10-1-1985

Parallel LISP

Thomas A. Rice
Purdue University

Leah H. Jamieson
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/ecetr

https://docs.lib.purdue.edu/ecetr/537

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
Parallel LISP

Thomas A. Rice
Leah H. Jamieson

TR-EE 85-2
January 1985

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
## Parallel LISP

**Title:** Parallel LISP

**Authors:** Thomas A. Rice and Leah H. Jamieson

**Performing Organization:**
Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

**Contract or Grant Number:** F30602-83-K-0119

**Report Date:** January 1985

**Number of Pages:** 99

**Distribution Statement:** Approved for public release: distribution unlimited

**Key Words:**
- LISP
- Parallel processing
- Symbolic processing
- Parallel programming

**Abstract:**
(over)
Projects in the past few years have looked into the problem of automatic parallelization of the LISP programming language. Since it appears to be feasible to adapt LISP to run on a general parallel computer, an interpreter to simulate the parallel execution of LISP has been developed. The design of the parallel LISP interpreter attempts to balance the conflicting goals of using a general model for the underlying parallel architecture and realizing an efficient (and therefore useful) simulator. The implementation can be used to study the execution characteristics of LISP in a parallel environment. It can also be used to derive information about architectural features which affect the performance of LISP on parallel machines.

This implementation will use a multitasking system and interprocess communication to simulate an MIMD machine. The implementation will include the formation, queuing, distribution, and execution of dataflow frames. Measurements derivable from the simulator include number of processor cycles, processor utilization, memory requirements, and speedup. Realistic LISP application programs can be used with the implementation to examine the feasibility and efficiency of parallel LISP. The interpreter presented here can therefore be used as a tool for studying the parallelism which is available in LISP programs and the architectural features needed to support this parallelism.
PARALLEL LISP

Thomas A. Rice
Leah H. Jamieson

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

TR-EE 85-2
January 1985

This research was supported by the United States Air Force Command, Rome Air Development Center, under contract number F30602-83-K-0119.
1. Abstract

Projects in the past few years have looked into the problem of automatic parallelization of the Lisp programming language. Since it appears to be feasible to adapt Lisp to run on a general parallel computer, an implementation will be developed. This implementation will be as general as possible in order to locate the tradeoffs between implementing Lisp on a general parallel computer versus having an efficient interpreter. This implementation can be used to study the execution characteristics of Lisp in a parallel environment. It can also be used to derive information about architectural features which affect the performance of Lisp on parallel machines.

This implementation will use a multitasking system and interprocess communication to simulate an MIMD machine. The implementation will include the formation, queuing, distribution, and execution of dataflow frames. Realistic Lisp application programs will be used with the implementation to examine the feasibility and efficiency of parallel Lisp. Measurements derivable from the simulator include number of processor cycles, processor utilization, memory requirements, and speedup. These tests will provide two main results. First, they will indicate possibilities for further gains by illustrating the bottlenecks in such a scheme. Second, they will help determine if it is indeed feasible to run Lisp on a parallel machine or if instead the overhead is too high for the application to be profitable. Most likely, some parallelism will be profitable. The simulation will provide information on the extent to which parallelism can be utilized.
2. Introduction

There have been several projects in the past few years that have looked into automatic parallelization of the Lisp [WiH81] programming language [Guz81a][GuN83]. Two different approaches have been taken in these projects. The AHR project at the University of Mexico [Guz81a] was directed toward designing and constructing a dedicated, parallel Lisp machine, whereas the approach of [GuN83] was toward adapting the AHR implementation for a specific SIMD (Single Instruction stream - Multiple Data stream) machine. Both of these projects, however, had common ground that is shared by many parallelizing projects. These projects either propose the design of a new machine or describe the implementation for a particular machine.

The project that is described here takes a different approach to the automatic parallelization of Lisp. The aim is to extract parallelism in a way that can be applied to any general MIMD (Multiple Instruction stream - Multiple Data stream) computer.

A parallel interpreter for the MACLISP [WiH81] dialect of Lisp has been developed. This interpreter assumes a general model of an MIMD architecture. The interpreter can be used to study the execution characteristics of Lisp in a parallel environment. Measurements from executing Lisp programs on the interpreter can be used to derive information about the architectural features which affect the performance of a Lisp program on an MIMD system.
3. Report Overview

This report follows the development and testing sequence of the implementation of a parallel Lisp interpreter. Section 4 describes the concerns of automatically parallelizing Lisp. In this section, four of the main concerns of this parallelization are described and possible approaches are suggested. The first two of these concerns are how to divide the tasks of parsing and executing Lisp programs to obtain an acceptable amount of parallelism without incurring too high a degree of communication and synchronization overhead. These are covered in 4.1 and 4.2.

Sections 4.3 and 4.4 discuss the second two of the four areas of concern. These areas are operation synchronization and information storage. Operation synchronization involves both maintaining data coherence and preventing deadlock. Information storage involves both the ways in which programs and data are stored as well as the ways in which these quantities are accessed by various processing units.

After the preliminary issues have been covered, sections 5 through 8 describe the design and development of several versions of the parallel interpreter. Throughout all of the designs, an MIMD (Multiple Instruction stream - Multiple Data stream) computer is assumed. This is a general model of an asynchronous parallel computer. Two specializations of this general model are used. The first of these, the interconnection network model, consists of a Control Unit (the CU), N Processing Units (the PEs), and an interconnection network that allows the PEs to exchange information. Each of the N PEs consists of a processor and a local memory. (N is typically a power of 2, although this is not a requirement for the interpreter.) It is also assumed that the individual PEs are not designed for I/O. The second model, the shared
memory model, still has a CU and N PEs. However, in place of an interconnection network, a shared memory module is added (the PEs still have local memory, though). For both of these models, the term Processing Unit (or PU) is used to refer to any of the PEs or to the CU. The operation environment is explained in detail in section 5.

The first MIMD model is developed in several steps. This is described in section 6. First, a recursive model is developed. Second, the recursion is converted into a simple serial simulation of multiprocessing. Next, the development of the interpreter into a multitasking simulation of multiprocessing is described. Several interpreter models are involved here. Each of the models differs from its predecessors in one or both of two ways. Either more of the basic MAACLISP Lisp operations are implemented or more parallelism is achieved. Section 7 describes the last interpreter using the first model. This interpreter model adds garbage collection and optimizations to increase efficiency and decrease elapsed execution time (both for the theoretical run time of the interpreter and the actual simulation elapsed time).

The interpreter developed with the second general MIMD model is covered in section 8. This interpreter still has all of the operations and options that were developed for the first general model MIMD interpreter. However, the function of the interconnection network is replaced by use of a shared memory module. Several final operations are implemented with this model, providing a full MAACLISP standard interpreter.

Section 10 of the report deals with the testing of the interpreter with an object recognition program. The results of these tests and recommendations for further improvements are presented in sections 10 and 11.
4. Parallel Interpretation: An Overview

Before beginning the design and development of a parallel interpreter, several concerns must be addressed. These concerns include the internal representation of the language, the ways in which a user program can be converted into this representation, and the ways in which this representation can be executed efficiently in parallel. Since the internal representation is central to this development, the next several sections will describe the conversion and execution of a program in terms of possible representations. The basic premise of these sections is that the internal scheme will be based upon a dataflow implementation. This schema has a better potential for good parallelism as compared to strict compilation since conventional parallelization methods such as loop expansion and automatic vectorization cannot be applied to Lisp. It will be seen that this notation can be adapted to run on a general MIMD (Multiple Instruction stream - Multiple Data stream) computer. Since the primary concern is efficient execution, this will be the first topic.

4.1. Parallel Execution

The basic concept behind automatic parallel execution of Lisp is a conversion of Lisp into a dataflow representation. Each s-expression in Lisp is translated into a set of dataflow nodes, or frames (see [GuN83], for example). Each of these frames consists of several parameters. The main parameter of the frame is the first item of the s-expression. This is typically either a Lisp function or a lambda definition. The case of a defined function will be considered separately. The other parameters consist of the arguments of the function and a pointer to the frame to which the result must be returned. Some of these arguments must be returned from other frames before execution
can begin. To account for this case, a counter is kept in each frame that records the number of arguments that still are awaiting evaluation. This provides the needed synchronization. The parallelism comes about by executing the functions of the frames in the various PEs (as well as in the CU) of a general MIMD machine as well as in the CU. This concept is illustrated by the simple example in Figure 1.

All of the frames are initially contained in the CU, which is responsible for overseeing the operation of the PEs. The CU will be used for distributing tasks (i.e., frame execution) to the PEs and collecting and collating the results. Parallelism is achieved by simultaneously executing frames that have all of their arguments ready.

The frames are divided into two main classes: distributable and nondistributable. These two classes come about from a consideration of the amount of gain that can be obtained by processing frames in parallel weighed against the overhead in setting up the parallel processing. Distributable frames are tasks for which the execution time exceeds the time it takes to transfer a frame to a PE and get back a result from the PE. For these cases, speedup can be obtained by processing the tasks in parallel. An example of such a task could be the Lisp operators QUOTIENT or TIMES (with multiple operands). In addition, special Lisp operators (such as a hardware FFT) could be assigned to special, dedicated PEs. A simple way of dealing with defined functions (as obtained via DEFUN) would be to assign a function to a dedicated PE. Alternatively, these functions could be expanded into a full set of nodes via their definitions. This latter approach provides more flexibility for scheduling and executing parallel nodes and thus has the potential for increased parallelism. This was the scheme used in the implementation.
(plus (times 2 3 (addl 0.5)) (difference 35 2))
Result: 42

Figure 1: Simple Example of Conversion of a LISP Expression to Dataflow Frames
Non-distributable frames involve two types of tasks. The first type is tasks that are so simple that the time to execute them is less than the time needed to transfer to frame to a PE and get a result from the PE. The Lisp operators ADD1 and ATOM fall into this category. The second type are tasks that involve operations that only the CU can perform. The Lisp operators PRINT and TERPRI fall into this category (assuming that only the CU can do I/O).

The CU maintains two queues of frames that are ready to be executed, one for each of the main classes of frames. Each time a PE is idle, a frame is removed from the distribute queue and is sent to that PE. When the PE has finished the execution of the frame, its result is send back to the CU and any frames that are waiting for this result are updated. This could cause additional frames to be added to the queues. While the PEs are executing frames from the distribute queue, the CU will process frames from the non-distribute queue. If there are no non-distributable tasks and all of the PEs are busy, then the CU can process a distributable task. If the PEs can send an interrupt to the CU when they have completed processing a frame, this scheme will have low synchronization overhead. Otherwise, the CU would need to poll the PEs after processing each non-distributable frame to see either that none of the PEs was idle or to collect results and distribute new frames for processing if any PEs were idle. Even with a moderate number of PEs (e.g., 16), the polling overhead would still be small.

Three main items remain to be considered. The first two of these are where the actual data are to be stored and how I/O is to be handled. The second of these is easily taken care of by treating I/O functions as non-distributable operations. Care must be taken if user defined functions are
distributed to PEs without preprocessing since these functions could contain I/O operations. In this case, the frame would need to spawn a new process to communicate through the CU. This could pose a problem if a Lisp program involves a large amount of I/O.

The case of data storage is more involved. It will be assumed that all of the basic atoms will be contained in the CU’s memory. Although this restricts some operations to the CU, it also this minimizes the amount of storage needed for strings since the actual atom need only be in one processor. The pointer to an s-expression could contain an augmented address that would consist of the address within the current processors memory prepended by the number of the processor that the memory is attached to. If a function must check the value of an atom, then that function should be classified as non-distributable since only the CU has direct access to the atoms. At the end of the execution, a sweep of the PE memories may be needed to collate the remaining active pointers in order to return the results. (The second machine model eliminated many of these problems.)

Tests with a branch-and-bound problem (the “n-queens” problem) [WiH81] demonstrated that in a general MIMD system, distributed nodes add a substantial amount of overhead since nodes must be transferred between processors repeatedly. Such transfers must be minimized or the parallelism of the system will be severely limited. There are two main alternatives to this approach. The first, restricting all nodes to the CU, was used in the initial implementation. A second, having a separate global memory and controller exclusively for the nodes, would allow more functions to be distributed.

The remaining item of the three that are being considered in this section is the problem of dynamic scoping. Dynamic scoping involves keeping track of
what local variables are associated with what functions as well as the order in which to access these variables. Dynamic scoping comes into play when a user defined function uses a previously defined variable name. In this case, the executing function needs to know which environment to use. To handle this, whenever a function is invoked it will create an environment that must be passed to any functions it invokes. If these functions are within the same PE (as per the earlier discussion on user defined functions), then all that is needed is an environment stack within that PE that must be checked when a variable is referenced. Thus, when a user function is reduced to frames it must perform variable references by searching the environment levels starting with its current level.

If user defined functions are fully expanded, then all of the environment concerns can be handled by the CU as long as certain critical types of frames (such as EVAL and LAMBDA operations) are labeled as non-distributable frames. This way, only one processing unit, the CU, needs to be concerned with environment references. If the environment were allowed to be spread across the PEs, then any variable reference could potentially need to check environments in all of the PEs. Therefore, the environment will be confined to the CU in an attempt to limit interprocessor communication. Confining the environment to the CU could be a bottleneck if a lot of variable manipulation is performed within functions (e.g., many SETQs). However, the alternatives, distributing the environment (for the interconnection network model) or allowing any PU to access the environment with the use of synchronization operations (for the shared memory model) have problems that are just as severe.
4.2. Parallel Parsing

Before a Lisp program can be executed in parallel, it must first be reduced to the basic frames and their dependencies. Although this could be done by a host machine, for a sizable program it might well pay off to prepare the frames in parallel. This can be done using the following scheme. First, assume that each PE contains a copy of the source code. This can be done by broadcasting the code to the PEs as it is read in by the CU. All of the frames will be stored in the CU in anticipation of the queues to be developed during execution. However, the PEs will be used to actually create the frames. To create a frame, a PE is passed a pointer to an s-expression that needs to be reduced to frames. The PE will do two things with this s-expression. First, it will set up the basic frame structure for the operation and return this frame to the CU. The CU will then coordinate the information needed so that this frame can return its results to the proper locations. (This information is in the form of pointers). Second, the PE will return a list of pointers to further s-expressions that need to be evaluated for the arguments of the initial s-expression. The CU will put these pointers into a queue of frames that need to be created and will link these new frames to the arguments that the original frame needs. By repeating this process until the frame creation queue is empty, all of the frames can be created and linked in parallel.

4.3. Function Evaluation Synchronization

The order of execution of s-expressions within the body of a lambda function is a major synchronization concern. That is, although the arguments to a function are to be evaluated in any order, the s-expressions within a function must be executed in the given order if side effects are to behave as the
programmer desired. This is true since each s-expression within a function body could have a complex dependence upon earlier ones due to side effects. For example, the following three s-expressions

\[
\begin{align*}
\text{(SETQ RT (SQRT (DIFFERENCE (TIMES B B) (TIMES 4 A C)))}} \\
\text{(SETQ A (TIMES 2 A))} \\
\text{(SETQ RT (QUOTIENT (DIFFERENCE RT B) A))}
\end{align*}
\]

must be executed in the order given. That is, the third expression must not be executed until after the second, even though all of its arguments are ready. (Although this is an example of poor programming, it serves the purpose of the example.) A dependency list could be constructed for each of the expressions in the body, but this task rapidly becomes non-trivial. It is quite possible that an expression could be dependent on every preceding expression since a list of Lisp statements is typically highly serial. One better way of handling this would be to add a dependency to each frame so that all of the frames corresponding to a given expression cannot be scheduled for execution until all the previous expressions has been completely processed. This could be accomplished by defining a dummy argument that requires the result of the previous frame. When this result is obtained, the waiting frame can be scheduled. (A more practical scheme that is logically equivalent was developed during the implementation.) Based on the form of typical sequences of Lisp expressions (such as those in the object recognition program described later), the reduction in parallelism that occurs with this method is more than offset by the time saved by not determining all of the data dependencies.
There is an additional area in which parallelism can be automatically applied to Lisp. This involves the MAPCAR function which routes multiple inputs to the same function. Obviously, this function could be scheduled as multiple frames, each with one of the inputs. If the arguments to the MAPCAR are simple s-expressions, then the expansion is straightforward. If the arguments are functions, then this scheme will still work since the MAPCAR would not execute until all of its arguments were ready.

All of the schemes discussed so far have used a data-driven protocol. That is, frames are scheduled when they have data ready. An alternative to this approach is a demand-driven scheme. Such a scheme would work backwards from the final output expression and demand results from certain other expressions. For example, a final PRINT expression would force its immediate predecessors to execute, which would in turn force their predecessors to execute, and so on. Although this scheme is theoretically sound, the overhead that would likely be needed to support it could be considerable (for example, finding the "final" expression is not necessarily a trivial task in and of itself). Further, if there is no extraneous code (i.e., no statements that produce neither results nor useful side effects), then as long as frames can be scheduled the data-driven scheme will still be efficient in comparison. Even though both schemes force the execution of expressions in a certain order, the demand-driven scheme requires dependencies to be traced from the end of execution to the beginning. Thus, the demand-driven scheme will not be considered further here. (See [KeL79] for more details on a demand-driven system).
4.4. Frame Structure

Now that the general model has been presented, the frame structure will be described. As has already been mentioned, the frame must contain fields for the operation to be performed, the number of arguments left to be evaluated, a distribute/non-distribute tag (which could be incorporated into the operation field), a possible tag field to delay later statements in a function body, and the arguments themselves. It must also contain a pointer to the previous frame. Since the frame returns its result to other frames, it need not have a result field as the result can be stored in one of the argument fields until the result is returned to the CU.

Next, the composition of the individual fields within each frame must be considered. The operation field must be at least seven bits to account for all of the basic Lisp functions. By using an eight bit field, the top bit could be used for the distribute/non-distribute tag. For each of the arguments, a pointer to a Lisp node will be needed. (A Lisp node is also known as a CONS cell). Since the nodes are generally in the PEs and the atoms are in the CU, this node must have two component fields that can address either the CU or the individual PEs. (Each Lisp node is defined as having two pointers. This is implicit to the definition of the language.) It must also be able to indicate the type of the argument since it might be a number (in one of a variety of formats), an array, an atom, or a list. The specific processing performed depends upon this type. Other fields may be needed for synchronization of specific operations. The exact composition of the frame is highly model and implementation dependent.
4.5. Initial Recommendations

After this brief examination, the strict execution sequence and the high overhead for frame transfers would appear to limit severely the amount of parallelism obtainable. This apparent limit stems from the small number of arguments to a typical function. However, if the depth of these functions is high, the number of frames that can be processed in parallel increases dramatically. In general, the amount of parallelism obtainable is very dependent upon the specific Lisp program being executed. Since it appears to be feasible to adapt Lisp to run on a general MIMD computer, an implementation will be developed. This implementation will use a multitasking system and interprocess communication to simulate an MIMD machine. The implementation will include the formation, queuing, distribution, and execution of the frames. This simulation will provide two main results. First, it will indicate possibilities for further gains by illustrating the bottlenecks in such a scheme. Second, it will help determine if it is indeed feasible to run Lisp on a parallel machine or if instead the overhead is too high for the application to be profitable. Most likely, some parallelism will be profitable. The simulation will provide information on the extent to which parallelism can be utilized.
5. The Machine Model and the Implementation Environment

The first basic model being assumed for the implementation is an MIMD machine with one master or control unit (the CU) and a set of slave processors (the PEs). All memory is strictly local to the individual processing units. Several limitations have been placed upon this model. Initially, it is assumed that the only channels of communication are between the CU and each of the PEs. That is, no use is initially made of the possibility of an interconnection network. This is a valid initial assumption since the CU will be basically be acting as a scheduler for frame execution and the PEs will be solely restricted to working with the frames themselves. If the implementation indicates that PE to PE communication is needed, then additional communication channels can be added. (This was done with the third multitasking model.) In the actual implementation, a separate channel for communication with the CU is established for each PE. However, this is equivalent to one channel that is multiplexed among the various PEs since only one channel is utilized at a time.

One modification to this model that could prove useful would be to add a separate unit that would act as a global memory or message pool. This could either be a specially designated PE or a separate processor. The purpose of this additional module would be to store information that multiple units might need access to. For example, if nodes were stored in such a unit, then list functions could be distributed instead of being restricted to the CU. If a PE is set aside to perform this task, then it can also provide the needed synchronization between the tasks accessing the nodes. This PE would in effect be an intelligent shared memory module. However, with such a module one must consider the extent to which contention for this resource might slow down execution of a Lisp program.
No assumptions are made about the size of the programs to be run on the interpreter or the amount of memory that they will need for frames, atoms, and nodes. Instead, dynamic memory allocation will be used to obtain as much memory as the interpreter needs (within the limits of the test system). However, the sizes of the various work queues are fixed so that an inordinate amount of time will not be spent searching for jobs in the queues instead of doing useful work. Overloaded queues could well be an indication that better results could be obtained by adding more PEs to handle the work. However, it could also be an indication that the system is being swamped by overhead or that the program simply has a high number of dependencies. Thus, if these limits cause problems, the implementation can be examined to determine the cause of the overflow. If additional queue space is needed, then the queue sizes can easily be expanded. (The queue sizes became as large as 500 for the object recognition program that was used to test the interpreter. This was due to the high number of dependencies in the program.)

Initially, only a limited subset of the standard Lisp data types will be supported. This limitation is being imposed simply to make the problem tractable. Once the interpreter is functional, additional data types can be added if the need for them is substantiated. Three data types will be supported initially. These types are integer, floating point, and character (or string) data. Both the integer and floating point data types have limited precision. Character strings are also of a limited length (this effectively limits the sizes of the atoms, too). Infinite precision integers and unlimited size character strings are specialized types that are generally separated from the limited types in Lisp interpreters.
Input to the interpreter will be in one of two forms. Either the input can be in a file or it can be input from the keyboard. In either format, the user will be required to specify that all input is complete before interpretation begins. Thus, the interpreter will effectively be a batch system rather than a full interactive system. This is probably more representative of the operation of a general MIMD machine as compared to a full interactive system.

The actual interpretation will proceed in three main phases: input, parsing, and execution. The limitations of the input phase have already been discussed. It will be assumed that each of the PEs has access to the entire program text. This could be accomplished by loading the program into the PEs at the same time that it is loaded into the CU.

After the input has been received, a preliminary parsing phase will be performed. This phase provides an initial set of frames for execution. As has been suggested, this parsing can be done in parallel. This is the reason for all of the PEs to get the initial code. With this, only an address of a section of code need be passed to a PE instead of passing the entire section of code.

Finally, the interpreter will enter the execution phase. This phase consists of scheduling and executing frames that have already been parsed and that are not waiting on any other results. Note that it is quite possible (and indeed probable) that the execution of some frames (such as LAMBDA functions) will result in the creation of more frames. Thus, parsing can also occur in the execution phase. As long as synchronization is maintained so that a frame which is in the process of creation is not accidentally scheduled, this concurrent parsing and execution poses no problems.
6. Implementation of a Parallel Lisp Interpreter

Now that the general approach toward the task of implementing a parallel interpreter has been presented, the actual development will be described. This presentation will follow the order in which the interpreter was implemented. This order progresses from a serial program with a recursion point through a serial model using circular buffers for CU-PE communication and finally to a multitasking model using interprocess communication. All of the models were developed using the ‘C’ Programming Language [KeR78].

6.1. A Recursive Model

The first model of the interpreter uses a recursion point instead of multiprocessing. The idea behind this is that once a recursive version is functional, the recursion point can be converted into a spawning point for PE tasks. This model also fits well into the task of interpreting Lisp since Lisp is primarily a recursive language. This model will only be used for the task of performing the input and initial parsing phases. The execution (and intermixed parsing) task will not be implemented until the multitasking model.

The input phase is performed by reading a file into the interpreter’s memory. This file is stored in a linked list that is allocated as the program is read in. Thus, there are no hard limits on the size of the program other than those imposed by the system. Various utility routines perform the tasks of advancing text pointers and getting characters and strings from this linked list, so that the precise formation of the list is not of primary importance in the implementation.
The initial parsing phase utilizes the recursive nature of Lisp. Recall that the parsing process creates frames for later execution. This frame creation is accomplished by first creating a father frame to which the main s-expression of a program will return a result. (Only one main s-expression is allowed with this initial model.) With this initial return point established, the actual parsing begins. This is accomplished by calling a frame creation routine with a pointer to an s-expression that is expected to return a result. This routine creates a frame and supplies the necessary parameters for the frame. These parameters include the function identifier, the total number of arguments for the function of the s-expression, the number of unevaluated arguments, and the actual evaluated arguments (i.e., any initial constants). The frame also contains the location of its father frame, the location in the father frame at which the result of executing the current frame must be stored, and the current environment level.

In the process of parsing an s-expression, it is likely that other s-expressions will be encountered. These expressions must return results to the current frame. The locations of these expressions must then be passed to the frame creation routine with the current frame being considered the father frame. This is the recursion point.

Some types of s-expressions must be handled differently. For example, in the case of a DEFUN (a Lisp function definition), the location of the function definition code is saved for later access. In the case of a LAMBDA expression, frame creation is postponed until all of the parameters become ready during execution. This is done since a new environment level is needed when a LAMBDA function is expanded. It is more efficient to delay the expansion until all of the inputs are ready so that the environment need only be initialized
If a non-numeric constant expression (an atom or a list) is encountered where an argument is expected, then another routine converts this expression into nodes and atoms. Recall that all nodes and atoms will be restricted to the CU when multiprocessing is first added.

Error handling is limited to reporting system problems (such as lack of memory or file access errors) or simply reporting any detected parsing errors. Since parsing errors are usually due to missing or excess parentheses and since such an error is usually catastrophic in its effect upon a Lisp program, no error recovery is attempted.

When the initial parsing is complete, this model outputs a list of the atoms, nodes, and frames that were created from the input program. This allowed the parser to be verified by using a substantial input program that utilized all of the various facilities of the parser. Although this manual verification was tedious, it was an essential step that had to be performed before the model was updated. Once this verification was obtained, work progressed on the next model: a simulated multiprocessing model.

### 6.2. A Simulated Multiprocessing Model

With the recursion model performing the initial parsing phase correctly, the next task is to change the recursion point into a spawning point for PE tasks. The first model for this purpose will still be a serial program. However, all communication between the CU and the PEs will be required to use routines that access a pair of circular buffers for each CU-PE communication channel. In the first true multiprocessing model, all that should need to be done is
change the actual communication routines so that they perform the desired communication. The various invocations of these functions will be unchanged. This allows the debugging of the first multitasking model to be limited mostly to the actual multitasking since CU-PE communication will already have been confined to model (and implementation) dependent subroutines. (For this model, the communication routines simply maintain sets of circular buffers.)

Again, only the parsing phase was implemented with this model. This interpreter model executes a set of routines that simulate the operation of the CU and the PEs one at a time. The main difficulty with this scheme is that the operations occur in a deterministic order instead of with varying orders as would likely occur in a multiprocessor system. However, this simplification allows the majority of the PE task creation processes to be debugged without having to be concerned about any non-determinism that might be present in a true MIMD machine.

The main tasks assigned to PEs were the parsing of s-expressions into frames. If a PE encountered additional points that needed parsing or if it encountered a non-numeric constant, then this information was returned to the CU without further processing. In the case of an additional processing point, this information was added to a queue of s-expressions that needed parsing. This allowed one main s-expression to be split among a large number of PEs for parsing. In the case of a literal expression, the CU performed the translation from the program text into nodes and atoms. The CU then saved this information in a literal queue. When the frames were complete, this literal information was then linked into the frames. Note that this delay is necessary since a literal point may be returned to the CU for processing long before the frame to which it is related is completely parsed.
The outputs and verification procedures for this model were the same as for the recursive model. With this model verified, work could begin on the first multitasking model. This model will simulate multiprocessing with multitasking.

6.3. A Multiprocessing Lisp Interpreter

Once the serial simulation model was functional, it was possible to proceed to the next step - simulating multiprocessing using multitasking. This method of simulation has several advantages over simple serial simulation of parallel processing. First and foremost, the nature of the multitasking system that was used (described in the following section) was utilized in such a way as to prohibit the unintentional sharing of any data space between processes. The only way to pass information between processes in the simulation is through special routines that access the interprocess communication facilities of the system. This provides strict enforcement of the concept of separate processing units with local memories. Second, the natural execution time variations gives an element of non-determinism to the simulation that would be difficult to obtain in a single process serial simulation. Finally, by applying different weightings to the various tasks performed by the processes (e.g., by adding fixed or deterministic time delays), a good perspective of the possible causes and/or cures for execution bottlenecks can be obtained.
6.3.1. The Initial Multitasking Model

Simulation of multiprocessing by multitasking was performed on a Dual Vax 11/780 [GoM82] running the 4.2BSD [42BSD] distribution of the Unix† operating system. The Unix operating system provides a function that allows a process to create (or fork) a copy of itself. This copy is identical to the original (including the data space) except for the value that is returned by the function invocation. This facility is utilized in the simulation by having the initial process fork off several copies of itself. The initial process then becomes the logical CU. The other processes become the PEs. Since the data space is duplicated, the concept of every process having a copy of the user program that is to be interpreted is enforced simply by reading the input before the process forking. Also, by setting a specific variable before the process forking, a unique number is assigned to each PE. This facilitates debugging since a PE encountering an error condition can notify the CU of the problem as well as the location of the problem.

Communication between the processes is accomplished by using the 4.2BSD Interprocess Communication (IPC) system [42BSD]. Although the details of the communication protocol are not of critical importance to the simulation, one point needs to be reemphasized. In this simulation, there is a full duplex communication channel between the CU process and each of the PE processes. However, since the channels are polled sequentially, the effect is the same as if there were only one channel of communication that could connect the CU to the PEs. Any scheme that allows the CU to distinguish between the messages received is logically equivalent to this scheme.

† Unix is a trademark of Bell Laboratories.
Other primitive communication between the CU and the PEs (such as abort signals and startup signals) are provided by the Unix signal mechanisms. These mechanisms are not a fundamental part of the simulation. Indeed, they are only encountered when the processes simulating the PEs are started and terminated and if a severe error condition (such as a memory access fault) occurs.

With this model established and the communication channels functional, the previous serial simulation that used circular buffers for communication was converted to a multiprocessing model. The only major changes to the model were the modification of the CU-PE communication routines. These routines were changed from buffer access routines to IPC access routines. Since the previous model had already been used to isolate the communication between the CU and the PEs, the new multitasking model became functional with a minimum of effort after the multitasking environment was established.

The main result obtained from this model was that transfers of nodes and atoms between the CU and the PEs could be a great cause of parallel overhead. Thus, as was already done in the previous model, all nodes and atoms were restricted to the CU. This restricts the degree of parallelism obtainable since contention for nodes and atoms could be a bottleneck. A later model removed this restriction.

6.3.2. The First Execution Multitasking Model

With a successful multitasking parser functional, the next step in the interpreter simulation was to add the execution phase. This phase was initially limited to output and arithmetic operations. List operations and functions
requiring interactive parsing were not implemented until the next model.

The output routines were the first execution routines created since every successfully terminating Lisp program produces some output. Since all of the nodes and atoms are limited to the CU, it makes sense that the printing function should also be restricted to the CU (i.e., it should be a non-distributable function). When one considers that in an actual MIMD machine the CU is likely to be the only processing unit with access to external peripherals such as terminals, printers, or an external I/O processor, this restriction makes even more sense. The printing functions were required not only to be able to handle the basic numeric data types, but were designed to be able to handle list expressions as well. That is, given a node (that could well be linked to other nodes and atoms), the routine is capable of printing the proper list structure that this node and its descendants represent.

The QUOTE function was implemented next as a testing function for the PRINT function. Again, since the QUOTE function returns nodes and atoms as results, it was also limited to the CU. Implementation of the QUOTE function required a modification to the parsing routines since the second argument to the function must be treated differently from a standard argument. (The same situation holds for the first argument of the SETQ operator). Thus, a flag was added to the parser that indicated the nature of the argument being examined.

The next functions to be added to the interpreter were simple arithmetic operations (such as ADD1 and TIMES). For the initial testing, these were also restricted to be non-distributable functions. Implementation of these functions added a new item of concern: type conversion. The input to an arithmetic routine could well be an atom that was created by a constant expression or by
a list operation. The input could even be a string representation of a number that was formed by list operations. If a function is to be distributed, however, it must not rely upon access to atoms. This necessitated the creation of a simple pre-execution routine. Before a frame is executed, the types of the fields are examined. If the field type indicates that an atom value is needed, then the necessary value is obtained from the atom and this value is substituted into the frame. Automatic type conversion within the routines (e.g., conversion of integer data to floating point data in case of overflow) was also implemented.

Once these math operators were functional in the CU, they were converted into distributable operators. The only changes required for this step were the addition of communication routines to send a frame to a PE for execution and to obtain the result of the execution from the PE. These routines use the same communication channel routines that have already been described, so the simulation is still not explicitly dependent upon the particular set of communication primitives used.

Also at this time a standardized set of higher level communication synchronization primitives was created. Initially, it was thought that Unix signals could be used to signal the results of different types of CU-PE communication. This rapidly proved infeasible and was also outside of the model that had been developed. The scheme that was substituted was to have each message between the CU and the PEs preceded by a single ASCII character. The particular character indicates the message type. Since the message class character is sent at the same time as the rest of the message, polling the communication channels is now a simple matter of seeing if the input buffer for that channel is empty. If it isn’t, the first character in the buffer indicates the message type (and thus the length of the message). This
message can then be removed from the buffer, either leaving the buffer empty or leaving the next message-type character as the first element in the buffer.

To provide flexibility in the initial models, the responsibility for determining if a function can be distributed is allocated to a subroutine that performs a conditional check. This allows the distribution type of the function to be changed easily during testing. A final implementation could easily change this conditional to checking a single bit of the function id field (as was mentioned in the initial discussion). (The final model developed compromised between these two ideas by using an in-line code table lookup technique.)

With the basic support for distributed execution ready, several additional arithmetic functions were implemented, some as distributable functions (e.g., QUOTIENT and PLUS) and some as non-distributable functions (e.g., SUB1). This allowed testing and debugging of the system with simple mathematical problems. From this point on, every new function was tested for all of its input classes as well as for the right types and numbers of inputs. Due to the modular nature of the frame structure, once a function interfaces correctly to a few different types of routines, it will also interface correctly with the rest of the routines. This makes verification of the operation of the interpreter tractable.

The next feature to be added to the interpreter was a reporting feature that provides an image of the current state of the simulation at various time instants. This is done by writing a single line to a specific output file from the CU each time the PEs are polled. This line indicates which of the PEs are active with a task and which PEs are idle. It also indicates if the CU is acting only as a distributor or if it is executing a non-distributable task at the time of the polling. Two such output files are created. One is created during the initial
The parsing phase that occurs just after input. The other is created during the execution phase (which will become the execution/interactive-parsing phase in a later model).

One final operation was added to this model. This was the SETQ function. This is one of the more involved operations since it must perform a search of the environment as well as create or modify an environment entry. By using the environment structure that was discussed previously, SETQ first searches the current environment level (passed as an argument to a routine) for the variable name in question. If the name is found, then the corresponding entry is modified. Otherwise, the search continues to the next higher environment level. If the search reaches the top level and the variable has still not been found, then a new entry is created in the top level. This corresponds to a global variable. An option flag was also incorporated into these routines to force a variable to be set in the current level and to return an error condition if it is already set. These options will be used in a subsequent model by routines supporting LAMBDA functions to set up the initial values of a function's parameters. Since the environment is handled exclusively by the CU, SETQ is a non-distributable function.

Two additional cases of variable evaluation will now be considered. First, the variables "t" and "nil" are automatically defined in every level of the environment. These "variables" are also special in as much as they cannot be reset to a different value than their defaults. Second, variable references need to be performed whenever a function is scheduled for execution. This is done just prior to checking for pre-execution operations (such as atom copying for arithmetic operations that can be distributed). This involves a subset of the searching routines that SETQ uses.
6.3.3. The Second Execution Multitasking Model

Once the goal of the previous model, simple distributed execution, was accomplished, work could proceed on expanding the power of the interpreter. Three main objectives were realized with this second model. These were the execution of multiple programs, the addition of simple list operations, and the addition of user defined functions. The realization of these objectives produced the first version of the interpreter that could realistically be used for simple programs. Several minor goals were also realized.

6.3.3.1. Main Objectives

The first objective was to allow multiple programs (i.e., multiple main s-expressions) in the input. Without this capacity, a user could not define functions, set variables, and then call a function without creating a controlling program for all of these functions. Clearly, the capacity for multiple programs is needed. This capacity was obtained by saving the location of the next main s-expression to be executed (if any). When execution of one main expression is complete, the parsing and execution phases restart with the next expression. Only the top level of the environment and function definitions (obtained from DEFUN operations) are preserved between the execution of these main expressions. This enhancement is an expansion of the functionality of the interpreter, not an increase in its parallelism.

The second objective called for the implementation of several list operations (such as CAR, CDR, LIST, and APPEND). Since these operations require the manipulation of nodes and atoms, they must be non-distributable functions for the current machine model. Once these operations were implemented, a workable subset of the basic Lisp operations was available.
The third objective, addition of user defined functions, was the major task in the development of this second multitasking model. LAMBDA functions (functions that are used only in one location of the code) were added first. For the purposes of initial scheduling, a LAMBDA frame is considered to be the same as any other non-distributable frame. When all of its parameters are ready, the execution of the LAMBDA frame begins. The first step of this execution is to get the location of the function definition that was stored in the frame during the parsing of that frame. This definition is then searched for the beginning keyword LAMBDA. If this keyword is not found, then the function is not a valid operation and an error is reported.

If the keyword is found, then the setup of the LAMBDA environment begins. This is done by creating a new environment level whose father environment is the environment of the father frame of the LAMBDA frame. This environment initially has no entries. Entries for this environment are obtained by matching variable names from the parameter list following the LAMBDA keyword with the arguments to the function. The numbers of parameters and arguments are required to match. Parameters are set in this particular environment level by the use of the special options that were designed into the SETQ supporting routines. These routines can also be used to force the creation of local variables (such as would be created by the PROG operator). This setting of local variables for the parameters insures correct dynamic scoping for variable accesses. Since all of the values of a variable exist and are accessible simultaneously, this interpreter uses deep-binding. (Many serial Lisp interpreters use shallow-binding, which only allows access to the latest definition of a variable.) Both the environment entries and the environment level are reclaimed when the execution of the LAMBDA frame is
With the environment ready, execution of the LAMBDA can begin. The body of the LAMBDA is a set of (possibly atomic) s-expressions. If the s-expression is atomic, then the value of the atom is stored in the first argument location of the LAMBDA frame. This location is free for this use since all of the arguments have already been transferred to parameters in the environment. If the expression is not atomic, then it is assumed to be an s-expression for execution. As was discussed earlier, if the body of the LAMBDA contains multiple expressions, then they should be executed in sequence due to data dependency problems that arise due to possible side effects of the Lisp operators. This is accomplished by first scheduling the current s-expression for parsing and execution. Subsequent expressions are scheduled later. The LAMBDA frame is considered the father of the frame that will result from the parsing. The result will again be returned to the first argument location in the LAMBDA frame. Next, the pointer fields in the LAMBDA frame are advanced and a flag is set to indicate that the frame has already evaluated its parameters and is now executing s-expressions (the total argument count field is used for this purpose since it is no longer needed). Finally, the unevaluated argument count of the LAMBDA frame is set to one and the frame is rescheduled.

When the parsing and execution of the s-expression from the LAMBDA frame are complete, the frame will again be scheduled for execution since it will again have zero unevaluated arguments. If there are additional s-expressions to be evaluated, then they are scheduled in the same way as the first one. Otherwise, the execution of the LAMBDA frame is complete and the return result, which is the result of the last s-expression, is already in the proper return location. Therefore, the LAMBDA frame can now return a result and complete.
The major change in the execution sequence that is caused by the LAMBDA function is that parsing and execution can now occur concurrently. Therefore, it is crucial that these operations be synchronized in some way or it is possible that one frame could return a result before the parsing of the frame to which it is to return a result is complete. This synchronization is accomplished by examining the function identifier fields of both the frame being considered for execution and the frame that would receive a result from this execution. If either of these fields is zero then the frame is not scheduled. (Zero is an invalid function identifier. It is also the initial value of the function identifier field). A zero field in the frame being examined indicates that the frame itself is still being parsed. A zero field in the father frame indicates that the frame that would receive the result is not completely parsed. If execution were to be scheduled for a frame whose father node was not parsed, then if a result were returned from execution before the parsing completed, the return of the parsed father frame would destroy the result. (The shared memory model presented in section 8 required an additional flag for this synchronization.)

The addition of DEFUN-produced functions is simple once LAMBDA frames can be executed. When a DEFUN is encountered, its parameters (which are actually the body of the function) are not evaluated. Instead, a function cell is created that contains the location of the body of the function and the name of the function. When such a function is invoked, the arguments are converted into entries in a new environment level in much the same way as the arguments for a LAMBDA function were. At this point, the frame is effectively a LAMBDA frame. It is therefore converted to a LAMBDA frame and execution proceeds accordingly. This way, there can be any number of
invocations of a particular function in execution concurrently.

6.3.3.2. Minor Additions

In addition to these major objectives, two minor additions were made at this stage of the development. First, the `t` and `nil` atoms were considered distributable. Second, error handling and reporting were standardized. With the addition of several functions that can return a `t` or `nil` result, a method was needed to be able to return the atomic value of `t` and `nil` from a distributed frame. If this ability was lacking, then all conditional functions would be non-distributable. The method of accomplishing this goal proved simple under the environmental context already established. Since the variables in question always have a fixed value in every environment level, the values can be returned merely by returning the variable names `t` or `nil`. When the frame to which this variable value is returned is scheduled for execution, then the variable name will be properly expanded into its correct value.

Error handling was also standardized with this model. Several types of Lisp errors are reported. These include parsing errors, passing the incorrect number of arguments to an expression, passing the incorrect type of arguments to an expression, and mismatched or missing parentheses. No attempt at error recovery is made since the usual cause of these errors is a parenthesis error, which can drastically change the nature of the program. Instead, when such an error is encountered the user is notified and execution stops. Errors that occur due to system faults (such as lack of available memory), interpreter errors, or communication errors (such as an unknown message type) are also reported.

All errors are classified into one of five levels, numbered from zero through four. A level zero error indicates a possible problem such as a minor missing
implementation feature. A warning message is issued and the program attempts to continue. A level one error is an indication that a program tried to use a feature that is still under development. This error (and all succeeding levels of error) cause execution to terminate. Level two errors report standard Lisp errors such as mentioned above. Unimplemented Lisp functions also fall into this category since they appear to be undefined functions.

A level three error is issued when the interpreter detects that its execution is in error. For example, an error message of this level would be issued if the interpreter tried to process an unknown type of operand. Level four errors are strictly limited to reporting system faults. However, they could well indicate an interpreter fault (e.g., if the interpreter is not freeing memory correctly) or user errors (e.g., giving the name of a non-existent file for the programs to be executed).

6.3.4. The Third Execution Multitasking Model

The original intention for expanding the interpreter from the second model involved adding the COND operation and most of the remaining Lisp operators. However, the severe limitations imposed on the performance of the interpreter by nodes and atoms being limited to the CU indicate that the model needs to be improved first. Although the initial operation of the interpreter indicated that the CU could function well as a scheduler and distributor, it becomes a bottleneck when it also has to perform all list operations. The restriction that the CU perform all list operations originated from the restriction that all of the nodes and atoms be in the CU. Several possible alternatives will be considered.
The first possible method of modifying the model would be to include additional message types so that the CU could transfer nodes and atoms to and from the PEs. This scheme has the advantage of being highly compatible with the current model. Thus, it would still be close to the idea of a generalized model instead of a specific model. However, this method still has problems since the CU will have a larger amount of communication to perform. This additional communication could well outweigh any gains achieved.

Another alternative is the addition of a global node and atom pool. This pool could either be a global memory with mutual exclusion operations or another PE that is connected to the other PEs through an interconnection network. This pool would have a very limited number of functions. It would primarily serve as the storage location for all nodes and atoms. It would have only two types of operations: it could receive a request for a node or atom from a PE and in response it would return a specific node or atom, or it could receive a node or atom from a PE and return the true address of that node or atom. This pool would not perform any other processing besides maintaining the lists of free and used nodes and atoms.

This scheme frees the CU from having to deal with accessing nodes and atoms before sending a frame to a PE for execution. It also allows almost every operation (with the exception of scheduling and environment manipulation operations) to be distributed. Any processing unit, including the CU, which requires a specific node or atom must go through the access pool. However, it has the disadvantage of requiring all node and atom access to go through one unit. This might not be less of a bottleneck than requiring the CU to handle all of the communications since the CU would have to perform accesses to this pool as well. This bottleneck would be minimal in a system with true global
memory, but to have a large parallel MIMD machine with global memory is unlikely be feasible. (A way to solve the contention problem for the interpreter was developed with the shared memory model of the interpreter. This interpreter model eliminated read/read and read/write conflicts. In addition, the shared memory was partitioned is such a way as to reduce contention further.)

A third alternative that is based upon the idea of a node and atom pool is a two-level access scheme. In such a scheme, the pool would act as a cache for the nodes and atoms. With this method, when the CU creates a node or atom, it stays in the CU as it does with the model from the previous section. When a PE creates a node (or atom), it would first be created in the global pool. The cache access and synchronization takes place during a node access.

When a PE needs a node or atom, it sends a request to the pool. The pool would look for the node or atom. If it is found, then it is returned. If it isn't found, then a request is sent to the CU for the node. The node is then copied into the pool and a copy is sent to the PE. When the CU needs access to a node or atom, the procedure is reversed. (The pool is checked second. If the pool is accessed, a copy is kept in the CU.) If a frame is not returned before creation of all of its nodes and atoms is confirmed, then the synchronization problem caused by trying to modify a node before it is created is eliminated.

The main concern for synchronization is when a node is modified. (Recall that creation of new nodes does not pose problems). When a node is modified, then before the frame is returned, any copies of the node must be deleted. That is, if the frame is in a PE, then the node will be modified in the pool and the CU node must be deleted. This could be accomplished by sending a message to the CU consisting of a delete atom/node message character and the address of
the item to be deleted. Both the CU and the pool would need to keep lists of all non-local nodes since the address of the node in the unit will not be the same as the address of the node in the remote unit.

What this comes down to is a bidirectional hardware cache between the PEs and the CU for node and atom access. It might be more reasonable to restrict the cache to be unidirectional. This could be accomplished as follows: the access path that causes problems is when a PE creates or modifies an existing node. To handle the first case, creation, instead of returning created nodes to the cache, return them directly to the CU. They will then be there if the CU needs them and they can still be accessed via the cache. The second case, modification, can be synchronized by not allowing PEs to modify a node. That is, node modification operations (such as NCONC) would be restricted to the CU. Also, before modifying a node, any copies of it in the cache would need to be deleted. This restricts the cache to be unidirectional and keeps synchronization at a minimum. The cache could still be a PE, just like the global node and atom pool could have been.

This scheme has a major problem, however - it requires a major change to the MIMD model if the cache is to be implemented directly in hardware. This tends to change the aim of the project from implementing Lisp on a general MIMD machine to designing a specific machine for Lisp. This is not desired.

There is still one more alternative to be considered. This alternative is the converse of the method in [GuN83]. That method allowed nodes to be distributed across the PEs. When a frame needed information from a node in another PE, the frame was transferred to the other PE. The converse of this is to transfer the nodes to the PEs that need them. This allows node operations to be distributed without putting the burden of controlling the nodes on the
CU or a centralized pool. If a packet-switched interconnection network [Sie85] is available as part of the MIMD machine, then several node transfers could take place concurrently.

The main concern in this case is still what happens when a node is modified. This can be handled by distributing a function that modifies a node to only the PE that contains the node that is to be modified. Provisions for this type of task distribution were incorporated into all of the multitasking models just in case of such a situation.

Implementation of a multitasking model to simulate these additional features is a logical extension of the current model. Transfers of nodes between the PEs and the CU could be accomplished by adding a new message type and using the existing communication channels. However, transfers of nodes between PEs would require the addition of an additional set of channels. The interconnection network itself could be simulated by an additional process in the multitasking domain.

One implementation specific detail is of importance here. The current model uses a simple address field to point to a controlling node in an argument field for lists and atoms. This is in place of an actual node in the argument field as was initially proposed when the model was being developed. This modification greatly reduced the amount of information that needs to be sent via a communication channel when a frame is distributed. The problem is that this address does not allow for the case of a non-local (non-CU) address. There are two ways in which this can be handled. First, the actual nodes could be put in the argument fields. This causes problems with large amount of data transfers when a frame is initially distributed.
Alternatively, a tag containing the processing unit number can be associated with each address in an argument field. This minimizes the amount of information that must be transferred when a frame is first sent, but it will require the fetching of nodes from PEs later. This apparent disadvantage disappears when one recalls that in order for a node to be put in the frame during parsing, the entire node must be returned to the CU. Thus, this case would require the node to be transferred from a PE to the CU and then from the CU to (a possibly different) PE. Contrast this to transferring a simple address and a tag back and forth from PE to CU and then transferring a node from a PE to another PE. The improvement is even greater if the node is in the same PE, since this eliminates the node transfer completely.

By comparing these possible methods of distributing list operations, the method of distributed nodes seems to be the most promising. Therefore, the third multitasking model will utilize this concept. This model now becomes a model of a general MIMD machine with the addition of a packet switched network. However, several modifications need to be made to the implementation in order to implement this model change. The modifications include changing some of the internal data structures to account for general node addresses and the addition of an additional process and additional communication channels to account for the interconnection network. Another Unix signal will also need to be utilized to give the network the capacity of interrupting the PEs and the CU.

Since the previous model had to be changed, this new model was developed in several steps. First, the data structures were modified and the software was modified in order to interface properly with the new structures. With the new structures installed and functional, the additional process and
communication channels were installed. This addition caused the initial synchronization to become somewhat complex. The actual synchronization mechanism is illustrated in Figure 2.

At this point, the actual networking can be installed. To illustrate the actions that the network process must perform, consider the following interactions that the network must support. The first interaction is PE_i requesting a node from PE_j. When PE_i requests the node, it will wait until the network returns the node. During the wait, PE_i will service requests for nodes (and atoms) from itself. It will service NO other requests, however (such as frame execution). This is accomplished by restricting servicing to the network communication channel channel. The network interrupt is disabled during this wait.

The second illustration is of the actions that must occur when PE_j receives an interrupt from the network. This interrupt would be caused by a PE requesting a node (or atom). When this interrupt is received, PE_j goes into a network service state (effectively an interrupt service routine) until all requests from the network have been satisfied. The network interrupt is disabled during this time. The only service requests from the network will be for nodes and atoms. When all of the requests have been fulfilled, the PE resumes its previous operations (i.e., it returns from the interrupt).

For purposes of network communication, the CU is considered to be another unit tied into the interconnection network. All requests must be in the buffers BEFORE an interrupt is sent. Otherwise, a PE could get a network interrupt and find an empty buffer. It would then think that it had serviced the buffer. Also, since the entire buffer is serviced when an interrupt is received, one pending interrupt is sufficient.
Actual connection sequence in the implementation:

Key:
- = passage of time increases in the downward direction
- = interprocess signal
- = interprocess communication link (IPC)
+ = crossover of | and - (no interference)

CU
--
Set up the net ports.

PEs
---
Idle

Network
------
Set up the net ports.

Prepare to listen on the ports.

Receive the network existence signal and signal the PEs to connect to the CU

Receive the startup signal from the CU. Connect to the CU.

Accept the PE connections.

Connect to the network (the network process must already exist since the CU signal was received)

Accept the PE connections.

Enter run state.

Accept the CU connection. Signal the CU that execution can begin.

Enter run state.

Receive the start signal. Begin execution.

Figure 2

Initialization Synchronization
The main concern with distributed nodes and access of those nodes via a network is the prevention of deadlock. That is, the case of multiple PEs waiting on each other forever must be prevented. The interrupt priority scheme just presented will do this, since a PE will still service network requests while it is waiting for other information to be returned to it. For example, consider the sequence in Figure 3 where PE_i and PE_j request nodes from each other. This is a worst-case scenario. It can be seen that the interrupt priority scheme and the servicing of requests while waiting for a node (or atom) is sufficient to prevent deadlock.

Testing of the third model began by implementing the five main list operations, CAR, CDR, LIST, CONS, and APPEND. This testing revealed another bottleneck in the system. This bottleneck is due to the total equality between the atom nil and the empty list. Since these entities are fully equivalent, operations that encounter an atom where a list is expected must check to see if that atom is nil. If so, it can be treated as an empty list. If not, an error condition must be reported. A similar situation occurs when an operation expecting an atom receives a list.

Since nil and the empty list are equivalent, only one internal representation is actually required. Since the internal representation of an empty list is simpler than the internal representation of a nil atom, the former will be used. (The empty list representation is simpler since only a null pointer is needed instead of a pointer to a character string and the string itself.) Therefore, the next last step in the actualization of the third model was the conversion of these two representations into one. Although this is a major theoretical change, the actual code was modified with a minimum of effort.
PE i

initial request for node from PE j issued to the network (interrupt pending to PE j)

PE i enters service-wait state

PE i receives request from PE j; this request is serviced.

PE i receives the requested node and continues operation. Network interrupts are re-enabled.

PE i receives the network interrupt that was pending. Since the cause of this interrupt was already serviced, PE i will find no service requests and will continue with normal operation.

PE j

initial request for node from PE i issued to the network (interrupt pending to PE i)

PE j enters service-wait state

PE j receives the requested node from PE i. PE j exits the service-wait state.

PE j receives the interrupt that was pending and enters the network-service state.

PE j receives the request from PE i; this request is serviced.

PE j exits the network-service state and resumes execution.

Figure 3

Network Service Example
6.3.5. The Fourth Execution Multitasking Model

Several advances were made with the fourth model, including the COND operation, the EQUAL and PROG operators, advanced resource usage reporting, and array operations.

The COND operator presents some problems because its syntax is substantially different from most of the other Lisp operators. That is, the first s-expression in a list must be treated differently from subsequent expressions, and execution of the subsequent expressions depends upon the result of the first expression. Compare this to an arithmetic function where all s-expressions at the same level can be scheduled for execution concurrently. The problem stems from the fact that COND is strictly a serial operator. Therefore, to insure correct automatic parallelization, it must remain basically serial.

Forcing COND to be a serial operator effects only the top level of the execution scheduling of the COND. That is, the test expressions are executed one at a time and then other tests or executions are scheduled after the test returns. If the tests or other expressions are themselves Lisp operations, they can be executed in parallel as usual.

The COND operation is performed by scheduling lambda operations in much the same way as execution of a DEFUN does. These operations are set to return their results to the first argument of the COND frame, so that testing of the first expression for nil is simplified. If the last test fails, a nil will automatically be returned. If a test succeeds, all subsequent s-expressions in the same list as the test operation will be scheduled and executed. From this point on, execution of the COND is identical to execution of a LAMBDA with the exception that the current environment must not be freed when execution completes.
With the COND operator functional, one of the primary operators that is used with in COND, EQUAL, was implemented. EQUAL is a substantially more complex case than some of the earlier logical operators since it must also work on general s-expressions. This requires a recursive comparison of subexpressions if the upper levels of the expressions being compared match. Recursion is actually used in the implementation. This is also a distributable operation since it does not need to perform any environment accesses or adjust the scheduling tables. Therefore, it uses the interconnection network to get copies of any non-local nodes it might need. These copies will be discarded when EQUAL completes its execution since they will not be modified. Note that string atoms will never need to be fetched since the uniqueness of atom names in the master atom list means that atoms can be checked for equality simply by comparing their addresses.

Since this model must be tested with various algorithms, some method of reporting statistics for the operation of the implementation is needed. The particular information that needs to be reported is how much time the various processes have used, how many bytes they have read from and written to other processes, and how many interrupts they have received (from the interconnection network). To make testing of the implementation tractable, an additional Lisp operator, SYSTEM, was defined. This operator accepts one numeric argument and performs various system functions. Execution of this function with an argument of one results in the desired statistics being appended to a file. A semaphore file is used in the process so that the results will be synchronized.

In order to keep the results file from becoming jumbled, a locking mechanism is used to insure that only one process can write to the file at any
given time. All other processes trying to write to the file will block until the process currently writing is done writing. If the amount of information that each process must write to the results file is small, this blocking will not degrade system performance. In fact, the desired information plus a code for the specific process is encoded into one line of less than 80 characters.

The only major type of operation yet unimplemented at this point is iteration. Iteration in Lisp is accomplished via the PROG operator. A PROG will exit if it runs out of s-expressions or if a RETURN operator is executed. Looping is accomplished via the GO operator. The initial setup and execution of a PROG (and PROG2 and PROGN, for that matter) is very similar to a LAMBDA operation. The first list after the operator name is still the parameter list, but in this case all of the parameters are initialized to nil. Thus, the parameter list is really a declaration list for local variables.

The difference in set up between a PROG and a LAMBDA involves the labels used by GOs. After the parameter list, each list is still executed in order. However, if an atom is encountered in place of a list, then that atom is considered to be a label. Therefore, before execution can begin, the body of the PROG is scanned for labels. When a label is encountered, a variable entry is created in the environment corresponding to the label. This entry contains the location of the label in the PROG. By using the environment for storing label information, nested PROGs are handled correctly and duplicate labels are treated as an error automatically. After the labels have been located, execution of the PROG can begin. This execution is identical to the execution of a LAMBDA with the exception that atoms (labels) are simply skipped over.

There are two operators that are designed to function within a PROG. These are the RETURN and GO operators. When a RETURN is encountered,
a backwards search of the father frames of the return frame is performed to find the PROG from which to return. When this frame is returned, any intermediate frames that are waiting on the completion of the execution of the return must be canceled since it makes no sense to get a result from a return operation. Therefore, once the father PROG is found, the value to be returned is transferred to that father frame. Then any frames up to the level of the PROG that are waiting on the execution of the RETURN are removed from the waiting queue and the frames are returned to a free space list. Since this reclamation must occur in the CU, RETURN is a non-distributable operation. Once any intermediate frames have been reclaimed, the master PROG frame is rescheduled for execution. A flag is set so that the PROG will exit with its return value correctly after freeing its environment entries.

One particular trouble case is handled specially for the RETURN and GO operators. Consider the expression

(l (return 2) (return 3))

which is clearly garbage. This type of problem is indicated in the interpreter by an intermediate frame between the RETURN (or GO) and the PROG having more than one unevaluated argument. When this type of situation is detected, an error message indicating the problem is issued and execution ceases.

The GO operator is handled almost identically to RETURN. Again, it must be called from a PROG and intermediate frames must be freed. If the single argument to a GO is a variable with a type of LABEL, then the location of the continuing execution code is transferred to the PROG frame and the PROG frame is rescheduled. Otherwise, if the argument is an atom, then the atom may also be the name of a variable. Thus, a search of the local environment level is performed in this case. GOs are not allowed to exit from a
PROG. (That is, you cannot jump out of a loop).

One more data class, the array, and its supporting operations are needed in order to support the execution of image processing algorithms. Arrays are declared (and automatically created) using the ARRAY operator. In Lisp, an array is accessed via a function call where the name of the array is used as the operator. Therefore, the array information is stored in the same internal structure that a function definition is. Memory for the array is allocated in the CU when the array is declared. Since only one element of an array is accessed at a time, array accesses are considered environment operations and are thus confined to the CU.

As has been mentioned, array access is accomplished via a function call. This function call performs the address computation for the array element to be accessed. Bounds checking can easily be performed at this point. Although some Lisp interpreters do not do bounds checking, this interpreter does since faulty memory accesses can destroy parameters vital to the multitasking environment. The overhead of bounds checking is well justified by this concern. In addition, bounds checking can always be disabled at a later date or a system function could be added to disable the checking.

Values are stored in an array by using the STORE operator. The first argument of this function is a list that looks identical to an array array access expression. Therefore, when this syntax is encountered, the array function must determine if the calling function is a STORE. If it is not, then the value of the array element is returned. If the calling function is a STORE, then the address of the array element is returned. This allows the STORE to modify the array. Since the array function must thus be able to examine its father frame, this is another reason for array operations to be confined to the CU. (Recall
that all frames are handled by the CU in this model.)

If an array is small, it can be initialized by using SETQs. However, for moderate or large sized arrays, this scheme rapidly becomes impractical. Further, it is usually desirable to be able to initialize arrays from and store them to files resident on the supporting system. For example, for image processing, the images are often stored as a serial file with one byte for each pixel. Therefore, to provide the necessary interface with the underlying system, system functions were created that allow fixnum and flonum arrays to be moved to and from external files directly. These external files can either be byte-wise data (as just described) or ASCII values separated by white space. Other data formats could be added with a minimum of effort.

Finally, improvements in several system procedures were made in this model. For example, the lexical search for operators was converted from a linear search (easy for debugging the interpreter) to a binary search (better for speed). Also, the function call that was being used to determine whether or not a function could be distributed to a PE was replaced by an in-line table lookup. A similar in-line coding scheme was used for invoking the proper function to execute a frame.

Command line options, including multiple input files and combined file and keyboard input, were added at this time. Many additional Lisp operators were implemented to provide a good working set. These new operators included the READ operator as well as arithmetic, logical, and list operators.
6.3.6. The Fifth Execution Multitasking Model

An examination of the state of the interpreter after the completion of the fourth model revealed a serious problem in its structure. This problem originated in the method of storing the code. Recall that through the fourth execution model that it was assumed that every processing unit had access to a copy of the code. Although this was a reasonable original assumption and was trivial to implement under the given operating system, it might not be a generally valid assumption. It might be that the memory of the PEs is limited so that storing duplicate copies of the program text would be wasteful of system resources. In addition, if the source program was changed then the code in all of the PEs would have to be updated simultaneously.

This method of text storage also makes the EVAL operator (the heart of Lisp) inefficient since a list would have to be converted into text and distributed to all the PEs before parsing could begin on the expression. Therefore, the major change from the fourth model to the fifth model was a change in the internal representation of the program. Instead of storing a copy of the text in each unit, one copy is kept in the GU. Also, this copy is stored as a list with the same structure as a Lisp data list. This agrees more closely with other Lisp interpreters. This is also a good time in the development at which to make such a switch since the list structure is fairly solid. If, instead, such a change had been made before it was decided to allow distributed nodes, then the effort required for the conversion would have had to have been repeated when the list structure was revised.

At the same time that the parsing structure was modified, provisions were made to allow the CU to parse the program and produce frames. Until this point, frame creation was restricted to the PEs. System functions were created
to allow the parsing to be distributed or restricted to the CU under the control of the programmer.

With the internal structure of code and data being identical, implementation of the EVAL operator became straightforward. The main concern is that the code to be evaluated must be in the CU. However, data could well be distributed across the PEs. Therefore, the data list is copied into the CU before the EVAL is executed. The main routines needed to support this already existed since a similar situation occurs for the APPEND operator. From this point, all that needs to be done is create a frame from the data and return the result of the evaluation. Since this involves adjusting the scheduling queues, EVAL cannot be a distributed operation. (Recall that scheduling operations occur only in the CU.) By combining the PRINT, READ, EVAL, PROG, and GO operators along with multiple file input, a pseudo-interactive multitasking interpreter resulted. This made further testing of the interpreter easier.

This revision of the parsing structure also simplified the task of implementing some of the few major unimplemented operators that remained. Specifically, FUNCALL, MAPCAR, and APPLY were implemented. All of these functions involve some type of transformation of data to code and then creating or adjusting execution frames. The simplest of these is the FUNCALL operator.

To execute a FUNCALL, the first argument must be converted into code. If the first argument is an atom, the it is either the name of a Lisp operator or a user defined function. In this case, it is a simple matter to change the function identifier of the current frame accordingly. However, the first argument could also be a list (such as a LAMBDA operation). In such a case,
the list would need to be copied to the CU just as was the case for the EVAL operator. Finally, the remaining arguments need to be moved within the frame. That is, the second through the \(n\)-th arguments for the FUNCALL become the first through the \((n-1)\)st arguments for the new function. Since the frame itself can be modified, rescheduling the frame will automatically cause the execution of the new function. Since the frame is modified, this operation also cannot be distributed. However, there is no reason why the new function created by the FUNCALL cannot be distributed. Indeed, after the FUNCALL frame has been modified, it is indistinguishable from a frame created in the “normal” manner.

Implementation of the MAPCAR operator required more coordination and interaction with the scheduler. The MAPCAR operator takes any Lisp operator (or user defined function) and one or more data lists and then applies a specified operator to subsets taken from the lists. The results of these operations are then returned in a new list. This is the closest Lisp comes to a true parallel operator. Therefore, a high degree of parallelism should be obtainable in its implementation.

The multitasking implementation consists of creating multiple frames, one for each data set obtained from the lists. Each of these frames is passed the function identification information using the same technique that was used to modify a FUNCALL frame. These subsequent frames return their results to a linked list. This allows any size list to be used in a MAPCAR. (If the results were returned to the MAPCAR frame directly, the list size would be limited by the frame size.) However, the MAPCAR frame is still considered to be the father frame of these subsequent frames. This causes the unevaluated argument count of the MAPCAR frame to be decremented every time one of the new
frames completes. If the number of unevaluated arguments in the MAPCAR frame is reset to be the number of new frames created, then the MAPCAR frame will be rescheduled when all of the new frames complete execution.

By setting a flag bit, one can tell if execution of a MAPCAR frame is due to this rescheduling or the original call. Another field of the MAPCAR frame is used to save the address of the head node of the linked list. When the MAPCAR is scheduled the second time, the arguments from the linked list are converted into a Lisp list for return and the linked list that was used for the MAPCAR rescheduling is returned to the free memory pool. Note that a high degree of parallelism is obtainable here since a large number of frames can be executed concurrently after they are created by the MAPCAR. Again, due to the fact that frame creation and scheduling is intrinsic to this operation, MAPCAR cannot be a distributed operation.

Finally, the APPLY operator was implemented. The operation of APPLY is a cross between the operations of MAPCAR and FUNCALL. Although it takes only one data list as an argument, it applies a function to that argument. Since only one new frame is needed, the existing APPLY frame can be converted into the new frame using a methodology similar to the one that FUNCALL uses. The data list is converted into arguments using the same strategy that MAPCAR uses.
7. System Response Improvement

To be able to simulate the operation of a parallel interpreter on realistic problems, two main issues remain to be addressed. These are reducing the amount of memory required by the interpreter and decreasing the cpu time that the simulation takes. Reducing system memory usage by implementing garbage collection for the interpreter will be discussed first.

7.1. Garbage Collection

Garbage collection occurs on the basic entities of nodes and atoms. It is not needed for the frame constructs since a frame is automatically returned to a free pool when it has been executed. Garbage collection is triggered when there have been a set number of requests for new nodes in any of the processing units. This limit can be varied depending upon how often the delays caused by garbage collection can be tolerated and how much memory the interpreter can be allowed to use. The state of the interpreter must be stabilized during garbage collection so that information that is in transition is not damaged or destroyed. For example, if a LIST operation is in execution in a PE while garbage collection occurs, then some of the nodes in the list could erroneously be freed. The actual delay before garbage collection can begin is typically quite small, since the only events that must be waited on are frames that are currently in execution.

Garbage collection occurs in three basic steps. First, the automatic trigger occurs and the state of the interpreter is stabilized. Second, every active node and atom is marked. This involves examining the nodes in the frame queues as well as environment variables, arrays, and function cells. Often, a list will have
to be traced across PE boundaries. The communication and synchronization involved here is not trivial. Finally, a master linked list of all nodes is traced in each PU and all unmarked nodes are returned to a free pool.

The automatic trigger can occur either in the CU or in one or more of the PEs. If the trigger occurs in the CU, a flag is set so that garbage collection will occur at the proper time. If the trigger occurs in a PE, then a special message is sent to the CU which also results in a flag being sent. This flag is checked in the main scheduling and execution loop in the CU. If it is set, then the state of the interpreter is stabilized. This is done by waiting until none of the PEs has a task in execution. No further tasks are scheduled during this time.

Once the state of the interpreter is stable, the nodes (and atoms) can be marked. This is accomplished by performing a trace of all active lists and atoms. As was mentioned before, all possible sources of active nodes must be considered. Due to the large number of data structures that can contain active nodes at any one time, determining which lists and atoms must be marked is not a trivial task.

Tracing a list involves setting a flag in the current node and then recursively following both of the links of the node and marking these sublists. Tracing of a particular sublist ends when neither of the links of the current node under consideration are node pointers. This tracing is complicated by the fact that lists usually cross PE boundaries. That is, a list may have a master node in the CU, followed by a node in PE₂, followed by a node in PE₃, and so on. Thus, it must be possible to trace the lists across PE boundaries. The CU is used to coordinate this activity.
When the CU finds that it has to trace a node that is not in the CU, then the general address of the node is added to a stack of external nodes to be marked. Periodically, the CU will send the nodes from this list to the PEs followed by a synchronization signal. When the PEs receive these node addresses, they will continue the tracing. If a non-local node address is encountered in the PE during this tracing, then this address is sent back to the CU for later distribution to the proper PU. When a PE has finished all of the tracing tasks that it was sent by the CU, it then returns a synchronization signal to the CU. This way, the CU knows when a PE is still performing tracing. When the CU has marked all of its nodes, it waits for the PEs to trace all of their nodes. If the stack of general node addresses is empty after all of the PEs have finished tracing, then the entire node marking task has been completed.

Atom garbage collection occurs along with node garbage collection. If the CU encounters an atom during tracing, it simply sets a flag in the atom data structure since the atom must be in the CU. If a PE encounters an atom during tracing, it sends the address of the atom back to the CU so that the CU can perform the marking.

After all of the active nodes and atoms have been marked, the actual freeing of nodes and atoms can begin. This is done by tracing through a master list of the nodes (and atoms, in the case of the CU) that are contained in a processing unit. The CU initiates this freeing by sending a message to each of the PEs once marking has been completed. If a node (or atom) is unmarked, then it is returned to the free pool in its respective processing unit and its address is removed from the master list. When a PE finishes this collection, it returns a message to the CU. After the CU receives these messages from all of
the PEs, then garbage collection is complete and regular execution can resume. The automatic triggering mechanisms in every processing unit are reset at the end of this process. (Section 8 describes the modifications of these scheme that were necessary when property operators were added.)

In addition, a Lisp system call can be used to force garbage collection to occur. This allows the user to prevent garbage collection from occurring during a section of code with real time constraints.

7.2. Decreasing Simulation Time

By testing the interpreter with several Lisp programs (from [WiH81]), it was found that even programs of moderate size took too long to execute. If programs with realistic time execution requirements are to be run on the simulator, then the elapsed time response of the simulator must be improved. Three main areas were focused on in an attempt to improve performance. These areas are the interrupt/signal mechanism, the interprocess communication routines, and the multitasking scheduler.

The first area examined was the signal mechanism that was being used to send interrupts from the interconnection network to the processing elements. An option was created whereby the signal interrupt mechanism could be replaced by a fairly efficient polling mechanism. Since performance was greatly degraded by this change, it was felt that the signal mechanism was not a cause of poor execution times.

Second, an alternate interprocess communication mechanism was installed using shared memory instead of the 4.2BSD IPC routines. Pairs of circular buffers within the shared memory were used as communication channels. By
accessing the pointers for these buffers in the correct order, the need for
semaphores was eliminated. That is, no synchronization is needed in the
accessing of the shared memory. The final form of this implementation
involved using Vax 11/780 assembly language to perform the block reads and
writes via the buffers.

Finally, the scheduling mechanism itself was examined. Much of the
current execution time of the interpreter is spent in wait loops while a process
is waiting on information. By using a special local system call [Gob84], a
process could be forced to swap itself out instead of simply remaining idle. This
allows the next process to begin its operations sooner. By combining this
modification and the shared memory modification, slightly better results were
obtained. However, the execution time is still far from satisfactory.
8. A New Parallel Lisp Machine Model

Since after several enhancements the interpreter still used too much elapsed and cpu time, several tests runs with a 4 PE model were performed in an attempt to isolate the problem. In particular, the classic "n-queens" problem [WiH81] was used since it utilized most of the capabilities of the interpreter. Program runs with n=4 and n=5 took approximately three orders of magnitude longer than when run with the serial Lisp interpreter on the support system. While some performance degradation was expected, this result is clearly unacceptable. Examination of the results of these simulations showed that the PEs were being utilized fairly well, but that the quantity of information that was being transferred was unacceptably large. For example, execution of the 5-queens problem required over 42000 internal interpreter cycles, each of these typically transferring one or more nodes and atoms via the interconnection network. Having to route all node and atom transfers through the network is a great handicap to the interpreter. This is a problem not only in respect to reducing the elapsed time and cpu time that the interpreter requires, but also in respect to a true multiprocessing model. Due to the way in which the multitasking model has been developed, its communication needs accurately follow those of a true multiprocessing interpreter. Therefore, alternatives need to be explored in an effort to reduce the amount of communication necessary between the PUs.

If the data entities that are being transferred repeatedly with the current model were stored in such a way that these transfers could be reduced or eliminated, then the communication bottleneck problem would be greatly reduced. Since the nodes and atoms tend to be accessed by all of the PUs, it makes sense to make them equally accessible to all of these units. In addition,
the current interpreter already enforces data coherence by allowing only one unit to have control of a node. Only the unit in control of a node can modify that node. In effect, only the controlling unit of a node has write permission on that node. Therefore, it seems practical to keep the nodes and atoms in a limited shared memory instead of keeping them in separate local memories.

The question naturally arises of whether shared memory is a feasible alternative if the main effort is to avoid collisions. That is, will there be a reduction in the amount of overhead time for transferring information between a section of shared memory and a PU as compared to transferring information between two PUs via an interconnection network? In the case of a general MIMD machine, the answer to this is not clear cut. However, due to the structure of the interpreter, many of the problems with shared memories can be avoided.

This shared memory would not need any circuitry to handle write collisions since the interpreter already avoids such collisions. Furthermore, if every PU manages its own subspace of the shared memory, then there would be no need for a global memory manager beyond what the interpreter already provides. All that would be needed is a memory allocation structure within the shared memory and a semaphore mechanism so that this structure could be accessed with lockouts. Since the only time mutual exclusion is needed is when a particular unit is grabbing more memory, performance should not be harmed by the memory being locked occasionally.

With a shared memory for nodes and atoms, a PU would not need to submit a request via the interconnection network to get a node. Instead, it could directly access the shared memory. Requests for atom creation would still need to be passed to the CU, but this is nowhere near as common an
operation as node access.

Transfers could be reduced further by using shared memory for the frame data structures. Instead of transferring a frame to a PE for execution, all that would need to be done is transfer the address of the frame. This would reduce the actual amount of communication between the CU and the PEs considerably since the PEs would only access what they needed in a frame. Furthermore, this frame shared memory could be entirely separate from the atom and node shared memory since they do not need to interact directly.

In general, additional collisions can be avoided by partitioning the shared memory according to its various uses. With this scheme, one shared memory could partially replace the interconnection network and another could partially replace the CU/PE channels. These shared memories would contain nodes and atoms, and frames, respectively. (Other shared memory modules for additional uses are described later in the report). The amount of communication traffic can thus be made more manageable by the addition of shared memory. If the shared memory modules were dual ported, then only accesses to non-local quantities (such as a node controlled by another PU) would even have a chance of collisions.

Therefore, the interpreter will be modified to model the inclusion of shared memory. This is a tradeoff between trying to implement a fully general parallel interpreter and trying to implement an efficient interpreter. The approach taken here is a compromise. Note that the shared memory model is still a general MIMD model; only one of the features of the fully general model has been changed. Initially, only the node and atom shared memory will be implemented since this would remove a large burden from the interconnection network. Then a shared frame memory will be added. Since multiple processes
can share memory on the current test system, the response time of the interpreter should be reasonable. Indeed, shared memory can be used with the current interpreter model to simulate the communication channels.

This change in the model reflects back to the overall aim of the project. The purpose of this project is not just to create a parallel Lisp interpreter, but also to derive information about architectural features which affect the performance of Lisp on an MIMD system. The results that made the change in models desirable also have implications toward the types of MIMD machines that could efficiently run a parallel Lisp interpreter. In particular, the parallel computer should have some type of shared memory. This shared memory can either be of the conventional multiport type, or it can be a set of memory banks connected to the processing units through an interconnection network. This model description is in contrast to a machine having only local memories and an interconnection network with no capability for shared memory. The quantity of data transfers necessary for any Lisp program with a large proportion of non-numeric operations seems to make it unlikely that any performance gains could be achieved with such a model.

8.1. The First Shared Memory Model (Model 1S)

Before converting the interpreter software to reflect a new model, all of the software was converted into RCS (Revision Control System) [Tic82] format. This allows any model of the interpreter to be retrieved without storing multiple copies of the software. This method also allows for easy recovery of a partially developed model if program changes cause unexpected types of errors.
With this conversion complete, the interpreter was then converted to reflect the new shared memory model. This new model consists of a central control unit, a group of processing elements, and two shared memory modules. As will be explained later, no interconnection network is needed for this model. One of the shared memories contains all of the nodes and atoms. Now, when a processing unit needs a node, it can directly access the shared memory instead of going through a network. The PE address associated with a node now indicates which processing unit has write permission on a node. The shared memory does not need the capability to handle write conflicts since only one unit will ever have write permission on a particular node.

The second shared memory module contains the execution frames. By keeping these frames in shared memory, there is no need to pass an entire frame for execution. Instead, only the address of the frame needs to be passed to a PE. A PE will then access only the fields that it needs from a frame. Again, write conflicts will not occur since only one processing unit will have write permission on a particular frame. Distributed parsing can also take advantage of the second shared memory module. This required the addition of another flag field to the basic frame structure since the field that was previously used for this function, the function identifier field, is set before the frame is completely parsed.

One additional advantage of having the frames in shared memory is that large frames (frames with many arguments) will not seriously degrade the performance of this model of the interpreter. With the previous model, the entire frame needed to be transferred between processing units, so large frames were a severe limitation. Now, a large frame only causes an increase in shared memory usage. With proper partitioning of the shared memory, this increased
traffic should not degrade the performance of this model severely.

Reads from both of these shared memory modules could be accomplished using a simple bus arbitrator and a data packet communication scheme. Such hardware would most likely be simpler than a general interconnection network. This simplification is enhanced by the fact that this new model no longer needs to be able to accept and process interrupts. (The old model needed the capability to accept and process interrupts from the interconnection network in order to prevent deadlock).

The remaining function of the interconnection network, signaling the CU to create atoms for PEs, has been replaced by creating a third frame classification: a conditionally distributable frame. The operators that require atom creation that were distributable in the old model are the CONS and LIST operators. Both of these operators will require the creation of new atoms only if one of the arguments is a pure numeric. (Other operators that were not in this model were made conditionally distributable when the interpreter was expanded.)

When a conditionally distributable frame is ready for execution, the types of its arguments are checked to see whether it will require the creation of atoms. This check is very quick. If no atoms will need to be created, then the frame is added to the distribute queue. Otherwise, it is added to the non-distribute queue. The alternative to this scheme would be to allow any processing unit to create atoms. This would involve adding more synchronization operations and would complicate garbage collection. Such complication is not justified for the special case of pure numerics for these two operators.
With these modifications complete, the new model was again tested with the n-queens problem. It was found that the 8-queens problem could be solved in 26 hours of elapsed time. This solution required 1,785,104 internal cycles of the interpreter. With the previous model, only the 5-queens problem could be solved within this time span. Obviously, this is a substantial improvement for a problem solved using a branch-and-bound algorithm. The ratio of simulator execution time to the execution time on the system Lisp is now only about 350. Considering that the simulator must still perform a large amount of process switching and that it has not been run through an optimizing compiler, this ratio is acceptable. It is now reasonable to use the interpreter with non-trivial Lisp programs. Indeed, by using an optimizing compiler and by adjusting some of the process control parameters, it should be possible to reduce the ratio further.

### 8.2. Enhancements to Lisp Model 1S

The implementation of a shared memory model allows the addition of several operators that could not be implemented efficiently with any of the previous models. The two main areas of enhancement to this model are the property operations and the so-called "dangerous" operations. In addition, this model allows additional function types, such as FEXPRs and LEXPRs, to be added efficiently. These additional function types have more flexibility than the single function type (EXPRs) that had been implemented in the previous models. A listing of the final classifications of the Lisp operators for the parallel interpreter is given in Appendix A.
8.2.1. Properties

The first enhancement to be discussed is the addition of the Lisp property operators. In this new model, each atom has associated with it an ordered linked list of properties. Each of the entries of this list, referred to as property cells, contains a name field, a control structure field, and a pointer to the next element in the list. The control structure field is identical in composition to the standard argument field of an execution frame. Thus, the value of a Lisp property has the same format as any other datum in the interpreter.

Initially, the linked list of each atom is empty (the atoms have no initial property values). Property cells are allocated dynamically and are freed when a node is released by the garbage collection routines or when a property is deleted. The entire process of garbage collection is complicated by the addition of properties to the interpreter. Previously, an atom was considered to be a terminal point in a trace of a set of nodes. Now, an atom can cause other atoms and nodes to be traced as well. This is a major conceptual change, since it is now possible to have a closed loop within a trace path. Thus, care must be taken in the tracing of a list in order to avoid getting into an infinite loop (consider the simple example of an atom that has itself as one of its own properties). This problem was solved by modifying the tracing scheme and taking care with the order in which the trace flags are examined and set.

By taking advantage of the shared memory model, some of the property operations can be implemented as distributed functions. This is implemented in the interpreter by the addition of a third logical shared memory module. (In an actual MIMD interpreter, the divisions between the logical shared memory modules need not correspond to the divisions between the physical shared memory modules.) In addition to the standard property cells, a special location
must be set up in the node and atom shared memory module. This location is an atom that is used to store the necessary information for the properties of nil. Since there is no actual atom representing nil in the interpreter, this addition is necessary.

Before presenting the actual implementation method of the property operations, a more in depth examination is in order. This discussion is needed to illustrate the reasoning behind making some of the operations distributable and some of them non-distributable. First, mutual exclusion is required when adding or deleting a property to an atom. This is necessary to maintain the integrity of the property list. It is also desirable that this exclusion be on a per-atom basis, not on a per-module basis (i.e., locking all of the property shared memory module), so that property operations upon several different atoms can occur at the same time. Note that it is not necessary to require full exclusion when the property operation being performed is a search for a property. As long as the property list of an atom is not being changed, there is no problem with multiple search taking place concurrently.

If searches of a property list are more common than additions and deletions, then the mutual exclusion constraint should not harm system performance. Also, since at any one time only one addition or deletion can take place for the properties of a particular atom, these operations are implemented as non-distributed functions. The overhead of the additional semaphore operations that would be needed to make these operations distributable would likely not be justified by the frequency of their occurrence. However, there is nothing implicit in the model to prevent this from being changed later.

Mutual exclusion is implemented via the use of a pair of semaphores for each atom (and for the special property list associated with the nil atom).
These semaphores are referred to as "is_search" and "can_search." They indicate whether a search is in progress and whether a search can currently take place (respectively). The algorithms for the interaction of the routines with the semaphores are presented in Figures 4 and 5. In these figures, a line surrounded by curly braces is considered to be an indivisible operation.

```plaintext
can_search ← 0;
wait_for( is_search .equal. 0 );
/* do the operation (PUTPROP, REMPROP, OR DEFPROP) */
can_search ← 1;

Figure 4: Semaphore Interaction for Property Addition and Deletion

loop: wait_for( can_search .equal. 1);
is_search ← is_search + 1;
if (can_search .equal. 0) then
  {is_search ← is_search − 1;}
goto loop;
/* do the operation (GET) */
is_search ← is_search − 1;

Figure 5: Semaphore Interaction for Property Searching
```

Recall that the semaphores are specific to each atom, not common to the property shared memory module.
8.2.2. List Surgery Operations

There are several Lisp operations that are impractical in a multiprocessing architecture without shared memory. Among these are the operators that directly perform list surgery (e.g., RPLACA and RPLACD). However, for the shared memory model that has been developed, implementation of these operations is straightforward. However in a model without shared memory (such as the original interconnection network model), one would have to be concerned with distributing the operation to a specific processing unit. Just checking for this possibility would add several steps into the main distributor function, thus reducing the performance of the system.

Since these list surgery operations tend to be very dangerous even in serial Lisp, very few provisions have been made to protect the user in these situations. The only concession that has been made is that the value of an address field is set to nil before it is set to a new value. Due to the order in which the operations occur, if a node is accessed by another processing unit before the surgery is complete, a nil will be seen instead of another (possibly erroneous and incomplete) address.

The main pair of list surgery operators are RPLACA and RPLACD. These operations replace the address of the CAR and CDR of a list, respectively. Both of these functions are conditionally distributable.

The next list surgery operator is NCONC. This operation is similar to append, but it performs surgery upon the cells instead of copying the lists. This is a distributable function. Since NCONC may require modifying cells that were originally created by many different processing units, without shared memory this operation could require distribution to several processing units in sequence.
8.2.3. FEXPRs and LEXPRs

FEXPRs and LEXPRs are function types that are more flexible than the previously implemented function type (EXPRs). All of these function types are created via the DEFUN operator. A FEXPR provides the advantage of not evaluating its arguments and of allowing a variable number of arguments. A LEXPR evaluates its arguments, but still allows a variable number of arguments. An additional Lisp operator, ARG, is associated with LEXPRs.

The first step of implementing these function types is to provide a mechanism whereby the function cell indicates the type of function to which it is pointing. Since the same types of cells are used to indicate arrays and functions and since the field of these cells that indicates the type of an array is unused in the case of a function, this field can be utilized for the purpose of function type indication. For the case of LEXPRs, additional information is needed. The parameters of a LEXPR can be considered to be bound to the execution frame associated with the LEXPR. Thus, the execution frame structure must indicate whether or not it is associated with a LEXPR and must also indicate how many parameters with which it was invoked.

The next concern is the parsing of code containing these function types. The routine that performs function identification was modified to recognize these types and to pass the information back to the main parsing routine. This routine then uses this information to determine if the arguments of a user defined function need to be evaluated. If distributed parsing is to be performed, then the function cells must also be in a shared memory module since the PEs will need to access them. (The PEs will only read the function cells; they will never write to them). If a function has not yet been defined when an invocation of this function is being parsed, a warning message is given
to the user and the function is assumed to be of the standard form (EXPR). This should only occur in very poor Lisp code.

Finally, execution of these function types must be considered. This execution is split into two areas: the execution of the DEFUN creating such a function, and the execution of an actual invocation of such a function. The execution of a DEFUN for FEXPRs and LEXPRs is a straightforward expansion of the normal case. The only addition is that DEFUN must now properly set the function indicator flag in the associated function cell. Execution is somewhat different for both FEXPRs and LEXPRs.

When a FEXPR begins execution, all of its arguments are combined into a list. This is done by calling the same routines used to implement the Lisp operator LIST and by setting the argument count to one. At this point, execution can continue as if the function was a standard EXPR.

Execution of a LEXPR is more complicated. The only parameter of a LEXPR is the number of arguments. This is set in the proper environment level using the argument count of the execution frame. To access the actual parameters, the Lisp operator ARG must be used. ARG is passed a single number which indicates which parameter is to be obtained. Execution of an ARG begins by tracing back through the tree of execution frames until it finds one that is associated with a LEXPR. The desired argument is then obtained from this frame. Note that since all of the frames are in shared memory, this trace back is simple. If no father LEXPR frame is found, an error message is issued since an ARG has no meaning outside of a LEXPR. Finally, garbage collection has to treat LEXPR frames specially due to the way in which they store their parameters.
9. Testing of the Parallel Lisp Interpreter

The testing of the interpreter was divided into two phases. First, the binary was made available to a set of graduate students who were making heavy use of Lisp. (A manual is provided in Appendix B.) Second, a property-based object recognition program was used to test the ability of the interpreter to automatically parallelize Lisp programs.

9.1. User Testing

Although extensive testing of the interpreter had been performed during its creation, the possibility of operational errors always exists. These errors are of two kinds. The first kind, an error in the implementation of a function, is usually caught fairly quickly and is easily fixed. The second kind of error, a synchronization error between the PUs, is much more difficult to detect and subsequently fix. Most of the errors of this type are due to a frame being scheduled for one type of operation before it is ready. Substantial "real" programs are often required to locate such errors. It was felt that experienced Lisp users could provide valuable assistance in locating errors in the operation of the interpreter as well as in providing suggestions for improved operation.

To simplify the process of correcting such errors in the future, several enhancements were added to the interpreter. These enhancements are in the form of routines that check the status of the interpreter before and after every internal execution cycle. In particular, the validity of all internal execution frames and environment variables are checked, and all of the node and atom lists are checked for contamination. Although these checks increase the execution time of the interpreter by a factor of ten or more, they are invaluable for isolating subtle errors and race conditions.
9.2. Simulation of Object Recognition

To test the interpreter fairly, an object recognition program was acquired from local graduate students [HuF84]. It was felt that testing the interpreter using software developed by other users would give more accurate results since those users would not know how to take advantage of the parallelism of the interpreter. This test is presented simply as an example of the use of the interpreter. More extensive testing is in progress.

The test program uses property based operations for object recognition. This particular program was chosen since it fit the main criteria for a simulation program. First, its instruction mix is similar to sample programs provided by DMA (the Defense Mapping Agency). Second, it is based upon a task that is similar to the tasks suggested by DMA. Finally, its execution time is large enough to provide accurate results, but short enough to allow a reasonable number of simulation runs. A typical run took 15 to 20 minutes of elapsed time and took 2000-3000 internal interpreter cycles.

The program is only about 250 lines of Lisp code. However, the library is over 300 lines of Lisp expressions which load in the appropriate data. Many of the operations in the main program are complex combinations of property operations and function applications (such as MAPCARs, APPLYs, and FUNCALLs).

9.2.1. Test Data and Procedures

A set of eight random objects was used for the testing phase. Each of these objects was formed by randomly choosing properties from the master object database (using a uniform distribution). Since the objects so formed tend to be poor matches for any one item in the database, the recognition
program tends to do more work in order to try to identify the random object. The results of the simulations using the various objects were so close as to not justify testing with additional objects.

The interpreter was tested with the random objects for one, two, four, and eight PE models (involving two, three, five, and nine total PUs, respectively). Five trials were performed for each of the objects for each model for a total of 160 trials. Again, the results had such a low deviation as to negate the need for further tests. Indeed, some of the simulation runs yielded identical results within the precision of the model. The simulation runs took three weeks of elapsed time with simulations in progress approximately 20% of that time. Some of the simulations had to be repeated since the original size specifications for the shared memory area and some of the waiting lists proved to be inadequate. It was found that a realistic program could need 1.5M of shared memory and could have as many as 500 frames on some of the internal waiting lists due to the dependencies of the program.

Only a few measurements could be obtained without seriously affecting the operation of the interpreter. The most important of these was the utilization of the PUs. The scheduler of the interpreter already keeps track of whether a PU is busy or idle. Since this information is determined for each internal cycle of the interpreter, it was a simple matter to have variables keep track of the total utilizations of the PUs. This information led directly to the calculations of the speedups since there were no overhead cycles. That is, there were no tasks scheduled in the parallel interpreter that would not be scheduled for a serial Lisp interpreter.
9.2.2. Results

The results of the simulation are broken down into two phases. The first phase involves setting up global lists and reading in the database and the objects to be examined. This is a purely serial phase: no parallelism was observed. For tasks that involved repeated recognition of objects from the same database, typically only a small fraction of this phase would need to be repeated for each new object.

The second phase is the recognition phase. This involves examining the database and voting upon possible choices for the object. Substantial parallelism was observed for this phase.

The quantities that bear more detailed examination are presented in the following tables. In every table, the label “cycle” refers to an internal operation cycle of the interpreter. Typically one PU can execute one operator or parse one s-expression in a cycle. Since the number of cycles is almost entirely dependent upon the scheduling of tasks by the interpreter, the results are highly independent of any other tasks that the test system might be performing while the simulator is being run. Tables 1 through 5 present results for the two phases.
Table 1
Initialization Results (Serial Phase)

<table>
<thead>
<tr>
<th>Model</th>
<th>Cycles (every object, every trial)</th>
<th>Efficiency</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PE</td>
<td>122</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>2 PE</td>
<td>122</td>
<td>0.3333</td>
<td>1</td>
</tr>
<tr>
<td>4 PE</td>
<td>122</td>
<td>0.2</td>
<td>1</td>
</tr>
<tr>
<td>8 PE</td>
<td>122</td>
<td>0.1111</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2
Recognition Results: Parallel Phase

1 PE Model

<table>
<thead>
<tr>
<th>Unit</th>
<th>Cycle Averages</th>
<th>Average Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE 0</td>
<td>2107.5</td>
<td>0.5086</td>
</tr>
<tr>
<td>CU</td>
<td>3443.8</td>
<td>0.8310</td>
</tr>
<tr>
<td>Total</td>
<td>4144.0</td>
<td>0.6702</td>
</tr>
</tbody>
</table>
Table 3
Recognition Results: Parallel Phase
2 PE Model

<table>
<thead>
<tr>
<th>Unit</th>
<th>Cycle Averages</th>
<th>Average Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE 0</td>
<td>1594.0</td>
<td>0.5181</td>
</tr>
<tr>
<td>PE 1</td>
<td>1308.7</td>
<td>0.4253</td>
</tr>
<tr>
<td>CU</td>
<td>2703.0</td>
<td>0.8785</td>
</tr>
<tr>
<td>Total</td>
<td>3076.8</td>
<td>0.6082</td>
</tr>
</tbody>
</table>

Table 4
Recognition Results: Parallel Phase
4 PE Model

<table>
<thead>
<tr>
<th>Unit</th>
<th>Cycle Averages</th>
<th>Average Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE 0</td>
<td>1281.4</td>
<td>0.5199</td>
</tr>
<tr>
<td>PE 1</td>
<td>863.7</td>
<td>0.3504</td>
</tr>
<tr>
<td>PE 2</td>
<td>727.7</td>
<td>0.2952</td>
</tr>
<tr>
<td>PE 3</td>
<td>636.0</td>
<td>0.2580</td>
</tr>
<tr>
<td>CU</td>
<td>2273.7</td>
<td>0.9225</td>
</tr>
<tr>
<td>Total</td>
<td>2464.7</td>
<td>0.4708</td>
</tr>
</tbody>
</table>
Table 5
Recognition Results: Parallel Phase
8 PE Model

<table>
<thead>
<tr>
<th>Unit</th>
<th>Cycle Averages</th>
<th>Average Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE 0</td>
<td>1222.5</td>
<td>0.4940</td>
</tr>
<tr>
<td>PE 1</td>
<td>720.7</td>
<td>0.2912</td>
</tr>
<tr>
<td>PE 2</td>
<td>475.8</td>
<td>0.1923</td>
</tr>
<tr>
<td>PE 3</td>
<td>344.0</td>
<td>0.1390</td>
</tr>
<tr>
<td>PE 4</td>
<td>271.8</td>
<td>0.1098</td>
</tr>
<tr>
<td>PE 5</td>
<td>231.7</td>
<td>0.0936</td>
</tr>
<tr>
<td>PE 6</td>
<td>197.9</td>
<td>0.0800</td>
</tr>
<tr>
<td>PE 7</td>
<td>173.3</td>
<td>0.0700</td>
</tr>
<tr>
<td>CU</td>
<td>2278.8</td>
<td>0.9209</td>
</tr>
<tr>
<td>Total</td>
<td>2474.49</td>
<td>0.2664</td>
</tr>
</tbody>
</table>
Table 6
Speedups

<table>
<thead>
<tr>
<th>Number of PEs</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.340</td>
</tr>
<tr>
<td>2</td>
<td>1.825</td>
</tr>
<tr>
<td>4</td>
<td>2.354</td>
</tr>
<tr>
<td>8</td>
<td>2.398</td>
</tr>
</tbody>
</table>

By examining these tables, we can see that the results are quite good for the 1 and 2 PE models. The utilizations of all of the PUs are fairly high and reasonable speedups are obtained. (The utilization of the CU is high for all of the models since the CU grabs distributable tasks when it is idle). However, the results are not as good for the 4 and 8 PE models. While the utilization of PE 0 is about the same, the utilizations of the remaining PEs have dropped substantially below the utilization of PE 1 in the 2 PE model. This drop in utilization is reflected in the speedup results. The speedup of the 4 PE model is only moderately better than that of the 2 PE model. Indeed, the speedup of the 4 PE model is only a factor of 1.29 better than the 2 PE model even though the 4 PE model has 1.67 times as many processing units. The results for the 8 PE model are virtually the same as those for the 4 PE model. No significant gains were achieved by adding 4 more PEs to produce an 8 PE model.

In addition, the 4 and 8 PE models would tend to have more problems with shared memory conflicts. Even though the structure of the interpreter prevents logical shared memory conflicts, there might still be physical conflicts. These conflicts would be caused by logically independent sections of shared
memory sharing the same physical device. This is especially true if large
capacity memory chips are used to construct the shared memory. Proper
interleaving schemes could reduce this problem to some extent, but it is
doubtful that it could be entirely eliminated.

No feasible method of measuring shared memory usage was obtained for
use with the simulator. Since the simulator uses true shared memory, the only
way to measure its access patterns would be to interrupt the interpreter at
every instruction, examine the instruction being executed, and determine if this
instruction was referencing shared memory. Not only would this vastly increase
the time necessary to perform a simulation, it would also affect the timings of
the interpreter and thus invalidate the results.
10. Further Research and Recommendations

There are several areas for possible improvement of the interpreter. Some of these improvements are only applicable for the use of knowledgeable programmers (i.e., users who are aware that they are executing a Lisp program on a parallel machine and would like to take advantage of that fact). Other improvements are specific to various parallel computer architectures or algorithm characteristics. However, these improvements all share the common feature of being relatively minor extensions of the current interpreter. (Appendix C describes the overall structure of the interpreter software and it should be used as a guide in making any changes.)

10.1. Enhancements for Knowledgeable Users

Throughout the development of the parallel Lisp interpreter, it was assumed that the user of the interpreter would be running programs that were originally designed to run on a serial Lisp interpreter. The only differences that such a user would notice between the functionality of a serial interpreter and the parallel interpreter are that the AND and OR operators would function somewhat differently (evaluating all of their operands concurrently) and that the careless use of global variables could cause data coherence errors (due to the way in which the environment is accessed). However, a more experienced user might actually want to take advantage of the underlying architecture of a parallel computer.

The simplest addition for such an experienced user would be the addition of a function to force the parallel execution of other functions. Also, this function should be of such a form as to be compatible with serial interpreters (for portability). Such a function is actually trivial: the LEXPR
creates a function that can execute a number of functions concurrently on the parallel interpreter. In addition, this function will still allow any programs using it to be run on a serial interpreter.

Another addition to the interpreter that might be desirable for the use of a knowledgeable programmer is a set of synchronization operations. Since counting semaphores are already used within the supporting code for property operations, this code could easily be used as a basis for adding explicit synchronization abilities.

10.2. Internal Modifications

The main area of serialism still remaining in the interpreter involves the ARRAY and STORE operators, as well as the code that accesses elements of arrays. Throughout the development of the array access section of the interpreter, it was assumed that most array accesses would be done sequentially (such as in a PROG loop) instead of concurrently. Thus, the array operations were made non-distributable so as to avoid the need for synchronization operations. (Recall that the opposite situation is present for property operations. Concurrent execution is likely for property operations so the GET operator is distributable and counting semaphores are used for synchronization with the other property operators). If it turns out that a specific set of user programs does indeed access array elements concurrently, then array operations could be made distributable too.
10.3. Hardware Specific Modifications

Depending upon the specific parallel machine that the parallel interpreter is to be run on, there are enhancements that could be made to the interpreter in order to increase the execution speed and/or the power of the interpreter. In the following sections several such enhancements are presented and their merits are discussed.

10.3.1. PE Task Assignment

If the configuration of a specific parallel computer is such that shared memory is available but the access cost of this memory is high (such as a high conflict rate), then an alternative approach would be to allow more than just the basic Lisp operators to be distributed to the PEs. By allowing user defined functions to be distributed (thus keeping all of the environments created by such a function local to a PE), the use of shared memory could be reduced. Such user distributed functions could also involve compiled Lisp code and/or access routines for special purpose hardware (such as an FFT engine). However, the cost of transferring any results from non-shared memory to the shared memory must also be considered.

10.3.2. Multi-user Operation

For some image processing tasks (such as creating maps from digitized images), it might be desirable to have several users working concurrently with the same data. If each PE is attached to some type of workstation (such as a terminal and a graphics display), then with supporting software modifications the interpreter could run in a multi-user mode. Each PE would need a supporting routine for communication with the Control Unit. Then the main
I/O routines could use an additional argument to indicate and direct communication to and from the workstations.

This scheme is actually a form of intelligent resource management. The structure of the interpreter performs the management to some extent, while allowing the users to add to the basic capabilities.

10.3.3. Non-shared Memory Machines

The problem of poor performance on interconnection network based parallel machines was discussed in detail earlier. However, there are still situations where it might be advantageous to implement a special version of the parallel interpreter for such a machine. For example, if user functions are made distributable or if the individual PEs have workstations (both possibilities that have been discussed previously), then the number of accesses via the interconnection network could be drastically reduced.

Again, this is a problem of intelligent resource management. The interpreter provides a basic set of capabilities, but the users must take responsibility for using these capabilities efficiently. This problem is more severe for an interconnection network based machine than it is for a shared memory based machine.

10.4. Recommendations

In the preliminary tests that were performed to exercise the interpreter, significant speed gains were realized only for the cases in which a very small number of processors was used. Clearly more extensive testing is needed to assess the extent to which automatic parallelization of existing Lisp programs is feasible, and to determine the amount of parallelism which is likely to be
present in such programs. The limitations on the parallelism found in the test program can be attributed to the "Lisp style" in which it was written. Typical Lisp style involves creating algorithms containing a large number of functions with a small number of arguments each. This results in an execution pattern with many restrictive dependencies. A better style for parallel Lisp would be to create algorithms containing a small number of functions with a large number of arguments each. This would remove some of the restrictions which reduced the efficiency of the interpreter.

If one specifically wanted to take advantage of the power of a parallel machine and also wanted to be able to use Lisp, then the approach developed here could be quite useful. It is certainly true that, because of the generality of the machine model, it is unlikely that the parallel Lisp simulated here will be as fast as a dedicated machine designed specifically to handle Lisp structures and operations. However, the interpreter can serve as a tool to study the way in which parallelism can be extracted from Lisp programs and to determine the attributes which a more specialized parallel Lisp architecture should have.
11. References


Appendix A:
Distributions of Operators and Data Types

Table A1
Distributable Operators:
(CU or PE, data independent)

<table>
<thead>
<tr>
<th>abs</th>
<th>and</th>
<th>append</th>
<th>atan</th>
<th>boole</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>cdr</td>
<td>cos</td>
<td>difference</td>
<td>eq</td>
</tr>
<tr>
<td>equal</td>
<td>exp</td>
<td>expt</td>
<td>fix</td>
<td>float</td>
</tr>
<tr>
<td>get</td>
<td>greaterp</td>
<td>last</td>
<td>length</td>
<td>lessp</td>
</tr>
<tr>
<td>log</td>
<td>lsh</td>
<td>max</td>
<td>min</td>
<td>minusp</td>
</tr>
<tr>
<td>nconc</td>
<td>not</td>
<td>null</td>
<td>numberp</td>
<td>or</td>
</tr>
<tr>
<td>plus</td>
<td>quotient</td>
<td>remainder</td>
<td>reverse</td>
<td>sin</td>
</tr>
<tr>
<td>sqrt</td>
<td>times</td>
<td>zerop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A2
Conditionally Distributable Operators:
(CU or PE, data dependent)

<table>
<thead>
<tr>
<th>assoc</th>
<th>cons</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>delq</td>
<td>list</td>
<td>member</td>
</tr>
<tr>
<td>rplaca</td>
<td>rplacd</td>
<td>subst</td>
</tr>
</tbody>
</table>

Table A3
Non-distributable Operators: (CU only)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add1</td>
<td>apply</td>
<td>arg</td>
<td>array</td>
<td>atom</td>
<td></td>
</tr>
<tr>
<td>boundp</td>
<td>comment</td>
<td>cond</td>
<td>defprop</td>
<td>defun</td>
<td></td>
</tr>
<tr>
<td>eval</td>
<td>explode</td>
<td>funcall</td>
<td>function</td>
<td>gensym</td>
<td></td>
</tr>
<tr>
<td>implode</td>
<td>load</td>
<td>mapcar</td>
<td>minus</td>
<td>prin1</td>
<td></td>
</tr>
<tr>
<td>print</td>
<td>prog</td>
<td>putprop</td>
<td>quit</td>
<td>quote</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>readch</td>
<td>remprop</td>
<td>return</td>
<td>set</td>
<td></td>
</tr>
<tr>
<td>setq</td>
<td>store</td>
<td>sub1</td>
<td>terpri</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A4
Other operators: Implemented via user defined functions

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>mapcan</td>
<td>prin</td>
<td>prog2</td>
<td></td>
</tr>
<tr>
<td>progn</td>
<td>sort</td>
<td>sortcar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table A5
Data Type Distributions

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>within frame</td>
</tr>
<tr>
<td>floating point</td>
<td>within frame</td>
</tr>
<tr>
<td>atom</td>
<td>shared memory</td>
</tr>
<tr>
<td>node</td>
<td>shared memory</td>
</tr>
<tr>
<td>function cell</td>
<td>via shared memory</td>
</tr>
<tr>
<td>property</td>
<td>via shared memory</td>
</tr>
<tr>
<td>array</td>
<td>CU memory</td>
</tr>
</tbody>
</table>
**Appendix B:**

**Parallel Lisp Manual Page**

MLISP(1) UNIX Programmer’s Manual MLISP(1)

**NAME**
m lsp - Multitasking MACLISP Lisp interpreter.

**SYNOPSIS**
mlisp [-h[elp]] [-i] [-s] [-S] [ filename [filename] ...]

**DESCRIPTION**
Mlisp is a multitasking Lisp interpreter. Its purpose is to investigate the automatic parallelization of the Lisp programming language for an MIMD (Multiple Instruction stream - Multiple Data stream) computer. Both shared memory and Unix domain sockets are used for interprocess communication.

The command line options are:
- **-h** Display a summary of the command line options.
- **-i** Manual input mode. After all named files are loaded, the standard input will be read for additional input. The reading will stop upon an EOF.
- **-s** Look for a file called “.mlisprc” in the current directory and load it if it exists. This file is often used to load a start up command sequence.
- **-S** Silent mode. This suppresses the display of the normal startup messages. These messages are normally in a file called “mlisp.message” in the directory that is used for supporting the interpreter.

The interpreter normally operates in a batch job fashion. Upon invocation, the interpreter processes any command line arguments and displays a start up message. Files listed after any command line options are then loaded and executed. All files that are loaded in this way are echoed to the standard output before the multitasking environment is established. Lisp programming constructs can be used to simulate an interactive environment. Due to the complex control structure, no error recovery is attempted.
If no arguments or files are given, the interpreter displays an initial command option summary and internal system call summary. It then prompts for input from the standard input channel. Input ends when an EOF is encountered. If there are command line options but no files are given, then the options will be executed as required but the interpreter will not prompt for any input. This is useful for checking the validity of a start up file and for obtaining a listing of the internal system calls.

CAVEATS
Due to the size of the interpreter and the vast amounts of interprocess communication that it must perform, this Lisp runs approximately 250 times slower than the system Lisp.

SEE ALSO
lisp(1), socket(2)

AUTHOR
Thomas Rice
Appendix C:
A Parallel Lisp Interpreter: Code Overview

This document provides a brief overview of the various files and routines that comprise the parallel Lisp interpreter developed under DMA contract by T. Rice during the time frame 11/83 through 11/84. This overview is not intended to be a tutorial on Lisp or on the theory of the parallel interpreter itself. Rather, this overview is intended to serve as a guide to anyone who wishes to gain a deeper understanding of the parallel interpreter or anyone who is charged with maintaining or updating the parallel interpreter.

The interpreter is written in the 'C' programming language under 4.2BSD Unix. Several of the features of the interpreter are specific to the development site. First among these is the implementation of shared memory by a specific system call which makes the (shared) text space writable. Second, sections of code are written in Vax 11/780 machine language to reduce execution time and to access semaphores via indivisible operations. If the interpreter is to be ported to another machine, these changes must be accounted for.

Finally, since the scheduler on the Vaxen on the Engineering Computer Network only allows two processes per uid to be in the run list concurrently, a special provision was made to allow the interpreter to run. The interpreter must be run from an account with a uid less than 100. A special account was set up with just this in mind.

Following are the specific descriptions of the files and the routines therein.

ldefs.h
This file contains all of the definitions needed for the interpreter. In addition, there are several flags (set via #define's) within this file. Most of these are obvious. Some of the unobvious ones are:

SERIAL - used for initial serial testing
TARSHARE - use shared memory for communication
TARSOCK - use IPC sockets for communication (faster). TARSHARE and TARSOCK are exclusive - one and only one of these should be defined.
INET - establish an interconnection network
FINDFREED - do massive checking of the state of the interpreter for every internal cycle. This is for finding bugs of pan-galactic proportion.

Also, the sizes of the structures for atoms, nodes, and frames must always be a multiple of four. This is required for the semaphores contained therein, since the semaphores must be aligned on long word boundaries. (Thus the start of a semaphore variable must also be offset by a multiple of 4 bytes from the beginning of such a structure.)

array.c
This file contains the main routine and support functions for Lisp array operations. This includes the ARRAY and STORE operators as well as array access via functions.

cond.c
The support routines for parsing and executing conditional statements are in this file. In particular, the routines to parse conditional statements and
return the appropriate result are contained.

cu_tf_pe.c
This file contains the support routines to allow communication between the control unit and the processing elements. In particular, the CU side of the communication is handled by the routines in this file. Most of the routines in this file are called by routines in lparse.c or lexecute.ctrl.c. The routines for the other side of the communication are in pe_tf_cu.c.

cvalexpr.c
This file contains the support routines to create the initial atoms and s-expressions from the (initial) linked-list text. In particular, the main routine creates the proper node structure for constant s-expressions. Some of these functions are recursive.

danger.c
This file contains the routines for performing the "dangerous" lisp operations (i.e., list surgery). ALL of these routines require the shared memory model. Nconc, delq, rplacq, rplaca, and delete are considered to belong to the list surgery class of operations.

defun.c
This file contains the routines to create function cells as well as to instanciate lambda execution when a function is referenced. This includes the necessary environment manipulations. Note that array access appears to be a function call in Lisp, so array access is also handled by these routines. The arg operator is also implemented in this file.

environ.c
Environment access and support for parallel lisp is done by the routines in this file. This includes the low level functions for setting a variable in an environment and getting the value of a variable from an environment. The variables t and nil are always defined. The environment can be searched level by level automatically or the search can be restricted to one level. A similar situation applies to setting a variable. The gensym operator is implemented in this file.

equal.c
This file contains routines to determine if two entries are equal (or eq). All explicit support routines are also in this file. Some of these routines are recursive. The member and assoc operators are implemented in this file.

eval.c
The file contains all the major support routines for the eval and funcall operators.

gcollect.c
All support routines for garbage collection of nodes and atoms are contained in this file. The global define FINDFREE can be set in ldefs.h in order to turn on massive tracing of garbage collection. This should only be done as a last resort in trying to find nasty bugs. The order of operations and the synchronization procedures for these routines is VERY strict. Be careful if you change even the slightest detail.

inter.c
This file contains the routines that support the simulation of an interconnection network. It also contains the main process for the network. The network is simulated by a separate process so that the model does not
explicitly depend upon the network implementation. Interrupt priorities are critical: they must not interrupt the Read and Write routines, but they must be able to interrupt general execution. Study the current priorities carefully before changing anything. (This file is not used in the shared memory model).

**lambda.c**

Lambda function parsing, scheduling, and execution are handled by routines in this file. Many of these routines are called by routines in other files (such as cond.c, defun.c, and prog.c).

**lexecute.c**

This file contains the execution routines for the parallel execution of a Lisp program that has been previously reduced to frames by lparse(). The entry and exit points of these routines are all the same. The specific operators supported in this file include printing, simple variable setting, comments, input reading, and the implode and explode operators.

**lexecute.ctrl.c**

This file contains the controlling routines for the parallel execution of a Lisp program that has been previously reduced to frames by lparse(). This is the main scheduler and thus the heart of the interpreter. If you don’t fully understand it, don’t worry - I’m not sure I do either anymore. There are two automatically initialized arrays in this file: the positions of elements in this file must agree with the indices assigned to operators in Idefs.h.

**lexecute.list.c**

This file contains the execution routines for the parallel execution of a Lisp program that has been previously reduced to frames by lparse(). This file contains list operations such as car and cdr. Routines to force a numeric argument into an atom and to copy a list given a starting point are also included.

**lexecute.logic.c**

This file contains the execution routines for the parallel execution of a Lisp program. In particular, it contains the execution routines that return a logical result (t or nil) or operate on logical operands (predicate operations).

**lexecute.math.c**

This file contains the math execution routines for the parallel execution of a Lisp program that has been previously reduced to frames by lparse(). The arithmetic and general mathematical operators are in this file.

**lisp.multi.c**

Lisp programming language on a generalized parallel machine. This is the main procedure file. All shared memory is initially created at the beginning of this file since shared memory is actually writable text space. Argument parsing, process creation, and initial signal handling routines are included in this file.

**lparse.c**

This function takes a Lisp program that has been read into a linked-list and creates the frames for the execution of the program. Since parsing can be distributed, synchronization variables are used to be sure a frame is not executed before parsing is complete. Most global variable initialization is done in this file.
lparse.io.c
This file contains the input and output routines to support the functions in lparse.c.
lparse.util.c
This file contains the memory allocation and management routines to support the functions in lparse.c.
lread.c
This file contains routines to open and read in a file into a linked list of text blocks. This is for the READ operator. This uses dynamic memory allocation.
lstring.c
This file contains the support routines to create the initial atoms and s-expressions. This includes the tasks of atom and node creation and management. Several related memory allocation and management routines (for atoms, nodes, and properties) are also in this file.
lsystem.c
This file contains the execution routines for the various lisp functions that are specific to this interpreter. Many features can be accessed by system calls instead of by having to recompile the interpreter. Among the more useful system calls are automatic array reading and writing to and from Unix files.
mapcar.c
This file contains all of the main routines and support routines for the mapcar and apply operators. Garbage collection support routines for mapcar are also included.
pe_tf_cu.c
This file contains the support routines to allow communication between the control unit and the processing elements. In particular, the PE side of the communication is handled by the routines in this file. The routines for the other side of the communication are in cu_tf_pe.c.
preeexec.c
This file contains the routines to convert numbers represented by atoms into pure numerics. This decreases the amount of information that needs to be passed and processed by the processing units.
prog.c
This file contains the routines for the setup and execution of block and loop operations. This includes the prog, prog2, progn, go, and return operators. Again, much of this is similar to the processing in cond.c, defun.c, and lambda.c.
properties.c
This file contains the support routines for managing the Lisp property operators. The actual memory allocation support is in lstring.c.
readin.c
Routines to open and read in a file into a linked list of text blocks are in this file. There is critical interaction with global variables by these routines. These routine use dynamic memory allocation.
shared.c
The definitions and routines for interprocess communication via shared memory are contained in this file. In addition, all shared memory allocation
routines are in this file. Many of these routines require mutual exclusion (which is done via Vax 11/780 assembly language).

util.c
This file contains support routines for a parallel Lisp Interpreter. In particular, format conversion, error handling, and statistics reporting are all in this file (some statistics reporting is in lparse.c and lexecute.ctrl.c).