate-next S)
             (quit))))

(defun signal (S)
  (replace-car (cdr S) '*free*)
  (activate-next S))

(defun activate-next (S)
  (if (not (null (cdr (cdr S))))
      (if (replace-car-eq (cdr S) '*busy* '*free*)
          (if (not (null (cdr (cdr S))))
              (progn (replace-cdr S (cdr (cdr S)))
                     (activate (car (cdr S))))
              (progn (replace-car (cdr S) '*free*)
                     (activate-next S))))))
```

Fig. 3. Algorithm 3: Implementation of semaphores.

primitive operations for dealing with suspended tasks:

- `(suspend` *F*`)` creates a suspension *S* of the current task *T*, then creates a new task *T'*, in which *F* is called with *S* as its argument. When *S* is activated, *T* will return from the `suspend` call.
- `(activate` *S*`)` restarts a task from the suspension *S*. The current task continues execution following the call to `activate`. The restarted task resumes execution by returning from its call to `suspend`.

Algorithm 3 also uses the primitive operation `(quit)`, which terminates the task that executes it.

In Algorithm 3, a free semaphore is represented as a list

```
(*semaphore* *free* ...)
```

that is, an object *S* such that `(car (cdr` *S*`))` is the symbol `*free*`. Any items in the list following the symbol `*free*` are suspensions of tasks that are waiting for the semaphore. The procedure `activate-next` activates the first suspended task in this list and updates the semaphore carefully, so that multiple concurrent calls to `activate-next` on the same semaphore will result in the activation of only one task. `activate-next` converts the free semaphore into a busy semaphore, represented as a list

```
(*semaphore* ...)
```

where the ellipsis once again stands for a list of suspensions, the first of which is the suspension that was activated by `activate-next`.

The procedure `signal` converts a busy semaphore *S* back into a free one, by replacing `(cdr (cdr` *S*`))`, which contained the suspension of the task currently using the semaphore, with the symbol `*free*`. `activate-next` is then called to activate the next task that should proceed, if any.

The procedure wait creates a suspension of the task that calls it and then uses the procedure nconc, defined previously, to put that suspension at the end of the list of suspensions waiting on the semaphore. activate-next is then called, so that if the semaphore is free, it will be made busy and a waiting task reactivated.

Desirable properties in a semaphore implementation are *efficiency* (e.g., no busy-waiting) and *fairness* (a pending wait should not be forever starved out by a stream of subsequent wait requests). Algorithm 3 avoids busy-waiting, but fairness is a tricky property, especially in a language such as Multilisp that does not promise fair scheduling among all tasks. The reader will recall that our implementation of nconc may cause starvation, calling into question the fairness of a wait primitive that uses nonc. wait is, however, fair in the sense that requests will be served in the order in which their nconc requests complete. Furthermore, the effect of this loophole in the fairness of semaphores is hard to distinguish from the effects of unfairness in the Concert Multilisp scheduler itself, discussed in Section 4.3.

Algorithm 3 can be optimized in a couple of respects. First, the overhead of creating a suspension when performing a wait operation on a free semaphore can be avoided in most cases by a preliminary check, implemented in the procedure fast-wait shown below, which immediately returns (with the value nil) if applied to a free semaphore with no suspended tasks:

```
(defun fast-wait (S)
  (if (and (null (cdr(cdr S)))
           (replace-car-eq (cdr S)'*busy* '*free*))
      nil
      (wait S)))
```

Second, the expense of nconc operations on long lists of suspended tasks can be reduced by performing the nconc operations on a separate pointer in the semaphore representation that points to the last conscell in the list of waiting tasks (this implementation is not shown, in the interest of brevity). This extra pointer may occasionally not point to the very last cell in the list (because nconc's running concurrently may have added some conscells), but this is not a serious problem, since nconc will always find the end of the list and append additional suspensions there.[16]

4.2.3 *Implementation of Futures.* replace and replace-if-eq are clumsy to use for implementing futures, so MCODE makes special provisions for futures. MCODE represents a future (the kind of object returned by an expression (future *X*)) as a special type of object containing a Lisp value, a task queue, a *determined flag*, and a lock. Initially the determined flag is false and the task queue is empty.

When an MCODE instruction needs to examine the type or value of an operand, it first checks the type field to see if the operand is a future. If so, it checks the future's determined flag. If this flag has the value true, then the value component of the future is fetched and used as the operand to the instruction (of course, it must be checked, recursively, to see if *it* is a future!). Otherwise, the

[16] This extra pointer is a "hint" of the sort discussed in [41].

future is not yet determined and execution of the instruction cannot proceed. To avoid busy-waiting in this case, the task is suspended and added to the future's queue of waiting tasks, all of which will be activated when the future becomes determined.[17] This query, suspension, and enqueueing must all occur as an atomic operation, lest a task "fall through the cracks" by querying a future simultaneously with the future's becoming determined. Although this is not a single-*location* operation like `replace` and `replace-if-eq`, it is a single-*object* operation and therefore has most of the same advantages.

Once a future is determined, the data structure used to represent the future becomes vestigial: its only function is to serve as an indirect pointer to the actual value, adding a small amount of overhead to each reference. To eliminate this overhead, the Multilisp garbage collector replaces references to determined futures by references to their values as it performs its scan.

4.3 Task Management

The creation and destruction of tasks is dictated largely by `pcall` and `future` forms appearing in a program; however, the definition of Multilisp allows considerable latitude in task scheduling decisions. Task scheduling in Concert Multilisp has two primary goals: preserving the locality of programs and avoiding the creation of excessive numbers of parallel tasks.

A classical difficulty for concurrent architectures occurs when there is too *much* parallelism in the program being executed. A program that unfolds into a very large number of parallel tasks may reach a deadlocked state where every task, to make progress, requires additional storage (e.g., to make yet more tasks), and no more storage is available. This can happen even though a sequential version of the same program requires very little storage. In effect, the sequential version executes the tasks one after another, allowing the same storage pool to be reused. By trying to execute all tasks at the same time, the parallel machine may run out of storage. Ideally, parallel tasks should be created until the processing power of the parallel machine is fully utilized (we may call this *saturation*) and then execution within each task should become sequential.

Concert Multilisp uses an *unfair* scheduling policy to produce this behavior. Each task has two possible states: *active* and *pending*. When a processor evaluates an expression such as `(pcall +` A B`)`, it creates two tasks. The scheduler devotes all its resources to only one task, while the other is relegated to a LIFO pending queue associated with that processor. If the system is saturated, this task will remain pending until the active task has finished, as would occur in sequential execution. But if there are idle processors in the system, one of them can pick up the pending task while the other task is still active. This mechanism prevents the combinatorial explosion of parallelism that is possible if A and B recursively invoke other `pcall`'s. A similar policy is applied to the tasks created by an expression `(future` X`)`. The task newly created to evaluate X is kept active, while the parent task is moved to a pending queue. This somewhat

[17] This implementation is quite similar to the semaphore implementation given above, except that the determination of a future activates all waiting tasks, whereas a `signal` operation on a semaphore activates only one waiting task.

counterintuitive policy produces the same order of execution on a saturated machine as if the `future` operator had been omitted[18] and therefore limits task queue growth to the same magnitude as stack growth in a sequential implementation.

Normally each processor has just one active task.[19] When this task completes, the processor looks at its queue of pending tasks to find one to activate. If this queue is empty, it looks in the pending task queues of other processors to find a task to steal and activate. Thus a task will eventually be executed by the same processor that created it, unless some other processor has run out of tasks.[20] This strategy should help preserve locality of memory references.

This system of active and pending tasks is similar to that of Keller [36, 37]. Its impact on free storage use is dramatic: for many programs that we have written, increasing the number of simultaneously active tasks per processor (in effect, decreasing the degree of unfairness in the scheduler, since a processor's active tasks are scheduled in a round-robin manner) has no effect on the speedup achieved through parallelism, but quickly increases the amount of heap storage required, until the available memory is exceeded. The effect of the unfair scheduler on locality has yet to be quantified.

This unfair scheduling policy is plausible because of Multilisp's current orientation toward obtaining parallelism through relaxation of precedence constraints, rather than through speculative execution of additional eager-beaver operations. In the eager-beaver case, it would be important to prevent the speculative operations from overwhelming the machine, to the exclusion of the main line of the computation. The best scheduling policy for this case is somewhat program dependent, but often it might take the form of specifying a ratio of the effort to be applied to the speculative computation versus the main line, with the understanding that a fair scheduling algorithm would be employed to divide the machine's time according to this ratio. Adding such a mechanism to the current Concert Multilisp scheduler, while preserving the advantages of the current unfair scheduler, presents an interesting design problem that should be explored further.

[18] If an active task must be suspended, for example, while waiting for a future or a semaphore, its processor will find another task to activate. This is a departure from the strict LIFO discipline discussed above and may be viewed as a loophole in the Concert Multilisp implementation of resource usage control via unfairness. On the other hand, in a saturated system, tasks that would be waiting for a future tend to be pending while tasks calculating the values of futures tend to be active. Thus relatively few references to futures actually result in suspension of the referencing tasks (Section 4.5 gives some relevant measurements). As for semaphores, they appear very sparingly in Multilisp programs we have written, so the effect of this loophole has not been noticeable.

[19] The number of active tasks on a processor may exceed one in special situations, such as when tasks are brought to a processor to interact with I/O devices attached at that processor. When a processor has more than one active task, it divides its attention among them in a round-robin manner.

[20] When a processor takes a task off another processor's queue, an argument can be made that it would be best to take the *oldest* task, rather than the newest, as is presently done in conformance with the LIFO discipline mentioned above. It is plausible to suppose that the oldest task is more likely to be the root of a substantial tree of computation, and in general it would seem that locality would be enhanced by moving large quanta of computation between processors, rather than small quanta. This hypothesis deserves an experimental test.

4.4 Heap Management

The heap management algorithms developed for Concert Multilisp require a common memory addressable by all processors. Although such an architecture requires more memory bandwidth as the number of processors increases, some proposed architectures fitting this description, such as the NYU Ultracomputer [21, 48], have very large numbers of processors. In any case, the algorithms furnish a case study in multiprocessor Lisp implementation, and they form the basis for expected extensions that relax the requirement that all memory be addressable by every processor; therefore it is worthwhile to outline them here.

The garbage collection algorithm of Concert Multilisp is based on the copying, incremental garbage collector of Baker [6]. In this garbage collector, the processor interleaves periods of garbage collection activity with periods of computation, but each period of garbage collection is short and performs only a small part of a complete garbage collection pass. There is never a long, system-wide pause for garbage collection.

An alternative organization for garbage collection on a multiprocessor is to dedicate certain processors exclusively to garbage collection and others exclusively to computation [15]. Concert Multilisp is not organized this way since it offers less flexibility in adjusting the fraction of system resources allocated to garbage collection.

4.4.1 *The Baker Garbage Collection Algorithm.* The simple Baker garbage collection algorithm divides the heap into two *semispaces*: *oldspace* and *newspace.* During each garbage collection period, the processor works on relocating accessible objects from oldspace to newspace. When all accessible objects have been relocated into newspace, oldspace can be discarded. Then the semispace designations can be swapped: the old newspace becomes the new oldspace, and copying continues in the opposite direction as before.

Figure 4 illustrates the organization of memory in more detail. Newspace is bounded by the addresses *BOTTOM* and *TOP* and is divided into three regions by the two pointers *MOVED* and *NEW.* We name these regions as follows:

- The region between *BOTTOM* and *MOVED* is the *black* region.
- The region between *MOVED* and *NEW* is the *gray* region.
- The region between *NEW* and *TOP* is the *empty* region.

The garbage collection algorithm preserves the following invariants:

(1) The empty region is empty (contains no objects).
(2) The black region contains pointers only into newspace; it contains no pointers into oldspace.
(3) Pointers into the heap from the exterior always point into newspace.

Whenever the processor needs to allocate some more storage (either for the user program or for relocation of an object from oldspace), it obtains that storage from the empty region, incrementing *NEW* as needed.

When the processor is in garbage collection mode, it works on enlarging the size of the black region by incrementing *MOVED.* The invariant that the black

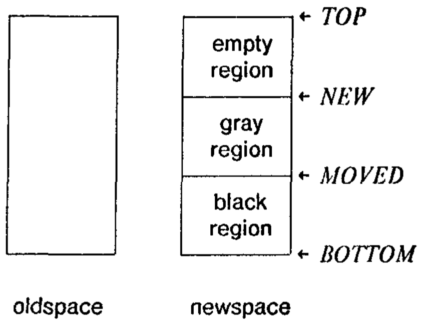


Fig. 4. Memory organization for the Baker garbage collection algorithm.

region contains no pointers into oldspace must be preserved. Hence, if the area added to the black region contains any pointers into oldspace, they must be converted to pointers into newspace. In general, this involves two kinds of operations: copying objects from oldspace to newspace (the *moving* operation), and updating pointers so that they point to the newspace copy of an object (the *updating* operation).

Given a pointer into oldspace that must be updated, the object that it points to may or may not already have been relocated into newspace. If it has been relocated, only the updating operation should be performed; otherwise the sharing relationships present in the heap would be destroyed. Therefore, when an object is relocated, its old copy in oldspace is marked with a *forwarding address*, the location of the new copy in newspace. Thus every object is relocated at most once.

Objects relocated from oldspace become part of the gray region, since *NEW* is incremented to make room for them. Eventually, however, all accessible objects in oldspace will have been relocated, and increases in the size of the black region will not lead to further increases in the size of the gray region. Thus *MOVED* eventually catches up to *NEW* and the gray region is eliminated.[21] At any point after this, the semispace designations may be swapped.[22]

Invariant (3) concerns pointers into the heap from the exterior (which we call *external pointers*), such as processor registers and the stack. Given this invariant, any external pointers may be stored into newspace, even into the black region, without violating the other invariants. Preserving this invariant requires special care (1) when fetching data from the heap into an external location, such as a processor register, and (2) when swapping semispaces. In case (1), we know that the location being fetched from is a location in newspace (we can fetch data directly from only those locations that are pointed to by an external pointer, and all external pointers point into newspace), but we do not know whether it is in the black region or the gray region. If the location is in the gray region, it might contain a pointer into oldspace, and our invariant would be violated by delivering

[21] If *NEW* reaches *TOP* before *MOVED* reaches *NEW*, then there is insufficient heap storage and the algorithm fails.

[22] It is not necessary to swap semispaces immediately; it is permissible to wait for the empty region to be used up before swapping semispaces and resuming garbage-collection activity.

this pointer to the exterior. So the pointer must first be updated into a newspace pointer, if necessary by moving the object it points to, and then delivered to the exterior.

To preserve invariant (3) when swapping semispaces, *all* external pointers must be converted to newspace pointers as part of the semispace-swapping operation. The semispace swap is accomplished by setting *BOTTOM*, *MOVED*, and *NEW* to the bottom of the new newspace (the old oldspace), and setting *TOP* to the top of the new newspace. Then all the external pointers are converted to pointers into the new newspace by relocating the objects they point to (this will cause *NEW* to be incremented). Processing can then continue, interspersed with periods of additional garbage collection activity.

The semispace swap includes the potentially time-consuming operation of converting all pointers in the stack, if the stack is not located in the heap. This delay could compromise the real-time properties of the garbage collection algorithm, but fortunately in Concert Multilisp, the majority of a task's stack is kept in the heap, as discussed in Section 4.1. Only a small top-of-stack cache, outside of the heap, needs to be scanned when swapping semispaces.

4.4.2 *The Concert Multilisp Garbage Collector.* In Concert Multilisp, every processor has its own oldspace and newspace in its local memory. Thus each processor has its own private newspace in which to create objects, eliminating contention between processors for allocation from the heap and increasing the fraction of accesses that go to local memory. The aggregate of the individual processors' oldspaces can be thought of as the system's oldspace, and the individual processors' newspaces can be thought of collectively as the system's newspace. Each processor's newspace is divided, as before, into empty, gray, and black regions, obeying invariants that may be stated as

(1) The empty region is empty.
(2) The black region contains pointers only into newspace—either the newspace of the same processor, or that of any other processor.
(3) External pointers all point into the newspace of some processor.

As before, gray regions may contain pointers to any newspace or oldspace.

During its garbage collection periods, each processor P increments its *MOVED* pointer to enlarge its black region. If a pointer to an object X in oldspace is encountered during this process, it is relocated. If X already has a forwarding address in the newspace of some processor, that forwarding address is used. This preserves the sharing relationships in the heap. If X does not have a forwarding address, then X is relocated into the newspace of processor P (no matter which oldspace X was in). This new location becomes the forwarding address of X. In a multiprocessor environment, this portion of the algorithm has some synchronization requirements, whose discussion we postpone for a moment.

The semispace swap is an activity that must be coordinated among all the processors. As long as any processor still has a gray region in its newspace, no processor can discard its oldspace, because the gray region could contain pointers into any processor's oldspace. As soon as all gray regions are gone, the processors may swap semispaces, but all of them must perform the swap at the same time;

otherwise, one processor could swap semispaces early and begin to build up a black region in its new newspace. This region could contain pointers into other processors' newspaces. Then if the other processors swapped semispaces later, these pointers would become pointers into oldspace, violating the invariant for black regions. Thus this multiprocessor garbage collection algorithm contains one global synchronization point: at some point after all gray regions have been eliminated, and before any processor's empty region runs out, all processors must suspend processing activity, swap spaces, and then resume processing.

The algorithm has some additional, nonglobal synchronization requirements, alluded to above, which occur during the moving and updating operations associated with the relocation of pointers in gray regions, which must be performed atomically. Only one processor (the processor in whose newspace the pointer appears) can try to update a pointer in expanding its black region, but any processor might at the same moment try to read the pointer, which will also require it to be updated. Thus several processors can simultaneously attempt the updating operation on the same pointer. Alternatively, several processors may simultaneously access different pointers to the same object and hence collide trying the moving operation.

Both types of collision are quite unlikely (experience with Concert Multilisp running on eight processors shows less than one collision per second). Therefore, any locking mechanism that assures the necessary atomicity will do—contention for locks is not heavy. Concert Multilisp associates a lock bit with each pointer, to handle the updating operation, and a lock bit with each object in the oldspace, for the moving operation.

4.4.3 *Discussion.* This garbage collection algorithm has been implemented and functions correctly. Three aspects of the algorithm invite further comment: (1) its impact on locality of reference, (2) its reliance on shared memory, and (3) the efficiency of its use of memory.

This garbage collection algorithm has the intriguing potential of dynamically enhancing the locality of reference in the heap. Whenever an object X is relocated, it is relocated into the newspace of a processor that has at least one reference to X. Thus if a processor P creates a data structure and later gives a pointer to that structure to processor Q, without keeping a copy of the pointer for itself, the entire data structure will eventually migrate to the newspace of Q, where it is presumably more likely to be used. The significance of this effect, however, is as yet unmeasured.

The algorithm has been stated as an algorithm for a shared-memory multiprocessor, but its only real requirement is for a common address space, so that every processor can name every memory cell, and so that a given address names the same cell no matter which processor uses the address. If this requirement is satisfied, the algorithm will function correctly, although its performance will be poor if access to remote memory cells is too slow. The magnitude of this deterioration depends on the locality of reference in the system. Probably the situation can be improved by changing the garbage collection algorithm to "batch" its remote accesses (at the cost of some additional complexity) so that many remote accesses can be active simultaneously, making their latency less significant than the bandwidth available for them.

The efficiency of memory use by incremental garbage collectors has been studied by Baker [6]. The only new wrinkle in our multiprocessor algorithm is that the oldspace and newspace are divided up into pieces that are dedicated to different processors. This raises the possibility of failure because one processor has run out of newspace for new allocation, even though there may be plenty of room left in other processors' newspaces. This effect has (unfortunately) been observed frequently on Concert Multilisp. It can be ameliorated by allocating plenty of extra storage to each processor, but there is a better solution. A processor's newspace (or oldspace) need not be one contiguous block. Instead, memory can be organized into rather large parcels (say 64K or 128K bytes each). If a processor runs out of newspace before the system is ready for a semispace swap, the processor can request an extension parcel from a pool that is set aside for that purpose, and carry on. (Alternatively, as suggested by one of the referees, the extension parcel could be obtained out of the newspace of another processor.) This parcel may be in a less favorable position for access by the processor, but at least the system can continue to operate. A scheme such as this is only a minor extension of the basic garbage collection algorithm, and dynamically compensates for differences in the memory requirements of different processors.

4.5 Performance of Concert Multilisp

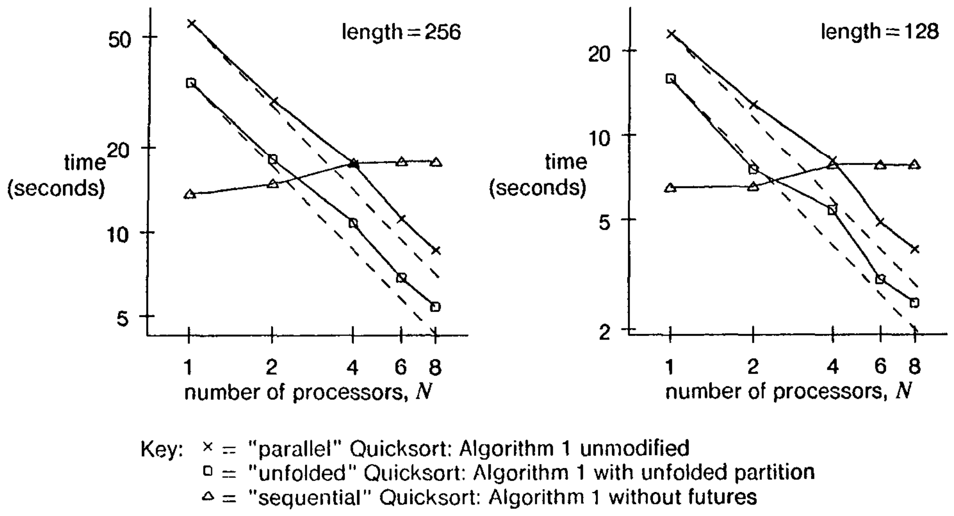
The current Concert Multilisp implementation, running on one processor, performs at roughly the same level as interpreted Franz Lisp code [16] running on the same processor. Given a suitably parallel Multilisp program, this performance has been observed to improve by very nearly a factor of n when n processors are used, for $n \leq 8$.

The rate of execution of MCODE instructions depends on two factors: the instruction mix and how much time the processor is currently spending on garbage collection. On the MC68000 processor running at an 8-MHz clock rate, the execution rate is approximately 2000 to 6000 MCODE instructions per CPU second. The lower rates occur in programs that use futures extensively—the instructions that implement futures take longer individually, and they also allocate more space in the heap, increasing the proportion of time devoted to garbage collection.

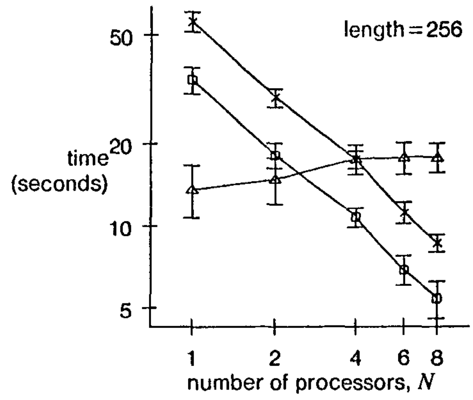
The Quicksort program of Algorithm 1 was run on Concert Multilisp for different sizes of randomly ordered input lists and different numbers of processors.[23] Also measured were a "sequential" program obtained from Algorithm 1 by removing all uses of future, and an "unfolded" version of Algorithm 1 obtained by a two-way unfolding of `partition` such that each recursive call to `partition` processes two elements of its argument list instead of one. The statistics measured were (1) total execution time, (2) the number of futures created, and (3) the number of futures that were waited for by one or more tasks before being determined.

These measurements are subject to variability from two sources. First, semispace swaps and the ensuing garbage collection activity occur at arbitrary

[23] The procedures `bundle-parts`, `right-part`, and `left-part` of Algorithm 1 were open-coded in the version of the algorithm used to obtain these figures, reducing somewhat the number of procedure calls required.



(a)



(b)

Fig. 5. Performance of Quicksort on Concert Multilisp. (a) Execution times. The dashed lines are the curves of linear speedup. (b) Execution times with standard deviations.

moments with respect to program execution, causing some executions of the same program to take longer than others. Second, the performance of the Quicksort algorithm itself depends somewhat on the ordering of the particular list being sorted. Due to the variability from these sources, each sort was performed four times on each of ten different randomly generated lists of each length. The graphs in Figure 5 show the mean execution times obtained from these measurements for lists of length 128 and 256. For lists shorter than 128 elements, the standard deviation of the measurements is large enough to cast considerable doubt on the shape of the execution time curves, but for lists of 256 elements the standard deviations are modest, as shown in Figure 5b.

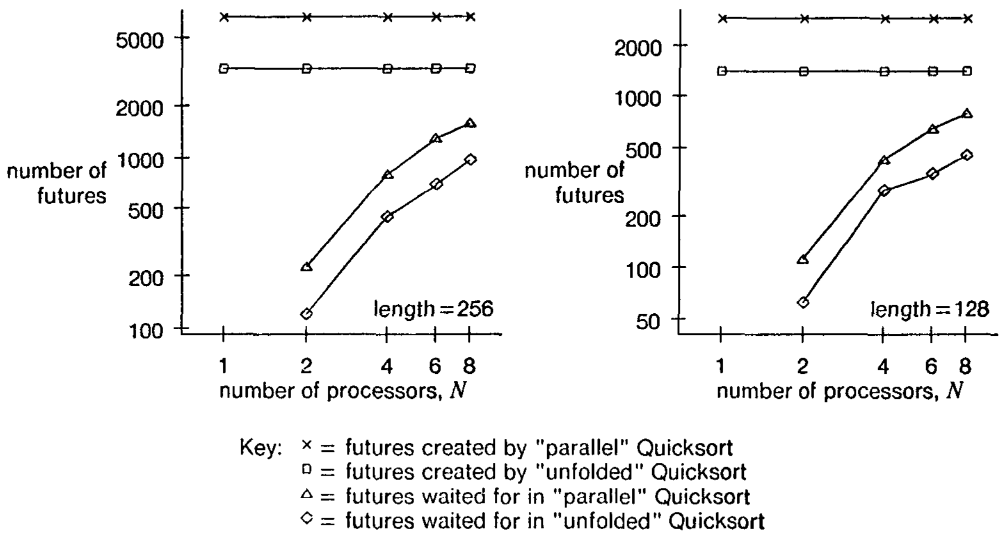


Fig. 6. Future creation and access statistics for Multilisp Quicksorts. The number of futures waited for is zero when only one processor is used; this point is not shown on these log-log plots.

The curves have noticeable kinks at $N = 4$ processors. These are probably due to variations in the amount of memory available for heap storage in the different experiments. The 1-, 2-, and 4-processor measurements use a single region of 1.5 megabytes of memory, divided among the participating processors. Thus the amount of memory allocated to each individual processor falls as more processors are added. Since the sequential Quicksort runs on only one processor, this tends to increase the amount of garbage collection required, causing some increase in average execution time as the number of processors increases. The parallel algorithms, however, are able to use all of the available memory and therefore do not suffer from this effect.

Beyond four processors, the amount of memory per processor remains constant at 3/8 megabyte, and therefore the total amount of memory available for heap storage increases. This does not benefit the sequential sort, whose execution time curve can be seen to level off, but appears to benefit the parallel sorts, whose curves can be seen to resume a more rapid descent.

Figure 6 gives some statistics about the use of futures by the parallel and unfolded Quicksort programs: the average number of futures created and the average number of futures that had one or more tasks waiting for their value at the time they were determined. The number of futures created by each program is independent of the number of processors used and depends only on the input list. Since the only difference between the sequential and parallel Quicksorts is the insertion of futures, the overhead per future can be calculated from the execution statistics on one processor as

$$\frac{\text{(parallel running time)} - \text{(sequential running time)}}{\text{number of futures created by the parallel algorithm}}.$$

This number is about 6.5 milliseconds. For comparison, the cost of a procedure call is approximately 1.3 to 1.6 milliseconds.

The number of futures waited for is much less than the total number created, especially for small numbers of processors. This shows the effect of Concert Multilisp's LIFO scheduler: since tasks that are created to determine futures have priority over the tasks that use those futures, most futures already have a value before their first use. In fact, other experiments suggest that most of the futures that are never waited for correspond to tasks that complete before their parent tasks ever leave pending status.

The Quicksort programs use futures for very fine-grained tasks; in fact, no other Multilisp program that has been written shows a larger ratio between execution time on one processor with and without futures. Programs using futures with coarser granularity suffer much less performance degradation; for example, a Quicksort using an ordering criterion more complex than simple integer comparison suffers proportionately less of a penalty for the use of futures. Nevertheless, it would be desirable to be able to use futures efficiently even at the level of granularity of our parallel Quicksort program. Whenever Concert Multilisp's LIFO scheduler defers a parent task until the completion of a child task, the expense of task creation has been wasted; the child task might as well have executed as a subroutine in the parent task. This appears to be true in a large fraction of all task creations; thus an implementation that selectively ignores certain uses of `future` could dramatically reduce the cost of programming with futures, without reducing the parallelism available. This may be one approach to making futures useful at finer levels of granularity. The performance of the unfolded Quicksort illustrates the performance improvements obtained by one particular strategy of selectively eliminating certain futures.

Although it may be possible to improve the performance of futures relative to other operations, Concert Multilisp remains uniformly slow in absolute terms. This slowness is due to the predominance of operations that are not particulary efficient on the MC68000:

- Many field extractions and insertions are required for examination and construction of tagged data.
- Every access to a datum in the heap entails checking bits in the datum to see if it is a pointer into oldspace or is locked.
- All stack pops or pushes require a check against the bounds of the current task's top-of-stack cache.
- Interpretive overhead results from the fact that the MC68000 does not directly execute MCODE; however, compilation of Multilisp into MC68000 code is unattractive, due to the length of the code that would have to be generated for the operations listed above.

It is reasonable to suppose that these performance figures could be uniformly improved by one to two orders of magnitude with a suitably designed processor, without invoking any aggressive hardware technology. This would, of course, increase the load on the system used for communication between the processors and the heap. However, a "multiprogramming" design, in which each processor could keep several (e.g., three to ten) tasks active simultaneously, could go a long way toward masking the latency of the interconnect medium, by ensuring that a processor still has work to do, even if some of its tasks are blocked awaiting the completion of remote accesses.[24] Thus the bandwidth of the interconnect medium may take on more significance than its latency, allowing some interesting new memory system designs. The development of hardware architectures for Multilisp-like languages promises to be a rich area of research; however, before we embark seriously on it we need to know more about the properties of the programs to be run.

5. CONCLUSION AND DISCUSSION

A strong motivation for the present work is the need for more actual experience with techniques for programming parallel machines. If the Multilisp project progresses as expected, its most important results will be

- Empirical validation of programming language ideas, such as `pcall` and `future`, which have been around for awhile, but which have not been used on an actual parallel machine. In the course of writing programs in Multilisp, we will be testing, fine-tuning, and quite likely adding to this set of ideas.
- Development of a body of working, parallel programs that can be used as benchmarks for future research into architectures for parallel computation.
- Exploration of solutions to various problems that lie between the programming language and the system level. Exception and error handling [28], debugging, and the previously discussed issues of fairness in scheduling are three examples of such problems. The parallel execution environment, and in particular the `future` construct of Multilisp, require standard solutions to these problems to be reexamined.
- Development of implementation algorithms, such as those discussed in Section 4, for Lisp-like langauges on parallel machines.

Many of the design goals of Multilisp are the same as those of any other general-purpose high-level language: applicability to a wide range of problems, support for a good software engineering discipline, and independence from the details of the underlying implementation. In addition, Multilisp has the goal of making it easy to write programs with large enough amounts of parallelism to be able to utilize large multiprocessors. Key features of Multilisp that work toward these goals are the use of lexical scoping, the status of procedures as first-class data objects, the support of a garbage collected heap, and the `pcall` and `future` constructs for introducing parallelism.

On the implementation level, this paper presents schemes for synchronization, task scheduling, and garbage collection in a multiprocessor environment. These algorithms, notably the Concert Multilisp garbage collector, should be widely applicable to other parallel Lisp dialects. Some of the techniques, such as the use of unfair scheduling to help control resource usage, also raise intriguing new semantic questions for language developers to consider.

[24] This approach is used by the Denelcor HEP-1 processor [51].

ACKNOWLEDGMENTS

The current state of progress of this work could only have been possible with the help of many individuals, notably Juan Loaiza, who has performed a great deal of the Multilisp implementation work, and has also contributed to the conceptual development of Multilisp. In particular, he should be credited with developing the ideas behind the semaphore implementation described in this paper. The Concert multiprocessor, on which the implementation is being carried out, exists as the result of a team effort, to which Randy Osborne, Tom Sterling, Tom Anderson, Sharon Gray, Dan Nussbaum, Peter Nuth, and Doug Robinow have all made outstanding contributions. Thanks are due also to Don Allen, Curtis Scott, and Jim Miller for their encouragement and their debugging efforts in implementing the Butterfly machine version of Multilisp. Thoughtful comments by Richard Zippel, Juan Loaiza, and the referees have materially improved the presentation and content of this manuscript, and are gratefully acknowledged.

REFERENCES

1. ABELSON, H., AND SUSSMAN, G. *Structure and Interpretation of Computer Programs.* Massachusetts Institute of Technology Press, Cambridge, Mass., 1984.
2. ACKERMAN, W., AND DENNIS, J. VAL—A Value-Oriented Algorithmic Language. LCS Tech. Rep. TR-218. Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass., 1979.
3. ANDERSON, T. The Design of a Multiprocessor Development System. Tech. Rep. TR-279, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass., Sept. 1982.
4. ARVIND, GOSTELOW, K. G., AND PLOUFFE, W. An Asynchronous Programming Langauge and Computing Machine. Rep. TR114a, University of California, Irvine, 1978.
5. BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978).
6. BAKER, H. Actor Systems for Real-Time Computation. Tech. Rep. TR-197, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Mass., Mar. 1978.
7. BAKER, H., AND HEWITT, C. The Incremental Garbage Collection of Processes. Artificial Intelligence Laboratory Memo 454, Massachusetts Institute of Technology, Cambridge, Mass., Dec. 1977.
8. BBN. Development of a Voice Funnel System: Quarterly Technical Report. BBN Reports 4845 (Jan. 1982) and 5284 (Apr. 1983). Bolt, Beranek, and Newman, Cambridge, Mass.
9. BOUKNIGHT, W. J., ET AL. The Illiac IV system. *Proc. IEEE 60*, 4 (Apr. 1972), 369–388.
10. BROOKS, F. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Reading, Mass., 1975.
11. COHEN, S., ROSNER, R., AND ZIDON, A. Paralisp Simulator (reference manual). Hebrew University Computer Science Dep. Res. Rep. 83-2, Jerusalem, Israel, Jan. 1983.
12. CORRELL, S. S-1 uniprocessor architecture. S-1 Project 1979 Annual Report. Lawrence Livermore National Lab., Livermore, Calif., 1979.
13. DEMINET, J. Experience with multiprocessor algorithms. *IEEE Trans. Comput. C-31*, 4 (Apr. 1982), 278–288.
14. DIJKSTRA, E. W. The structure of the "THE" multiprogramming system. *Commun. ACM 11*, 5 (May 1968).
15. DIJKSTRA, E. W., ET AL. On-the-fly garbage collection: An exercise in cooperation. In *Language Hierarchies and Interfaces (Lecture Notes in Computer Science 46).* Springer Verlag, New York, 1976.
16. FODERARO, J. K., SKLOWER, K., AND LAYER, K. The Franz Lisp Manual. University of California UNIX distribution, 1983.

17. FRIEDMAN, D., AND WISE, D. Aspects of applicative programming for parallel processing. *IEEE Trans. Comput. C-27*, 4 (Apr. 1978), 289–296.
18. FRIEDMAN, D., AND WISE, D. CONS should not evaluate its arguments. In S. Michaelson and R. Milner (Eds.), *Automata, Languages and Programming*, Edinburgh University Press, Edinburgh, 1976, pp. 257–284.
19. GABRIEL, R. P., AND MCCARTHY, J. Queue-based multiprocessing Lisp. Presented at the *ACM Symp. Lisp and Functional Programming* (Austin, Tex., Aug. 1984).
20. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
21. GOTTLIEB, A., ET AL. The NYU ultracomputer—Designing an MIMD shared memory parallel computer. *IEEE Trans. Comput. C-32*, 2 (Feb. 1983), 175–189.
22. GOTTLIEB, A., LUBACHEVSKY, B., AND RUDOLPH, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 164–189.
23. GURD, J., KIRKHAM, C., AND WATSON, I. The Manchester prototype dataflow computer. *Commun. ACM 28*, 1 (Jan. 1985), 34–52.
24. HALSTEAD, R. Architecture of a myriaprocessor. In *IEEE COMPCON Spring 81* (San Francisco, Feb. 1981), 299–302.
25. HALSTEAD, R. Architecture of a myriaprocessor. In J. Solinsky (Ed.), *Advanced Computer Concepts.* La Jolla Institute, La Jolla, Calif., 1981.
26. HALSTEAD, R. Implementation of Multilisp: Lisp on a multiprocessor. In *Proc. ACM Symp. Lisp and Functional Programming* (Austin, Tex., Aug. 1984), 9–17.
27. HALSTEAD, R. Reference Tree Networks: Virtual Machine and Implementation. M.I.T. Laboratory for Computer Science Tech. Rep. TR-222, Cambridge, Mass., July 1979.
28. HALSTEAD, R., AND LOAIZA, J. Exception handling in Multilisp. Presented at the *1985 Int. Conf. Parallel Processing* (St. Charles, Ill., Aug. 1985).
29. HAYNES, C., FRIEDMAN, D., AND WAND, M. Continuations and coroutines. In *Proc. ACM Symp. on Lisp and Functional Programming* (Austin, Tex., Aug. 1984), 293–298.
30. HENDERSON, P., AND MORRIS, J. H. A lazy evaluator. *Proc. 3rd ACM Symposium on Principles of Programming Languages* (1976), 95–103.
31. HEWITT, C. Viewing control structures as patterns of passing messages. Working Paper 92, Artificial Intelligence Laboratory, M.I.T., Cambridge, Mass., Apr. 1976.
32. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978).
33. HOARE, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.
34. ICHBIAH, J. D., ET AL. Preliminary ADA reference manual. *SIGPLAN Not. 14*, 6, Part A (June 1979).
35. IEEE Task P796/D2. Proposed microcomputer system 796 bus standard. *IEEE Comput. 13*, 10 (Oct. 1980), 89–105.
36. KELLER, R. Rediflow multiprocessing. *IEEE COMPCON Spring 84* (San Francisco, Feb. 1984).
37. KELLER, R., AND LIN, F. Simulated performance of a reduction-based multiprocessor. *IEEE Comput. 17*, 7 (July 1984), 70–82.
38. KERNIGHAN, B., AND RITCHIE, D. *The C Programming Langauge.* Prentice-Hall, Englewood Cliffs, N.J., 1978.
39. KNUEVEN, P., HIBBARD, P., AND LEVERETT, B. A language system for a multiprocessor environment. In *Proc. 4th Int. Conf. Design and Implementation of Algorithmic Languages* (Courant Institute of Mathematical Studies, New York, June 1976), 264–274.
40. KUCK, D., MURAOKA, Y., AND CHEN, S.-C. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. Comput. C-21*, 12 (Dec. 1972), 1293–1310.
41. LAMPSON, B., AND SPROULL, R. An open operating system for a single-user machine. In *Proc. 7th Symp. Operating Systems Principles* (Asilomar, Calif., Dec. 1979), 98–105.
42. MARTI, J., AND FITCH, J. The Bath concurrent Lisp machine. *EUROCAM '83(Lecture Notes in Computer Science).* Springer Verlag, New York, 1983.
43. MCCARTHY, J., ET AL. LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Mass., 1965.

44. McGRAW, J., ET AL. SISAL—Streams and Iteration in a Single-Assignment Language. Language Reference Manual (version 1.0), Lawrence Livermore National Lab., Livermore, Calif., July 1983.
45. MOLLER-NIELSEN, P., AND STAUNSTRUP, J. Experiments with a Multiprocessor. Tech. Rep. PB-185, Aarhus University Computer Science Dep., Aarhus, Denmark, Nov. 1984.
46. RETTBERG, R., ET AL. Development of a Voice Funnel System: Design Report. BBN Rep. 4088, Bolt, Beranek, and Newman, Cambridge, Mass., Aug. 1979.
47. RUSSELL, R. M. The CRAY-1 computer system. *Commun. ACM 21*, 1 (Jan. 1978), 63–72.
48. SCHWARTZ, J. Ultracomputers. *ACM Trans. Program. Lang. Syst. 2*, 4 (October. 1980), 484–521.
49. SEITZ, C. L. The cosmic cube. *Commun. ACM 28*, 1 (Jan. 1985), 22–33.
50. SHAPIRO, E. Y. A Subset of Concurrent Prolog and Its Interpreter. Institute for New Generation Computer Technology Tech. Rep. TR-003, Jan. 1983.
51. SMITH, B. J. A pipelined, shared resource MIMD computer. In *Proc. Int. Conf. Parallel Processing*, 1978.
52. STEELE, G. L. Rabbit: A Compiler for Scheme. Tech. Rep. AI-TR-474, Artificial Intelligence Lab., M.I.T., Cambridge, Mass., May 1978.
53. SUGIMOTO, S., ET AL. A multimicroprocessor system for concurrent Lisp. In *Proc. 1983 Int. Conf. Parallel Processing* (June 1983).
54. TURNER, D. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9*, 1 (Jan. 1979), 31–49.
55. WEINREB, D., AND MOON, D. Lisp Machine Manual. Symbolics Corp., Cambridge, Mass., 1984.
56. WENG, K. Stream-Oriented Computation in Recursive Data Flow Schemas. Tech. Memo TM-68, M.I.T. Laboratory for Computer Science, Cambridge, Mass., Oct. 1975.

Received September 1984; revised July 1985; accepted July 1985