Proceedings of the ISCA
8th International Conference

INTELLIGENT SYSTEMS

Denver, Colorado
June 24-26, 1999

Editors: M. Kantardzic

A Publication of
The International Society for
Computers and Their Applications - ISCA

ISBN: 1-880843-28-5

In the Proceedings of the 8th International Conference on Intelligent Systems, Colarado, USA, June 1999, pp. 1- 6

A Scalable Distributed Associative Multimedia Knowledge Base System for the Internet

Stephen W. Ryan and Arvind K. Bansal

Department of Mathematics and Computer Science

Kent State University, Kent, OH 44242 - 0001, USA

E-mail: sryan@mcs.kent.edu and arvind@mcs.kent.edu

Abstract

This paper describes a system to retrieve multimedia knowledge on a cluster of heterogeneous architectures distributed over the internet. In this system, knowledge is represented using facts and rules in an associative logic-programming model. Associative computing facilitates search by content and exploits data parallel computation. It also provides a flat data model that can be easily mapped onto heterogeneous architectures. The implementation uses a message-passing library for architecture independent communications, uses object oriented programming for modularity and portability, and employs Java to provide a graphical user interface, multimedia capability, and accessibility via the internet.

Keywords: distributed associative heterogeneous Internet knowledge retrieval

1. Introduction

During the design of complex objects such as combustion engines or automobiles engines, the model has to be continuously and incrementally modified to incorporate changes based upon new simulation results [9]. An interactive system to facilitate the design process needs to be intelligent and graphically based to relieve the designer from low level coding and to cut down on design cycle time. A large library of components along with their attributes is needed so that matching components can be retrieved from a multimedia knowledge base in realistic time, facilitating software reuse. A system for designing complex objects therefore depends on three major factors: high performance retrieval from very large knowledge bases to retrieve best matching components, intelligent abstract modeling to interconnect simulations of various components, and high performance scientific computing for precise simulation. Since the architectures of computers are very different and are continuously changing, we need a system that is architecture independent, scalable, and portable.

With the recent advances in fast internet connectivity, it has become possible to share heterogeneous resources such as multimedia knowledge bases, applications, and computing power across arbitrary distances and among a large number of users. Ideally, one would like to be able to access a desired resource transparently, regardless of where on the network or on what type of computer system it resides. By doing so, the entire internet can be treated as a large virtual computer: distributed resources can be accessed simultaneously over the internet in a realistic time.

In this paper, we describe a system which integrates the logic-programming paradigm [12], heterogeneous computing [8, 19], the associative computing paradigm [16, 17], the object-oriented paradigm, and Java [6] to provide a multimedia knowledge retrieval system for heterogeneous clusters of computer systems distributed over the internet.

In this paper, we extend our work on a distributed associative knowledge based system [18] by providing an internet interface, multimedia capability, and remote loading of knowledge domains. The current system provides a web based multimedia tool for efficient multimedia knowledge retrieval on the internet, and can be invoked using a web browser from anywhere on the internet.

The model distributes knowledge on multiple servers either to utilize different knowledge domains or to exploit data parallelism on a distributed knowledge domain. A coordinator is used to coordinate and collect data from multiple servers. Major processing is done either within the servers or at the client site to reduce the overhead of data transfer. The model is suited for internet based cluster computation involving multimedia knowledge retrieval from distributed sources, the use of internet as a large virtual multimedia computer, and the integration of intelligent reasoning and scientific computing.

In the knowledge base, components and their attributes are represented as logical facts and rules. Logic programming [12] provides a declarative programming model and intelligent reasoning capability. A message passing library [8, 19] provides architecture independence within a local cluster of knowledge servers. The object oriented implementation provides modularity and portability. Associative computing [5, 16, 17] facilitates search by content of a large knowledge base of components and data parallel computing on a cluster of knowledge servers in a distributed knowledge domain. The associative data representation [5, 16] provides a uniform mechanism to represent complex data structures across heterogeneous architectures, and reduces the

overhead of linearization of complex data structures. The scheme is illustrated in Figure 1.

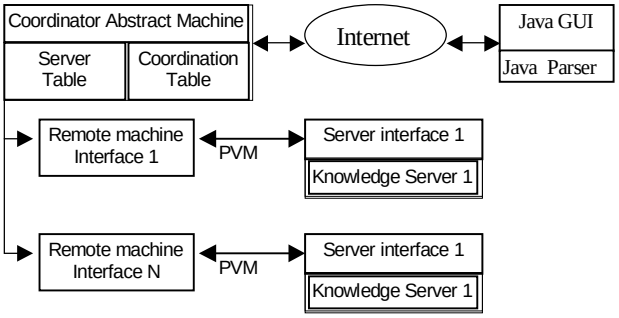

Figure 1: A model for multimedia distributed knowledge base retrieval on the internet

The main contributions of this paper are as follows:

1. A multimedia knowledge base can be stored on and retrieved from clusters of high performance processors in a distributed fashion. A needed multimedia knowledge domain can be loaded from the server to the client's workstation to reduce the overhead on the server.
2. To the best of our knowledge, this is the first attempt to intelligently simulate a complex object using a virtual intelligent computer on the internet.
3. The use of the message passing paradigm and the associative model of computing makes distributed knowledge retrieval transparent to the user.
4. The coordinator based knowledge engine distributes computation on a cluster of servers connected through a message passing library to reduce the data transfer overhead.

Although, our application is for the intelligent design of complex objects, the project can also be used as an intelligent broker to collect information from multiple commercial search engines and perform advanced queries on the resulting data to prune undesired results.

The paper is organized as follows: Section 2 describes briefly the heterogeneous associative model of logic programming. Section 3 describes a distributed version of the model. Section 4 describes the Java interface that provides multimedia capability, and explains the multimedia capability through a simple example. Section 5 describes the object-oriented implementation of the distributed model and an object oriented implementation of the Java based graphical user interface. Section 6 describes the related works, and Section 7 concludes the paper.

2. Background - The Heterogeneous Associative Model

In this section, we describe the heterogeneous associative logic programming model for knowledge base systems. This model [1, 2, 3] exploits associative search to match the clause heads with a query in a data parallel manner, with execution of compiled clause bodies. In this model, data is represented as an indexed association of fields to facilitate search by content. The presence of associative memories in hardware, although not necessary, automatically improves the efficiency of the model. Our model is based upon the efficient software implementation of associative operations, and indexing based upon compile time storage of vectors having same value in a specific field. Low level implementation details, however, are outside the scope of this paper.

In the implementation of the associative logic programming model, the left hand side of a logic program is represented as a two dimensional associative table with parallel fields for the names and arguments in the clause heads. The right hand side of the program is compiled into low level abstract instructions.

A data parallel binding environment is generated during unification of a goal with the corresponding clause heads or during the execution of built-in predicates. It consists of a sequence of frames, each containing a set of associative Boolean vectors to mark unifiable facts. The model also uses a set of global registers for holding the bindings (or pointer to binding vectors) of arguments in the current goal, an associative control stack to store states of computations for previous procedure invocations, and an associative table to handle aliasing of logical variables. The global registers are analogous to those in the Warren Abstract Machine [21]. The control stack is an association of time stamps with previous states of execution. Each state is represented as an associative frame, and uses associative vectors to facilitate fast backtracking. Variable aliases are indicated by Boolean vectors and are tracked by an alias management table. The alias management table is an associative table using bit vectors and logical ORing of bit vectors to derive the union of two sets of aliased variables in case members of two sets are aliased by an instruction. A detailed explanation of this model and the corresponding abstract instructions is given in [2, 3].

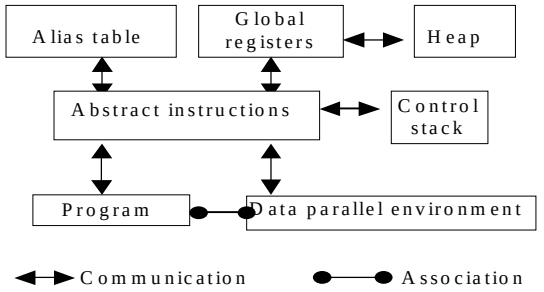

Figure 2: The heterogeneous associative model

3. Distributed Heterogeneous Knowledge Base System

The distributed system consists of two types of abstract machine: the *associative abstract machine* described

above and a *coordinator* abstract machine (see Figure 1). The coordinator launches server processes on a local or remote host. Each server contains an associative abstract machine as illustrated in Figure 2, with the ability to receive goals and send solutions to the coordinator using a message-passing interface such as PVM [19] or MPI [8].

3.1 Abstract Data Structures

The coordinating abstract machine needs three data structures that supplement those in the base abstract machine. They are an associative server table, a coordination table, and an associative binding area.

The *associative server table* stores information about the predicates handled by the servers. For each server/procedure pair there is an entry of the form (*server-id, procedure-id, clause-count)* in the server table. The *server-id* uniquely identifies a server. The *procedure-id* is a reference to an entry in the procedure table. The *clause-count* is the number of clauses the server has that match the goal. The *coordination stack* is a sequence of *coordination-vectors*, Boolean vectors marking the executable procedures that have not returned the bindings for the current goal, associated with the server table. The association with the server table facilitates the marking of servers that have yet not returned their bindings. Initially the Boolean vector is the same as the server capability vector. Upon receiving a *failure* signal from a server, the corresponding bit in the coordination vector is reset. The coordinator backtracks to a previous coordination vector when the current coordinator vector becomes empty. The *associative binding area* stores the bindings incrementally as they are received from the servers. The data elements in the associative binding area are (*server-id, time-stamp, variable-id, value, type)* for each bound argument. A Boolean vector associated to this table is used to identify the vectors bound to a register at a given time stamp.

3.2 Distributed Abstract Instructions

There are four abstract instructions in the coordinating abstract machine that facilitate distributed processing. They are: *get_servers*, *broadcast_goal*, *receive_binding*, and *repeat_else_try*

The *get-servers* instruction takes a procedure-id as an argument and returns a coordination vector that identifies the servers that can solve the goal and which have not yet sent all the bindings for the goal. The *broadcast_goal* instruction sends a goal to the servers indicated by a server filter vector. The *receive_binding* instruction is executed repeatedly to retrieve the bindings from the servers. The arguments for a *receive-bindings* instruction are a coordinating vector and a binding vector. The bindings from each server in the server table are added to the associative binding area. The binding filter vector points to the new binding vectors. If all the matching servers transmit *failure*, the coordinator backtracks. The abstract instruction *repeat_else_try* enables the repeated execution of the *receive_bindings* instruction with the capability to backtrack in the absence of a binding. A detailed explanation of the abstract instruction set is given in [18].

3.3 The Execution Model

A schema file prepared by the user specifies a list of remote hosts and the file names of the knowledge bases to be loaded onto the various hosts. The coordinator reads the schema file and initiates the server processes. After each server process has been successfully initialized and has loaded its data, it reports back to the coordinator with a list of the available procedures. The coordinator builds up the server table from these reports.

To solve a goal, the coordinator performs an associative search on a server table to obtain a filter vector identifying all servers that have the corresponding procedure. The result of this search initializes the corresponding coordination vector. The coordinator broadcasts a goal to the servers using an abstract instruction *broadcast_goal*.

Upon receiving a goal, each server first concurrently searches its facts and then its rules for a solution. After finding a set of solutions, the servers wait for further instructions from the coordinator. After broadcasting a goal, the coordinator queries each matching server for solutions. The coordinator polls one server at a time. Each server, when prompted, transmits the bindings for the goal arguments to the coordinator. Anticipating further requests, the servers backtrack to find additional solutions. Meanwhile, the coordinator stores the received bindings in its associative binding area along with the current time-stamp and the server-id.

After collecting the first set of solutions from the servers, the coordinator reports these solutions to the user. Upon further request from the user, the coordinator requests a new set of solutions from the matching servers. Additional bindings received from the servers are added to the associative binding area. A server sends *a failure* in the absence of additional solutions. After receiving a *failure*, the coordinator removes that server from the list of matching servers by resetting the corresponding bit in the coordination vector. This process is repeated until the list of servers is empty. This condition corresponds to an empty coordination-vector. When the coordination vector becomes empty, the coordinating process backtracks and tries other rules to solve the goal.

3.4 Serving Multiple Subgoals

After receiving a request to solve a subgoal, a server generates initial solutions. Upon request, the server sends the resulting bindings to the coordinator, and proceeds to generate alternate solutions. However, the coordinator's next request may be to solve the next subgoal instead of requesting the alternate solutions. The server then saves its state, including the alternate solutions to the first subgoal, and proceeds to solve the second subgoal. If the coordinator backtracks, it first requests more solutions to

the second subgoal. After reporting *failure* to the coordinator, the server reverts back to its previous state with the alternate solutions to the first subgoal. The coordinator receives the *failure* message and removes this server from the list of servers for the second subgoal. If the list of servers becomes empty for the second subgoal, the coordinator backtracks and requests alternate solutions to the first subgoal.

4. Web Based Multimedia Interface

In this section, we describe the implementation of the Java graphical user interface and front end to the Distributed Associative Logic Programming System. Included in this front end are the following:

1. a lexical analyzer and a parser for the input of logical goals and assertions,
2. a compiler to generate instruction code and data for the Associative Logic Program engine,
3. communication routines for configuring and communicating with the distributed logic engine,
4. a graphical user interface. This interface provides the ability to retrieve multimedia knowledge bases (including image and sound) easily from the internet via URLs.

The Java interface is shown in Figure 3. It consists of a text window for viewing the text of a logic program and a text entry field for entering new facts, rules and goals. From the File menu, existing programs are opened either from a local file or from the internet by specifying a URL. A compiled version of the program can also be saved to the local disk by selecting the Compile menu. The program is automatically compiled when launching a server. The first option on the Servers menu will spawn a server process on any host in the currently configured cluster, compile the current program, and load it into the new server.

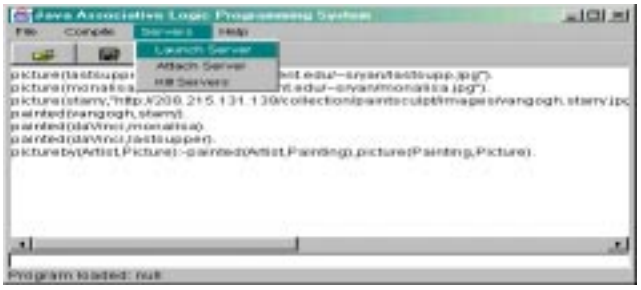

Figure 3: Launching a knowledge server

We demonstrate the use of the system through a small knowledge base about famous artists and their paintings. The knowledge base contains two types of facts and one rule as follows:

painted(Artist, Paintings).
picture(Painting, URL).
pictureby(Artist,Picture) :-
painted(Artist,Painting), picture(Painting,Picture).

The rule will match the artists in the multimedia knowledge base with images of their work on the internet. There may be several sites on the internet which store paintings of the great artists. With this system, it would be possible to access this distributed multimedia knowledge, create a server for each individual knowledge base on a local high performance computer cluster and query the knowledge transparently as if it were a single local knowledge base.

5. Object Oriented Implementation

This section describes an object model needed to provide modularity in the distributed execution of the heterogeneous associative logic program system and the object model for the Java based graphical user interface.

5.1 Class Hierarchy

There are two primary classes in the object model: the *abstract-machine* class and the *program* class. The *abstract-machine* class represents an abstract machine, and the *program* class encapsulates the associative representation of a logic program. Figure 4 illustrates the overall class structure.

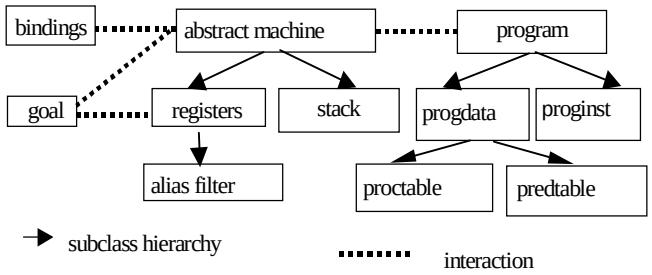

Figure 4: Object-oriented data representation

The public interface of the *abstract-machine* class allows for loading a program, solving a goal, requesting alternate solutions, and retrieving binding information for the goal arguments. The private member functions of the *abstract-machine* class include functions for executing the instruction code of the program and for backtracking and controlling the flow of the program. The two major subclasses of the *abstract-machine* class encapsulate the *registers* and the *control stack*.

The *program* class has two subclasses: *progdata* and *proginst*. The subclass *progdata* represents an associative table of clause-heads. The subclass *proginst* represents the compiled code for the clause bodies. The subclass *progdata* has two subclasses: *predtable* and *proctable*. The subclass *predtable* maps predicate names to a numeric predicate-id. The subclass *proctable* is used for fast lookup of the predicate and the entry point in the compiled code for each procedure. All of the components of the *program* class are public and manipulated directly by the abstract machine

The associative data types are encapsulated in the two classes *associative-filter* and *associative-vector*. The *associative-filter* class supports logical operations. The *associative-vector* class is used to represent associative data vectors. It is implemented using the C++ template facility to support arbitrary data types.

5.2 Object Oriented Implementation of Java Interface

The structure of the Java application is shown in Figure 5. The user interface is integrated with a parser (the ALParser class) and an array of RemoteAbsMachine objects that provide the interface for communicating with server processes, encapsulated by the ALPServer class.

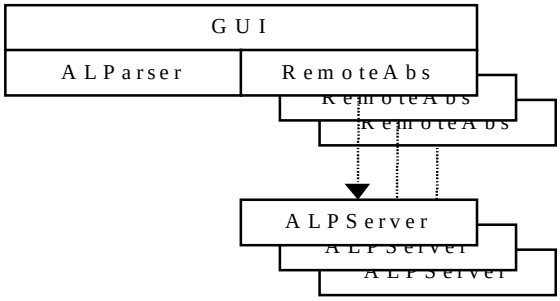

Figure 5: Structure of the Java application

The ALParser class (see Figure 6) accepts logical statements and builds an abstract representation. Within the parser, there is a lexical analyzer (the LexAn class), a symbol table (the SymbolTable class), and a representation of the logic program (the ALProgram class).

The ALProgram class contains two representations of the program. The first is an array of procedures, each of which is an array of clauses. This is the representation that is built via the parser and corresponds directly to the text of the program. The second representation of the program is the tabular data and instruction code that is used by the associative logic programming engine.

A logic program is passed through the lexical analyzer and parsed. This populates the symbol table and the initial representation of the program in the Program class. At compile time, the associative representation of the program is created within the ALProgram object and then written to disk or passed to the server via the RemoteAbsMachine class.

The RemoteAbsMachine class masks the remote nature of the ALPServer. Requests made to a RemoteAbsMachine are actually packed into messages and sent to the server process via the message passing system. Each RemoteAbsMachine object can represent a single server or a coordinator managing multiple distributed servers.

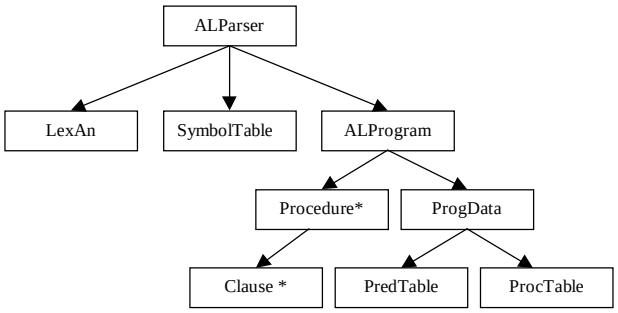

Figure 6: A Java based parser

After all servers have been launched, goals may be submitted via the text entry line of the user interface. If multiple servers have been launched, then the goal will be submitted to all of those servers. Textual results of the query will be displayed in the text window. Multimedia results, such as images or sounds, will be displayed accordingly on the screen.

6. Related Works

We are not aware of any other distributed associative models for retrieving multimedia knowledge from heterogeneous clusters. There are logic programming models [4, 13, 15, 20] for internet programming that have been developed. JINNI [20] is a web-oriented logic programming system that utilizes Java. It relies on the native BinProlog engine when high performance is required. W-ACE [15] is a constraint-based logic programming system that is capable of web based logic programming. W-ACE supports active servers (servers aware of client activity) and active clients (clients working on a program loaded from the server).

BinProlog and ACE are some of the fastest WAM based logic programming implementations [21]. However, they do not support content-based knowledge retrieval. Our model supports content-based knowledge retrieval, cluster computing, and is scalable due to the use of associative data structures. For local concurrency among servers W-ace exploits fine grain AND-OR parallelism suitable for tightly coupled servers. Fine grain parallelism may have high overhead of data transfer on a loosely coupled network. We believe that all these models of internet based intelligent knowledge retrieval and web based computing will provide insight to this new, uncharted, and fast growing field of virtual intelligent resource sharing among computers on the internet.

Our model has evolved due to the need to model a complex simulation on a cluster of high performance architectures in a realistic time. It is efficient on loosely coupled cluster of servers since coordinator based parallelism exploits both data parallelism and coarse grain parallelism. The use of the message-passing libraries in our model provides a natural capability to interface with distributed simulation software developed in other languages.

7. Conclusions and Future Work

In this paper, we have discussed the generic architecture of an abstract machine for the distributed execution of logic programs on a heterogeneous collection of computers. The object-oriented implementation is portable, flexible and extensible. The use of message-passing libraries provides scalability and architecture independence. The performance results show that distributing the knowledge domain on multiple local servers connected through a high-speed intranet provides significant performance improvement. Similarly, porting individual knowledge domains to the client side over the internet also reduces data transfer and computation overhead.

Currently, in collaboration with Javed Khan [11], we are looking into the development of an internet based multimedia intelligent language which supports content based associative image retrieval [11, 14]. We are also developing interfaces to CORBA and scientific languages such as Fortran to interoperate with scientific computing simulations in the field of high performance engine design.

Acknowledgments

This research was supported in part by NASA Lewis Research Center through a NASA Grant. The authors also acknowledge Greg Follen for useful discussions and continued support of this project. The authors also acknowledge Javed Khan for useful discussions on other applications of this work.

References

1. 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, Volume 5, Number 3*, (1992) pp. 247-262
2. A. K. Bansal, "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. A. K. Bansal, "A Framework of Heterogeneous Associative Logic Programming," *International Journal of Artificial Intelligence Tools, Vol. 4, Nos. 1 & 2*, (1995), pp. 33 - 53.
4. P. Bonnet, S. Bressan, L. Leth, and B. Thomsen. Towards ECLiPSe Agents on the INTERNET, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, (1996).
5. J. A. Feldman, and D. Rovner, "An Algol Based Associative Language," *Communications of the ACM, Volume 12, Number 8*, (1969) pp. 439 - 449.
6. J. Gosling, B. Joy, and G. Steele, "The Java Language Specification," Addision-Wesley, also see http://www.javasoft.com
7. D. Gries, *The Science of Programming*, Monograph, Springer Verlag, New York, 1987.
8. W. Gropp, E. W. Lusk, and A. Skjellum, "Using MPI: Portable Parallel Programming with Message Passing Interface," *MIT Press*, 1994
9. P. T. Homer and B. Schlichting, "Using Schooner to support distribution and heterogeneity in the Numerical Propulsion System Simulation Project," *Concurrency Practice and Experience, Vol. 6(4)*, 1994, pp. 271-287
10. K. Hwang, and F. A. Briggs, *Computer Architecture and Parallel Processing*, Mcgraw Hill Book Company, New york, USA, (1984).
11. J. Khan, "Intermediate Annotationless Dynamical Object-Index-Based Query in Large Image Archives with Holographic Representation," Journal of Visual Communication and Image Representation, Vol. 7, No 4. 1996, pp. 378 – 394
12. R. Kowalski, *Logic for Problem Solving*, Elsevier-North Holland, (1979).
13. S. W. Loke, and A. Davison, Logic Programming with the World-Wide Web. *Proceedings of the 7th ACM Conference on Hypertext*, pages 235 - 245. ACM Press, (1996).
14. V. Ogle and M. StoneBraker, "Chabot: Retrieval from a Relational Database of Images," IEEE Computer 29, 1995, pp. 18-22.
15. E. Pontelli and G. Gupta, "W-ACE: A Logic Language for Intelligent Internet Programming," *Proceedings of the Ninth International Conference on Tools with Artificial Intelligence*, Newport Beach, CA, USA, 1997, pp. 2 -10.
16. J. L. Potter, *Associative Computing*, Plenum Publishers, New York, (1992).
17. J. Potter, J. Baker, A. K. Bansal, S. Scott, C. Ashtagiri, "Associative Model of Computing," *IEEE Computer*, November 1994, 19 - 25
18. S. Ryan and A. K. Bansal, "A Scalable Heterogeneous Associative Logic Programming System," Proceedings of the Ninth International Conference on Tools with Artificial Intelligence, Newport Beach, California,. November 1997, pp. 37 – 44.
19. V. S. Sunderam et. al., "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience, No. 2*, (1990), pp. 315 - 339.
20. P. Tarau, "Jinni: a Lightweight Java-based Logic Engine for Internet Programming," *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, (1998)
21. D. H. D. Warren, "An Abstract Prolog Instruction Set," *Technical Report 309*, SRI International, (1983).