A General Framework for Heterogeneous Associative Logic Programming

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

August 6, 1996

Associative computation is characterized by search by content and data parallel computation. Search by content paradigm is natural to scalable high performance heterogeneous computing since use of tagged data makes data independent of implicit addressing mechanisms. In this paper, we present an algebra for associative logic programming, an associative resolution scheme, and a generic framework of associative abstract instruction set. The model is based on integration of data alignment and use of two types of bags: data element bags and filter bags of boolean values to select and restrict computation on data elements. Use of filter bags integrated with data alignment reduces the computation and data transfer overhead; use of tagged data reduces overhead of preparing data before data transmission. The abstract instruction set has been illustrated by an example. Performance results for a simulation in a homogeneous address space are presented.

Keywords: Artificial intelligence, Associative computing, Data parallel computing, Heterogeneous computing, Scalable high performance computing, Logic programming.

1 Introduction

Associative computation is characterized by seamless intertwining of search by content and data parallel computation [14]. This intertwining facilitates the integration of logic programming paradigm and data parallel computations on a heterogeneous collection of massive parallel machines. This work generalizes the associative computation model which started with our earlier work to exploit associative computation on SIMD architectures [1, 2].

Heterogeneous collection of machines have different address space, and there is no uniform mechanism to handle this heterogeneity. Current popular approaches to handle heterogeneity [15] use message passing paradigm for information transfer. However, the structured data, if transferred across heterogeneous address space, has to be linearized (unpacked) in one address space, and restructured (packed) in another address space to make use of message passing paradigm.

1. Unpacking algorithm goes through chain of references to linearize the data,
2. Packing algorithm has to build up chain of references, and
3. Many implicit attributes have to be tagged during message transfer.

In order to reduce this overhead, associative computing paradigm uses sedentary data, uses duplicate copies of tagged data in heterogeneous address space to reduce data transfer whenever possible, and exploits intertwining of search by content and data parallel computation. The motivation for this paper is to formalize the rules for associative computation model of logic programming, and develop a general framework of the instruction set which will facilitate implementation of the associative model of logic programming on a heterogeneous collection of different massive parallel architectures. The major contribution in this paper is the extension of associative computation formalism to resolution process of logic programming, and description of a generic framework of the abstract instruction set. This formal framework is the basis for our effort to develop heterogeneous abstract machines suitable for heterogeneous collection of high performance machines having heterogeneous address space.

In addition, the associative computational model has direct applications for a class of problems which require integration of high performance symbolic computing, knowledge retrieval, and scientific computation. Some examples are geographical knowledge bases, genome sequence analysis, and intelligent simulation of complex engine processes.

2 Preliminaries

Basic definitions of logic programming are available in [11]. A logic program is a set of Horn clauses of the form A :- B_1, ..., B_N ($N \geq 0$). Facts, known as the data base part, are clauses with empty body; Rules are clauses with non-empty body. Solving a logical query is based on repeated resolution of a goal to conjunction of subgoals using unification - a combination of pattern matching and value binding process - until the goal can be looked up in the database (facts).

A *multiple-occurrence variable* occurs more than once in a goal. A *shared variable* occurs in more than one subgoals of a clause. A *producer* is the first occurrence of a shared variable, and generates a bag of values for a shared variable. Other occurrences of the shared variable are *consumers*. *Aliased* variables share the same value; binding one of the aliased variables automatically binds others with the same value. Aliasing is an equivalence relationship. We will denote aliasing by symbol $\simeq$.

A *simple fact* has no multiple occurrence variable, and a *complex fact* has at least one multiple occurrence variable. A complex clause is either a complex fact, or a clause with non-empty clause-body.

2.1 Associative Computation

Associative computation has two major components: search by content and data parallel computation. Use of search by content facilitates the use of heterogeneous address space since no implicit indexing scheme is used to address a data element. Using this facility, data can be distributed across different address space, and different processors may work on different data elements exploiting data parallelism.

A generic architecture on which an associative computation model can be implemented is an ordered collection 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. An associative search of a field for a specific value sets up a bag of Boolean results which are used to filter an abstract unit of computation on different data elements of a bag. An instruction is broadcast to each cell simultaneously to initiate the same abstract computation [1] on different data elements. The flow of control is effected by generating, saving, and transferring the

[1] an abstract computation may have multiple different mutually exclusive definitions, and each definition may be a sequence of commands

bit-vectors corresponding to Boolean results. A SIMD architecture satisfies this criteria. However, the only major difference between SIMD computation and other generic architecture is that SIMD architecture supports fine grain implicit synchronization for every data parallel computation.

2.2 Abstract Data Representation

A bag is a collection of data items with multiple possible occurrences of a value. Two bags are associated if the individual elements in the bags are indexwise paired. The advantage of pairing is that the knowledge of an element in one bag is sufficient to derive the value of the corresponding element in the other bag. We define two types of data structures, namely, a D-bag and an F-bag to explain associative computations. Intuitively, D-bags are used to represent data elements, and F-bags are used to filter the data elements.

A *D-bag*, denoted by $\mathcal{D}$, is defined as an ordered bag which includes null value $\perp$ and top value $\top$. $\perp$ denotes the absence of any value, and $\top$ denotes an undefined value. $\top$ can be instantiated with any value. For example, $\{$ *2,* $\perp$*, 3* $\}$ is a D-bag. However, $\{$ *2,* $\perp$*, 3* $\} \neq \{$ $\perp$*, 2, 3* $\}$ since D-bags are ordered. The null element $\perp \preceq$ every element in the D-bag. We denote a D-bag with all elements equal to $\perp$ as ϕ. A D-bag is mapped on a generic architecture such that by accessing same index on S_i - the local storage on the architecture - all the elements of a D-bag can be accessed from different cells.

A D-bag of M-tuples of the form $\{ < d_{11}, ..., d_{1M} ..., < d_{N1}, ..., d_M > \}$ is stored as an association of M D-bags indexwise 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 aligned D-bags as $\mathcal{D}_1 \bigoplus \mathcal{D}_2 \bigoplus ..., \bigoplus \mathcal{D}_M$. An F-bag is a D-bag with Boolean values *true* and *false*. The Boolean values *true* and *false* are treated synonymously with the values "1" and "0" respectively. We treat *false* (or "0") $\prec$ *true* (or "1"). A F-bag of *1s* is denoted by $\mathcal{F}^1$, and a F-bag of *0s* is denoted by $\mathcal{F}^0$.

We also define the notion of D-inclusion, D-equality, D-union, and D-intersection of two D-subbags. These notions are different from there set-theoretic counterparts due to the presence of order in D-bags and F-bags, and inclusion of $\perp$ and $\top$ in D-bags, and pairwise comparison of corresponding elements. A D-bag $\mathcal{D}_1 = \{ d_{11}, ..., d_{1N}\}$ is 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 the sake of clarity, We refer to the inclusion of a D-bag by D-inclusion, and denote D-inclusion by $\sqsubseteq$. For example, $\{$ *4,* $\perp$*, 5, 6* $\} \sqsubseteq \{$ *4, 3, 5, 6* $\}$ since $\perp \prec$ *3*. D-union of two D-subbags $\{d_{11}, ..., d_{1N}\}$ and $\{d_{21}, ..., d_{2N}\}$ 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 $d_{1I} \preceq d_{2I}$. We denote D-union by $\bigsqcup$. For example, $\{ \perp, b, c \} \bigsqcup \{ a, b, \perp \}$ derives $\{ a, b, c \}$. D-intersection of two D-subbags $\{d_{11}, ..., d_{1N}\}$ and $\{d_{21}, ..., d_{2N}\}$ 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 $d_{2I} \preceq d_{1I}$. we denote D-intersection by $\sqcap$. For example, $\{$ *2, 3,* $\perp$ $\} \sqcap \{$ $\perp$*, 3, 4* $\}$ derives the D-subbag $\{$ $\perp$*, 3,* $\perp$ $\}$. D-equality of two D-bags $\mathcal{D}_1$ and $\mathcal{D}_2$ derives an F-bag $\mathcal{F}$ such that $\forall I_{1 \leq I \leq N}\ d_{3I}\ (\in \mathcal{F}) = 0$ if $d_{1I} \neq d_{2I}$ or $d_{1I} = \perp$ or $d_{2I} = \perp$; $d_{3I} = 1$ if $d_{1I} = d_{2I}$ and $d_{1I} \neq \perp$ and $d_{2I} \neq \perp$. We denote D-equality by $\doteq$. For example, $\{ a, b, \perp\}$ $\doteq \{a, c, \perp\}$ derives the F-bag $\{1, 0, 0\}$.

We denote application of an F-bag on a D-bag by $\mathcal{D} \bigotimes \mathcal{F}$. Under the assumption *false* $\prec$ *true*, D-union of F-bags is implemented by logical-OR of the corresponding logical bit-vectors, and D-intersection of F-bags is implemented by logical-AND of the corresponding logical bit-vectors.

3 An Algebra for Associative Computation

Our model is based upon seventeen 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 update*.

3.1 Laws of Data Association

This subsection describes three rules of association of data. Rule (1) describes the formation of association, Rule (2) describes the isomorphism of nested associations, and Rule (3) describes the isomorphism under permutation of an association. The implication of isomorphism of association is that the information in an association is not altered by nesting or by permutation.

Rule (1) states that a D-bag of M-tuples is given by the association of M D-bags such that corresponding elements are aligned by index. For example, $\{$ *a, 2, 3, 4* $\} \bigoplus \{$ *b, 5, 6, 7* $\}$ is equivalent to $\{ < a, b >, < 2, 5 >, < 3, 6 >, < 4 , 7 > \}$.

Rule (2) states that nested associations are isomorphic For example, $\{$ *1, 2* $\} \bigoplus (\{$ *3, 4* $\} \bigoplus \{$ *5, 6* $\})$ is isomorphic to $(\{$ *1, 2* $\} \bigoplus \{$ *3, 4* $\}) \bigoplus \{$ *5, 6* $\}$, and both are isomorphic to $\{ < 1, 3, 5 >, < 2, 4, 6 > \}$.

Rule (3) states that permutations in an association are not symmetric but 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) $\in \mathcal{D}_2 \bigoplus \mathcal{D}_1$.

3.2 Laws of Associative Search

This subsection describes two laws of associative search. Rule (4) describes the mapping of a D-bag to an F-bag based upon an associative search of an element, and Rule (5) describes the derivation of information from other D-bags by searching in one of the D-bags in an association. Rule (4) is necessary for data-parallel goal reduction, and Rule (5) is necessary for deriving the bindings of variables in a goal after unification.

Rule (4) 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* }.

Rule (5) 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 $\{4, \top, \top \}$ 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 $\{ < 4, 5, 6 >, \bot, < 4, 9, 10 > \}$.

3.3 Laws of Selection

In this subsection, we describe six rules for selection of data elements from an association using an F-bag or computation on F-bags. Computation on F-bags reduces computation overhead when selecting subbag of data elements; transmission of F-bags across heterogeneous address space reduces data movement overhead in comparison to the transmission of association of D-bags. Rule (6) is used to pick up data elements from the corresponding tuples; Rule (7) - Rule (10) are used to reduce the data movement if D-bags of an association are present on different address space; Rule (8) is used to

reduces the creation overhead of subbags of a D-bag. Rules (11) and (12) have been implicitly used to derive the property of many low level computations.

Rule (6) 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, $\{\ 3,\ 5,\ 6\ \} \bigotimes \{0,\ 1,\ 0\}$ derives $\{\ \perp,\ 5,\ \perp\ \}$.

Rule (7) 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 derives the D-bag $\{ < 4,\ a >, \perp, < 6,\ c > \}$. Similarly, $\{(2,\ 3), (4,\ 5), (6,\ 7)\} \bigotimes \{\ 1,\ 1,\ 0\ \}$ 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 $\{ < 2,\ 3>, <4,\ 5>, \perp\}$.

Rule (8) 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, $\{\ 5,\ 6,\ 7\ \} \bigotimes \{\ 1,\ 0,\ 1\ \}$ derives the D-bag $\{\ 5,\ \perp,\ 6\ \}$. While $\{\ 5,\ 6,\ 7\ \} \bigotimes \{\ 1,\ 0,\ 0\ \}$ derives the D-bag $\{\ 5,\ \perp,\ \perp\ \}$.

Rule (9) 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, $\{2,\ 3,\ 4\} \bigotimes (\{1,\ 0,\ 0\} \bigsqcup \{0,\ 1,\ 0\ \}\ ) \equiv (\{2,\ 3,\ 4\} \bigotimes \{1,\ 0,\ 0\}) \bigsqcup (\{2,\ 3,\ 4\} \bigotimes \{0,\ 1,\ 0\ \}) \equiv \{2,\ 3,\ 4\} \bigotimes \{1,\ 1,\ 0\}$ which derives $\{\ 2,\ 3,\ \perp\ \} \equiv \{\ 2,\ \perp,\ \perp\} \bigsqcup \{\ \perp,\ 3,\ \perp\}$.

Rule (10) 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, $\{2,\ 3,\ 4\} \bigotimes (\{1,\ 1,\ 0\} \sqcap \{0,\ 1,\ 1\ \}\ ) \equiv \{2,\ 3,\ 4\} \bigotimes \{0,\ 1,\ 0\}$ which derives $\{\ \perp,\ 3,\ \perp\ \} \equiv \{\perp,\ 3,\ 4\} \sqcap \{3,\ \perp,\ 4\}$.

Rule (11) states that F-bag formed by D-equality on two associations of D-bags $\mathcal{D}_{11} \bigoplus ... \ \mathcal{D}_{1M}$ and $\mathcal{D}_{21} \bigoplus ... \ \mathcal{D}_{2M}$ is equivalent to $\mathcal{D}_{11} \doteq \mathcal{D}_{21} \sqcap ... \ \mathcal{D}_{1M} \doteq \mathcal{D}_{2M}$. For example, $\{\ \perp, < 2,\ 3,\ 4\ >, < 5,\ 6,\ 7\ > \} \doteq \{\ \perp, < 2,\ 3,\ 4\ >, < 6,\ 6,\ 7\ > \equiv \{\ \perp,\ 2,\ 6\} \doteq \{\ \perp,\ 2,\ 6\} \sqcap \{\ \perp,\ 3,\ 6\} \doteq \{\ \perp,\ 3,\ 6\} \sqcap \{\ \perp,\ 4,\ 7\} \doteq \{\ \perp,\ 4,\ 7\} \Rightarrow \{\ 0,\ 1,\ 0\ \} \sqcap \{\ 0,\ 1,\ 1\ \} \sqcap \{\ 0,\ 1,\ 1\ \} \Rightarrow \{\ 0,\ 1,\ 0\ \}$.

Rule (12) 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, $\{2,\ 3,\ 4\} \times \{\ true\ \} \equiv \{2,\ 3,\ 4\} \bigotimes \{1,\ 1,\ 1\}$ which results into the D-bag $\{2,\ 3,\ 4\}$.

Rule (13) states that Cartesian product of a D-bag $\mathcal{D}$ with *false* is equivalent to $\mathcal{D} \bigotimes \mathcal{F}^0 \Rightarrow \phi$.

3.4 Laws of Data Parallel Computation

In this subsection, we describe two rules for data parallel computation.

Rule (14) 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 abstract computation on every element of the associated fields. Any computation involving $\perp$ maps onto $\perp$. For example, $\{2,\ 3,\ \perp\} *^D \{3,\ 4,\ \perp\}$ derives $\{6,\ 12,\ \perp\}$.

Rule (15) 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 derives the D-bag $\{8,\ 12,\ 16\}$.

3.5 Laws of Associative Update

In this subsection, we describes two rules for associative update: Rule (16) describes associative insertion of a tuple in an association, and Rule (17) describes data parallel release of tuples from an association.

Rule (16) 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 to ($\mathcal{D}_1^U$ $\bigoplus$, ..., $\bigoplus$ $\mathcal{D}_N^U$) where $\mathcal{D}_I^U$ denotes the updated bag. Each $\mathcal{D}_I^U = \mathcal{D}_I \bigcup \{d_I\}$.

Rule (17) states that by associatively searching in one field, the associated data elements in the other field can be released in constant number of computations. For example, associative search for a tuple {4, $\top$, $\top$ } from the tuple { *< 4, 5, 6 >, < 3, 7, 9 >, 4, 9, 10 >* } derives an F-bag { *1, 0, 1* }. Complement of { *1, 0, 1* } derives { *0, 1, 0*}. Application of { *0, 1, 0*} derives the D-subbag as { $<$ $\perp$, *< 3, 7, 9 >*, $\perp >$ } deleting all the tuples which have value *4*.

3.6 Extending Associative Rules for Logic Programming

In this Section, we extend the algebra of associative computing to handle logical variables and their bindings. A D-bag in logic program contains both constants and variables. These extended rules have been used to exploit associative computation for binding, goal reduction, identification of bag unifiable clauses, and selection of a unifiable clause. We will refer to D-bag of substitutions as D-substitution.

Rule (18) states if a variable X is matched with a D-bag then it generates an F-bag $\mathcal{F}^1$ and the D-bag of resulting substitutions is given by a D-bag $X_I/$ {d_1, ..., d_N} which is also equivalent to D-substitution { X_I/d_1, ..., X_I/d_N}.

Rule (19) states that if a constant is matched with a D-bag { d_1, ..., d_N} then it generates an F-bag { f_1, ..., f_N} as follows:

$f_I = 1$ if d_I = the given constant
$f_I = 1$ if d_I is a variable
$f_I = 0$ otherwise

The resulting D-bag of substitutions { σ_1^D, ..., σ_N^D} is given by:

$\sigma_I^D = \top$ if d_I = the given constant
$\sigma_I^D = d_I$/the constant value if d_I is a variable
$\sigma_I^D = \perp$ otherwise

For example, matching a value d with the D-bag { *X, Y, d, b*} gives the F-bag {*1, 1, 1, 0* } and the correspondingD-bag of substitutions as { *X/d, Y/d*, $\top$, $\perp$}.

Rule (20) states that data-parallel matching a pair of multiple occurrence-variable < X, X> with a pair of D-bags $< d_{11}, ..., d_{1N}>$ and $< d_{21}, ..., d_{2N} >$ derives an F-bag { f_1, ..., f_N } by extending data parallel equality as follows:

$f_I = 1$ if $d_{1I} \sqcap d_{2I}$ and d_{1I} and d_{2I} are not variables
$f_I = 1$ if $d_{1I} \not\simeq d_{2I}$ and d_{1I} or d_{2I} is a variable
$f_I = 1$ if $d_{1I} \simeq d_{2I}$
$f_I = 0$ otherwise

The resulting D-bag of substitutions for above four cases are given as follows:

$\sigma_I^D = X/(d_{1I} \sqcap d_{2I})$
$\sigma_I^D = \{X/d_{1I},\ d_{1I}/d_{2I}\}$ or $\{X/d_{1I},\ d_{1I}/d_{2I}\}$
$\sigma_I^D = \{d_{1I}/X,\ d_{2I}/X\}$
$\sigma_I^D = X/\perp$ otherwise

For example, matching a multiple-occurrence variable $< X,\ X >$ with a pair of D-bags of the form $< \{a,\ b,\ Z\},\ \{c,\ b,\ W\} >$ will derive the F-bag $\{0,\ 1,\ 1\}$ and the D-bag of substitution as $\{X/\perp,\ X/\top,$ $\{\ X/W,\ W/Z\}\ \}$.

3.7 Substitution and Composition

Our definition of substitution and application is similar as [12]. Given a substitution θ and a logical term t, the application of θ on t is denoted by $t\ \theta$. A *D-substitution* is a D-bag of substitutions such that every element is a $\perp$ meaning that match has failed, a $\top$ meaning that matching is syntactically successful, or a set of the form $\{\ X_{I1}/t_{I1},\ ...,\ X_{IN}/t_{IN}\ \}$ where t_I could either be a constant symbol, a variable, or a D-bag of constants, a $\perp$, or a$\top$. Given two D-substitutions ${\Sigma^D}_1$ of the form $\{\ \sigma_{11}^D,\ ...,$ $\sigma_{1I}^D,\ ...,\ \sigma_{1N}^D\}$ and ${\Sigma^D}_2$ of the form $\{\ \sigma_{21}^D,\ ...,\ \sigma_{2I}^D,\ ...,\ \sigma_{2N}^D\}$, the composition ${\Sigma^D}_1 \bullet {\Sigma^D}_2$ is given by a D-bag $\{\ \sigma_{31}^D,\ ...,\ \sigma_{3I}^D,\ ...,\ \sigma_{3N}^D\ \}$. Each σ_{1I}^D is either a $\perp$, a $\top$, a variable, a constant symbol, a singleton value X_I/t_I, or a set of values $\{\ X_{I1}/t_1,\ ...,\ X_{IN}/t_{IN}\ \}$; each σ_{2I}^D is either a $\perp$, a $\top$, a variable, a constant symbol, a singleton value Y_I/w_I, or a set of values $\{\ Y_{I1}/w_{I1},\ ...,\ Y_{IN}/w_{IN}\ \}$. Each σ_{3I}^D is:

(i) $\perp$ if $\sigma_{1I}^D = \perp$ or $\sigma_{2I}^D = \perp$
(iia) σ_{1I}^D if $\sigma_{2I}^D = \top$
(iib) σ_{2I}^D if $\sigma_{1I}^D = \top$
(iii) $X_I/t_{1I} \sqcap\ t_{2I}$ if $X_I = Y_I$
(iv) $\{\ X_I/t_I,\ Y_I/w_I\ \}$ if $X_I \neq Y_I$ and $t_I \simeq w_I$
(v) $\{\ X_{I1}/t_{I1}s_{2I},\ ...,\ X_{I1}/t_{I1}s_{2I}\} \bigcup s_{2I}$

Application of $\perp$ on a logical term will derive $\perp$, and application of $\top$ on a logical term will derive the logical term. For example, given the D-substitutions $\{\ \perp,\ \top,\ X_3/a,\ \{X_{41}/P,\ X_{42}/c\}\}$ and $\{\ Y_1/a,$ $Y_2/b,\ X_3/b,\ \{\mathrm{P/d}\}\ \}$ derives D-substitution $\{\ \perp,\ Y_2/b,\ X_3/\perp,\ \{\ X_{41}/d,\ P/d,\ X_{42}/c\}\ \}$.

4 Associative Model of Logic Programming

In this section, we define abstract data representations, a set of basic computations on these data representations, and describe the computation model for associative logic programming.

The associative computation model maps a logic program as a pair of associations of the form $< \mathcal{L} \oplus \mathcal{N} \oplus \mathcal{P} \oplus \mathcal{A}_1 \oplus$,..., $\oplus\ \mathcal{A}_N,\ \mathcal{L} \oplus \mathcal{C} >$. The first element of the pair represents a 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 sequence of compiled abstract instructions of the corresponding clause-body; $\mathcal{N}$ is the priority of the clause-heads to be selected for resolution; $\mathcal{P}$ is the D-bag of procedure-names; $\mathcal{A}_I$ $(1 \leq I \leq N)$ is the D-bag of I_{th} argument in set of the clause-heads in a program; $\mathcal{C}$ is a D-bag of 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.

Binding environment contains two parts: data parallel binding environment and shared space (popularly called heap). The binding environment in the model is represented as an association of

D-bags and F-bags. The binding is associated with time stamp for efficient recovery (based upon associative search on time stamp) of previous environment upon backtracking and is associated with index S_I of the local address space for reference. An association of D-bags is used to represent a set of tuples such that each D-bag represents the elements occurring at the same position in the tuples. Searching for a data item in a D-bag derives an F-bag which is used to select a D-subbag. A data parallel computation performs the same computation on every element of a D-bag or F-bag. The details of a binding environment for an implementation for a SIMD based architecture is given in [2], and is outside the scope of this paper. Readers can come with their own scheme within this framework of computation.

4.1 Associative Resolution

A goal is represented as a tuple of data elements. Given a conjunctive goal of the form :- $A_1 \bigwedge$, ..., $\bigwedge A_N$ ($N \geq 0$), a subgoal A_I is picked at random, and each individual argument is matched with the corresponding D-bag using rules (18), (19), or (20). At the end of each matching, the resulting F-bag $\mathcal{F}_I$ is D-intersected with $\sqcap_{J=0}^{J=I-1}\mathcal{F}_J$ (where $\mathcal{F}_0$ is $\mathcal{F}^1$) - the cumulative result of previous F-bags. The corresponding D-substitution is given by Σ_I^D. If $\sqcap_{J=0}^{J=I}\mathcal{F}_J = \mathcal{F}^0$ then further data parallel reduction is terminated and backtracking occurs. At the end of successful data parallel goal reduction, the D-substitution, denoted by Θ^D, is given by $\Sigma_1^D \bullet ... \bullet \Sigma_N^D$. During composition of D-substitution, associative computation and full data parallelism is exploited for rules (i) to (iv). However, in presence of aliased variables, rule (v) enforces unification. To avoid the sequentiality, unification of aliased variables is deferred until a specific clause is picked up for resolution.

The D-bag of unifiable clauses is selected by applying $< \mathcal{L} \oplus \mathcal{N} \oplus \mathcal{P} \oplus \mathcal{A}_1 \oplus$,..., $\oplus \mathcal{A}_N > \otimes \sqcap_{I=0}^{I=N} \mathcal{F}_I$. The search strategy for the resolution tree is based upon prioritizing the selection of the unifiable clauses in the order simple facts followed by complex facts and complex rules. The rationale for this selection is to avoid the nonterminating branches, and to improve the execution efficiency to derive solutions. This strategy is directly supported by search-by-content, and is more powerful compared to textual order strategy as illustrated:

Example 1:

p(X, Y) :- q(X, Z), r(Z, Y).
p(a, b).
q(X, Y) :- q(Y, X).
q(b, c).
r(c, 4).
r(c, 5).

In the above example, a the simple fact in procedure *p/2* is picked up first to get an early solution. For an alternate solution of *p/2* complex clause is picked. However, nonterminating branch in *q/2* is avoided; execution of *q/2* gives value of variable *Z* as *c* which when applied in subgoal *r/2* returns a D-bag { *4, 5* }.

The cumulative D-substitution $\Sigma_1^D \bullet ... \bullet \Sigma_N^D$, denoted by Θ^D, is of the form { θ_1^D, ..., θ_N^D} where θ_I^D is a partial unifier for each potentially unifiable clause-head. The corresponding substitution for alised variables, denoted by θ_I^S is derived after selecting the potentially unifiable clause. The final mgu θ_I is derived by $\theta_I^D \bullet \theta_I^S$. θ_I is $\perp$ if either θ_I^D or θ_I^S is $\perp$. It can easily be verified that if an element in $\sqcap_{I=0}^{I=N} F_I$ is *0* then the corresponding θ_I^D is $\perp$. if a goal A_J is used for resolution with a clause of the form A :- B_1, ..., B_Q ($Q \geq 0$) then, after the resolution the resolvent, is given by :- (A_1, ..., A_{I-1},

B_1, ..., B_Q, ... A_{I+1}, ..., A_N) θ. If $\theta = \perp$ then application of $\perp$ to previous substitutions derives $\perp$ which is equivalent to failure; alternate θ_I is selected according to associative search strategy. Other than this difference, the treatment for resolution tree and the proof for soundness is very similar to [12], and has been omitted from this paper.

5 An Associative Abstract Machine

In this section, we describe briefly an associative abstract instruction set based on the above model. A detailed scheme for the implementation of associative computation model on SIMD architecture is given in [2]. Readers can come up with their own implementation model based on this framework.

5.1 An Abstract Instruction Set

The Abstract instruction set of the Dprolog compiler has been divided into five classes: *instructions for data parallel goal reduction*, *instructions for data selection instructions for data movement*, *instructions for control flow*, and *instructions for data-parallel computation*. Data movement instructions are used to transfer data between registers, transfer data between between D-bags in the data parallel binding environment, between heap and registers, and between two heterogeneous address spaces. Control instructions are used to test $\mathcal{F}^0$, backtrack to select another binding for a producer, and save environment on the control stack. Logical parallel instructions are used to select the unifiable clauses by deriving the D-intersection of the corresponding F-bags, finding the D-union of aliased sets. and partial handling of negation for facts using complement. Arithmetic computation instructions are of three types, namely, *scalar-scalar* $\rightarrow$ *scalar*, *vector-vector* $\rightarrow$ *vector*, and *scalar-vector* $\rightarrow$ *vector*. A description of key abstract instructions is given in Figure 1.

5.2 Applying Instruction Set to Solve a Query

For associative goal reduction, a sequence of instructions *data_parallel_arg_match* or *data_parallel_unify*, *d_intersect_F-bag*, and *backtrack_if_*$\mathcal{F}^0$ are called repeatedly until all the goal-arguments are processed or the F-bag of unifiable clauses is empty.

For handling simple facts, *select_next_simple_fact* derives the alternate fact to be processed. In case, there are no alternate simple facts, the control is transferred to code area where complex clauses are handled.

For handling complex clauses, *compliment_F-bag* and *d_intersect_F-bag* are used to remove D-bag of simple unifiable facts from D-bag of unifiable facts; *select_next_clause* is used to process next unifiable complex clause; *unify* is used for handling aliased variables; and *copy_reference* is used to pass the bindings from a goal to a subgoal.

Producer consumer relationships are divided in two categories:

1. A producer produces a D-bag $\mathcal{D}_1$, and the corresponding consumer is a function working on consumer occurrence. This is handled by data parallel computation on every data-element bound to the consumer variable. The example for the first case is a user-defined goal followed by an arithmetic operation, a set operation, or a comparison operation.

2. A producer produces a D-bag and consumer is in a goal which consumes one scalar value at a time. This case includes goals with the corresponding consumer occurrence using associative

data_parallel_arg_match $\mathcal{A}_I, \mathcal{U}_I$	matches a goal argument with the corresponding D-bag $\mathcal{A}_I$ (see Rules 18 and 19), derives the F-bag $\mathcal{F}_I$ which is D-intersected with $\mathcal{U}_{I-1} = \sqcap_{J=0}^{J=I-1}\mathcal{F}_J$) to derive new value of $\mathcal{U}_I = \sqcap_{J=0}^{J=I}\mathcal{F}_J$)
data_parallel_unify $\mathcal{A}_I$, $\mathcal{A}_J, \mathcal{U}_I$	perform data parallel equality test between two D-bags $\mathcal{D}_I$ and $\mathcal{D}_J$ to handle multiple-occurrence goal variables (see Rule 20). The resulting F-bag is d-intersected with $\mathcal{U}_{I-1} = \sqcap_{J=0}^{J=I-1}\mathcal{F}_J$) to derive new value of $\mathcal{U}_I = \sqcap_{J=0}^{J=I}\mathcal{F}_J$)
unify A_1, A_2	performs conventional unification of two logical terms A_1 and A_2, and is used to unify aliased variables
d_intersect_F-bag $\mathcal{F}_I$, $\mathcal{F}_J$, $\mathcal{F}_K$	computes and stores $\mathcal{F}_I \sqcap \mathcal{F}_J$ into $\mathcal{F}_K$
d_union_F-bag $\mathcal{F}_I$, $\mathcal{F}_J$, $\mathcal{F}_K$	computes and stores $\mathcal{F}_1 \sqcup \mathcal{F}_2$ into $\mathcal{F}_K$
d_compliment_F-bag $\mathcal{F}_I$, $\mathcal{F}_J$	computes and stores $\neg\ \mathcal{F}_I$ into $\mathcal{F}_J$
save_bag_id A_I, $\mathcal{D}_I$	stores the reference of D-substitution (of variable in A_I) $\mathcal{D}_I$ in the heap. $\mathcal{D}_I$ itself is stored in the data-parallel environment
save_subbag_id A_I, $\mathcal{D}_I$, $\mathcal{F}_I$	saves the references D-bag $\mathcal{D}_I$ and F-bag $\mathcal{F}_I$ needed to compute D-subbag $\mathcal{D}_I \bigotimes \mathcal{F}_I$.
load_next_binding A_I	picks up next binding from D-substitution of a variable argument corresponding if all the elements of the corresponding D-substitutions are not $\perp$. If all the elements of $\mathcal{D}_I$ are $\perp$ then backtracking takes place. After an element is picked, the corresponding element in D-substitutions is replaced by $\perp$

Figure 1: A Generic Framework of Associative Abstract Instruction Set

load_scalar_binding A_I	picks up a scalar binding for a variable in argument A_I
copy_reference A_1, A_2	copies the reference of the binding of a variable in argument A_1 of a goal to a variable in argument A_2 of a subgoal. The data may require transfer of data from one address space S_I to another address space S_J. The advantage of copying the reference is that communication and structuring overhead moving large D-bags of attributes of a data element are reduced
*backtrack_if_*$\mathcal{F}^0$ $\mathcal{F}_I$	backtracks and restores previous environment if $\mathcal{F}_I$ is $\mathcal{F}^0$
*repeat_until_*$\mathcal{F}^0$ $Label_I$	saves label $Label_I$ on the control stack, and reverts the direction of control flow from backtracking to forward flow
select_next_simple_fact $\mathcal{F}_I$, $Label_I$	selects another unifiable simple fact from $< \mathcal{L} \bigoplus \mathcal{N} \bigoplus \mathcal{P} \bigoplus \mathcal{A}_1 \bigoplus$,..., $\bigoplus \mathcal{A}_N$, $\mathcal{L} \bigoplus \mathcal{C} > \bigoplus \mathcal{F}_I$. After selection of clause, new value of $\mathcal{F}_I$ is derived by marking that entry *0* in $\mathcal{F}_I$. If $\mathcal{F}_I$ is $\mathcal{F}^0$ then control is passed to code sequence starting from label $Label_I$
select_next_clause $\mathcal{F}_I$	selects randomly another unifiable complex clauses identified by $< \mathcal{L} \bigoplus \mathcal{N} \bigoplus \mathcal{P} \bigoplus \mathcal{A}_1 \bigoplus$,..., $\bigoplus \mathcal{A}_N$, $\mathcal{L} \bigoplus \mathcal{C} > \bigotimes \mathcal{F}_I$. After the selection of a clause, new value of $\mathcal{F}_I$ is derived by marking that entry *0* in $\mathcal{F}_I$. If $\mathcal{F}_I$ is $\mathcal{F}^0$ then backtracking occurs.
call $Label_I$	saves the current environment on the control stack, and passes control to label $Label_I$ corresponding to first instruction of a procedure
return	restores the previous environment, and *returns* from the called procedure

Figure 1: A Generic Framework of Associative Abstract Instruction Set (continued)

search in the clause-heads. This is handled by repeated backtracking to fetch a new value and processing the value until the D-bag bound to the producer is empty. This repeated backtracking is achieved by a pair of abstract instructions *repeat_until_*$\mathcal{F}^0$ and *load_next_binding*.

In addition, there are multiple data parallel arithmetic operations, data parallel arithmetic comparison operations, data parallel logical operations.

6 An Illustrative Example

In this section, we explain commonly used abstract instructions through a compiled program. Example 2 illustrates the compilation of programs with simple facts, complex facts, and complex clause with shared variables.

Example 2:

The program has three procedures, namely, *p/2, q/2, r/2*. The procedure *p/2* illustrates compilation 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 simple fact and a complex clause. The right hand side of the complex clause in *r/2* exhibits producer-consumer relationship: variable *Y* has producer occurrence in subgoal *p/2* and consumer occurrence in subgoal *q/2*.

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). * Simple facts *

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

6.1 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 A_I^G and the corresponding D-bag in the program by $\mathcal{D}_I$. $\mathcal{A}_I$ denotes both A_I^G and $\mathcal{D}_I$; U_K denotes $\sqcap_{J=0}^{J=I-1}\mathcal{F}_J$); $\mathcal{F}_K^F$ denotes F-bag of simple facts; $\mathcal{F}_K^{UF}$ denotes F-bag of unifiable simple facts; $\mathcal{F}_K^C$ indicates unifiable complex clauses; and $\mathcal{F}_K^T$ denotes a F-bag to store temporary result. We give abstract description of the instructions:

% Data parallel goal reduction for procedure *p/2*.

p/2:	*data_parallel_arg_match*	$\mathcal{A}_1$,	$\mathcal{F}_0^U$		% match first argument of goal and store F-bag in $\mathcal{F}_0^U$
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$			% backtrack if F-bag $\mathcal{F}_0^U$ is empty
	data_parallel_arg_match	$\mathcal{A}_2$,	$\mathcal{F}_0^U$		% match second argument and store cumulative result in F-bag $\mathcal{F}_0^U$
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$			% backtrack if F-bag $\mathcal{F}_0^U$ is empty
	d_intersect_F-bag	$\mathcal{F}_0$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UF}$	% Identify unifiable simple facts in $\mathcal{F}_0^{UF}$

	save_bag_id	$\mathcal{A}_1$	$\mathcal{F}_0^{UF}$		% save D-substitution for first argument
	save_bag_id	$\mathcal{A}_2$	$\mathcal{F}_0^{UF}$		% save D-substitution for second argument

% Handling simple facts for procedure *p/2*

	compliment_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_1^T$		% store all complex clauses in F-bag $\mathcal{F}_1^T$
	d_intersect_F-bag	$\mathcal{F}_1^T$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UC}$	% identify F-bag of unifiable complex clauses
	select_next_simple_fact	$\mathcal{F}_0^{UF}$,	$Label_{10}$		% select another unifiable simple fact if present else jump to process complex clause

% Complex fact processing for procedure *p/2*

$Label_{10}$:	*select_next_clause*	$\mathcal{F}_0^{UC}$			% select one unprocessed unifiable complex clause
CF_p/2:	*unify*	$\mathcal{A}_1$,	$\mathcal{A}_2$		% unify aliased variable X in *p/2*
	return				% return from *p/2*

% Instructions for procedure *q/2*. Instructions have same meaning as in *p/2*

q/2:	*data_parallel_arg_match*	$\mathcal{A}_1$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	data_parallel_arg_match	$\mathcal{A}_2$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	d_intersect_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UF}$
	save_bag_id	$\mathcal{A}_1$,	$\mathcal{F}_0^{UF}$	
	save_bag_id	$\mathcal{A}_2$,	$\mathcal{F}_0^{UF}$	
	compliment_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_1^T$	
	d_intersect_F-bag	$\mathcal{F}_1^T$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UC}$
	select_next_simple_fact	$\mathcal{F}_0^{UF}$,	$Label_{23}$	
$Label_{23}$:	*select_next_clause*	$\mathcal{F}_0^{UC}$		

% Instructions for procedure *r/2*. Data parallel goal reduction has same meaning as *p/2*.

r/2:	*data_parallel_arg_match*	$\mathcal{A}_1$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	data_parallel_arg_match	$\mathcal{A}_2$,	$\mathcal{F}_0^U$	
	*backtrack_if_*$\mathcal{F}^0$	$\mathcal{F}_0^U$		
	d_intersect_F-bag	$\mathcal{F}_0^F$,	$\mathcal{F}_0^U$,	$\mathcal{F}_0^{UF}$
	save_bag_id	$\mathcal{A}_1$,	$\mathcal{F}_0^{UF}$	
	save_bag_id	$\mathcal{A}_2$,	$\mathcal{F}_0^{UF}$	

	compliment_F-bag	$\mathcal{F}_0^F$, $\mathcal{F}_1^T$	
	d_intersect_F-bag	$\mathcal{F}_1^T$, $\mathcal{F}_0^U$,	$\mathcal{F}_0^{UC}$
	select_next_simple_fact	$\mathcal{F}_0^{UF}$, $Label_{34}$	
$Label_{34}$:	*select_next_clause*	$\mathcal{F}_0^{UC}$	

% Instructions to Handle Complex Rule

CC_R1:	*copy_reference*	$\mathcal{A}_1$, $\mathcal{A}_1$	% copy X in *r/2* to X in subgoal *p/2*
	copy_reference	$\mathcal{A}_2$, $\mathcal{A}_2$	% copy Y in *r/2* to Y in subgoal *p/2*
	call	*DP_p/2*	% call procedure *p/2*
$Label_{39}$:	*load_data*	$\mathcal{A}_1$, *2*	% load *2* in $\mathcal{A}_1$
	copy_reference	$\mathcal{A}_2$, $\mathcal{A}_2$	% copy Y in *r/2* in Y in subgoal *q/2*
	*repeat_until_*$\mathcal{F}^0$	$Label_{42}$	% load $Label_{42}$ on stack and go to next instruction if binding of $\mathcal{A}_2$ is non-empty else backtrack
$Label_{42}$:	*load_next_binding*	$\mathcal{A}_2$	
	call	*DP_q/2*	% call procedure *q/2*
	return		% return to top level

6.2 Performance Evaluation

We developed and implemented the associative computation model and its generic abstract instruction set for a homogeneous address space. 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 10 ns clock supported by current technology, and three clock cycles (load-execute-store cycle), the associative look up speed is 1.2 million $\times$ 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 one hundred and twenty MCPS (million computations per second) for thousand facts.

When two subgoals of a rule share variables, and the second subgoal is not a data parallel computation, 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 four 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.

7 Conclusion

In this paper, we describe a formal model of associative logic programming for heterogeneous address space and a generic abstract instruction set. The model and the abstract instruction set will act as a framework which can be altered to exploit the advantages of specific architecture. The advantage of data parallel goal reduction is that data can be distributed on different machine and each machine may execute same piece of code on different data sets. The use of bit-vectors reduces the data transfer overhead when data is distributed across different machines. The simulation results for homogeneous address space is very encouraging. We are currently modifying the abstract machine for heterogeneous address space using popular message passing paradigm tools [15].

References

[1] Bansal, A., and Potter, J., "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.

[2] Bansal, A., "An Associative Model to Integrate Knowledge Retrieval and Data Parallel Computation", *International Journal on Artificial Intelligence Tools*, Volume 3, Number 1, 1994, pp. 97 - 125.

[3] Bansal, A., "Towards a Formal Computational Model for Associative Logic Programming", *Proceedings of the International Workshop on Data Parallel Implementation of Declarative Languages*, Geneoa, Italy, 1994, Uppsala University, Sweden, pp. 11 - 20.

[4] Bansal, A., Prasad, L., and Ghandikota, M., "A Formal Associative Model of Logic Programming and its Abstract Instruction Set", *International Conference of Tools with Artificial Intelligence*, November 1994, pp. 145 - 151.

[5] Dwork, C., Kanellakis, P., and Mitchell, J., " On the Sequential Nature of Unification", *The Journal of Logic Programming*, (June 1984), pp. 35-50.

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

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

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

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

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

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

[12] Lloyd, J., *Foundations of Logic Programming*, (Springer-verlag, New York, 1984).

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

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

[15] Sunderam, V. S., " PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience, 2*, pp. 315-339 (1990).

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

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