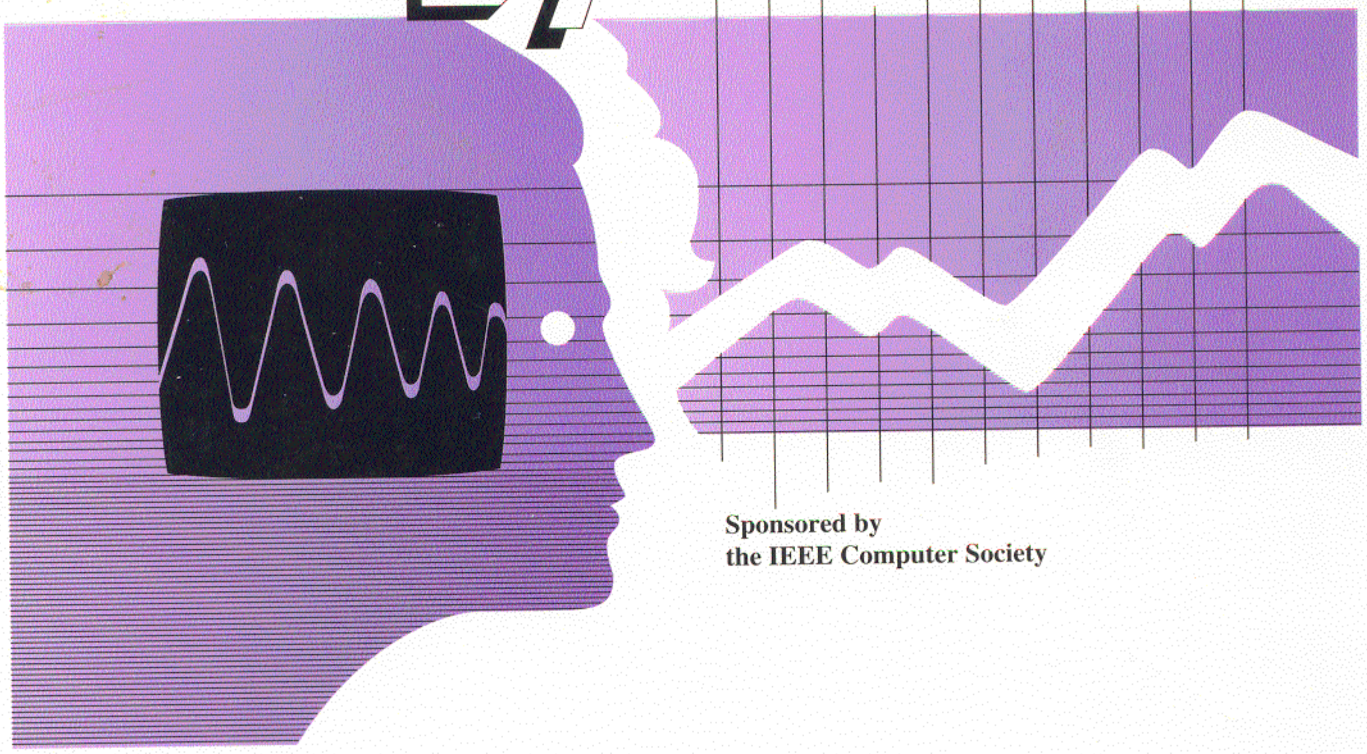
Proceedings

Ninth International Conference on

TOOLS WITH ARTIFICIAL INTELLIGENCE

TAI 97

November 3–8, 1997
Newport Beach, California



Sponsored by
the IEEE Computer Society

A Scalable Distributed Heterogeneous Associative Logic Programming System

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 distributed implementation of a scalable heterogeneous associative logic programming model, and describes an abstract instruction set for the distributed version of the model. Associative computation exploits data parallel computation. The implementation uses PVM for architecture independence, and uses object oriented programming for modularity and portability. Performance results on a cluster of IBM RS 6000 are presented.

Keywords: Artificial intelligence, Associative computing, Data-parallel computing, Distributed Computing, Heterogeneous computing, Logic programming, Knowledge base, Scalable high performance computing, Symbolic computing

1. Introduction

The advent of fast distributed networks has made it possible to distribute large knowledge bases over multiple computer systems, and to partition complex computing tasks by distributing sub-tasks over multiple processors. Different types of sub-tasks, a symbolic versus a numeric task for example, can be mapped onto processors of different architectures according to their capability. This is known as heterogeneous distributed computing.

In this paper, we describe an architecture and a distributed implementation of a heterogeneous distributed associative logic programming model based upon the theory developed in [4]. In this model, the Logic programming paradigm, heterogeneous computing, the associative computing paradigm, and the object oriented paradigm have been integrated. Logic programming provides a declarative programming model, the PVM message passing library provides architecture independence, the low level implementation using the object oriented paradigm provides modularity, and associative computing integrates data parallel computing with associative search by content.

The distributed heterogeneous associative model distributes knowledge on multiple servers either based upon the different domains or to exploit data parallelism. A coordinator is used for coordinating and collecting data from servers, while major processing is done within the servers.

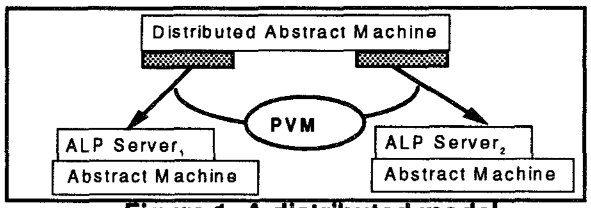


Figure 1: A distributed model

We believe that the model is suited for complex computation involving knowledge retrieval from distributed sources and the integration of symbolic and numeric computing. The distributed model consists of multiple cells each executing a sub-component of a complex computation. Each cell consists of a coordinator and multiple servers where the servers could be suited either for knowledge retrieval, symbolic computing or numeric computing.

The main contributions of this paper are as follows:

1. The model is mapped to a heterogeneous set of architectures in a user transparent manner.
2. The model supports modularity, and is scalable to any number of machines.

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 paradigm.

Associative Computing searches and selects data elements for processing according to their contents [9, 10]. 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 new associated vector. A *filter vector* is an associated vector of Boolean values resulting from a data parallel logical operation. Filter vectors are used to select records for further processing.

Data parallel computing refers to 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 to 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.

Parallel Virtual Machine (PVM) [12] is a library of message passing functions which allows processes on a heterogeneous network of computer systems to communicate and exchange data. The use of PVM makes the data transfer and network communication transparent to the user.

Logic programming [8] is a popular declarative paradigm suitable for high-level reasoning and knowledge representation. In a logic program, knowledge is represented by a set of facts and rules which describe relationships between data objects.

2.1. The Heterogeneous Associative Model

The heterogeneous associative logic programming model [4] exploits associative search to match the clause-heads with the query in a data parallel manner, and exploits compiled execution of clause bodies.

The data structures in this model are: the program representation — an associative table with parallel fields for the names and arguments in the clause heads, a data parallel environment associated with the program representation, a set of global registers, an associative control stack, and an associative table to handle aliasing of logical variables. The global registers are analogous to those in the Warren Abstract Machine [13], and are used to store pointers to variable bindings. The control stack contains information about the current state of execution and uses associative vectors to facilitate fast backtracking. Variable aliases are indicated by filter vectors and are tracked by the alias management table. A detailed explanation of this model and the corresponding abstract instructions is given in [2, 3, 4].

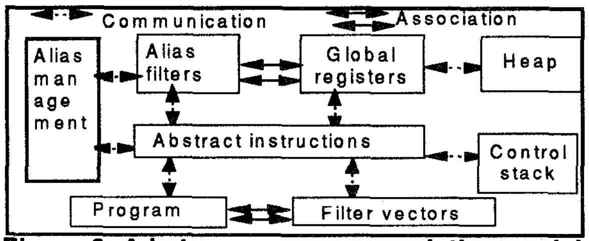


Figure 2: A heterogeneous associative model

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, Y) :- p(X, X1), q(X2, Y).

Example 1 illustrates a simple logic program. The corresponding compiled abstract instruction code is given in Figure 3. A symbol $<P_n>$ denotes a procedure, a symbol $<U_n>$ denotes a universal filter vector, a symbol $<L_n>$ or $<DP_n>$ denotes a program label, a symbol $<B_n>$ denotes the binding filter vector, a symbol $<C_n>$ denotes a clause filter vector, and a symbol $<A_n>$ denotes the *nth* argument of the current predicate.

```
DP0: match_register_arg       A0, U0
     test_and_backtrack       U0
     match_register_arg       A1, U0
     test_and_backtrack       U0
     and_bit_vectors          F0,U0, B0
     store_vector_id          B0, A0
     store_vector_id          B0, A1
     compliment_bit_vector    F0, T1
     and_bit_vectors          T1, U0, C0
     test_and_return          B0, L10
L10: try_me_else              C0
R3:  copy_logical_register    A1, A0, P1
     copy_logical_register    A0, A1, P1
     call                     DP0, P1
     return
DP1: match_register_arg       A0, U0
     test_and_backtrack       U0
     match_register_arg       A1, U0
     test_and_backtrack       U0
     and_bit_vectors          F0, U0, B0
     store_vector_id          B0, A0
     store_vector_id          B0, A1
     compliment_bit_vector    F0, T1
```

```
       and_bit_vectors          T1, U0, C0
       test_and_return          B0, L25
L25:   try_me_else              C0
DP2:   match_register_arg       A0, U0
       test_and_backtrack       U0
       match_register_arg       A1, U0
       test_and_backtrack       U0
       and_bit_vectors          F0, U0, B0
       store_vector_id          B0, A0
       store_vector_id          B0, A1
       compliment_bit_vector    F0, T1
       and_bit_vectors          T1, U0, C0
       test_and_return          B0, L36
L36:   try_me_else              C0
R1:    copy_logical_register    A0, A0, P0
       load_new_variable        A1, V4, P0
       call                     DP0, P0
       continue                 L40
L41:   load_new_variable        A0, V5, P1
       copy_logical_register    A1, A1, P1
       call                     DP0, P1
       return
LD:    load_program             PR2
       load_program             PR2
       load_program             PR2
```

Figure 3: The compiled code for Example 1

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 arguments are matched with the arguments of 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. The clause heads of rules are annotated by a clause filter-vector C_0. 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.

The compiled code for rules copies the goal arguments in global registers [3, 4] and calls the subgoals. The three *load_program* instructions load the three procedures in the program representation.

3. Object Oriented Implementation

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.

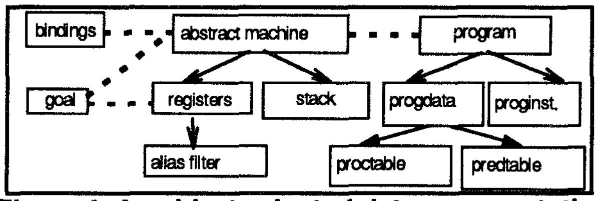


Figure 4: An object-oriented data representation

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, 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.

4. The Distributed Heterogeneous System

The Distributed system consists of two types of abstract machines: a *coordinator* abstract machine and a *server abstract machine* (see Figure 5). The coordinator launches the 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 via a message passing paradigm. In addition, all servers share a procedure table with the coordinator.

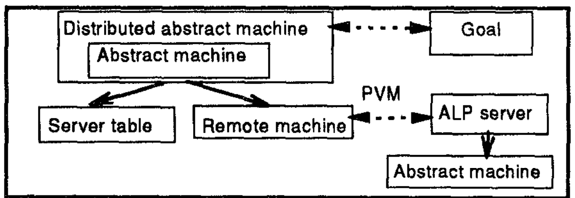


Figure 5: A distributed heterogeneous system

4.1. Abstract Data Structures

Two additional data structures — an associative server 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 process. The *procedure-id* is a reference to the entry in the procedure table which describes a goal. The *clause-count* is the number of clauses the server has that match the goal.

The *associative binding area* stores the bindings incrementally as they are received from the servers. The data elements in associative binding area are *server-id, time-stamp, variable-id*, and the *value* and *type* of each bound argument. A filter vector associated to this table is used to identify 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 procedures it is able to serve. 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 identifying all servers which have clauses that match the goal. The goal is broadcast to these servers. Upon receiving the 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. 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.

After collecting the solutions from all of the matching servers, 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.

4.3. Serving Multiple Subgoals

At a particular instance, 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 the alternate solutions to the first subgoal.

4.4. Distributed Abstract Instructions

There are four new abstract instructions in the coordinating abstract machine [12]: *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 which identifies the servers that can solve that goal.

The *broadcast_goal* instruction broadcasts 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 the 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 control the execution of the *receive_bindings* instruction.

5. A Distributed Logic Program Example

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

Server 1	**Server 2**
p(1, 2). p(2, 3). p(3, 4).	p(1, 4). p(2, 5). p(3, 6).
p(X,Y) :- q(Y,X).	same as server 1.
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, Y) :- p(X, X1), q(X2, Y).	same as server 1.

To simplify the trace, the programs in both servers have the same structure and differ only in their facts. The distributed compiled code is given in Figure 7.

```
DP0:  get_servers            P0, U0
      test_and_backtrack     U0
      broadcast_goal         P0, U0
      repeat_else_try        L4, L9
L4:   receive_bindings       U0, B0
      store_vector_id        B0, Arg0
      store_vector_id        B0, A1
      test_and_backtrack     B0
      return
L9:   try_me_else            R0
R3:   copy_logical_register  A1, A0, P1
      copy_logical_register  A0, A1, P1
      call                   DP0, P1
      return
DP1:  get_servers            P1, U0
      test_and_backtrack     U0
      broadcast_goal         P1, U0
      repeat_else_try        L18, L23
L18:  receive_bindings       U0, B0
      store_vector_id        B0, A0
      store_vector_id        B0, A1
      test_and_backtrack     B0
      return
L23:  try_me_else            R0
DP2:  get_servers            P2, U0
      test_and_backtrack     U0
      broadcast_goal         P2, U0
      repeat_else_try        L28, L33
L28:  receive_bindings       U0, B0
      store_vector_id        B0, A0
      store_vector_id        B0, A1
      test_and_backtrack     B0
      return
L33:  try_me_else            R0
R1:   copy_logical_register  A0, A0, P0
      load_new_variable      A1, V4, P0
      call                   DP0, P0
      continue               L37
L38:  load_new_variable      A0, V5, P1
      copy_logical_register  A1, A1, P1
      call                   DP0, P1
      return
LD:   load_program           PR2
      load_program           PR2
      load_program           PR2
```

Figure 7: The distributed compiled code

5.1. 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 each server to transmit a list of its procedures. This information is inserted into the server table as shown in Table 1.

Table 1. The server table for Example 2

Server	Procedure-list
1	0, 1, 2
2	0, 1, 2

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 DP_0.

The *get_servers* instruction performs an associative match against the server table with the procedure P_0. The resulting universal filter vector U_0 represents matching servers. The *test_and_backtrack* instruction tests U_0. In the absence of servers, this instruction triggers a backtrack of the coordinator. In this example, the filter vector U_0 identifies two servers.

The *broadcast_goal* instruction builds a goal consisting of procedure P_0 and the two unbound arguments, and broadcasts the goal to the servers indicated by the universal filter vector U_0.

The *repeat_else_try* instruction puts the repeat label L_4 on the control stack. After the return instruction sees the label L_4 on the stack, the program counter is reset to L_4, and the execution of the following code is repeated. A *try_me_else* label is also placed on the stack to handle failure. Upon backtracking, the control is transferred to the location given by the *try_me_else* label. 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.

Table 2: The binding area (first bindings)

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 against the associative binding area by matching against the current timestamp *0*. The next two *store_vector_id* instructions store a pointer to the binding vector B_0 in the registers for both the arguments. The *test_and_backtrack* instruction would backtrack for empty binding vector B_0; and the control would be transferred to the *try_me_else* label L_9. Since B_0 is not empty, the return instruction places the repeat label L_4 in the program counter and returns. The distributed abstract machine reports the current bindings to the user and waits. Upon further request from the user, the distributed abstract machine resumes execution at label L_4, and requests bindings from the servers. 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.

Table 3. The binding area (second bindings)

Index	Server	Time-stamp	Argument1	Argument2
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

The distributed abstract machine again builds a new binding vector B_0 by matching against the timestamp 1. The instructions *store_vector_id*, *test_and_backtrack* and *return* are executed as before, and the new bindings are reported to the user. Upon further request from the user, the control is transferred to the repeat label L_4. 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_9. The *try_me_else* instruction transfers control to the code for the rule at the label R_3. The *copy_logical_register* instructions allocate new registers for the arguments to the subgoal *q(Y, X)*. The *call* instruction then transfers control to the label DP_1 — the entry point for the compiled code of the procedure *q/2*. Both the 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.

5.3. An Execution Trace of the Servers

Here we trace the execution of the servers. The code executed by the servers is illustrated in Figure 3, and the various states of the servers are illustrated in Table 4.

Let us assume that a user gives the goal *r(X, Y)* to the 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 DP_2. First the *match_register_arg* instruction identifies all of the matching clauses. For the first server, the filter vector points to the fact *r*(*2, 2*) and the rule *r*(*X, Y*) :- *p*(*X, X1*), *q*(*X2, Y*). The resulting filter vector is AND'ed with the fact filter, F0, and the result is stored as the binding for the two arguments. The complement of this filter vector, which represents the matching rules, is saved for further processing. The binding filter vector is then tested. Since the binding vector is not empty, the program returns the control. The server waits in state 1 for further instructions.

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 from q/2
8	5	second solution from p/2 rule
8	7	no more solution of 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 solutions for p/2
12	7	no more solution for q/2
13	6	no more solution for p/2

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 returns control to the *try_me_else* instruction at label L_{36}. The control is transferred to the code of the only rule for the procedure *r/2*, at label R_1. The server sets up the arguments and calls the 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. As before, 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. This time 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.

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, Y*) :- *p*(*X, X1*), *q*(*X2, Y*), 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 as shown by 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 as shown in 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, Y*) :- *p*(*X, X1*), *q*(*X2, Y*), 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 to the subgoal *p/2*, and broadcasts a request for additional bindings for *p/2*. After receiving the new solutions to the subgoal p/2, the coordinator continues execution of the rule and requests the servers 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 it's control stack. The coordinator finally fails.

6. 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
1	20000	230 ms
2	10000	130 ms
4	5000	90 ms
6	3333	90 ms
8	2500	120 ms

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.

7. Current Limitations and Future Work

It would be desirable to automate the distribution of data and the creation of servers based on run-time requirements.

The template-based implementation of associative vectors has made it possible to handle complex data types. However, implementation of complex data-types is still underway.

It is possible to have access to a vast and widely distributed knowledge base using loose communication between multiple coordinators since the servers report to their coordinator about their capabilities. The formal model of communication between coordinators of different cells still has to be developed.

With the addition of complex data types and further refinement of the communication model, it will be possible to integrate other types of servers (specializing in numerical computation, for example).

8. Conclusion

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

Acknowledgments

The first author was supported by the summer internship program at the NASA Lewis Research Center in Cleveland, Ohio. We thank Greg Follen at NASA for that support and for providing the equipment and facilities to run the benchmarks. This work provides the basis for further research supported by a NASA GSRP grant from the NASA Lewis Research Center.

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, 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), pp. 145–151.
4. 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.
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. D. Gries, *The Science of Programming,* Monograph, Springer Verlag, New York, 1987.
7. K. Hwang, and F. A. Briggs, *Computer Architecture and Parallel Processing,* Mcgraw Hill Book Company, New york, USA, (1984).
8. R. Kowalski, *Logic for Problem Solving,* Elsevier-North Holland, (1979).
9. J. L. Potter, *Associative Computing,* Plenum Publishers, New York, (1992).
10. J. Potter, J. Baker, A. K. Bansal, S. Scott, C. Ashtagiri, "Associative Model of Computing," *IEEE Computer,* November 1994, 19 - 25
11. S. Ryan, "Scalable High Performance Distributed Execution of Heterogeneous Associative Model of Logic Programming," *MS Thesis,* Department of Mathematics and Computer Science, Kent State University, Kent, OH 44242, January 1996, 83 pages.
12. V. S. Sunderam et. al., "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience, No. 2,* (1990), pp. 315 - 339.
13. D. H. D. Warren, "An Abstract Prolog Instruction Set," *Technical Report 309,* SRI International, (1983).