CUCS-96-84

LOGIC PROGRAMMING
USING PARALLEL ASSOCIATIVE OPERATIONS

S. Taylor, A. Lowry, G. Q. Maguire Jr., S. J. Stolfo

Department of Computer Science, Columbia University, NY 10027

LOGIC PROGRAMMING
USING PARALLEL ASSOCIATIVE OPERATIONS

S. Taylor, A. Lowry, G. Q. Maguire Jr., S. J. Stolfo

Department of Computer Science, Columbia University, NY 10025

Abstract

In order to provide performance improvements in the execution of large logic programs, it is highly desirable to investigate the relationships between logic, data-base systems and knowledge-based systems in the context of *massively parallel architectures*. This paper presents a model for the interpretation of logic programs in this type of environment and overviews the algorithms under development.

An interpreter that implements the model has been demonstrated in simulations on a number of small programs. Implementation requires that only a small set of hardware primitives be available, these have been successfully implemented on a working prototype machine, DADO.

Current research aims to develop the model into a practical and efficient logic programming system for use on the machine.

1. Background and Introduction

Logic programming is a programming methodology based on symbolic logic, involving the use of Horn clauses. These are universally quantified first order axioms containing at least one positive literal, a restricted form of the general clause encountered in first order predicate calculus. As a consequence of their formal mathematical semantics, Horn clauses provide an explicit declarative interpretation. In addition, Kowalski [11] has assigned them a procedural semantics that provides a basis for their use in programming and effectively defines *how* a program is to be executed. The logic-based languages [1, 19, 26] that have been developed as a result are based on this procedural interpretation of Horn clauses. Under this interpretation, terms in the body of a clause constitute subgoals that must be examined in order to satisfy the head goal. For pedagogical reasons, in this paper we will refer to the syntactic structure and terminology of Prolog when discussing logic programming formalisms. Thus the following clause

logician(X) - human(X), teaches(X, logic)

may be read declaratively as

> *<u>for every</u> X, X is a logician <u>if</u> X is human <u>and</u> X teaches logic*

or procedurally as

> *to solve the goal of finding if X is a logician, solve the goal of finding if X is a human and solve the goal of finding if X teaches logic*

Execution of a logic program, comprising a set of Horn clauses, involves the proof of a user *directive* using the clauses in the program. The results of the execution are the possible binding sets for variables occurring in the directive, each of which is existentially quantified, or failure if the proof cannot be constructed. Since the meaning of a program is essentially declarative in nature, different control strategies can be used to construct the proof. Prolog [20, 26] employs a simple depth-first search of the AND/OR tree defined by clauses in the program. A number of other logic-based language implementations [16, 19] have provided breadth-first search and more sophisticated control structures in an attempt to improve flexibility and efficiency on von Neumann machines [1, 27].

The close relationship between logic and data-base systems [7, 18] provides an excellent theoretical basis for the development of both data-base systems and knowledge-based systems. Logic programming languages provide a natural framework for the implementation of these systems since they provide both the basic inference mechanism required and a uniform representation for factual and procedural

This research is supported cooperatively by International Business Machines Corporation, Digital Equipment Corporation, Intel Corporation, Valid Logic Systems Inc. and Defense Advanced Research Projects Agency under contract N00039-82-C-0427

CH2007-3/84/0000/0058 $01.00 © 1984 IEEE

knowledge Kunifuji and Yokota [12] have shown that Prolog, augmented with the *set-of* meta-predicate, is a relationally complete query language and have outlined methods to interface data-base operations and problem solving systems within a Prolog environment The architectural model of this and other systems [10] involve separate reasoning and relational searching engines

The model described in this paper uses a single inference engine for logic programming in a massively parallel environment It may be implemented using *associative operations* similar to those discussed in the literature on data-base machines and associative processors [23] The model makes use of parallelism inherent in logic programs and as such, does not require the use of additional non-logical annotations A number of opportunities exist for parallel execution of logic programs [5 15] The form described in this paper is an extension of the model termed *search parallelism* [5]

2. The Model

The model presented may be viewed conceptually as the configuration shown in figure 1, making no commitment to a particular connection topology A supervisory, *control processor* (CP) communicates with a large number (on the order of many thousands) of *processing elements* (PE's) in a tightly coupled environment that is *single instruction, multiple data stream* (SIMD) [6] in style but differs in that it allows remote procedure invocation There is no global memory each PE has it's own local memory and the PE configuration may be associatively addressed Since the PE's are assumed to be relatively simple in nature the system may effectively be viewed as an intelligent memory where processing and storage are extensively intermingled

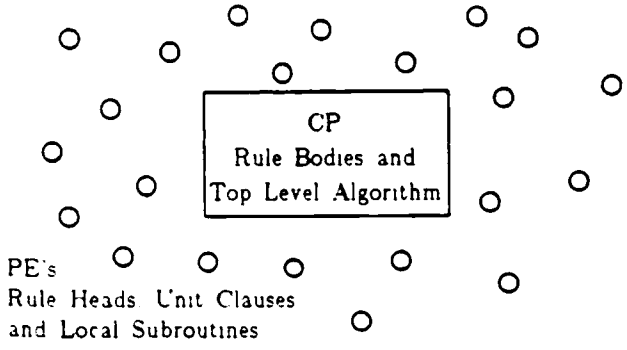


Figure 1

Communication is always initiated by the control processor through a small number of *primitive hardware instructions* The synchronization of the system is assumed to be inherent in the operation of these primitive instructions, and they must be efficiently implemented In order to describe the semantics of the instructions, three registers are of interest

CPR the control processor register used for communication
PER the processing element communication register
EN the enable flag resident at each PE when true, the PE may participate in communication with the CP

The necessary and sufficient set of primitive hardware instructions required to implement the model are defined as follows

ENABLE	Set EN to true, in every PE
BROADCAST	If EN is true then copy CPR to PER, in every PE
REPORT	If EN is true then copy PER to CPR assuming only one PE is enabled
MIN-RESOLVE	Copy min(PER) to CPR, (min(PER) is the minimum value held in a PER register in any enabled PE)

By *broadcasting* an appropriate code these associatively based instructions can be used to invoke local subroutines at each PE Local subroutines perform various functions which include memory management, unification and manipulation of variable bindings Their execution may effect the contents of the three communication registers

The algorithms described in the following sections may be implemented using only this small set of instructions The instructions have been demonstrated on an existing architecture suggesting that a practical implementation is realizable

2.1 The Distribution of Information

In order to reduce both the time spent in communication and the speed of various matching algorithms, the information stored locally at each PE is kept in a tokenized form Tokens are typed pointers into a symbol table kept at the control processor, the sole area in the system that the print form of the program is maintained [8]

The performance of many sequential algorithms can be enhanced by using the associatively based mechanisms

previously described [9]. In general, however, if a significant amount of information is to be accessed, a good hashing or table lookup technique is likely to be as effective due to the communication costs involved. When a logic program is loaded, unit clauses and rule heads are distributed freely throughout the PE's. Rule bodies are stored in the control processor rather than at the PE's. This eliminates the cost that would otherwise be incurred in transferring them from the PE's to the control processor upon rule activation. A token is stored with each rule head to identify the corresponding rule body for retrieval from the control processor when needed. Each unit clause and rule head is assigned to one and only one PE, however each PE may contain several pieces of information. The distribution of this information maybe completely arbitrary and thus considerable flexibility in allocation strategy exists. Intelligent distribution schemes can easily improve performance. At present, clauses are allocated to a particular PE based on the static complexity of clauses already at the PE and if possible, no two clauses within a PE use the same predicate. More sophisticated run time allocation techniques are presently under consideration.

2.2 The Top Level Algorithm

An abstract algorithm, which resides at the control processor and supervises the execution of logic programs, is presented in the appendix. The algorithm traverses the AND/OR search space defined by the clauses in the program using four principle operations, *unification*, *join*, *substitution* and *purging*. Each operation makes extensive use of the primitive hardware instructions detailed previously in order to efficiently manage variable bindings made during program execution.

In summary, the algorithm accepts a directive from the user and expands the entire first level of the search space, producing sets of bindings for variables in the directive. It then loops, attempting to collect a binding set from the PE configuration and print it as a result. Binding sets take two forms and are generated during unification and join operations as the algorithm proceeds. ***Simple binding sets*** are those involving only unit clauses (i.e., clauses with no body) while ***complex binding sets*** are those which include bindings made when a rule head is involved in the operations. Consider the following example.

Goal: f(1)
Contents of PE 5: {Clause 1, unit clause, f(X)}
Contents of PE 2: {Clause 2, rule, f(Y) - body(2)}
Contents of Control Processor: {body(2) - h(Y), g(Z)}

On completion of unification, PE 5 will contain a simple binding set (i.e., [X/1]) while PE 2 will contain a complex binding set (i.e., [Y/1-->body(2)]). When a simple binding set is reported to the control processor, the bindings are immediately printed as a result. When a complex binding set is reported further inferences must be made in order to elaborate the search space and complete the proof. In this case, the rules referenced in the complex binding set must be activated. This is achieved by accessing the relevant body using the token associated with each binding (e.g., body(2)), making any relevant instantiations (e.g., Y/1), renaming variables, and then broadcasting the instantiated body (e.g., <h(1),g(_1)>) to the PE's to be solved. This causes the search space to be partially elaborated beyond the first level of the execution tree. The algorithm terminates when no further binding sets can be collected, at which point the user is prompted for a new directive.

Since a directive may consist of a conjunction of goals, all goals must be solved in order to satisfy the directive. In attempting to solve a particular goal, the control processor broadcasts the goal to each PE which immediately begins searching for clauses relevant to the goal. To determine which clauses are relevant, a unification algorithm is executed which, if successful, produces a set of bindings for variables involved in the unification operation. Since the unification at each clause is independent of the others in the program, they may be carried out in parallel. After a goal is broadcast and unification has begun, a number of processors may become idle because they do not contain relevant clauses. By broadcasting additional goals, prior to the completion of the first goal, these idle processors may be used to solve the additional goals in parallel. Since each goal is solved independently, each producing a distinct set of variable bindings, and because goals may share variables, it is necessary to eliminate conflicting bindings once all goals are complete. This may be carried out using a join [3] operation which constructs a result set for the conjunction as a whole.

Since a conjunction will exist as the body of a clause, the variable bindings found using the join operation must be transformed into a form consistent with the clause head. This operation involves a substitution algorithm (explained in more detail in section 5) which, when complete, allows the space occupied by variable bindings in the conjunction to be reclaimed. The actual reclamation of space is carried out using a purging algorithm.

The very nature of the problem involves the parallel exploration of a possibly exponential search space, it is

important to minimize, as far as is possible, the space used to store bindings The algorithms generate a *frontier set* which comprises a complete characterization of the search space cast in terms of the rules which must eventually be activated in order to yield all possible results This represents the minimum information that must be maintained in order to traverse the search space As results are found they are printed, yielding back the space they occupy

3. The Unification Operation

A linear unification algorithm [17] operates locally at each PE on conjunctions broadcast from the control processor Consider the following somewhat idealized scenario which is intended to illustrate the technique

Conjunction to be unified a b
Contents of PE 1 a_1
Contents of PE 2 b_1

The following sequence of events results

1 The CP broadcasts the conjunction 'a, b' to every PE

2 PE 1 begins unifying $<a\ a_1>$, at the same time PE 2 begins unifying $<a,b_1>$, fails quickly and progresses to unify $<b,b_1>$

3 PE 1 completes unifying $<a,a_1>$, attempts to unify $<b,a_1>$ and fails quickly

4 PE 1 and PE 2 complete unification

The binding sets created are identified by a *level number* that is associated with each goal Thus bindings for 'a' are tagged 'level 1' and those for 'b' are tagged 'level 2' distinguishing them A *sequence number* is also allocated to each binding set that relates to the position of the unifiable clause in the order of the program text The unification process reduces the time taken to find all bindings for a conjunction to the time taken to unify the most complex single goal plus a small constant for each failure goal When failure of a conjunction occurs, it is detected quickly by probing to see if there were any results at all for each particular goal

Having obtained binding sets it is inevitably necessary at some point to transmit them across the network Consider the following example supplied by Paterson and Wegman [17]

{G(F(x1 x1) F(x2,x2), F(xn-1,xn-1)), G(x2,x3, xn)}

If an explicit representation of the unifier is used, an exponential amount of space is required, techniques for sharing structure (DAG's) can be employed to overcome the problem [17] In a distributed environment, additional problems occur when the unifier must be transferred across the network In particular if the structure of the n'th binding is traversed and transmitted explicitly, the transmission requires exponential time This problem may be overcome using a similar paradigm to sharing structure namely, *common substructures are transmitted only once* This requires that a *send mark* be associated with each variable at the PE in which it resides The first time the binding for a variable is transmitted the send mark is set Further attempts to transmit the variable binding cause it's name to be sent rather than its structure Since names are pointers into the symbol table in the control processor, the change of referencing environment (PE environment to control processor environment) resulting from the transmission does not affect the structure which may now be reconstructed at the receiver These algorithms are presently being implemented in the available hardware and take a substantially different approach to the problem than that advocated for FFP machines [14]

4. The Join Operation

Consider the following clause which exhibits a familiar problem experienced in data-base systems, that of shared variables that occur in a conjunction

grandfather(X, Z) - father(X, Y), parent(Y, Z)

Results for each goal in the clause body are represented by sets of variable bindings that were created during unification and are distributed throughout the PE configuration When the solutions for each constituent goal are available, inconsistent bindings for variables that are common to different goals (e.g., 'Y' in the above clause) must be eliminated The remaining solutions form the result set for the conjunctive goal taken as a whole, and thus for the clause head Consider the following distributed binding environment resulting from unifications using the the above rule body:

Contents of PE 14
father(paul, jane) bindings [X/paul, Y/jane]
level=1 sequence=1

Contents of PE 5
father(john, mary). bindings: [X/john, Y/mary]
level=1 sequence=2

Contents of PE 7:
father(alex, andy) bindings: [X/alex, Y/andy]
level=1 sequence=3

Contents of PE 11:

parent(andy, mark) bindings: [Y/andy, Z/mark]
level=2 sequence=4

Contents of PE 3

parent(mary, eddy) bindings: [Y/mary, Z/eddy]
level=2 sequence=5

The set of atomic formulae (both unit clauses and rule heads) represented in the PE configuration may be regarded as comprising several *relations*, each the *extension* of some goal literal. Viewed in this way, the elimination of binding conflicts may be carried out by applying a relational equi-join [3] algorithm to sets of variable bindings for goals occurring in the conjunction under consideration. In outline the algorithm proceeds as follows:

```
FOREACH binding_set IN smallest_relation DO
{
    enable(smallest_relation), % distinguished by level
    binding = report_an_unused_binding,
    enable(largest_relation),
    broadcast(binding),          % parallel operations
    broadcast_command(match_common_variables),
    broadcast_command(form_union_of_results),
}
```

This algorithm forms an ordered set of result bindings. The ordering is maintained by an array calculation involving the sequence numbers and provides the opportunity to ensure the correct operation of sequentialities which may occur in some code segments (e.g. I/O operations). The above example forms the following simple binding sets:

Contents of PE 5
father(john, mary)
bindings [X/john, Y/mary, Z/eddy]
level=1 sequence=4

Contents of PE 7
father(alex, andy)
bindings [X/alex, Y/andy, Z/mark]
level=1 sequence=5

Bindings from rule heads may be included in the join result along with those from unit clauses. As a consequence, possible rule activations that may take place appear in the result but are not executed until they are required. This lazy evaluation technique ensures the correct operation of certain code sequences (e.g. streams) and also prevents unnecessary work.

At the completion of a join operation, all binding sets (both simple and complex) are left at PE's distributed throughout the system and *not* at the control processor.

On a conventional von Neumann architecture, this algorithm is expensive to compute (O(n log n)). However, using the primitive hardware operations outlined earlier, it can be computed in time that is strictly proportional to:

- the size of the smallest set of results for any goal involved in the join, and
- the number of common variables joined over.

If the number of results at a particular PE becomes large during some stage of the algorithm, individual results must be redistributed. This adds an additional cost that is proportional to the size of the result relation. The algorithm is based on a technique described in a doctoral dissertation by Shaw [23], however, since the match phase of the operation must be carried out over logical terms, unification must be used rather than a simple symbol matcher.

Since the results for the constituent parts of the conjunction are all available prior to the use of the join and the size of the result sets may be obtained efficiently using the available hardware primitives, the algorithm may use an optimal ordering in considering results over the whole conjunction. This is a similar notion to that used by Warren [27] in the implementation of an interpreter that reordered the execution of goals in relational data-base queries.

When a join result involves the bindings made on the head of a rule, the complex bindings created have a special representation of the form:

{<common bindings>, <rule 'a' bindings>, <rule 'b' bindings> <rule 'n' bindings>}

This may be interpreted as:

> *to complete this part of the search space, activate rules a,b,...n*

Many such binding sets may exist within the PE configuration. The *common bindings* shown in the above representation form the most restrictive set of bindings for all rule invocations in the complex binding. This set is formed during the join operation.

using unification If unification fails the complex binding will not appear in the result set and no work would be expended in proving the rules in the complex binding It is not yet clear that, in practical applications, forming and maintaining the common bindings is in general less costly than attempting to prove the rules involved Consider the following example [4]

```
1  permute([] [])

2  permute(PL1 [PH|PL2]) -
                  delete(PH PL1 PL3)
                  permute(PL3 PL2)

3  delete(DA [DA|DL] DL)

4  delete(DB [DC|DL1] [DC|DL2]) -
                  delete(DB DL1 DL2)
```

A conventional Prolog interpreter would enter infinite recursion if the above clauses were used with the goal

permute(A, [a])

After printing the first result, which is 'A = [a]', backtracking causes the 'delete' goal to be retried in clause 2 This second call has the first argument instantiated to 'a' and the other two arguments uninstantiated The delete algorithm then attempts to find all possible lists that 'a' may be deleted from, by virtue of the depth first search involved, using clauses 3 and 4 The algorithms in this paper are able to utilize all the information available to prevent infinite recursion in a similar manner to breadth first search An explanation of how the above example is executed will serve to clarify how this is achieved

Initially as a result of unifying the goal with the head of clause 2 the following binding set is created

[A/PL1 a/PH []/PL2] ('a/b' - "a is bound to b")

Clause 2 is now activated and two goals are posted (shown here with renamed variables signified by '?'), delete(a ?PL1 ?PL3) and permute(?PL3 [])

delete(a ?PL1 ?PL3) produces bindings

[a/DA [DA|DL]/?PL1, DL/?PL3] a) from clause 3

[a/DB [DC|DL1]/?PL1, [DC|DL2]/?PL3] b) from clause 4

permute(?PL3 []) produces bindings

[[]/?PL3] c) from clause 1

The join operation when applied to the result sets for the two goals now creates the set of bindings

[a/DA [DA|DL]/?PL1, []/?PL3, []/DL] from a) and c)

Since ?PL3 was bound to [] in binding set c), at the time the join is carried out, the complex binding that would have occurred from the combination of binding sets b) and c) is discarded, thus preventing the call to 'delete' with uninstantiated arguments and non terminating recursion

5. The Substitution and Purge Operations

When a complex binding is reported to the control processor, it holds all the information required to complete some sub tree in the search space Each rule binding in the complex binding carries a pointer to the corresponding rule body, held in the control processor so that it can be retrieved and activated The results for rule bodies however, do not necessarily represent the results for the goal which invoked the rule Consider the following example

Goal	f(X)
Rule	f(a(Y)) - g(Y)
Fact	g(b)
Fact	g(c)

Solving the body of the rule generates values for 'Y', but the results for 'X' are of the form 'a(Y)' In order to minimize the space used by bindings, the initial binding set [a(Y)/X, b/Y, c/Y] can be reduced to a new set [a(b)/X, a(c)/X] by a process we have come to term *substitution* The initial binding set can now be deleted, using a process termed *purging* which reclaims the space they occupy As the search space is expanded, these techniques can be carried out in parallel across the new bindings found (b/Y and c/Y in this case) using the primitive hardware instructions described earlier The resulting set of bindings, maintained at the PE's, constitutes only the frontier set described earlier

Figure 2 shows how variables are conceptualized at the time the substitution operation takes place The state of the computation prior to the substitution operation is such that a complex binding from the present frontier set has been received at the control processor It carries a set of bindings, the *Top->Frontier bindings* which occurred when the complex binding was created These characterize the search space

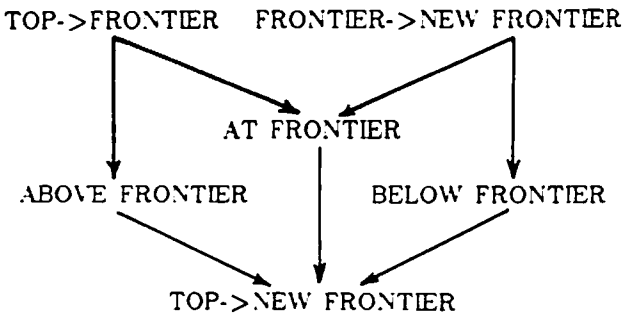


Figure 2

above the present frontier set As a result of activating some rule in the complex binding, bindings for those variables in the body of the rule are created using unification and join operations These bindings, the *Frontier->New Frontier bindings*, are distributed throughout the PE's and are to be included in the new frontier set created when the substitution operation occurs

Since variable bindings may occur in both directions (goal-head and vice versa) during unification, the two bindings sets described above effectively comprise three layers of bindings The *above frontier* bindings are those from variables occurring in rules above the present frontier set to terms in the rule being resolved (e g , X' in the previous example) The *at frontier* bindings are those for variables occurring in the rule being activated at the frontier set (e g , Y' in the previous example) The *below frontier* bindings are those for variables occurring in unit clauses or rule heads that some atomic formula, in the body of the rule being resolved, unified with

Substitution takes the three layers and creates a new set of bindings that does not contain any redundant information (e g , 'Y' in the previous example) This new set binds terms above the present frontier set to those in the new frontier set in parallel across the new frontier set Thus the bindings maintained are always from the top-level directive to the current frontier set The new frontier set is distributed throughout the PE's and a purging operation can now be used to reclaim the space occupied by any redundant information

6. An Example Search Space

An example will serve to illustrate how bindings are formed and resolved Consider the following set of abstract clauses in which each term is assumed to be some structure containing variables,

C1) t - a, b, c	C2) a_1 - d, e	C3) a_2
C4) a_3 - f	C5) b_1 - g	C6) b_2
C7) c	C8) d_1	C9) d_2
C10) d_3	C11) d_4	C12) e
C13) f_1	C14) f_2	C15) g

The subscripts shown serve only to clarify the explanation and are not actually part of the clauses Figure 3 shows the search space generated by the program while attempting to solve a top level directive 't' using the familiar AND/OR tree representation for illustrative purposes

If the program were executed by a conventional Prolog interpreter, consistent binding combinations from the following set would be printed as results

d_1 e g c	d_3 e b_2 c	f_1 g c
d_1 e b_2 c	d_4 e g c	f_1 b_2 c
d_2 e g c	d_4 e b_2 c	f_2 g c
d_2 e b_2 c	a_2 g c	f_2 b_2 c
d_3 e g c	a_2 b_2 c	

The model of execution described earlier constructs an identical set of results In the diagrams that follow, the ordering maintained by the algorithm is reflected by the left to right order in which the bindings appear on the page Complex bindings are marked '*' to distinguish them

In unifying the directive 't' with the head of clause C1, the following complex binding is created at the PE holding the head of C1

Stage 1

t *

When this binding is reported to the control processor it carries information which allows the body of clause C1 (the conjunction 'a, b, c') to be accessed from the symbol table, instantiated and broadcast to the PE's for unification A join operation then causes the following set of bindings to be generated at PE's distributed throughout the system

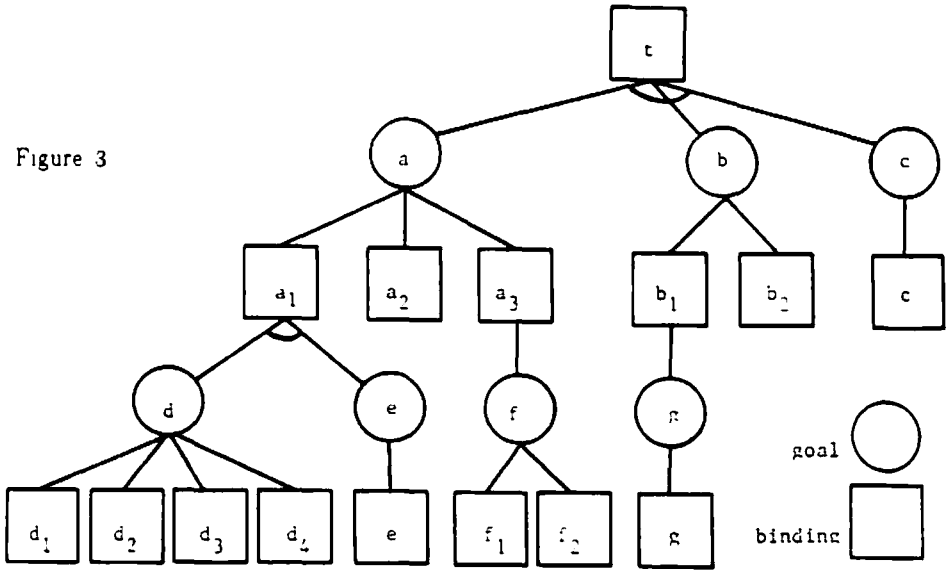
Stage 2

a_1 b_1 c * a_1 b_2 c * a_2 b_1 c *

a_2 b_2 c, a_3 b_1 c * a_3 b_2 c *

The leftmost binding is now reported to the control processor The binding carries the information that 'a_1' is contributed by the head of clause C2, 'b_1' by

Figure 3



the head of clause C5 and that 'c' was a unit clause This allows clause C2 to be accessed at the control processor and activated As a consequence goals 'd' and 'e' are broadcast to the PE configuration for unification and their results are joined The same operation then allows clause C5 to be activated causing goal 'g' to be resolved The results from both these clauses (C2 and C5) are then joined resulting in the following distributed binding environment

Stage 3

d_1 e g c d_2 e g c d_3 e g c d_4 e g c

a_1 b_2 c * a_2 b_1 c * a_2 b_2 c a_3 b_1 c *

a_3 b_2 c *

Since the first four bindings do not need expansion (because they are simple bindings), they are now printed Following cycles cause the search space to be completely elaborated as follows.

Stage 4

d_1 e b_2 c d_2 e b_2 c d_3 e b_2 c d_4 e b_2 c

a_2 b_1 c * a_2 b_2 c a_3 b_1 c * a_3 b_2 c *

Stage 5

a_2 g c a_2 b_2 c a_3 b_1 c * a_3 b_2 c *

Stage 6

f_1 g c f_2 g c a_3 b_2 c *

Stage 7

f_1 b_2 c f_2 b_2 c

7. The DADO Architecture

DADO [24, 25] is a highly parallel, tree-structured architecture based on VLSI technology It is our belief that DADO can provide significant, cost-effective performance improvements over sequential machines in a wide range of Artificial Intelligence applications The DADO prototype now under construction comprises 1023 processing elements (PE's) interconnected to form a complete binary tree Currently, each PE is implemented using an Intel 8751 microcomputer chip and an Intel 2186 8Kx8 RAM chip A special combinational I/O switch, implemented as a custom integrated circuit, is under development It provides high-speed communication facilities including, as a subset, the primitives described in this paper The speed of each primitive is expected to be approximately equal to a single 8751 instruction executed locally at a PE The full-scale version of the system, implemented entirely in custom VLSI, is expected to contain many thousands of PE's

An initial prototype version of the system, using 15 PE's, has been fully operational since April 1983 It's main purpose is to provide a software development environment The communication and synchronization facilities, which will be available in future versions of the machine, are presently implemented in firmware and are made efficient by pipelining instructions through the tree

It should be noted that the binary tree organization of DADO was chosen for reasons related to efficient implementation in VLSI [13] As is the case with many of the parallel algorithms under investigation, the DADO tree structure has no direct relevance to the logic programming algorithms outlined in this paper

8. Current and Future Research

It would be possible to use a number of alternative strategies to those presented in this report Three important variations are under consideration

- The complete evaluation of all results in a join is unnecessary, it would be preferable to only consider results from a single element of the smallest relation at any one point in the search This strategy would generally be more useful when all possible results are not required

- When goals in a conjunction do not share variables the join algorithm computes the cartesian product of the relations It would be preferable to avoid this overhead if the goals can be shown to be independent

- The substitution mechanism used is very sophisticated and requires considerable code resident at each processor The use of a simple stack in the control processor may be more effective however this method precludes parallel substitution

An interpreter using the basic inference engine described in this paper has been successfully demonstrated on a number of small logic programs under simulation The simulator is structured such that various code segments may gradually be delegated to the DADO tree for implementation and testing It is a goal of this research to generalize the model described into a practical logic programming system for use on the machine Various sections of the algorithms have already been implemented on the prototype system reported earlier A number of significant problems such as the use of negation have not yet been addressed However where possible the implementation will closely resemble conventional Prolog in order to encourage logic programmers to use the system

Investigations to determine how many processors will produce optimal running time in relation to the size of a given program and how best to organize run time processor allocation are still to be carried out A statistical analysis of Prolog source programs has begun in order to ascertain the level of parallelism that can be expected and tune various heuristics used in the model

The use of formalisms which allow Prolog to express concurrency [2, 21] lend themselves to a different form of architecture than that presented in this paper and attack a different set of problems Merging these formalisms in order to provide efficient support for formalisms such as object-oriented programming [22] and distributed AI applications is an attractive possibility

9. Conclusions

The motivation for this work is to investigate the ties between logic, data-base systems and knowledge-based systems in the context of massively parallel architectures The model presented displays a number of attractive qualities

- The small set of primitive communication operations needed have been successfully implemented and the algorithms are based on efficient operations that manipulate bindings in a non von Neumann architecture This suggests that a practical and efficient system is realizable

- No additional, non logical annotations are required

- Clauses may be placed at arbitrary PE's in the hardware configuration This creates considerable flexibility in the allocation strategies which can be used

- A number of clauses may be packed into a single PE As a result performance will gracefully degrade as packing density increases above the optimum level

- No replication of information is required

- The model presents a uniform method to handle both procedural and factual knowledge present in knowledge-based systems by virtue of the logic programming methodology used

- The use of parallelism is transparent to the user

While the parallel execution of a logic program is potentially exponential in space considerable effort has been expended to reduce the space required, in practice, to a minimum

The approach presented requires sophisticated algorithms, a number of avenues are being investigated to reduce their complexity Current research aims to generalize the model to a practical and efficient logic programming system for use on the DADO machine

Acknowledgments

The authors would like to extend their thanks to Doug DeGroot Harold Stone Chris Maio Daphne Tzoar Dan Miranker and Mark Lerner whose help enthusiasm and critical insights have proved invaluable during various stages of this research

References

1. Clark K L McCabe F G and Gregory S IC-PROLOG Language Features In *Logic Programming*, Academic Press, 1982, pp 243-266

2. Clark, K L and Gregory S PARLOG A Parallel Logic Programming Language Tech Rept DOC 83/5 Imperial College of Science and Technology March, 1983

3. Codd E F "Relational Completeness of Data Base Sublanguages" *Courant Computer Science Symposium 6: Data Base Systems Vol. 6* (1972) R Rustin (ed) Prentice-Hall, Inc

4. Coelho H Cotta J C and Pereira, L M *How to Solve it with Prolog*. Ministerio da Habitacao E Obras Publicas, Laboratorio Nacional de Engenharia Civil, Lisboa 1982

5. Conery J S and Kibler D F "Parallel Interpretation of Logic Programs" *Proceedings of 1981 conference on Functional Programming Languages and Computer Architecture Vol. 1* (1981), pp 163-170

6. Flynn D "Some Computer Organizations And Their Effectiveness" *IEEE Transactions On Computers Vol. C-21* (September 1972) pp 948-960

7. Gallaire H and Minker J (Ed) *Logic and Data Bases*. Plenum Press, 1978

8. Goto E Monocopy and Associative Algorithms in an Extended Lisp Information Science Laboratory, May, 1974

9. Hillyer, B K , et al Informal NON-VON Algorithms Department of Computer Science, Columbia University, August, 1982

10. Kellog, G Knowledge Management A Practical Amalgam of Knowledge and Data Base Technology AAAI System Development Corporation 1982, pp 306-313

11. Kowalski, R A Predicate Logic as a Programming Language IFIP Congress, 1974, pp 569-574

12. Kunifugi S and Yokota H PROLOG and Relational Data Bases ICOT Research Center October 1982

13. Leiserson, C E *Area-Efficient VLSI Computation* Ph D Th , Department of Computer Science Carnegie-Mellon University, October 1981

14. Mago, G "Data Sharing in an FFP Machine" *Conference REcord of the 1982 ACM Symposium on Lisp and Functional Programming Vol. 1* (August 15-18 1982) pp 201-207

15. Matsumoto Y , Nitta, K and Furukawa, K PROLOG Interpreter and its Parallel Extension Electrotechnical Laboratory/ICOT Research Center, 1982

16. McCord M C LP A PROLOG Interpreter written in LISP Tech Rept 85-82, University of Kentucky May 1982

17. Paterson M S and Wegman M N "Linear Unification" *Computer and System Sciences Vol. 16* (1978) pp 158-167

18. Pereira L M Porto A Monteiro L and Filgueiras M (Ed) *Proceedings of Logic Programming Workshop '83*. Universidade Nova De Lisboa Praia da Falesia Algarve/Portugal, 1983

19. Robinson J A and Sibert E E LOGLISP Motivation Design and Implementation In *Logic Programming*, Academic Press 1982, pp 299-313

20. Roussel P Prolog Manuel de Reference et d Utilisation Marseille-Luminy, 1975

21. Shapiro E Y A Subset of Concurrent Prolog and its Interpreter Weizmann Institute of Science/ICOT Research Center, January, 1983

22. Shapiro, E Y and Takeuchi, A Object Oriented Programming in Concurrent PROLOG Weizmann Institute of Science/ICOT Research Center, 1983

23. Shaw, D E The NON-VON Supercomputer Department of Computer Science, Columbia University, 1982

24. Stolfo, S J and Shaw, D E "DADO A Tree-Structured Machine Architecture For Production Systems" *Proceedings of the National Conference on Artificial Intelligence Vol. 1* (August 1982).

25. Stolfo, S J, Miranker, D and Shaw, D E Architecture and Applications of DADO, A Large-Scale Parallel Computer for Artificial Intelligence

Proceedings of the Eighth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence, Inc, Karlsruhe, West Germany, August, 1983, pp 850-854.

26. Warren, D H D Implementing Prolog - Compiling Predicate Logic Programs Tech Rept D A I 39/40, Department of Artificial Intelligence, Edinburgh University, May, 1977

27. Warren, D H D "Efficient Processing of Interactive Relational Database Queries Expressed in Logic" *IEEE* (1981), pp 272-281

APPENDIX - Supervisory Interpreter Code

The following abstract algorithm is executed at the control processor and supervises the execution of logic programs It uses four primary operations, namely, *unify*, *join*, *substitute* and *purge* These are implemented using the primitive hardware instructions detailed in Section 2

```
% First level for bindings
constant BASE_LEVEL = 2

repeat
{
  directive = get_directive()  % Read
  if (directive /= halt) then
    prove_print(directive)       % Prove, Print
}
until (directive = halt)



prove_print(conjunction)
{
 UNIFY(conjunction, BASE_LEVEL)
 % do unifications locally in PE's
 JOIN(BASE_LEVEL, length conjunction)
 % produce result for conjunction
 YIELD()
 % mark bindings at level 1, the result set level

 while "results remain" do
 {
  result = report_next_result()
  if simple(result) then print(result)
   else      % rules need to be expanded
   {
    level = BASE_LEVEL
    foreach head_binding in result do
    {
     body =
        lookup(head_binding body_ptr,symbol_table)
     body =instantiated(body, head_binding bindings)
     UNIFY(body, level)
     JOIN(level, length body)
     SUBSTITUTE(head_binding bindings, level)
     PURGE(level)
     level = level + 1 % different result sets put
                       %    on different levels
    }
    JOIN(BASE_LEVEL, size result)
    % form result set for expanded complex binding
    YIELD()
   }
 }
}
```