CUCS-107-84

||PSL: A Parallel Lisp for the *DADO* Machine*

Michael K. Van Biema
Mark D. Lerner
Gerald Maguire
Salvatore J. Stolfo

Columbia University

February 6, 1984

Abstract

We describe a system level programming language and integrated environment for programming development on the *DADO* parallel computer. In addition a set of language constructs augmenting LISP for programming parallel computation on tree structured parallel machine are defined . We discuss the architecture of the *DADO* machine and present several examples to illustrate the language. In particular we describe how thelanguage provides an integrated approach to the problem of parallel software design. Parallel algorithms may be designed analyzed on a sequential machine under simulation and then simply recompiled to run on a parallel machine. In concluding sections we outline the implementation using the Portable Standard LISP Compiler.

*This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-82-C-0427 and the New York State Science and Technology Foundation, as well as grants from Intel Corporation, International Business Machines, Hewlett Packard, Digital Equipment Corporation, Valid Logic Systems, and Bell Laboratories.

Table of Contents

1 Introduction

This paper describes the development of a system level programming language which greatly reduces the difficulty of programming tree structured machines. We view each node of the parallel machine as an abstract datatype which ay any point in time contains data and code; moreover, primitives and high-level functions are provided to manipulate these data objects. The approach provides an integrated environment for software development and debugging. We call this language ||PSL, standing for Parallel Portable Standard Lisp.

We discuss the difficulties of providing a system level programming language for a parallel machine and outline the difficulties encountered with a previously used language.

This language has been developed in the context of the *DADO* project. The final goal of this project is the development of both hardware and software systems designed for the rapid execution of Artificial Intelligence (AI) systems. Two AI systems are currently under development (see [Miranker 84] and [Taylor 84] for details):

- *Herbal*, named in honor of Herbert Simon and Allen Newell, inventors of the AI production system paradigm. *Herbal* is a parallel production system language which uses a modification of the Rete-match algorithm [Forgy 82].
- LPS, A *L*ogic *P*rograming *S*ystem which is a logic based language facility [Taylor 83].

This paper examines the system level programming language in which these systems are being developed. Before giving the language specification we briefly describe the structure and operation of the *DADO* machine.

2 The *DADO* Machine

DADO is a fine grain, binary tree-structured machine which will eventually contain many thousands of processing elements (PEs). A 1023 node machine is under construction with a planned completion date within a year. There is currently a 15 node prototype functioning at Columbia University.

The *DADO* machine works in conjunction with a host processor, which can be any machine with sufficient capacity to support the PSL [GRISS 81] environment. The host processor, also known as the Control Processor (CP), functions in several capacities:

- It works as a file server.
- It stores parts of the symbol table which are not needed in each PE.
- It provides a convenient user interface.
- It runs a high level debugger.
- It gathers statistics on the performance of programs as they are executed in the tree.
- It runs the simulator.

Within the *DADO* machine, each PE is capable of executing in either of two modes. In the first, which we will call *SIMD mode* (for *s*ingle *i*nstruction stream, *m*ultiple *d*ata stream [Flynn 72]), the PE executes instructions that are broadcast by some ancestor PE within the tree. A SIMD processor is in either an enabled or a disabled state. When enabled it executes the instructions received from its parent and passes them along to its children; when disabled it does not execute them locally, but continues to pass them on.

The second mode is *MIMD mode* (for *m*ultiple *i*nstruction, *m*ultiple *d*ata stream). When a *DADO* PE enters MIMD mode, its logical state is changed in such a way as to effectively "disconnect" it and its descendants from all higher-level PEs in the tree. In particular, a PE in MIMD mode does not receive any instructions that might be placed on the tree-structured communication bus by one of its ancestors. Such a PE may, however, broadcast instructions to be executed by its own descendants, provided they are in SIMD mode.

The *DADO* machine can thus be configured in such a way that an arbitrary internal node in the tree acts as the root of a tree-structured SIMD device in which all PEs execute a single instruction (on different data) at a given point in time. This flexible architectural design supports *multiple-SIMD* execution (MSIMD). Thus, the machine may be logically divided into distinct partitions, each executing a distinct task, and this is the primary source of *DADO's* speed in executing a large number of primitive pattern matching operations concurrently. This also generalizes to full MIMD mode as each node may function independently, forming its own degenerate tree (see [Stolfo 83] for details).

The host processor interfaces directly with the *DADO* root processor and functions in a manner analogous to a *DADO* PE in MIMD mode. When a PE enters MIMD mode it becomes the CP of its own subtree. The root PE executes all SIMD instructions locally as well as broadcasting them to its descendants. In this manner the semantics are the same as when the SIMD instructions are broadcast by the host processor. Careful attention has been paid to the design of the language to maintain consistency at the two points of local asymmetry in the tree: the root and the leaf nodes.

3 PPL/M: A First Pass

The antecedent of the language developments described in this paper was parallel PLM (PPL/M) [Stolfo 84]. It was our first parallel systems programming language. The language has been used to implement a small parallel production system interpreter and many of our comments are based upon this.

PPL/M suffered from many problems which diminished its effectiveness as a tool for the implementation of high level parallel algorithms. Nevertheless, the work with PPL/M suggested many language improvements. These improvements have been incorporated the new ||PSL language, and others were already in Lisp.

PPL/M was implemented using Intel's existing high level PL/M language as the core language. It is a standard block structured language based on PL/I, to which we added the necessary parallel processing primitives. It provided several primitive communication functions, which are still used in the ||PSL language. These are described in detail later. The difficulties with PPL/M, described below, include limitations in creating data structures, problems passing data structures around the tree, limited calling conventions, and the unavailability of recursion.

The most serious of these difficulties is the inability to pass arbitrary data structures around the tree. In a parallel environment, ease of specifying communication of data, as well as the efficiency of communication, are of the utmost importance. Later we present a simple and efficient solution to this problem.

As an example of the limitation of PPL/M communication constructs, the language does not directly support the parallel assignment of arbitrary expressions. Instead it requires that the user provide a detailed specification of how the data transfer will occur. For example, it is frequently necessary to transmit a list from one processor to many others, but PPL/M only permits the broadcasting (or reporting) of a single byte. The ||PSL language permits the user to do this with one statement.

Another limitation of PPL/M is the requirement that procedure invokations be rigidly defined within the block structure, and consequently it is difficult to write data driven programs. The new ||PSL language uses a more general calling mechanism that permits execution of any precompiled function at any time. For example, it is relatively simple for one function to pass another function a list of candidate functions, and the second level function can execute any of these.

The PPL/M language had other limitations as well. The availability of dynamic data structures, which we consider fundamental to development of algorithms, was not part of the language. The user was required to develop these mechanisms, and this increased development time while decreasing reliability.

Finally, the programming environment was powerful but too slow. It was limited to a development system with an in-circuit emulator. It took as much as a half hour to compile, link and execute a program. We note, on the other hand, that we used an old development system, that faster systems are now available, and this would have reduced development time.

The ||PSL system, on the other hand, is far more powerful and faster. It allows the programmer to use either of two simulators: a machine simulator at the language level, or a single processor simulator at the machine instruction level. Moreover, ||PSL programs, when executed on the hardware of the DADO machine, can make use of the function cell to embed debug and trace functions into the program.

4 ||PSL: Language Overview

As stated earlier our goal is to provide a system level programming language that sufficiently reduces the complexity of programming the *DADO* machine so that higher level programming languages and AI systems may be easily implemented. A natural choice for such a language is LISP. The University of Utah compiler generation tools make such a choice even more attractive.

LISP is an appropriate language for several reasons: it is both interpretable and compilable, it encourages independent small modular functions, and it has traditionally been used as the systems programming language for AI.

There are good reasons for this. For example, we have found that general list manipulation features are essential, and if these features are missing from the language, the user is obliged to provide them. Finally, the ability to manipulate the program as a data object is a valuable tool which is not available in most conventional languages.

However a full LISP implementation in each PE is not practical due to the limited storage capacity of the PEs. We therefore make the following restrictions on LISP data types allowing only:

- atoms
- s-expressions
- integers
- inums

These are known as LISP Objects. We have also extended the LISP language with two features provided by the PSL system. The first, RLISP, is an Algol-like syntactic form which is translated into typical LISP syntax by a preprocessor supplied with the system. In what follows we present all of our specifications in both RLISP and normal LISP form in order to give the reader the flavor of RLISP. The second extension is SysLISP which, unlike LISP, allows access to the actual machine bits, bytes and words. Most of the kernel functions such as the allocator and the garbage collector are written in SysLISP. The high level interface to the parallel communication functions is implemented in SysLISP as well.

4.1 Semantics of Parallel Communication Functions

Our language design decisions were based on the following view of the machine. Each node within the tree is viewed as an abstract datatype which at any point in time contains its own data and code. This abstract data type communicates with the rest of the tree by means of its current functional value. This is always the most recently computed value. The rest of the tree communicates with the abstract datatype by calling its functions, passing external values as parameters to these functions, as well as specifying which internal values are to be operated on.

In PSL all global variables must be declared explicitly. All of the global variables declared *SLICE* reside in each PE. All fluid variables must also be declared in a like manner. Undeclared variables are presumed to be locals. They are allocated on the runtime stack at the time of function entry. This is a major difference between interpreted and compiled code and we retain it for reasons of efficiency. We also feel that, due to the nature of parallel procedure execution, programmers are well advised to limit themselves to the use of globals and locals.

To declare a sliced global one calls the sliced declaration function as follows:

RLISP: `slice global` *<variable-name>*;

LISP: `(slice global` *<variable-name>*`)`

A function may also be declared with the slice attribute. Such functions are stored in compiled form in all PEs. All global and fluid declarations must precede reference to the variables, since different code will be compiled in these cases. The syntax for a sliced function definition is:

```
slice <function-type> Procedure <function-name> (<args>) <function-body>;
(slice <function-def> <function-name> (<args>) <function-body>)
```

In the above, *<function-type>* is any of the usual LISP function types (i.e. expr, fexpr, etc). <function-def> is any of the corresponding definition functions (i.e. de, df, etc.).

4.2 Parallel Procedure Activation

There are two alternatives for parameter passing for sliced functions. In the default case the evaluation of parameter forms proceeds as:

1. evaluate the form in the root processor
2. broadcast the value to the descendants
3. store in the parameter area (registers).

The sliced function is then invoked in the local PEs.

The *slice* parameter is the second case. In this case the variables used in the parameter form must be already present as global variables in all PEs. The parameters are broadcast to the PEs, which then locally evaluate the parameters. These are subsequently used in the evaluation of the sliced function. Sliced parameters must be declared. For example:

```
slice expr Procedure f (slice: arg1, arg2, slice: arg3); ...;
(slice de f((slice arg1) arg2 (slice arg3))....)
```

where arg1 and arg3 are sliced, and therefore are evaluated in each PE prior to function execution. Arg2 is evaluated in the root, and this value is used in all PEs.

4.3 The Parallel Communication Functions

In this section we define a complete set of communication primitives for global and local communication in the tree.

Global communication is accomplished by means of two instructions, one to send data down from the root, and another to send data up to the root. The *broadcast* instruction allows any MIMD processor to transmit a local value descendant SIMD PEs; these PEs forward the information to their children. The data propagates throughout the entire tree in an instruction cycle.

Sending data in the other direction, from a processor within the tree to the root, is accomplished with a *report* instruction. This sends a value from a single enabled PE to the root.

An additional instruction, *resolve*, selects one processor from the currently enabled processors. It is often used prior to a report instruction. The selection is made on the basis of an integer comparison between the values supplied by each PE. The minimum or maximum valued PE is selected depending on the form of the instruction. Ties are resolved in favor of the lowest numbered PE based on an inorder tree numbering. In all cases this resolve is completed within O(log n) times (where n is the number of PEs in the tree). It should be noted that a semi-custom integrated circuit has been designed for the 1023 element version of the machine which executes these operations in one machine instruction cycle.

The ||PSL language provides a special function for algebraically associative functions. Named the TAO function (for tree associative operation), it applies its functional argument to three inputs. One input is the local value from the PE, and the other two are the values returned by its children. This allows logarithmic time associative operations. If a node is SIMD disabled its value does not participate in the operation.

Local communications augment the above global communication operations. Local communication is accomplished by the *send* and *recv* instructions. In the case of the receive instruction, a value may be received from the parent PE or either of the children PEs. The send instruction is more limited -- it sends a value only to the children. Sending to a parent is not allowed as the semantics would be not be clear if both children tried to send simultaneously.

Parallel communication may also be accomplished by *implicit* communications, which is done by use of the LISP primitive *SETQ*.

The semantics of SETQ are defined to permit manipulation of the abstract datatypes stored in the descendant processors. There are three factors that determine the effect of a SETQ. These are:

1. State of the processor: MIMD or SIMD
2. The destination variable: MIMD or SIMD
3. The source variable: MIMD or SIMD

State of Processor	Destination Variable	Source Variable	Effect on MIMD PE	Effect on SIMD PE
MIMD	Simd	Simd	none	Local assignment in all PEs
	Simd	Mimd	none	Root value assigned to all PEs
	Mimd	Simd	Simd variable from Simd half assigned to Mimd variable	none
	Mimd	Mimd	Mimd destination assigned value of Mimd source	none
SIMD	Simd	Simd	not allowed	Simd value assigned to Simd variable
	Simd	Mimd	not allowed	not allowed
	Mimd	Simd	not allowed	not allowed
	Mimd	Mimd	not allowed	not allowed

This produces The exact syntax and semantics of the communication functions if given below:

(Broadcast <form>)

Places the value of the locally evaluated *form* into the Input variable of all descendant processors. The function returns T.

(Enable) Transfers a 1 into the enable bit of all SIMD PEs, and thereby permits them to execute instructions which are communicated from above. The function returns T.

(Disable) This disables a SIMD processor and causes it not to execute instructions locally, but to continue passing them to its children. It continues in this mode until receipt of an enable instruction. The function returns T.

(Min-Resolve <form>) (Max-Resolve <form>)

A parallel comparison of the value of all <form>s is performed in logarithmic time. The processor with the "smallest/largest" object remains enabled, and all other PEs are disabled. The function returns T to the root if any processor becomes the winner, and Nil if no processor wins.

(Report <form>)

The <form> is evaluated in the single enabled processor and its value is returned in the root processor as the functional value of Report. If more than processor is enabled when the Report function is called the first enabled PE in an inorder traversal is selected to do the report.

(Mimd <function-name>)

All SIMD enabled descendants logically disconnect themselves from their parents, change their state to MIMD, and begin execution of the specified function. The function returns T.

(ExitMimd) Unlike the previous instructions, this is a MIMD primitive which is executed by the processors when in MIMD mode. It is executed by MIMD nodes to return to SIMD mode. Under current semantics the processor will not execute further instructions until reconnection of the machine is complete. There is potential to change this in the future because the *DADO* I/O chip is designed to support interrupt driven multi-processing. Exit returns Nil if the processing element has any MIMD descendants.

(Sync) This MIMD primitive is executed by the processor which called the Mimd function. It forces the processor to wait until its MIMD descendants invoke the (Exit) function. The sync function in *DADO* hardware waits until all descendants return to the SIMD state. This function returns T.

(Send <form> <tree-neighbor>)

This transmits the value of the <form> to the Input variable of the designated <tree-neighbor>, which may be either the left child or the right child. Sending to the parent is not allowed as the semantics would be unclear if both children were enabled. The function returns T if the <tree-neighbor> is enabled and Nil if it is not.

(Recv <var> <tree-neighbor>)

The current functional value (Output variable) of the <tree-neighbor> is assigned to <var>. The Tree neighbor is either the parent, left-child or right-child. The function returns T.

(TAO <function> <form>)

(for tree associative operation), it applies its functional argument to three inputs <form>s. One input is the local value from the PE, and the other two are the values returned by its children. This allows logarithmic time associative operations. If a node is SIMD disabled its value does not participate in the operation.

The semantics of Send and Recv are not clear from the above description when the operand PE is in SIMD disabled mode. In these cases it is the status of the recipient PE that determines the semantics, not the status of the originator of the call. Specifically, it is always possible to receive data from a PE, but data will only be sent to an enabled PE. Data can be passed through a SIMD disabled PE.

4.4 ||PSL Examples

In this section we present code for two fundamental operations: loading the *DADO* tree with data, and *associative probing*, where data in the tree is matched against an external search string.

4.4.1 Sequentially Loading *DADO*

This example is rewritten from a portion of the PPL/M code implementing a small production system that runs on the prototype *DADO1* machine. It demonstrates how each processing element can be sequentially loaded with data from some external source. It functions by use of the *resolve* primitive to select one unused PE. All other processors are then disabled. The assignment to RECORD occurs in this designated processor. This process stops when all processors have been used, or the user data is exhausted.

```
Code                                      Explanation

In RLISP:

Sliced Expr Procedure DisableLoadedPEs ();
  If DONE then Disable;

Expr Procedure LoadTree (FromFile);
BEGIN
  Sliced Global RECORD;                   Declare RECORD resident in all PEs
  Sliced Global DONE;                     Declare DONE resident in all PEs


  ENABLE();                               Enable all PEs
  DONE:=Nil;                              Set DONE to Nil in all PEs

  While AnotherRecordp(FromFile) do       another record to put into tree
    Begin
      Next:=GetNextRecord(FromFile);      set Next to be new record
      ENABLE();                           Enable all PEs
      DisableLoadedPEs();                 If a PE has been loaded Disable it
      If MinResolve(DONE) then            If there is still a PE available to load
        Begin                             Only one such PE is enabled now
          RECORD:=Next;                   Load the PE with the Next record
          DONE:=T;                        This PE is now loaded
        End
      else error(No-More-PEs);
  End;
```

In LISP:

```
(SLICE DE DISABLELOADEDPES ()
   (COND (DONE DISABLE)))

(DE LOADTREE (FROMFILE)
  (PROG ()
    (SLICED GLOBAL RECORD)
    (SLICED GLOBAL DONE)
    (ENABLE)
    (SETQ DONE NIL)
    (WHILE (ANOTHERRECORDP FROMFILE)
      (PROG ()
        (SETQ NEXT (GETNEXTRECORD FROMFILE))
        (ENABLE)
        (DISABLELOADEDPES)
        (COND ((MINRESOLVE DONE)
               (PROG   ()
                     (SETQ RECORD NEXT)
                     (SETQ DONE T)))
              (T (ERROR (DIFFERENCE (DIFFERENCE NO MORE) PES))))))))
```

4.4.2 Associative Probing

This routine determines if any processor has a record with data that matches a particular constant. It uses the *minresolve* function to select only one processor. After the resolution is complete, the selected processor reports additional data back to the host processor.

```
Code                                     Explanation
```

In RLISP:

```
%% selector procedures
Sliced Expr Procedure NAME (sliced:REC); car(REC);
Sliced Expr Procedure AGE (sliced:REC); cadr(REC);
Sliced Expr Procedure IQ (sliced:REC); caddr(REC);

Expr Procedure Find-Student(WithIQ);
BEGIN
  Sliced Global RECORD;                  Declare RECORD resident in all PEs
  Sliced Global FOUND;                   Declare FOUND resident in all PEs

    Enable();                            Enable all PEs
    FOUND:=eq(WithIQ,IQ(RECORD));        See if a RECORD exists
                                          with desired IQ
    If MinResolve(FOUND)                 See if one found ...
      then Report(NAME(RECORD))           ... if so return name
      else NIL;                           ... otherwise return Nil.
END;
```

In LISP:

```
%% selector procedures
(SLICE DE NAME (REC) (CAR REC))
(SLICE DE AGE (REC) (CADR REC))
(SLICE DE IQ (REC) (CADDR REC))

(DE FINDSTUDENT (WITHIQ)
  (PROG ()
        (SLICED GLOBAL RECORD)
        (SLICED GLOBAL FOUND)
        (ENABLE)
        (SETQ FOUND (EQ WITHIQ (IQ RECORD)))
        (COND ((MINRESOLVE FOUND) (REPORT (NAME RECORD)))
              (T NIL))))
```

4.5 Implementation Issues

The PSL compiler functions within the control processor by translating LISP code into a set of CMACROs representing an instruction set for a primitive abstract machine. These CMACROs are then recursively expanded into the actual assembly or object code for the real machine. The compiler itself is completely written in PSL, as are the interpreter and other system components. This allows relatively rapid bootstrapping for a new machine.

The compiler uses a tagged representation of all LISP data types. A tagged object is called an item and in our implementation consists of three bytes (24 bits). The first byte contains 5 tag bits and 3 garbage collection bits. The remaining bytes are used for the information field. This information field is either immediate data, in the case of an inum, a pointer into the id space, or a reference to the heap space for other Lisp objects.

4.5.1 The Symbol Table

A PSL variable is a tagged item with an information field that contains a pointer into the identifier space (this is the local symbol table). This symbol table contains four entries for each id (variable): a pointer to the print name, a pointer to the property list for this id, a value cell, and a function cell. In our implementation both the hash table and print name are stored in the host processor since these structures are only needed at the user-machine interface.

Variables are represented in the rest of the tree as their offset into the id space. This saves considerable space in each PE. Further space economies may be achieved by sharing s-exprs upon consing. The technique called H-consing, which uses a hash table for cons cells in much the same way as a hash table is used for placing atoms into the symbol table, guarantees that equal s-expr's share the same physical location. Its use is highly recommended due to the significant space savings achieved by the sharing of structures. This may in fact become the system default.

4.5.2 Passing Lisp Objects

In order to pass arbitrary Lisp Objects around the tree with the communication functions we build a normalized relocation map of the object and then pass this relocation map. The object is then recreated from the relocation map at the receiving node. This approach has three distinct advantages.

First if the object is not too large a single byte may be used for each pointer instead of a full word. Secondly, if hash consing is used, for each string with multiple identical substrings the unique substrings will only be passed once. A pointer is passed to the unique substring occurrence for all other occurrences. This is particularly important in an environment where unification occurs as a large number of such strings with multiply occurring substrings are often produced. Finally, this method also allows the passing of self referential structures.

4.5.3 Garbage Collection

Garbage collection in a parallel environment is a well known problem. Our solution is not particularly elegant, but is practical. When a PE discovers it must do a garbage collect it interrupts the rest of the tree and the entire tree garbage collects at one time. We use a simple linked list garbage collector. Due to the imposed restrictions on LISP data types we allow only items which are all the same size (24 bits), consequently compaction is not required. This is accomplished at the price of excluding vectors and strings.

This restriction can be removed as needed by special allocation of a fixed segment with its own allocation mechanism. In the simplest implementation we simply use SysLISP to allocate a large chunk of storage, and compute an explicit pointer into this region. The more sophisticated approach is to have two garbage collection facilities, the string vector region having its own compacting collector and allocator.

4.5.4 Debugging: Calls with one level of indirection

The use of function-value cells in the symbol table has always been recognized as important for tracing, monitoring, and debugging. The need for these facilities is acute in a parallel environment, as it would be otherwise impossible to efficiently monitor a myriad of processors. For example, one major problem with a parallel machine is the monitoring and debugging of the PE contents. By changing the contents of the function value cell of an id we may easily insert a call to a debugging or monitoring routine. The price for this is one additional memory access. We also maintain separate data and code spaces so that the code space need not be reported to the CP in order to determine the state of a PE during debugging.

5 Conclusion

||PSL is a significant tool for the definition and implementation of parallel processing systems. It provides both a coherent language and a development environment for algorithm specification on a tree structured machine. In particular it provides effective communication between the tree and the external environment as well as global and local communication within the tree. The integrated debugging and simulation

facilities are expected to be of great aid during program development especially since they are all written in and accessed through a common language.

References

Griss, M. L., and Hearne, A. C. "A Portable LISP Compiler." *Software - Practice and Experience,* 11:541-605, June, 1981.

Intel, PL/M Language Reference Manual, 1979.

Johnson, S. C. *Yacc: Yet Another compiler Compiler,* Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.

Lesk, M. E. "Lex -- A lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

Lowerie, D., Layman T. , Daer D., and Randal, J. , "Glypnir – A programming Language for ILLIAC IV," Comm. ACM, 19 3, March 1975.

Miranker, D, "Herbal, A Production System for the *DADO* Machine," Technical Report (in preparation) Department of Computer Science, Columbia University, 1984.

Stolfo, S., Miranker, D. and Lerner, M., "PPL/M: A Systems Programming Language for the *DADO* Machine," Columbia University Department of Computer Science, 1984.

Stolfo, S., "The *DADO* Parallel Computer," Department of Computer Science, Columbia University Technical Report," submitted to *AI Journal,* 1983.

Stolfo, S., "Knowledge Engineering, Theory and Practice," Proceedings of the IEEE *Trends and Applications,* 1983.

Taylor, S., "LPS, A Logic Programming System: Motivations and Goals" Technical Report (in preparation) Department of Computer Science, Columbia University, 1984.

Taylor, S., Lowry A., Maguire, G. Q., and Stolfo, S. J. "Programming using Parallel Associative Operations," in 1984 International Conference on Logic Programming, February 6-9, 1984.

Tzoar, D., and Taylor, S., "Unification in a parallel Environment," Technical Report (in preparation) Department of Computer Science, Columbia University, 1984.