International Journal on

ARTIFICIAL INTELLIGENCE TOOLS

Architectures, Languages, Algorithms

Volume 9 • Number 3 • September 2000

World Scientific
Singapore • New Jersey • London • Hong Kong • Bangalore

International Journal on Artificial Intelligence Tools, Vol. 9, No. 3 (2000) 343-367

A Scalable Distributed Multimedia Knowledge Retrieval System on a Cluster of Heterogeneous High Performance Architectures

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

Received May 8, 1999
Accepted August 29, 1999

This paper describes a system to distribute and retrieve multimedia knowledge on a cluster of heterogeneous high performance architectures distributed over the Internet. The knowledge is represented using facts and rules in an associative logic-programming model. Associative computation facilitates distribution of facts and rules, and exploits coarse grain data parallel computation. Associative logic programming uses a flat data model that can be easily mapped onto heterogeneous architectures. The paper describes an abstract instruction set for the distributed version of the associative logic programming and the corresponding implementation. The implementation uses a message-passing library for architecture independence within a cluster, uses object oriented programming for modularity and portability, and uses Java as a front-end interface to provide a graphical user interface and multimedia capability and remote access via the Internet. The performance results on a cluster of IBM RS 6000 workstations are presented. The results show that distribution of data improves the performance almost linearly for small number of processors in a cluster.

Keywords: Associative computing, distributed computing, heterogeneous computing, Internet, knowledge retrieval, logic programming, modeling, simulation, symbolic computing.

1. Introduction

Much of the design process is incremental. A complex object consists of many smaller components which are further divided into smaller sub-components. In the current competitive market, the design of complex objects such as engines — automotive combustion engines or aircraft engines — has to be continuously modified to incorporate new research. This requires the design process to be interactive, and the designed object

have to be reconfigured in an iterative manner to facilitate changes based upon the simulation results. Previously designed components have to be modified incrementally to closely match the simulations. The need to continuously change previously designed components requires the management of a large library of the components along with their attributes, so that closely matching components can to be retrieved and modified in an iterative manner based on the simulation results.

Complete scientific simulation of all of the components as a whole is computationally prohibitive. In order to get a simulation of a complex object in realistic time, the different components of the object have to be simulated on different clusters of computers to exploit component level parallelism [14], and the results of the multiple simulations have to be reconciled. Since the architectures of high performance computers are very different and are continuously changing, it is necessary to design a simulation system which is architecture independent, supports high performance clusters of heterogeneous architectures, is scalable, and is portable. An additional requirement consists of modeling the interconnectivity of different components of the object using high-level symbolic reasoning. Symbolic reasoning is also needed for abstract rule based simulation of components where precise scientific computation is cost prohibitive in terms of time and resources.

With the recent advances in fast Internet connectivity, it has become possible to share heterogeneous resources such as multimedia databases, 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: information can be retrieved simultaneously over the Internet from multiple distributed multimedia knowledge bases, and can be processed at the client end in a realistic time.

In essence, the model for the simulation of a complex object requires integration of five major technologies as follows:

(i) High-performance retrieval from knowledge bases has been exploited for retrieving the best matching components.
(ii) Symbolic computing has been exploited to interconnect various component simulations.
(iii) Scientific computing has been interfaced for precise simulation.
(iv) Remote accessibility and visualization of the simulation results and component to facilitate provides user-friendly interactive modification of objects.
(v) The Internet has been used to share resources and to exploit the web as a giant virtual computer.

In this paper, we describe an architecture, an abstract machine, and an implementation of a heterogeneous distributed associative knowledge base model based upon the theory developed in [2, 4]. This model integrates the logic-programming paradigm, heterogeneous computing, associative computing, the object-oriented programming paradigm, and Internet-based programming using Java. Logic programming provides a

declarative programming model. In the knowledge base, components and their attributes are represented either as logical facts or rules. Associative computing integrates data parallel computing with associative search by content. Associative search by content facilitates efficient search of components from a large knowledge base of library of components. Associative data representation — complex data represented as a two-dimensional associative table — reduces the overhead to linearize complex data structures, and provides a uniform mechanism to represent complex data structures across heterogeneous message architectures. The message-passing library provides this architecture independence. The implementation using the object-oriented paradigm provides modularity and portability. The use of Java provides a graphical user interface, remote access through web browsers, and multimedia retrieval capability.

The model is suited for retrieval of knowledge distributed in clusters of high performance computers over the Internet (see Figure 1). A cluster-based computation model consists of multiple clusters, where each cluster consists of high performance architectures capable of symbolic and scientific computing. Each cluster possesses a coordinator and multiple servers. Knowledge is distributed on multiple servers either based upon the different knowledge domains or to exploit coarse grain data parallelism and object level massive parallelism present on the web. Coarse grain data parallelism is present since multiple knowledge bases are present on different clusters, and each cluster concurrently solves a goal. Object level massive parallelism [10] is present since multiple individual components are simulated simultaneously. A coordinator is used to manage and collect data from multiple servers, while major processing is done within the servers to reduce data transfer overhead.

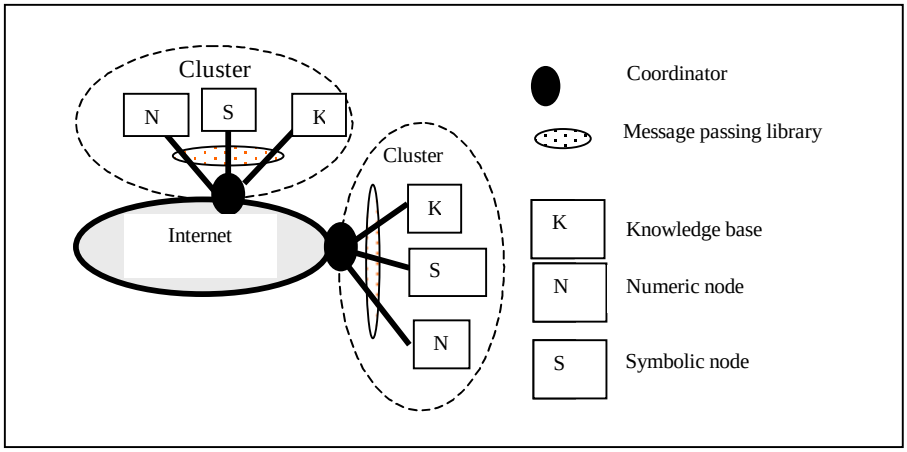

Fig. 1. Architecture of a distributed knowledge base on the Internet

The main contributions of this paper are as follows:

- The model is mapped to a heterogeneous set of architectures in a user transparent manner.

- The model supports modularity, and is scalable to any number of machines.
- The use of a message-passing library and the use of the associative model of computing make the distributed knowledge retrieval transparent to the user. The distributed knowledge base on the web is accessible to a user as a local knowledge base.
- The coordinator-based model delegates maximum computation locally on distributed servers to reduce the data transfer overhead,
- The abstract machine for distributed knowledge retrieval is generic and scalable.
- Java-based multimedia interface provides remote access capability.

The paper is organized as follows: Section 2 describes briefly the background, the definitions of the concepts, and the abstract machine for the implementation of associative logic programming model. Section 3 describes an object oriented implementation of the associative model of logic programming. Section 4 describes heterogeneous associative logic programming model, its distributed version, and the extension of the abstract instruction set for distributed model. Section 5 describes the execution behavior of the distributed model. Section 6 describes the Java interface that provides the multimedia capability. Section 7 describes an object-oriented implementation of the Java based graphical user interface. Section 8 describes the performance evaluation. Section 9 describes the related works, and Section 10 concludes the work.

2. Background and Definitions

In this section, we briefly describe the related concepts of four paradigms: associative computing, heterogeneous computing, logic programming, and object-oriented programming.

Associative Computing searches and selects data elements for processing according to their contents [9, 21, 22]. Data records are distributed among the processors, forming a table of parallel fields or *associated vectors* such that the data elements in associated vectors are accessed using the same index value. Vectors are processed simultaneously by broadcasting a single instruction to all processors. The results of the computation forms a filter vector — a new associated vector of Boolean values. Filter vectors are used to select records for further processing.

Data parallel computing refers to the simultaneous execution of some abstract computation on multiple data elements. Distributed computing refers to the distribution of computation over multiple computer systems and encompasses both the data parallel and process parallel computing models. In the absence of data dependency, distributing a computation can result in a near linear speed up.

The object-oriented programming paradigm is well suited for distributed computing using message passing between objects. The sub-components of a computation are

isolated and encapsulated in 'objects'. An object acts as a client by making a request to another object for some data or for some function to be performed. The receiving object acts as a server by replying to the client with the requested information.

A message passing library [13, 26] uses message passing functions to communicate and exchange data on a heterogeneous network. The use of a message-passing library makes the data transfer and network communication transparent to the user.

Logic programming [17] is a popular declarative programming paradigm suitable for high-level reasoning and knowledge representation. In a logic program, knowledge is represented by a set of facts and rules used to describe relationships between data objects.

Java [11] is an Internet programming language derived from C++ with additional support for multimedia display and portability over the Internet [5, 8, 10]. Java has also been interfaced with web browsers, and is being used extensively over the Internet for multimedia presentation. A major advantage of Java is its independence from heterogeneous architectures over the Internet. Java has also been interfaced with message-passing libraries to serve as a platform for cluster based multimedia display [28].

2.1. The heterogeneous associative model

The heterogeneous associative logic programming model [4] exploits associative search to match the clause heads with a query in a data parallel manner, and relies on execution of compiled code for the clause bodies. In this model, data is represented by an explicit association of fields (including index) to facilitate search by content. The presence of associative memories in hardware facilitates search by content and automatically improves the efficiency of the model. A software implementation of associative operations insures the data lookup efficiency for a large number of facts on sequential architectures as well.

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 instruction code.

In the compiled code for the right-hand side of the program, a data parallel binding environment is generated and modified during unification of a goal with the corresponding clause-heads or during the execution of built-in predicates. The data parallel binding environment consists of a sequence of multiple frames, each containing associative Boolean vectors to mark unifiable clauses and associative binding vectors to mark the bindings of the variables in matching facts or clauses. The model also uses a set of global registers for holding the bindings (or pointers to bindings) 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 variables. The global registers are analogous to those in the Warren Abstract Machine [29]. The control stack is an association of time stamps and all previous states of execution. Each state is

represented as an associative frame, and uses associative vectors to facilitate fast backtracking. The vectors are indexed with respect to the base of the current frame. Variable aliases are indicated by filter vectors and are tracked by the alias management table. The logical OR of two vectors creates the union of two sets of aliased variables when members of the two sets are aliased by an instruction. A detailed explanation of this model and the corresponding abstract instructions is given in [4].

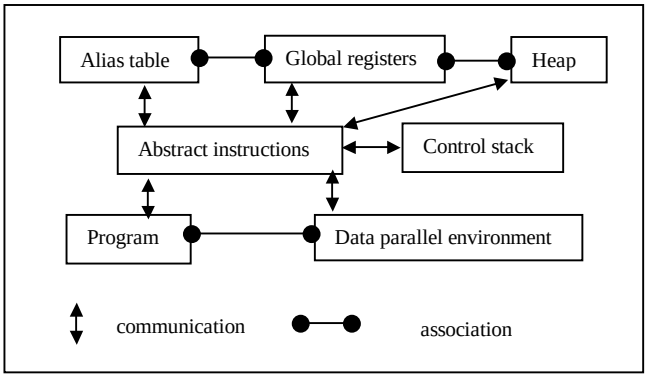

Fig. 2. A heterogeneous associative model.

A shared variable has two occurrences in a clause body: a producer occurrence and a consumer occurrence. The producer occurrence generates a value for the shared variable, and the consumer occurrence uses the value of a variable.

2.2. Notations for the abstract instructions

In this section, we define the notations used in the examples. We will show examples of the abstract instructions code for both the abstract machine of the server processes and the distributed abstract machine used by the coordinator.

A symbol <P/arity> denotes the start of a procedure <P> with an arity of <arity>. A symbol <P/arity>_C_N denotes the start of the Nth clause of the predicate <P/arity>. A symbol <U_N> denotes a universal filter vector — a vector of Boolean values associated with a set of clauses to identify unifiable clauses. A symbol <B_N> denotes a binding filter vector — a vector of Boolean values marking the values in the clause-head representation to which a variable in a goal will be bound after unification. A symbol <F_0> denotes a fact vector — a vector of Boolean values marking the facts in a knowledge base. A symbol <C_N> denotes a clause filter vector — a vector of Boolean values marking clauses with non-empty body. A symbol <A_N> denotes the Nth argument of the current goal. A symbol <T_N> denotes a temporary Boolean Filter vector. A symbol <R_N> indicates a global register, and a symbol <V_N> denotes a variable. A symbol <S_N> indicates a Boolean filter vector marking the servers which contain a particular predicate, and a symbol <L_N> denotes a label. Example 1 illustrates a simple logic program. The corresponding compiled abstract instruction code is given in Figure

3. Note that the vectors and variables have been indexed with respect the start of the current frame.

Example 1

p(1, 2). *p(2, 3).* *p(3, 4).*
p(X, Y) :- *q(Y, X).*

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

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

We describe the general behavior of the instructions as follows. There is one entry point in the code for each procedure in the logic program. First the goal procedure and arguments are matched with the clause heads in a data parallel manner. The resulting filter-vector U_0 is ANDed with the filter-vector F_0 which annotates the ground facts. The resulting binding filter vector B_0 indicates the bindings of the goal arguments with the corresponding values in the facts. A clause filter vector C0 annotates the clause heads. If facts were found, then control is returned. In the absence of matching facts or upon a request for more solutions, the compiled code for the rules is executed using the clause-filter vector C_0.

The compiled code for rules copies the goal arguments in global registers [3, 4] and calls the sub-goals. The three *load_procedure* instructions are used to load the three procedures in the program representation.

```
p/2:      match_arg              A0, U0        /* match the 1st argument*/
          backtrack_if_empty     U0            /* backtrack if no matching positions */
          match_arg              A1, U0        /* matching the 2nd argument*/
          backtrack_if_empty     U0            /* backtrack if no matching positions */
          and_bit_vectors        F0,U0, B0     /* mark the binding values in facts */
          store_vector_id        B0, A0        /* store the binding of the 1st argument*/
          store_vector_id        B0, A1        /* store the binding of the 2nd argument */
          comp_bit_vector        F0, T1        /* complement F0 into T1 */
          and_bit_vectors        T1, U0, C0    /* mark unifiable rules */
          return_if_empty        B0, L10       /* return if binding vector B0 is empty */
L10:      try_me_else            C0            /* next unifable clause until C0 is empty*/
p/2_cl:   copy_reference         A1, A0, q/2   /* build the 1st argument to call q/2 */
          copy_reference         A0, A1, q/2   /*build the 2nd argument to call q/2 */
          call                   q/2           /* branch to subroutine with label p/2*/
          return                               /* return from the procedure p/2*/
q/2:      match_arg              A0, U0        /* match the 1st argument*/
          backtrack_if_empty     U0            /* backtrack if no matching positions */
          match_arg              A1, U0        /* match the 2nd argument*/
          backtrack_if_empty     U0            /* backtrack if no matching positions */
          and_bit_vectors        F0, U0, B0    /* mark the binding values in facts */
          store_vector_id        B0, A0        /* store the bindings of 1st argument*/
          store_vector_id        B0, A1        /* store the binding of the 2nd argument */
          comp_bit_vector        F0, T1        /* complement F0 into T1 */
          and_bit_vectors        T1, U0, C0    /* mark unifiable rules */
```

```
        return_if_empty     B0, L25      /* return if binding vector B0 is empty */
L25:    try_me_else         C0           /* next unifable clause else backtrack */
r/2:    match_arg           A0, U0       /* match with the 1st argument */
        backtrack_if_empty  U0           /* backtrack no matching positions is */
        match_arg           A1, U0       /* match  the 2nd argument */
        backtrack_if_empty  U0           /* backtrack if no matching positions */
        and_bit_vectors     F0, U0, B0   /* mark the binding values in facts */
        store_vector_id     B0, A0       /* store the binding of 1st argument */
        store_vector_id     B0, A1       /* store the binding of 2nd argument */
        compl_bit_vector    F0, T1       /* complement F0 into T1 */
        and_bit_vectors     T1, U0, C0   /* mark the binding values in facts */
        return_if_empty     B0, L36      /* return if binding vector B0 is empty */
L36:    try_me_else         C0           /* next unifable rule else backtrack */
r/2_cl1: copy_reference     A0, A0, p/2  /* build the 1st argument to call p/2*/
        load_new_variable   A1, V3, p/2  /* build the 2nd argument to call p/2 */
        call                p/2          /* call the set of instructions at label p/2 */
L40:    continue            L40          /* save L40 and execute next instruction */
L41:    copy_reference      A0, V3       /* build 1st argument to call q/2 */
        copy_reference      A1, A1, q/2  /* build 2nd argument to call q/2 */
        repeat_else_backtrack L44        /* next value of shared variable */
L44:    load_next_vector_value A0        /* load the next value for variable Y */
        call                q/2          /* call the set of instructions at label q/2 */
        return                           /* return from this routine */
Load:   load_procedure      p/2          /* load the procedure p/2 into a table */
        load_procedure      q/2          /* load the procedure q/2 into a table */
        load_procedure      r/2          /* load the procedure r/2 into a table */
```

Fig. 3. The compiled code for Example 1

3. Object Oriented Implementation of Associative Logic Programming Model

This section describes an object model needed to provide modularity in the distributed execution of the heterogeneous associative logic program system.

3.1. 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.

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, including the implementation of the abstract instruction set, and functions 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 represents an associative filter vector. It supports logical operations and assignment. The *associative-vector* class is used to represent associative data vectors. It is implemented using the C++ template facility to support arbitrary data types. Functions are provided to manipulate the associative vector using associative techniques.

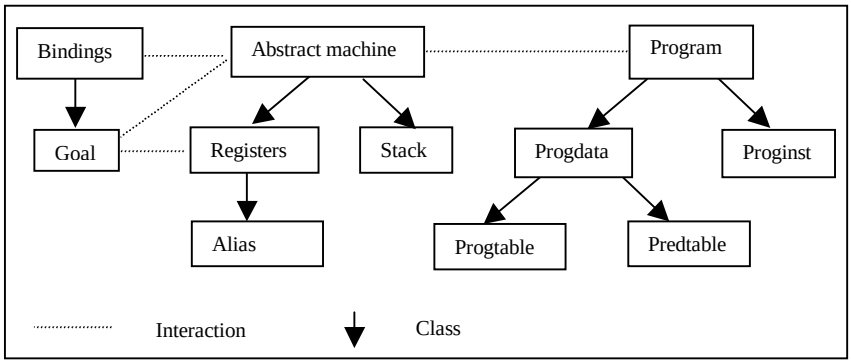

Fig. 4. An object oriented data representation

4. The Distributed Heterogeneous System

The distributed system consists of two types of abstract machines: a *coordinator* abstract machine and a *heterogeneous associative abstract machine* (see Figure 5).

4.1. Abstract data structures

The coordinator launches server processes on a local or remote host. Each server has a heterogeneous associative abstract machine as described in Subsection 2.1, with the additional ability to receive goals and send solutions to the coordinator using a message-passing interface such as PVM or MPI. Three additional data structures — an associative server table, a coordination table, and an associative binding area — are needed in the distributed model.

The *associative server table* stores the information about the predicates processed 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 last entry is the

server-capability vector which identifies the list of servers which can execute a procedure. An illustration for the program in Example 2 is given in Figure 7.

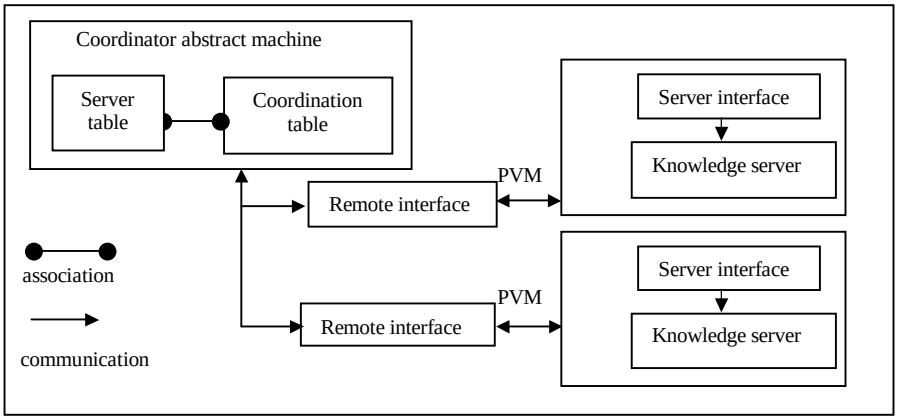

Fig. 5. A distributed heterogeneous associative system

The coordination table is a sequence of *coordination-vectors* — Boolean vectors marking the executable procedures which have not returned the bindings for the current goal. The coordination table is associated with the server table to facilitate the association of Boolean vectors with the procedures. Initially the Boolean vector is the same as the server-capability vector. Upon receiving a failure from a server, the corresponding bit in the coordinator vector is reset. The coordinator backtracks to previous the 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 of the form (*server-id, time-stamp, variable-id, value, type).* A filter vector associated with this table identifies the vectors bound to a register at a given time-stamp.

4.2. The execution model

A schema file prepared by the user specifies a list of remote hosts and the file names of the logic programs 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 program, it reports back to the coordinator with a list of the executable procedures. The coordinator builds up the server table from these reports.

To solve a goal, the coordinator performs an associative search on the server table to obtain a filter vector marking all servers with the corresponding predicate. The result of this search initializes the corresponding coordination vector. The goal is broadcast to these servers. Upon receiving a message, each server first searches its facts and then its rules for a solution. After finding a set of solutions, the server waits for further instructions from the coordinator.

After broadcasting the goal, the coordinator queries each matching server for solutions. The coordinator requests solutions one server at a time. However, this process can easily be made concurrent. 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 the associative binding area along with the current time-stamp and the server-id. The time-stamp is used in backtracking and is incremented before the solution of each subgoal.

After collecting the solutions from one server, the coordinator reports the 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. This process is repeated until the list of servers is empty. When this occurs, the coordinating process backtracks and tries other rules. The other rules in the coordinator also occur in servers, and servers have already sent the values from their corresponding domains. However, in the presence of shared variables, the bindings generated from one server have to be passed to other servers to generate a complete set of bindings. All the servers except a server S returning a binding spawn their procedures again to consume the bindings of the shared variable generated by S. Multiple servers concurrently consume the values of shared variables generated by other servers, and spawn multiple subgoals concurrently. The coordinator manages this process.

4.3. Serving multiple subgoals

At a particular instant, a server might have solutions for more than one subgoal. 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.

4.4. Distributed abstract instructions

There are four new abstract instructions in the coordinating abstract machine [23]: *get_servers*, *broadcast_goal*, *receive_binding*, and *repeat_else_try* that facilitate distributed processing.

The *get-servers* instruction takes a procedure-id as an argument and returns a filter vector that identifies the servers that can solve that 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 server vector and a binding vector. The bindings from each server in the server table are added to an associative binding area. The binding filter vector points to the new binding vectors. If all the matching servers transmit *failure*, this instruction releases the repeat label from the control stack, and 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 bindings. It puts a repeat label and *try_me_else* label on the control stack, which controls the execution of the *receive_bindings* instruction.

5. A Distributed Domain

In this section, we present the execution trace of a simple logic program working on knowledge distributed over two servers.

Example 2

The program in Figure 6 illustrates a distributed version of the program in Example 1. To simplify the compiled code, only the facts were altered. The compiled code consists of two types of codes: server-level code and coordinator-level code.

Server 1		***Server 2***	
p(1, 2).	*p(2, 3).*	*p(1, 4).*	*p(2, 5).*
P(3, 4).		*p(3, 6).*	
p(X,Y) :- q(Y,X)		*p(X,Y) :- q(Y,X).*	
q(2, 1).	*q(3, 2).*	*q(4, 3).*	*q(5, 4).*
q(2, 6).	*q(3, 7).*	*q(4, 8).*	*q(5, 9).*
r(2,2).		*r(2, 3).*	
r(X, Z) :- p(X, Y), q(Y, Z).		*r(X, Z) :- p(X, Y), q(Y, Z).*	

Fig. 6. A simple example of knowledge in two different domains

Figure 7 illustrates the compiled code for the coordinator. The major difference is that the coordinator spawns the subgoals, collects the bindings, interacts with the outside world, and coordinates the values of shared variables if bindings from one server are

passed to another server.To simplify the trace, the programs in both servers have the same structure and differ only in their facts.

```
p/2:      get_servers              p/2, S0        /* get servers for p/2 into S0*/
          backtrack_if_empty       S0             /* backtrack if no more servers */
          broadcast_goal           p/2, S0        /* broadcast p/2 to marked servers */
          repeat_else_try          L5, L10        /* repeatedly call L4 else branch to L9 */
L5:       receive_bindings         S0, B0         /* receive the bindings in B0 */
          store_vector_id          B0, A0         /* store the binding of the 1st argument */
          store_vector_id          B0, A1         /* store the binding of the 2nd argument */
          backtrack_if_empty       B0             /* backtrack if vector B0 is empty */
          return                                  /* return from the procedure p/2 */
L10:      try_me_else              C0             /* next server until vector C0 is empty */
p/2_cl:   copy_reference           A1, A0, q/2    /* build the 1st argument to call q/2 */
          copy_reference           A0, A1, q/2    /* build the 2nd argument to call q/2 */
          call                     q/2            /* call the procedure q/2 */
          return                                  /* return from the procedure p/2 */
q/2:      get_servers              q/2, S0        /* get servers for q/2 into filter vector  S0*/
          backtrack_if_empty       S0             /* backtrack if servers are exhausted */
          broadcast_goal           q/2, S0        /* broadcast q/2 to servers marked by S0 */
          repeat_else_try          L18, L23       /* call routine at L19 else branch to L24*/
L19:      receive_bindings         S0, B0         /* receive the bindings into vector B0 */
          store_vector_id          B0, A0         /* store the binding of the 1st argument */
          store_vector_id          B0, A1         /* store the binding of the 2nd argument */
          backtrack_if_empty       B0             /* backtrack if vector B0 is empty */
          return                                  /* return from the procedure q/2 */
L24:      try_me_else              C0             /* next server until vector C0 is empty */
r/2:      get_servers              r/2, S0        /* get servers for r/2 into vector  S0*/
          backtrack_if_empty       S0             /* backtrack if all severs are exhausted */
          broadcast_goal           r/2, S0        /* broadcast r/2 to servers marked by S0 */
          repeat_else_try          L28, L33       /* call routine at L29 else branch to L34 */
L29:      receive_bindings         S0, B0         /* receive the bindings into vector B2 */
          store_vector_id          B2, A0         /* store the bindings of the 1st argument */
          store_vector_id          B2, A1         /* store the binding of the 2nd argument */
          backtrack_if_empty       B0             /* backtrack if vector B0 is empty */
          return                                  /* return from the procedure r/2 */
L34:      try_me_else              C0             /* next server until C0 is empty */
r/2-r1:   copy_reference           A0, A0, p/2    /* build the 1st argument of p/2 */
          load_new_variable        A1, V3, p/2    /* build the 2nd argument of p/2 */
          call                     p/2            /* call procedure p/2 */
L38:      continue                 L37            /* continue to next level L39 */
L39:      copy_local_register      A0, V3         /* buildup the 1st argument of q/2 */
          copy_reference           A1, A0, q/2    /* copy the 2nd argument of q/2 */
          repeat_else_backtrack    L42, A0        /* Call L42 until A0 is empty */
L42:      load_next_vector_value   A0             /* load next value from A0 */
          call                     q/2            /* call procedure q/2  */
          return                                  /return from the procedure r/2 */

Load:     load_procedure           p/2            /* load the information of p/2 */
          load_procedure           q/2            /* load the information of q/2 */
          load_procedure           r/2            /* load the information of r/2 */
```

Fig. 7. The distributed compiled code

5.1. The execution behavior of the distributed abstract machine

To create a distributed abstract machine, a schema file with the host and path-name for each program is prepared as follows:

```
host1.domain1    /path/on/host1/program1
host2.domain2    /path/on/host2/program2
```

The command "dalps program schema" initiates the coordinator process. The first argument (the file name of the coordinator's program) is used to create a program object. A distributed abstract machine is then instantiated to execute the program. The coordinator process reads the schema file, and creates a server process for each entry.

Every server is implemented individually as a remote abstract machine object. The remote abstract machine spawns a server process on the specified host using PVM, loads the specified program into the server, and then acts as a communication interface between the coordinator and the server.

After creating the remote abstract machines, the distributed abstract machine requests that each server transmit a list of its procedures. This information is inserted into the server table as shown in Table 1.

Table 1. A server table for Example 2

Procedure id	Server id	Clause count	Server capability vector
p/2	1	4	1
p/2	2	4	1
q/2	1	4	1
q/2	2	4	1
r/2	1	2	1
r/2	2	2	1

5.2. An execution trace of the coordinator

Let us suppose that the user enters the goal *p*(*X, Y*). The distributed abstract machine initializes registers for the arguments *X* and *Y*, and starts executing the instructions from the label p/2.

The *get_servers* instruction performs an associative match against the server table with the procedure p/2. The resulting server capability vector S_0 represents matching servers. The *backtrack_if_empty* instruction tests S_0. This instruction triggers the backtracking of the coordinator in the absence of a server. In this example, the filter vector S_0 marks servers one and two.

The *broadcast_goal* instruction builds a goal consisting of procedure p/2 and the two unbound arguments, and broadcasts the goal to the servers indicated by the universal filter vector S_0.

The *repeat_else_try* instruction puts the repeat label L_{23} on the control stack. After the return instruction sees the label L_5 on the stack, the program counter is reset to L_5, and the execution of the following code is repeated. A *try_me_else* label L_{10} is also placed on the stack to handle failure. Upon backtracking, the control is transferred to the

label L_{10}. The *receive_bindings* instruction is used to report the bindings for the goal arguments from each eligible server. The initial set of received bindings corresponds to the ground facts. In this example, server 1 returns the bindings *p*(*1, 2*), *p*(*2, 3*) and *p*(*3, 4*); and server 2 returns the bindings *p*(*1, 4*), *p*(*2, 5*) and *p*(*3, 6*). The distributed abstract machine adds these bindings to its associative binding area as illustrated in Table 2. The time-stamp for these bindings is 0 (the initial time-stamp).

Table 2. The binding area for the goal p/2

Index	Server	Time-stamp	Argument 1	Argument 2
0	1	0	1	2
1	1	0	2	3
2	1	0	1	4
3	2	0	1	4
4	2	0	2	5
5	2	0	3	6

The distributed abstract machine builds a binding vector B_0 associated with the associative binding area by matching against the current timestamp *0*. The next two *store_vector_id* instructions store a reference to the binding vector B_0 in the registers for both the arguments. The *backtrack_if_empty* instruction would backtrack if the corresponding binding vector is empty; and the control would be transferred to the *try_me_else* label L_{10}. Since B_0 is not empty, the return instruction places the repeat label L_5 in the program counter and returns. The distributed abstract machine reports the current bindings to the user and waits. On request from the user, execution resumes at label L_5, and requests the next set of bindings from the servers. The timestamp is incremented by the *receive_bindings* instruction in order to differentiate the new bindings from those previously received. The bindings for the rule *p*(*X, Y*) :- *q*(*Y, X*) are added to the associative binding area at timestamp *1* as illustrated in Table 3.

The distributed abstract machine again builds a new binding vector B_0 by matching against the timestamp *1*. The sequence of three instructions — *store_vector_id*, *backtrack_if_empty* and *return* — is executed as before, and the new bindings are reported to the user. On further request from the user, control is transferred to the repeat label L_5. This time the *receive_bindings* instruction fails in the absence of additional solution for p/2 from the servers. The repeat label is removed from the control stack, and control is transferred to the *try_me_else* label L_{10}. The *try_me_else* instruction transfers control to the code for the rule at the label $p/2\text{-}cl_1$ The *copy_reference* instructions allocate new registers for the arguments to the subgoal *q*(*Y, X*). The *call* instruction then transfers control to the entry point for the compiled code of the procedure *q/2*. Both servers are requested to solve the goal *q/2*. The bindings from their ground facts are added to the associative binding area. Requests for further bindings result in *failure*, and the execution terminates.

Table 3. The binding area (second bindings)

Index	Server	Time-stamp	Argument 1	Argument 2
0	1	0	1	2
1	1	0	2	3
2	1	0	3	4
3	2	0	1	4
4	2	0	2	5
5	2	0	3	6
6	1	1	1	2
7	1	1	2	3
8	1	1	3	4
9	1	1	4	5
10	2	1	6	2
11	2	1	7	3
12	2	1	8	4

5.3. An execution trace of the servers

In this subsection we trace the execution of the code by the servers. The various states of the servers are illustrated in Table 4.

Let us assume that a user gives the goal *r*(*X, Z*) to a coordinating process. The distributed abstract machine broadcasts the goal to the matching servers. Upon receiving the goal, the servers load their registers with the given arguments and start executing their instruction code at label *r/2*. First a *match_arg* instruction identifies all of the matching clauses in the vector U0. For the first server, a filter vector points to the fact *r*(*2, 2*) and the rule *r*(*X, Y*) :- *p*(*X, Y*), *q*(*Y, Z*). The resulting filter vector U0 is ANDed with the fact filter F_0, and the resulting binding vector B0 is stored as the bindings for the two arguments. The complement of this fact vector F_0 ANDed with the vector U0 represents the matching rules in vector C0, and is saved for further processing. The binding filter vector B0 is then tested. Since the vector B0 is not empty, the program returns the control to the coordinator. The server waits in state 1 for further instructions.

Upon receiving the next request for bindings from the coordinator, the server extracts the bindings from the registers to get the vector of values. After transmitting the bindings (*2* and *2* in this example), the server looks for additional solutions. In this example, the server passes control to the *try_me_else* instruction at label L_{36}. The control is transferred to the code of the first (and only) rule of r/2 at the label r/2_cl. The server sets up the arguments and calls Procedures *p/2* and *q/2*. The server finds another set of bindings for r/2, and waits in State 2 for further instructions from the coordinator.

After receiving a request for the second set of bindings, the server sends the bindings to the coordinator. The server resumes search for additional solutions. This time the server backtracks, and picks up the additional solutions derived from the rule *p*(*X, Y*) :- *q*(*Y, X*) as given by State 3.

The server sends the requested bindings, and tries for additional solutions. The server (in State 4) replies with *failure*. Upon receiving this message, the coordinator uses the

rule for the procedure *r/2*, and requests the server to solve the subgoal *p*(*X, Y*). The server complies with the request and waits on state 5.

Table 4. Different states of the servers

State	Time	Description
1	1	bindings from r/2 — ground facts
2	2	bindings from r/2 — first solution
3	3	bindings from r/2 — second solution
4	4	no more solution for r/2
5	4	bindings from p/2 — ground facts
6	5	bindings from p/2 rule
7	5	solution from p/2 rule
7	6	solution form q/2
8	5	second solution from p/2 rule
8	7	no more solution from q/2
9	5	second solution from p/2 rule
10	6	no more solution for p/2
11	6	no more solution for p/2
11	7	solution from q/2 ground facts
12	6	no more solution for p/2
12	7	no more solution for q/2
13	6	no more solution for p/2

When requested, the server sends the bindings corresponding to its ground facts for the procedure *p/2*, and searches for additional solutions given by the rule *p*(*X, Y*) :- *q*(*X, Y*). Since the coordinator is working on the rule *r*(*X, Z*) :- *p*(*X, Y*), *q*(*Y, Z*), its next request is to solve the goal *q/2*. Upon receiving this request, the server increments its register and control stack pointers and begins working on the new goal. The second set of bindings for the previous goal *p/2* is unreported but is not overwritten due to the incrementing of the pointers. The bindings corresponding to the ground facts for the procedure *q/2* become the new current bindings (see State 7). Upon request from the coordinator, the servers report the current bindings for *q/2*. Since there are no additional solutions, the query fails (see State 8).

The coordinator reports the solutions of *p/2* and *q/2* to the user. On a request for additional solutions, the coordinator backtracks to the rule *r*(*X, Z*) :- *p*(*X, Y*), *q*(*Y, Z*), and requests additional solutions to the subgoal *q/2*. The server reports *failure* upon the coordinator's request. The abstract machine of the server backtracks to the state prior to receiving the request to solve the goal *q/2*, and awaits the next request from the coordinator (see State 9). The current bindings are now the second solution previously determined for the goal *p/2*. The coordinator backtracks, and broadcasts a request for additional bindings for *p/2*. After receiving the new solutions to the subgoal p/2, the coordinator requests the servers (except the one generating the bindings for Procedure p/2) for solutions to the subgoal q/2. The servers save the current state of the goal p/2, and compute solutions for the goal q/2 (see State 11). On request, the server returns the solutions for q/2. The server *fails* to find additional solutions for the goal *q/2* (see State 12).

When the coordinator backtracks and requests more solutions to the subgoal *q/2*, the server responds with *failure* and decrements its register and control pointers (see State 13). The coordinator continues to backtrack, and asks the server for additional solutions for the first subgoal *p/2*. The server again responds with *failure*, and empties its control stack. The coordinator finally fails.

6. A Web Based Multimedia Interface

In this section, we describe the implementation of the Java graphical user interface and front end [24] to the Distributed Associative Logic Programming System. The Java based front end contains:

(i) a lexical analyzer and a parser for the input of logical goals and assertions,
(ii) a compiler to generate instruction code and data for the Associative Logic Program engine,
(iii) routines for communicating with the distributed logic engine, and
(iv) a graphical user interface to retrieve multimedia information from the Internet via URLs.

The complete model with the Java Interface is shown in Figure 8. The Java interface provides a link from the user to an ALPS server or DALPS coordinator. A Java application parses and translates the query into an associative logic program query at the client's end. This translated query is then transferred to the coordinator or server.

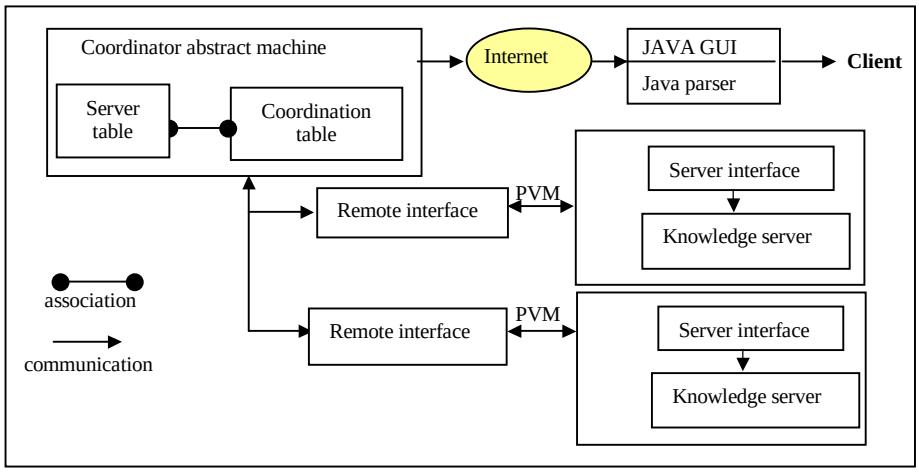

Fig. 8. A distributed heterogeneous associative system

The Java interface is shown in Figure 9. It consists of a window for viewing the text of a logic program and a text entry field for entering new facts, rules and goals. The existing programs are opened either from a local file or from the Internet by specifying a URL. Programs can be saved to the local disk or published to a web address.

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.

We demonstrate a multimedia query (see Figure 10) 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).

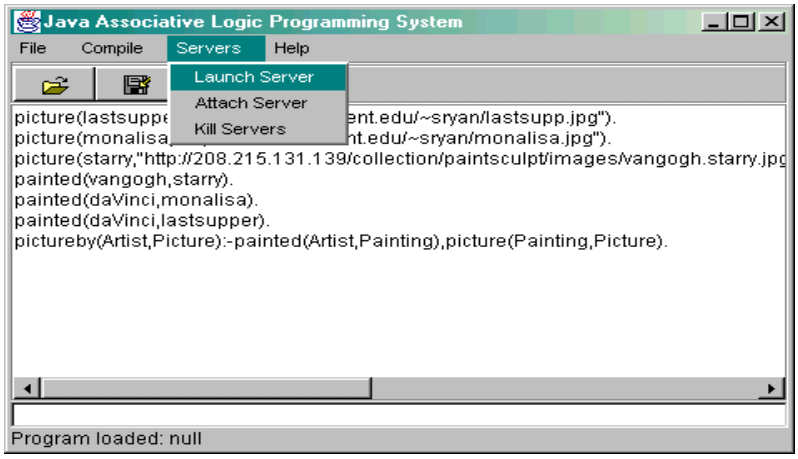

Fig. 9. Launching a knowledge server

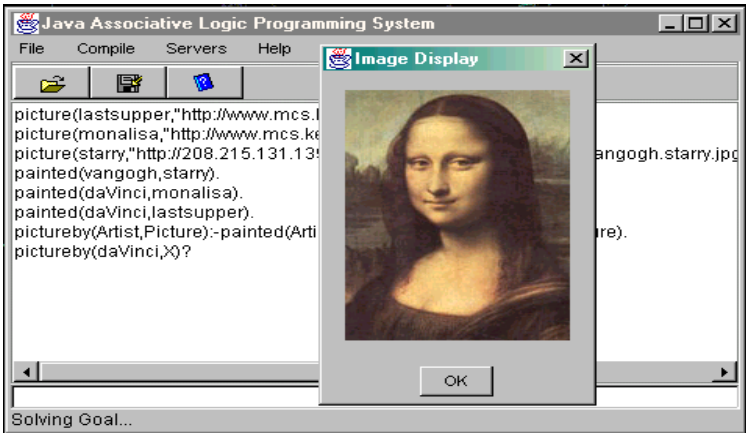

Fig. 10. An example of a multimedia solution to a query

The rule will match the artists in the multimedia knowledge base with images of their work on the Internet. There may be several such Internet sites. 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.

7. Object Oriented Implementation of Java Interface

The structure of the Java application is shown in Figure 11. The user interface is integrated with a parser, defined by the ALParser class, for handling the input from the user plus an array of RemoteAbsMachine objects each of which provides the interface for communicating with a server process, indicated here as an ALPServer.

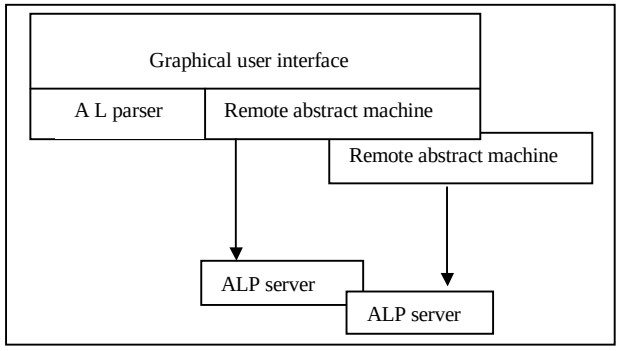

Fig. 11. The structure of Java application

The ALProgram class actually contains two representations of the program. The first is an array of procedures. Each procedure is an array of clauses. This is the representation that is built via the parser and corresponds directly to the textual version 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 passes through the lexical analyzer and is parsed. This populates the symbol table and the initial representation of the program in the Program class. At compile time (when the user selects 'Compile' from the Compile menu or launches a server for the program), the associative representation of the program is created within the ALProgram object. This is 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 against the 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, by specifying a schema file instead of a simple program, a coordinator that represents a system of distributed servers.

After all servers have been launched, goals may be submitted via the text entry line of the user interface. Launching of multiple servers results into the submission of the goal

to all of those servers. Textual results of the query are displayed in the text window. Multimedia results, such as images or sounds, are displayed accordingly on the screen.

The object-oriented implementation of the parser is illustrated in Figure 12. The ALParser class accepts a logical query from the user 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).

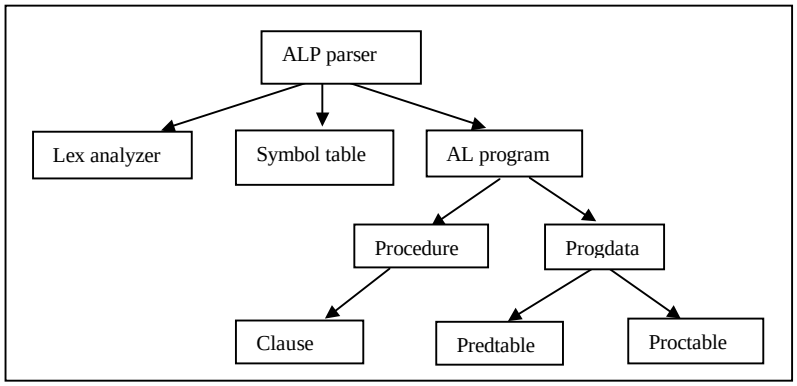

Fig. 12. A Java based parser for a logical query

8. Performance Evaluation

We tested the non-distributed (single processor) version of the abstract machine against the distributed version using a knowledge base of 20,000 facts. For the distributed version, we tested configurations of two, four, six, and eight IBM RS/6000 processors (with the facts distributed evenly on each) to study the overhead in going to a distributed paradigm. The results are summarized in Table 5. We conclude that there is definitely a speedup to be gained by distributing the data. The break-even point, where the communication costs negate any further distribution of the data, seems to be at six processors with a granularity of 3,333 facts.

Table 5. Performance results

Processors	Facts	Time (milliseconds)
1	20000	230
2	10000	130
4	5000	90
6	3333	90
8	2500	120

These tests show that the granularity of the data (for this test environment) should be more than 3,000 facts per processor. The model is thus appropriate for distributing large

knowledge bases. Allowing processors to perform more local computation will further reduce the communication overhead.

9. Related Works

We are not aware of any other distributed associative models to retrieve multimedia knowledge on a heterogeneous cluster of high performance computer systems. There are other logic programming models [6, 7, 18, 20, 27] for web based Internet programming that have been developed. JINNI [27] is a web-oriented logic programming system that utilizes Java. It relies on the native BinProlog engine when high performance is required. W-ACE [20] is a constraint-based logic programming system that is capable of web based logic programming.

BinProlog and ACE are some of the fastest implementation of WAM-based logic programming models [20, 27]. However, they do not support content-based knowledge retrieval. Our model supports content based knowledge retrieval, is cluster based, and is scalable to any architecture and number of processors 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 has a higher overhead of data transfer on a loosely coupled network.

Our model has evolved due to the need to model a complex simulation on a cluster of high performance architectures in a realistic time. Our distributed knowledge base system is efficient on a 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.

There is also work on using Java as a web based parallel language for cluster based multimedia computing [5, 8, 10] and as an interface with high performance Fortran programming. The goal of this project is to use Java in combination with message passing libraries such as PVM [26] and MPI [13] to treat the web as a large cluster based virtual computer. This work is complementary. Both our work and this work are designed to use the web as a massive parallel computer for high performance engineering design and simulation. Their work, however, focuses on high performance scientific simulation, while our work focuses on high performance knowledge retrieval from the component library and for high performance symbolic integration of multiple scientific simulations of different components of complex objects. We believe that the integration of these two approaches will be the key for the simulation of complex objects in realistic time. Both approaches will mutually benefit from these concepts.

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 computing and resource sharing among computers on the Internet.

Another interesting related work is the retrieval of images from a description of incomplete images [16, 19]. Currently we are investigating how to integrate into our system the retrieval of complete multimedia objects given partial images.

10. Conclusions

In this paper, we have discussed a generic architecture of an independent abstract machine for the distributed execution of logic programs on a heterogeneous collection of computers accessed via the Internet. The object-oriented implementation is portable, flexible and extensible. The use of a message-passing library provides scalability and architecture independence. The performance results show that distributing the data provides a significant performance improvement. More local processing will reduce overhead due to data transfer and can further improve performance.

Acknowledgments

This research was supported in part by NASA Lewis Research Center through a NASA GSRP grant. The authors also acknowledge Greg Follen and other researchers and NASA administration for useful discussions and continued support of this project.

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**, 5:3** (1992) 247 – 262.

[2] A. K. Bansal, *An Associative Model to Integrate Knowledge Retrieval and Data-parallel Computation*, International Journal on Artificial Intelligence Tools**, 3:1** (1994) 97 - 125.

[3] A. K. Bansal, L. Prasad, and M. Ghandikota, *A Formal Associative Model of Logic Programming and its Abstract Instruction Set*, Proceedings of the International Conference of Tools with Artificial Intelligence, (1994) 145--151.

[4] A. K. Bansal, *A Framework of Heterogeneous Associative Logic Programming*, International Journal of Artificial Intelligence Tools**,. 4:1 & 2** (1995) 33 - 53.

[5] D. Bhatia, V. Burzevski, M. Camuseva, G. Fox, W. Furmanski, and G. Premchandran, *WebFlow — A Visual Programming Paradigm for Web/Java based coarse grain distributed computing*, Concurrency Practice and Experience **9:6** (1997) 555-577.

[6] 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, Germany, (1996).

[7] D. Cabeza, M. Hermenegildo, and S. Varma, *The PiLLoW/CIAO Library for INTERNET/WWW Programming*, Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications, JICSLP'96, Bonn, Germany, (1996) 72-90.

[8] K. Dincer and G. Fox, *Using Java and JavaScript in the Virtual Programming Laboratory: a web-based parallel programming environment*, Concurrency Practice and Experience, **9:6** (1997) 521-534.

[9] J. A. Feldman, and D. Rovner, *An Algol Based Associative Language*, Communications of the ACM, **12:8** (1969) 439 - 449.

[10] G. Fox and W. Furmanski, *Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modeling*, Concurrency: Practice and Experience **9:6** (1997) 415-425, http://www.npac.syr.edu/users/gcf/01/terri/SCCS_793.

[11] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addision-Wesley, (1996), also see http://www.javasoft.com.

[12] D. Gries, *The Science of Programming*, Monograph, Springer Verlag, New York, (1987).

[13] W. Gropp, E. W. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with Message Passing Interface*, MIT Press, (1994).

[14] P. T. Homer and B. Schlichting, *Using Schooner to support distribution and heterogeneity in the Numerical Propulsion System Simulation Project*, Concurrency Practice and Experience, **6:4** (1994) 271-287.

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

[16] J. Khan, *Intermediate Annotationless Dynamical Object-Index-Based Query in Large Image Archives with Holographic Representation*, Journal of Visual Communication and Image Representation, **7:4** (1996) 378 – 394.

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

[18] S. W. Loke, and A. Davison, *Logic Programming with the World-Wide Web*, Proceedings of the 7th ACM Conference on Hypertext, Washington DC, USA, (1996) 235 – 245.

[19] V. Ogle and M. StoneBraker, Chabot*: Retrieval from a Relational Database of Images*, IEEE Computer **28:9** (1995) 40-48.

[20] 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) 2 -10.

[21] J. L. Potter, *Associative Computing*, Plenum Publishers, New York, (1992).

[22] J. Potter, J. Baker, A. K. Bansal, S. Scott, C. Ashtagiri, *Associative Model of Computing*, IEEE Computer, **27:11** (1994) 19 – 25.

[23] S. W. 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, (1997) 37 – 44.

[24] S. W. Ryan and A. K. Bansal, *Applying Java for the Retrieval of Multimedia Knowledge Distributed on High Performance Clusters on the Internet*, Proceedings of the International Conference on Practical Applications of JAVA, London, UK, (1999) 193 – 203.

[25] S. W. Ryan and A. K. Bansal, *A Scalable Distributed Associative Multimedia Knowledge Base System for the Internet*, Proceedings of the 8th International Conference on Intelligent Systems, Denver, Colarado, USA, (1999) 1 - 6.

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

[27] P. Tarau, *Jinni: a Lightweight Java-based Logic Engine for Internet Programming*, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, Manchester, UK, (1998) 1- 15.

[28] D. Thurman, *jPVM: A native methods interface to PVM for the Java platform*, http://www.chmsr.gatech.edu/research/projects/software.

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