Towards a Formal Computation Model of Associative Logic Programming

Arvind K. Bansal
Department of Mathematics and Computer Science
Kent State University, Kent, OH 44242 - 0001, USA
E-mail: arvind@mcs.kent.edu

1 Introduction

Associative computation is characterized by seamless intertwining of search by content and data parallel computation. This intertwining facilitates integration of knowledge retrieval and data parallel computations independent of number of elements. This work is an effort to generalize the associative computation model which started with our earlier work to exploit associative computation on SIMD architectures.

In this paper, we describe an architecture independent rules for associative computation, the advantages achieved by intertwining search by content, data parallel computation and alignment of abstract data,and an abstract instruction set which exploits associative computation and data alignment effectively. The major significance of the associative model is that it supports a large class of queries to derive unspecified relations based on incomplete information about attributes, knowledge discovery, reasoning about meta-relations - relations about relations, and queries which integrate search by content, inequality, and data parallel scientific computation.

The compiler and emulator for the current model have been implemented. The compiler has been written in C^{++} [18] and the emulator has been written using ANSI C [7] on HP 730. The emulator supports both scalar and data-parallel computations, and is portable to any architecture which supports a data parallel version of C. The benchmark results show that overhead of shallow backtracking and deep backtracking has been significantly reduced which allows seamless integration of knowledge retrieval, rule based reasoning, and data parallel scientific computation. Associative representation of data reduces the overhead of sequentiality caused by pointer based data representations [17].

Implications of these results are that the model can be successfully applied to data intensive problems such as geographical information systems, image understanding systems, statistical knowledge bases, and genome sequencing. For example, in geographical information systems, spatial data structures such as quad-trees and oct-trees can be represented associatively. Different regions having the same values can be identified using associative search on value. and integrated with intelligent rule based reasoning. In large knowledge bases, statistical queries can directly benefit from associative search by content, associative representation of structures, data parallel arithmetic computations, and data parallel aggregate functions. Genome sequencing schemes which need knowledge retrieval, efficient insertion and deletion from a sequence, and efficient manipulation of matrices for heuristic matching of sequences can directly benefit from this scheme.

2 Associative Computation Model

The basis of associative computation model is to represent the data as bags or association of bags, and perform data parallel computation with little data movement. We use two types of bags, namely, a D-bag and a F-bag. A bag is a collection of items such that there can be multiple occurrences of an element. A *D-bag*, denoted by $\mathcal{D}$, is defined as an ordered bag which also contains null elements $\perp$. For example, { *2, $\perp$, 3* } is a D-bag. However, { *2, $\perp$, 3* } $\neq$ { $\perp$, *2, 3* } since D-bags are ordered. $\perp \preceq$ every element in the D-bag. A D-bag $\mathcal{D}_1$ = { d_{11}, ..., d_{1N}} is D-included in another D-bag $\mathcal{D}_2$ = {d_{21}, ..., d_{2N} } if $\forall I_{(1 \leq I \leq N)}$ $d_{1I} \preceq d_{2I}$. For example, { *4, $\perp$, 5, 6* } is a *D-subbag* of { *4, 3, 5, 6* } since $\perp \preceq$ *3*. D-union of two D-subbags of a D-bag derives a new D-bag { d_{31}, ..., d_{3N} } such that $\forall I_{(1 \leq I \leq N)}$ $d_{3I} = d_{1I}$ if $d_{2I} \preceq d_{1I}$, $d_{3I} = d_{2I}$ if $d_{1I} \preceq d_{2I}$, or $d_{3I} = d_{1I}$ if $d_{1I} = d_{2I}$. For example, D-union of the D-subbags { $\perp$, *b, c* } and { *a, b*, $\perp$ } derives the D-bag { *a, b, c* }. D-intersection of two D-subbags of a D-bag derives a new D-bag such that $\forall I_{(1 \leq I \leq N)}$ d_{3I} = d_{1I} if $d_{1I} \preceq d_{2I}$, $d_{3I} = d_{2I}$ if $d_{2I} \preceq d_{1I}$, or $d_{3I} = d_{1I}$ if $d_{1I} = d_{2I}$. For example, D-intersection of D-subbags { *2, 3,*, $\perp$ } and { $\perp$, *3, 4* } derives the D-subbag { $\perp$, *3*, $\perp$ }.

Note that D-inclusion, D-union, and D-intersection of two D-subbags are different than usual definitions of union and intersection since D-inclusion, D-union, and D-intersection are based on pairwise comparison of elements in two D-bags. The truth values *true* and *false* are treated synonymously with the values "1" and "0" respectively. An F-bag is a D-bag which has either *true* and *false*. We treat *false* (or "0") $\preceq$ *true* (or "1"). A F-bag of *1s* is denoted by $\mathcal{F}^1$, a F-bag of *0s* is denoted by $\mathcal{F}^0$, and a F-bag containing both *1s* and *0s* is denoted by $\mathcal{F}$. F-bags are realized by logical bit-vectors. Under the assumption *false* $\preceq$ *true*, D-union of F-bags and logical-OR of the corresponding logical bit-vectors are equivalent, and D-intersection of F-bags and logical-AND of the corresponding logical bit-vectors are equivalent.

A D-bag of M-tuples of the form { < d_{11}, ..., d_{1M} >, ..., < d_{N1}, ..., d_{NM} > } is stored as M D-bags aligned to each other such that accessing I_{th} element of one bag also gives access to I_{th} element of other D-bags. We denote the physical aligned D-bags as $\mathcal{D}_1 \bigoplus \mathcal{D}_2 \bigoplus$..., $\bigoplus \mathcal{D}_M$ where $\bigoplus$ represents the association of two D-bags such that corresponding elements are aligned. We also denote application of an F-bag on a D-bag to select a D-subbag by $\mathcal{D} \bigotimes \mathcal{F}$. We also denote selection of a D-bag from an association as Π. For example, $\Pi_2(\mathcal{D}_1 \bigoplus \mathcal{D}_2)$ gives $\mathcal{D}_2$.

An architecture is a sequence of *processing cells*. Each cell is a quadruple < C_i, R_i, S_i, M_i > where C_i denotes a processing element (PE), R_i denotes a set of local registers, S_i denotes local storage, and M_i denotes a mask-bit. The D-bag of mask bits is set selectively to filter instructions. An instruction is broadcast to each cell simultaneously. The flow of control is effected by generating, saving, and restoring the mask bits based on the results of tests on local data. An associative search of a field for a specific value sets up the corresponding mask bit which is stored and manipulated during computations. SIMD architecture with content addressable memory satisfies this criteria. However, we do not limit the scope of the associative computation to SIMD architectures.

Our model is based upon following sixteen rules of associative computation. There are five types of laws of associative computation: *laws for data association*, *laws for associative search*, *laws for selection*, *laws for data parallel computations* and *laws for data parallel updation*. The first rule states that a D-bag of M-tuples is given by the association of *M* D-bags such that corresponding elements are physically aligned. For example, { *a, 2, 3, 4* } $\bigoplus$ { *b, 5, 6, 7* } is equivalent to { < *a, b* >, < *2, 5* >, < *3, 6* >, < *4* , *7* > }. The second rule states that resulting association is independent of level of sub-associations formed by individual D-bags. For example, { *1, 2* } $\bigoplus$ ({ *3, 4* } $\bigoplus$ { *5, 6* }) is equivalent to ({ *1, 2* } $\bigoplus$ { *3, 4* }) $\bigoplus$ { *5, 6* }, and both are equivalent to { < *1, 3, 5* >, < *2, 4, 6* > }. The third rule states that associations of D-bags are not symmetric. However, the associations are isomorphic: a pair (*x, y*) $\in \mathcal{D}_1 \bigoplus \mathcal{D}_2$ (such that $x \in \mathcal{D}_1$ and $y \in \mathcal{D}_2$) has a bijective mapping to (y, x) $\in \mathcal{D}_2 \bigoplus \mathcal{D}_1$. The implication of this rule is that the same information can be represented equivalently by permuting the order of association. The fourth rule states that associative search of a data element *d* in a D-bag $\mathcal{D}_1$ (of the form < d_1, ..., d_N > derives an F-bag $\mathcal{F}$ such that if d_j = d then the corresponding element in $\mathcal{F}$ is "1" otherwise "0". For example, associative search of an element *4* in the D-bag { 3, 5, 4, 7, 4, 9 } gives an F-bag {*0, 0, 1, 0, 1, 0* }. The fifth rule states the law of selection. The rule states that association of an F-bag with a D-bag selects the data elements whenever the corresponding element in F-bag is 1. For example, { *3, 5, 6* } $\bigotimes$ {*0, 1, 0*} gives { $\perp$, *5*, $\perp$ }. The sixth rule states that by associatively searching in one field, the associated data elements in the other field can be extracted. For example, associative search for a tuple {4, _, _ } from the tuple { < *4, 5, 6* >, < *3, 7, 9* >, ..., < *4, 9, 10* > } gives an F-bag { *1, 0, ..., 1* } which gives the selected D-subbag as { < *4, 5, 6* >, $\perp$, ..., < *4, 9, 10* > }. The seventh rule states that selecting data elements from two associated D-bags is same as selecting data elements from individual bags and then associating them. For example, ({ *4, 5, 6* } $\bigotimes$ { *1, 0, 1* }) $\bigoplus$ ({ *a, b, c* } $\bigotimes$ { *1, 0, 1* }) is equivalent to { < *4, a* >, < *5, b* >, < *6, c* > } $\bigotimes$ { *1, 0, 1* } which gives the D-bag { < *4, a* >, $\perp$, < *6, c* > }. For example, {(*2, 3*), (*4, 5*), (*6, 7*)} $\bigotimes$ { *1, 1, 0* } is equivalent to ({*2, 4, 6*} $\bigotimes$ { *1, 1, 0* }) $\bigoplus$ ({*3, 5, 7*} $\bigotimes$ { *1, 1, 0* }). The computation derives {*2, 4*, $\perp$} $\bigoplus$ {*3, 5*, $\perp$ } which is equivalent to { < *2, 3*>, <*4, 5*>, $\perp$}. The eight rule states *monotonicity* in selection. The rule states that data elements of a D-bag selected using an F-bag $\mathcal{F}_1$ includes the data elements selected using another F-bag $\mathcal{F}_2$ if $\mathcal{F}_1 \sqsubseteq \mathcal{F}_2$. For example, { *5, 6, 7* } $\bigotimes$ { *1, 0, 1* } gives the D-bag { *5*, $\perp$, *6* }. While { *5, 6, 7* } $\bigotimes$ { *1, 0, 0* } gives the D-bag { *5*, $\perp$, $\perp$ }. The ninth rule states that data elements of a D-bag selected by D-union of two different F-bags is same as results of selecting the data elements by applying individual F-bag on the D-bag and then performing D-union on the resulting D-subbags. For example, {*2, 3, 4*} $\bigotimes$ ({*1, 0, 0*} $\bigsqcup$ {*0, 1, 0* }) gives {*2, 3, 4*} $\bigotimes$ {*1, 1, 0*} which is equivalent to *2, 3*, $\perp$. The tenth rule states that data elements of a D-bag selected by D-intersection of two different F-bags is same as results of selecting the data elements by applying individual F-bag on the D-bag and then performing D-intersection on the resulting D-subbags of data elements. For example, {*2, 3, 4*} $\bigotimes$ ({*1, 1, 0*} $\sqcap$ {*0, 1, 1* }) gives {*2, 3, 4*} $\bigotimes$ {*0, 1, 0*} which is equivalent to { $\perp$, *3*, $\perp$ }. The eleventh rule states that Cartesian product of a bag $\mathcal{D}$ with *true* is equivalent to the set obtained by associating $\mathcal{F}^1$ with any D-bag, and is equivalent to $\mathcal{D}$ itself. For example, {*2, 3, 4*} $\times$ { *true* } is equivalent to {*2, 3, 4*} $\bigotimes$ {*1, 1, 1*} which results into the D-bag {*2, 3, 4*}. The twelfth rule states Cartesian product of a D-bag $\mathcal{D}_1$ with *false* is equivalent to the set obtained by associating $\mathcal{F}$ with an D-bag, and is equivalent to a null set. A null set ϕ is also represented as a D-bag with every element as $\perp$. The thirteenth rule states that if any two D-bags are associated and a data parallel computation is performed on the data elements of each bag then the operation is equivalent to performing the same operation on every element of the associated fields. Any computation involving $\perp$ maps onto $\perp$. For example, {*2, 3*, $\perp$} $*^D$ {*3, 4*, $\perp$} gives {*6, 12*, $\perp$}. The fourteenth rule states that if a scalar value *Val* is operated on a D-bag using a data parallel computation, then the data parallel computation is equivalent to taking Cartesian product of the singleton set { *Val* } with $\mathcal{F}^1$, and performing data parallel computation on the association ({ *Val* } $\times$ $\mathcal{F}^1$) $\bigoplus$ $\mathcal{D}_1$. For example, *4* * {*2, 3, 4*} is equivalent to {*4, 4, 4*} $*^D$ {*2, 3, 4*} which gives the D-bag {*8, 12, 16*}. The fifteenth rule concerns *associative update*. The rule states that if a tuple of the form < d_1, ..., d_N > is inserted in an association of bags $\mathcal{D}_1$ $\bigoplus$, ..., $\bigoplus$ $\mathcal{D}_N$, then the association is updated in a unit time to ($\mathcal{D}_1^U$ $\bigoplus$, ..., $\bigoplus$ $\mathcal{D}_N^U$) where $\mathcal{D}_I^U$ denotes the updated bag. Each $\mathcal{D}_I^U = \mathcal{D}_I \bigcup \{d_I\}$. The sixteenth rule states that by associatively searching in one field, the associated data elements in the other field can be released in a unit computation. For example, associative search for a tuple {4, _, _ } from the tuple { < *4, 5, 6* >, < *3, 7, 9* >, ..., *4, 9, 10* > } gives an F-bag { *1, 0, ..., 1* }. Complement of { *1, 0, ..., 1* } gives { *0, 1, ..., 0*}. Application of { *0, 1, ..., 0*} gives the D-subbag as { < $\perp$, < *3, 7, 9* >, ..., $\perp$ > } deleting all the tuples which have value *4*.

3 The Abstract Machine and Its Behavior

The compilation model maps the program as a pair of associations of the form $< \mathcal{L} \bigoplus \mathcal{P} \bigoplus \mathcal{A}_1 \bigoplus, ...,$ $\bigoplus \mathcal{A}_N, \mathcal{L} \bigoplus \mathcal{C} >$. The first element of the pair represents D-bag of clause-head tuples, and second element of the pair represents the clause-body tuples. $\mathcal{L}$ is the D-bag of labels connecting clause-heads to low level code of the corresponding clause-body, $\mathcal{P}$ is the D-bag of procedure-names, $\mathcal{A}_I$ is the D-bag of I_{th} argument in set of the clause-heads in a program, $\mathcal{C}$ is the D-bag of a sequence of compiled instructions corresponding to set of clause-bodies in the program such that each element $c_i \in \mathcal{C}$ is a sequence of instructions corresponding to one clause-body.

The computation model has following components:

1. an associative data parallel abstract instruction set,
2. an associative representation of clause-heads,
3. an associative heap to store the bindings of the output variables, shared variables, and scalar bindings.
4. a data parallel binding environment which is physically aligned to the association of parallel field representing clause-heads. The data parallel environment is used to store the F-bags derived during compile-time analysis, F-bags to mark unifiable clauses, and D-bags to store bindings for a variable.
5. an associative scheme to handle the aliased variables,
6. an associative control stack to store a control-thread,
7. a sequence of global registers - an associative equivalent of scalar registers in WAM to store the bindings of arguments of a subgoal.

A detailed scheme for the computation model for SIMD computers is given in [3].

Alignment of D-bags plays a major role in reduction of the overhead of data movement and associative retrieval of information from aligned fields. Data movement is reduced by using a F-bag to select a subbag from the original D-bag or association of D-bags. Physical alignment of the arguments of the same clause facilitates

(i) data parallel pruning of those clauses which do not share same values for multiple occurrence goal variables: arguments of the same clauses are pairwise compared in a data parallel manner. For example, for the given goal $a(X,\ X,\ Y)$ and a set of clauses $\{\ a(5,\ 6,\ 4)$, ..., $a(4,\ 4,\ 2)$, ..., $a(6,\ 7,\ 9)\}$ data parallel equality test between first two D-bags gives an F-bag $\{0,\ ...,\ 1,\ ...,\ 0\ \}$ which indicates non-unifiability of the clause-heads.

(ii) data parallel computation on two arguments of the same clause. For example, arithmetic comparison to identify whether second argument is greater than the third argument for the given goal $a(X,\ X,\ Z) \bigwedge X > Z$ and a set of clauses $\{\ a(5,\ 6,\ 4)$, ..., $a(4,\ 4,\ 2)$, ..., $a(6,\ 7,\ 9)\}$, a unit data parallel inequality test will return the F-bag $\{1,\ ...,\ 1,\ ..,\ 0\ \}$.

(iii) data parallel derivation of clauses satisfying conjunction, disjunction, negation of complex conditions. For example, finding out the set of clauses which satisfy both the conditions given above will need D-intersection of two subsets selected by F-bags $\{0,\ ...,\ 1,\ ...,\ 0\ \}$ and $\{\ 1,\ ...,\ 1,\ ...,\ 0\ \}$. D-intersection of two F-bags derives the F-bag $\{\ 0,\ ...,\ 1,\ ...,\ 0\ \}$ marking the unifiable clause-heads.

In conjunction with physical alignment of the program association with data parallel environment, the physical alignment of argument of the same clause also facilitates

(iv) selection of a subset of data from the program arguments without actual movement of data (associative computation rules 6 and 7). For example, given a set of clauses { *a*(*5, 6, 4*), ..., *a*(*4, 4, 2*), ..., *a*(*6, 7, 9*)}, and an F-bag { *1, ..., 1, ..., 0* }, the arguments the clauses *a*(*5, 6, 4*) and *a*(*4, 4, 2*) are selected for data parallel operation without data movement.

(v) Selected data from the program association can be moved to data parallel binding environment in a constant number of data parallel operations. For example, given a set of clauses $\mathcal{S}$ = { *a*(*5, 6, 4*), ..., *a*(*4, 4, 2*), ..., *a*(*6, 7, 9*) and a F-bag $\mathcal{F}$ = {*1, ..., 1, ..., 0*}, the operation $\Pi_3(\mathcal{S} \otimes \mathcal{F})$ gives the D-bag { *4, 2*, $\perp$ }.

One of the concerns in logic programs is to efficiently access and manipulate the aliased variables such that binding of one variable is also seen by the other aliased variable. Traditional systems use chain of references to access the bindings. In contrast, associative logic programming system can be benefited by associative search to derive and update the bindings of aliased variables. The associative computing paradigm handles aliasing very effectively due to physical alignment of F-bag selecting a subset of variables and data parallel D-union of two F-bags to derive a new F-bag which gives D-union of two D-bags. The major advantage of associative scheme over conventional scheme is that accessing bindings for aliased variables needs constant number of operations, and is independent of number of variables in the aliased set. A detailed implementation mechanism is given in [3, 4].

3.1 The Model Behavior

The model exploits run time execution efficiency both at the data level during data parallel goal reduction by treating the clause-heads as data for efficient pattern-matching, and control level during the execution of the code of the corresponding subgoals in the selected clause. The forward control flow is divided into three parts: pre-call processing, pre-clause processing, and clause-processing. Pre-call processing is used to perform data parallel goal reduction, and setting up the potential bindings for goal variables. Pre-call processing has four components, namely, *matching constant arguments in a goal to corresponding D-bags* $\mathcal{A}_I$*, data parallel equality of corresponding D-bags* $\mathcal{A}_I$ *and* $\mathcal{A}_J$ *to handle multiple-occurrence variables, and D-intersection of F-bags, derived by matching each goal arguments with the corresponding D-bag of clause arguments, to derive F-bag of unifiable clauses.* Pre-clause processing is used to test the presence of unifiable clauses using the F-bag of unifiable clauses, and passing control to the right clause nondeterministically using shared label. Clause processing is used to handle aliasing of variables, setting up the global registers, handling alternate bindings of shared variables during backtracking, and storing the current state into control stack, before starting next cycle. If at any time the F-bag representing unifiable clauses is empty then backtracking occurs, information is retrived from the control stack, and the previous environment is restored.

There are four rules for associative goal reduction. The first rule is the *rule of matching constants.* The rule states if goal argument is a constant then matching the goal argument Arg_I^G with the corresponding field $\mathcal{A}_I$ gives an F-bag $\mathcal{F}_{\mathcal{I}}$. The rule is used to prune those clauses which do not match for a particular argument. The second rule is the *rule of data parallel equality-test.* The rule states that if two goal arguments Arg_I^G and Arg_J^G have occurrence of same variable then F-bag is derived by performing equality test on both the D-bags $\mathcal{A}_I$ and $\mathcal{A}_J$. The third rule is the *conjunctive rule of non-unifiable clauses* which takes the D-intersection of F-bags derived by matching individual goal arguments with corresponding D-bags in the program. The fourth rule is rule of unification, and is generally used for handling aliased variables.

4 Implementation and Performance Evaluation

The Abstract instruction set of the Dprolog compiler has been divided into five classes, namely, *pattern matching instructions for goal reduction*, *data selection and data movement instructions*, *control instructions*, and *data-parallel computation*. Pattern matching instructions are mainly used during data parallel goal reduction, handling aliasing, and unification. Data movement instructions are used to transfer data between registers, transfer data between between D-bags in the data parallel binding environment, and between heap and registers. Control instructions test the F-bag of unifiable clauses and backtrack to select another binding for a producer. Logical parallel instructions are used to find the unifiable clauses to derive D-intersection of F-bags obtained by individual matches, finding the D-union of aliased sets, and handling negation by complementing the F-bag. Arithmetic computation instructions are of three types, namely, *scalar-scalar* $\rightarrow$ *scalar*, *vector-vector* $\rightarrow$ *vector*, and *scalar-vector* $\rightarrow$ *vector*. In this section, we explain the most commonly used abstract instructions through a compiled program.

Example 1:

The program has three procedures, namely, *p/2, q/2, r/2*. The procedure *p/2* illustrates the intertwining of simple ground facts and complex facts caused by presence of aliasing. The procedure *q/2* is a D-bag of simple facts. The procedure *r/2* is a mixture of ground fact and a non-unit clause. The right hand side of non-unit clause *r/2* exhibits producer-consumer relationship.

p(1,2). p(2, 3). p(3, 4).
p(X, X). * Complex fact due to aliasing *

q(2, 1). q(3, 2). q(4, 3). q(5, 4).

r(2, 2).
r(X, Y) :- p(X, Y), q(2, Y).

The Compiled Code: We give the compiled code for the above program, and the corresponding operational semantics of the abstract instructions. We will denote I_{th} argument of a goal by Arg_I^G and the corresponding D-bag in the program by $\mathcal{A}_I$. Arg_I denotes both Arg_I^G and $\mathcal{A}_I$, U denotes F-bag of unifiable clauses, F denotes F-bag of simple facts without aliased variables, B denotes F-bag of complex clauses (facts with aliased variables or clauses with non-empty clause-body), C indicates unifiable complex clauses, and T_I denotes temporary F-bag.

% Pre-call processing instructions for procedure *p/2*.

DP_p/2: *Match_Register_Arg* Arg_1, U_0

This instruction matches the goal argument Arg_I^G with the D-bag $\mathcal{A}_I$ and finds out D-intersection of resulting F-bag with old value of U_0 to derive new value of U_0.

Test_And_Backtrack U_0

This instruction restores the previous environment if $U_0 = \phi$ and backtracks.

Match_Register_Arg Arg_1, U_0
Test_And_Backtrack U_0

And_Bit_Vectors F_0, U_0, B_0

This instructions derives the D-intersection of two F-bags F_0 and U_0 and stores the result in a new F-bag B_0.

Store_Vector_Id B_0, Arg_1

This instruction stores the index of F-flag B_0 and the index of D-bag $\mathcal{A}_I$ into the heap as a reference to bindings for the variable Arg_I^G.

Store_Vector_Id B_0, Arg_2

% Pre-clause processing instructions for procedure *p/2*

Compliment_Bit_Vector	F_0,	T_1	
And_Bit_Vectors	T_1,	U_0,	C_0

Test_And_Return B_0, L_{10}

This instruction picks up another simple fact if F-flag B_0 is not empty otherwise it passes control to label L_{10}.

L_{10}: *Try_Me_Else* C_0

This instruction restores the previous environment if $C_0 = \phi$ otherwise it picks up next complex clause, and passes the control to the corresponding label.

% Clause Processing for procedure *p/2*

CF3_p/2: *Unify* Arg_1, Arg_2

This instruction is used to alias two arguments having same variable name.

Return

This instruction returns the control to calling Procedure.

% Instructions for procedure *q/2*

DP_q/2:	*Match_Register_Arg*	Arg_2,	U_0	
	Test_And_Backtrack	U_0		
	Match_Register_Arg	Arg_2,	U_0	
	Test_And_Backtrack	U_0		
	And_Bit_Vectors	F_0,	U_0,	B_0
	Store_Vector_Id	B_0,	Arg_1	
	Store_Vector_Id	B_0,	Arg_2	
	Compliment_Bit_Vector	F_0,	T_1	
	And_Bit_Vectors	T_1,	U_0 , C_0	
	Test_And_Return	B_0,	L_{23}	
L_{23}:	*Try_Me_Else*	C_0		

% Instructions for procedure *r/2*

DP_r/2:	*Match_Register_Arg*	Arg_1,	U_0	
	Test_And_Backtrack	U_0		
	Match_Register_Arg	Arg_2,	U_0	
	Test_And_Backtrack	U_0		
	And_Bit_Vectors	F_0,	U_0,	B_0
	Store_Vector_Id	B_0,	Arg_1	
	Store_Vector_Id	B_0,	Arg_2	
	Compliment_Bit_Vector	F_0,	T_1	
	And_Bit_Vectors	T_1,	U_0,	C_0
	Test_And_Return B_0,	L_{34}		
L_{34}:	*Try_Me_Else*	C_0		
P_2_R1:	*Copy_Logical_Register*	Arg_1,	Arg_1	

This instruction copies the reference to the actual value from Arg_1 of the clause-head to the register representing Arg_1 of the first subgoal. The advantage of storing reference is that data movement is avoided.

	Copy_Logical_Register	Arg_2,	Arg_2
	Call	*DP_p/2*	
	Continue	L_{39}	
L_{39}:	*Load_Register*	Arg_1,	T_1, *2*

This instruction loads value *2* in a physical register T_1 and sets up a reference in another register representing Arg_1.

	Copy_Logical_Register	Arg_2,	Arg_2
	Repeat_Else_Backtrack	L_{42}	

This instruction stores the label L_{42} into the control stack, and passes control to the next instruction. Upon backtracking, if the F-flag of unifiable clauses is empty then backtracking takes place. This instruction simulates failure driven iteration until all the unifiable clauses have been tested.

L_{42}:	*Load_Next_Vector_Value*	Arg_2

This instruction gets the index of the D-bag for the variable in Arg_2 from the heap, and retrieves the next value from the D-bag.

	Call	*DP_q/2*
	Return	

4.1 Performance Evaluation

The prototype emulator has been implemented on an HP 730 using ANSI C. The emulator is portable to any architecture which supports data parallel version of C. The results demonstrate that the number of operations needed for associative lookup is independent of number of ground facts. Thirty operations are needed to match a ground fact with two arguments. The number of operations is linearly dependent upon the number of arguments in a query. For each extra argument, nine extra operations are needed to load the value in registers, perform data parallel match, and perform logical ANDing of the previous bit-vector with the new bit-vector obtained during data parallel match.

For a 20 ns clock supported by current technology, and three clock cycles (load-execute-store cycle), the associative look up speed is six hundred thousands $\times$ number of facts for a set of facts with two arguments. In the presence of data parallel scientific computations intertwined with associative lookup, the peek execution speed is limited by the associative look-up speed which will be sixty MCPS (million computations per second) for thousand facts.

When two subgoals of a rule share variables, the data elements in vector bindings for shared variables are processed one at a time. This scenario is the worst case for execution, and the execution speed reduces to two hundred thousand logical inferences per second (LIPS) for one shared variable. The slow down is caused primarily due to the overhead of storing the control thread during forward control flow, register set up, and retrieving the control thread during backtracking. Our results show that the overhead of data parallel matching is less than the overhead of storing the control thread during forward control flow which makes the model suitable for handling flat programs with relations having a large number of arguments.

Note: Following References are limited due to space limitation.

References

[1] A. K. Bansal and J. Potter, "Exploiting Data Parallelism for Efficient Execution of Logic Programs with Large Knowledge Bases", *Proceedings of the Tools for Artificial Intelligence 1990*, Herndon, USA, (Nov. 1990), pp. 674 - 681.

[2] A. K. Bansal and J. L. Potter, "An Associative Model to Minimize Matching and Backtracking Overhead in Logic Programs with Large Knowledge Bases", *The International Journal of Engineering Applications of Artificial Intelligence*, Permagon Press, Volume 5, Number 3, (1992), pp. 247 - 262.

[3] A. K. Bansal, J. Potter, and L. V. Prasad, " Data Parallel Compilation and Extending Query Power of Large Knowledge Bases," In the Proceedings of the International Conference of *Tools for Artificial Intelligence 1992*, pp. 276 - 283.

[4] A. Bansal, "An Associative Compilation Model for Tight Integration of High Performance Knowledge Retrieval and Computing", International Journal on Artificial Intelligence Tools, World Scientific Publishers, in press.

[5] J. A. Feldman and D. Rovner, "An Algol Based Associative Language," *Communications of the ACM*, Volume 12, No. 8, August 1969, pp. 439 - 449.

[6] C. C. Foster, *Content Addressable Parallel Processors*, Van Nostrand Reinhold Co., New York, (1976).

[7] M. Ghandikota, "Implementing Abstract Instruction Set for Logic Programs on Associative Supercomputers", *MS Thesis, Department of Mathematics and Computer Science, Kent State University*, Kent, OH 44242, USA, December 1993.

[8] D. Gries, "The Science of Programming", Monograph, Springer Verlag, Newyork, 1987.

[9] T. Higuchi, T. Furuya, K. Handa, and A. Kokubu, " "IXM2: A Parallel Associative Processor for Semantic Net Processing", in *Proc. of Tools for Artificial Intelligence*, Herndon, USA, (Nov. 1990).

[10] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, Mcgraw Hill Book Company, New york, USA, (1984).

[11] P. Kacsuk, and A. Bale, "DAP Prolog: A Set Oriented Approach to Prolog," *The Computer Journal, Vol. 30, No. 5, 1987*, pp. 393-403.

[12] P. Kacksuk, "DAP Prolog," in *Execution Models of Prolog for Parallel Computers*, Research Monograph, MIT Press, 1990.

[13] K. Knobe, J. D. Lukas, G. L. Steele, "Massively Data Parallel Optimization", The 2nd Symposium of Massively Parallel Computation, Fairfax, Virginia, 1988, pp. 551 - 558.

[14] Kowalski, R., *Logic for Problem Solving*, Elsevier-North Holland, (1979).

[15] Z. Manna and R. Waldinger, "The Logical Basis for Computer Programming", Volume1: Deductive Reasoning, Addison Wesley, 1985.

[16] J. L. Potter, "Data Structures for Associative Supercomputers", *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Processors*, Fairfax, USA, (Oct. 1988), pp. 77 - 84.

[17] J. L. Potter, *Associative Computing*, Plenum Publishers, Newyork, (1992).

[18] L. V. Prasad,"Compiling Logic Programs to Incorporate Data-parallelism on Associative Supercomputers", MS Thesis, Department of Mathematics and Computer Science, Kent State University, Kent, OH 44242, USA, December 1993.

[19] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, (1986).

[20] A. Takeuchi and K. Furukawa, "Parallel Logic Programming Languages", *Lecture Notes In Computer Science, Vol. 225*, Springer Verlag, Newyork, (July 1986), pp. 242 - 254.

[21] D. H. D. Warren, "An Abstract Prolog Instruction Set", *Technical Report 309*, SRI International, (October 1983).