ICOT Technical Report: TR-292

TR-292

Knowledge Base Machine Based on Parallel Kernel Language

by
H. Yokota (Fujitsu), H. Itoh and T. Takewaki

August, 1987

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Knowledge Base Machine Based on Parallel Kernel Language

Hidenori ITOH, Toshiaki TAKEWAKI[◇], Haruo YOKOTA[†]

ICOT Research Center
Institute for New Generation Computer Technology
Mita Kokusai Bldg., 21F, 1-4-28 Mita, Minato-ku, Tokyo 108, Japan

[†] *Fujitsu Laboratories Ltd. Kawasaki*
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

ABSTRACT

This paper describes a knowledge base machine (KBM) that is being researched and developed from the viewpoint of parallel logic programming. From the idea of parallel logic programming, a parallel kernel language (PKL) has been developed for the Fifth Generation Computer System (*FGCS*) project. Our KBM is based on the PKL. It has a parallel inference mechanism and a parallel retrieval mechanism which are controlled by an operating system.

INTRODUCTION

One of the principles of this research was that logic programming can become a new, unifying principle in computer science [1]. This is because logic programming will cover computer architecture, new programming styles, programming language semantics, and database processing. Logic programming will also play an important role in such fields as linguistics and artificial intelligence.

The logic programming language Prolog was selected as the research tool. From Prolog, a parallel kernel language (PKL) was developed, corresponding to a conventional machine language. The PKL is the nucleus of hardware and software systems for the Fifth Generation Computer System (*FGCS*) project. Inference mechanisms and knowledge

[◇] Currently working at *Toshiba Corporation*

base mechanisms will be developed as hardware systems; basic and application software will be developed as software systems.

Development is being pursued in three stages. The first was from 1982 to 1984. We are now in the middle of the second stage, which is to end in 1988.

In the initial stage, a sequential kernel language [2] was developed with an object-oriented programming feature added to Prolog. The personal sequential inference machine (PSI) [3] was developed as an inference mechanism based on it. A relational database machine (DELTA) [4] was developed as a relational database storage and retrieval mechanism connected physically with the PSI by a LAN and logically with relational commands.

In the second stage, we are pursuing three activities.

First, the deductive database machine (PHI) is being developed. The PHI is composed of the PSI and dedicated hardware knowledge base engine (KBE) [5], connected by a bus. The technologies obtained through the development of DELTA's relational engine (RE) are integrated in the KBE. Expansion of the PHIs develops a distributed deductive database system. Second, the hierarchical memory mechanism for the global shared memory is being developed. This mechanism is composed of multi-ports and a memory unit [6, 7]. The third activity is the development of the retrieval processors [8] that communicate with inference mechanisms by data streams defined in PKL.

The hierarchical memory mechanism and retrieval processors will be integrated into the prototype of the parallel KBM model in the final stage.

PARALLEL KERNEL LANGUAGE

Guarded Horn Clauses (GHC) [9] was developed as the parallel kernel language (PKL) of the FGCS. using imposition of guards and restriction of nondeterminism where the clause which has passed the guard first is used for subsequent computation.

The pioneering works of GHC are Relational Language [10], Concurrent Prolog [11], and Parlog [12]. All of these languages are very similar and their common feature is that each has a guard part and a body part, separated by a commit operator (denoted by |). Of these languages,

GHC has the simplest syntax. Each clause written in GHC is in the following form:

```
H :- G1,...,Gm | B1,...,Bn.
```

where connectives :- and , are common to ordinary Horn clauses. The part of a clause before | is called the guard, and the part after it is called the body. The calculation results of the guard are only effective in the guard. A guarded clause with no head is a goal clause, as in Prolog.

The semantics of GHC are also quite simple. The execution of a GHC program proceeds by reducing a given goal clause to the empty clause under the following rules.

Rule 1: No piece of unification in the guard of a clause can instantiate a variable in the caller.

Rule 2: No piece of unification in the body of a clause can instantiate a variable in the guard until that clause is selected for commitment.

Rule 3: When there are several clauses of the same head predicate (candidate clauses), the clause whose guard succeeds first is selected for commitment.

Rule 1 is used for synchronization, *Rule 2* guarantees selection of one body for one invocation, and *Rule 3* can be regarded as a sequencing rule for *Rule 2*. Under the above rule, each goal in a given goal clause is reduced to new goals (or null) in parallel.

RETRIEVAL BY UNIFICATION

GHC is capable of handling parallel processes using variable bindings. A process is suspended until pieces of unification succeed or fail. This suspension is important for controlling processes running in parallel. Therefore, it is difficult to retrieve structured data containing variables with GHC. Although unification can be used for retrieving structured data in Prolog, it cannot be used the same way in GHC. If search conditions are checked in the guard part of GHC clauses, the check processes are suspended and not released. If condition checks are performed in the body part of GHC clauses, alternative clauses are never selected. To retrieve the structured data, the features described in the next section, should be added to GHC.

Since the structures in knowledge bases are expected to contain variables, special retrieval functions are required for GHC to operate knowledge bases. Moreover, GHC has no functions to update static data or to control concurrent updates. The functions are related to database operations. We introduced a model and a set of primitive operations on it to retrieve structured data containing variables [13]. The model is called a relational knowledge base. Since the model is based on a relational database model, it is easy to introduce update and concurrency control.

An object handled in a relational knowledge base is a term, a structure constructed from function symbols and variables. The definition of a term is the same as that in first order logic.

(i) 0-place function symbols and variables are terms.

(ii) If f is an n-place function symbols and $t_1,...,t_n$ are terms, then $f(t_1,...,t_n)$ is a term.

(iii) Terms are generated only by applying the above rules.

Since GHC is based on first order logic, terms can be handled from GHC. A subset of the Cartesian products of term sets is called a term relation.

Operations on term relations are enhanced relational algebra operations. The enhancement is achieved by extending the equality check between items of ordinary relations to a unification between items of term relations. Thus, join and restriction are extended to *unification-join* and *unification-restriction*. These are called *retrieval-by-unification* (RBU) operations [13].

Not only retrieval functions but updating, definition, and other functions are also needed to handle knowledge bases in actual use. The following operations are planned for handling term relations.

- Retrieval
  - Unification-join
  - Unification-restriction
  - Projection
  - Union
- Update
  - Change items
  - Insert tuples
  - Delete tuples
  - Add tuples
- Definition
  - Create a term relation
  - Erase a term relation
  - Make an index for an attribute
  - Remove the index for an attribute
- Miscellaneous
  - Count tuples

INTERFACE BETWEEN GHC AND RBU

There are two different aspects in the relationship between the GHC and RBU. As stated above, RBU is used for assisting GHC to retrieve terms as items of knowledge. GHC can control parallel retrieval processes of RBU. This section considers how to connect RBU with GHC to enhance the retrieval function of the PKL. The parallel control mechanism is described in the next section.

To use RBU from GHC, there are two approaches for the interface.

1) Provide built-in predicates for each RBU operation.
2) Provide a built-in predicate and deliver commands to the predicate as incomplete messages.

The difference between these two approaches is not very great. However, when the two systems are built independently, the interface between them in the second approach is simpler than in the first one. Therefore, we plan to use the second approach.

The interface is illustrated using a simple example. The following GHC program (Fig. 1) is an implementation of the Horn logic interpreter. Horn clauses are stored in a term relation named kb1 (Fig. 2), and retrieved by unification-restriction operations. The first attribute of a term relation is used for maintaining variable substitutions for a goal clause, the second for storing the head part of a Horn clause, and the third for storing the body part of the Horn clause.

In this case, the Horn set in kb1 pointed out the parent-ancestor relationship. The term an(x,y) indicates that y is an ancestor of x, and the term pa(x,y) indicates that y is a parent of x.

The following query is given for searching for the ancestor of "a".

```
?- solve(kb1,an(a,X),Result).
```

```
solve(KB,Goal,Result) :- true |
        loop(KB,cmd(C1,C2),[ [[Goal],[Goal|]] ],Result),
        rbu(cmd(C1,C2)).

loop(KB,cmd(C1,C2),[]          ,X) :- true    | X = [], C1=C2.
loop(KB,CMD       ,[ [G,R] |L],X) :- R = []   | X = [G|Y],
        loop(KB,CMD,L,Y).
loop(KB,cmd(C1,C3),[ [G,R] |L],X) :- R \= [] |
        C1 = [unification_restriction(KB,[1=G,2=R],[1,3],S)|C2],
        merge(L,S,N),
        loop(KB,cmd(C2,C3),N,X).
```

Fig. 1. Horn Logic Interpreter Written in GHC

kb1

G	[an(A,B)|Tail]	[pa(A,B)|Tail]
G	[an(A,B)|Tail]	[pa(A,C),an(C,B)|Tail]
G	[pa(a,b)|Tail]	Tail
G	[pa(b,c)|Tail]	Tail

Fig. 2. Example of a Term Relation

In each stage of iteration of the GHC program, the following RBU commands are generated to search the knowledge base (kb1) for clauses unifiable with the resolvents. The second argument of each item in the stream (Sn) is a resolvent of the resolution, and the first argument indicates the variable bindings for the goal clause corresponding to the resolvent.

Fig. 3 shows execution examples of RBU commands. Since S2 and S5 contain empty clauses as the resolvents, an(a,b) and an(a,c) are the answers of the query.

These iteration processes and retrieval processes can execute in parallel if the stream is long enough. That is to say, the system

```
unification_restriction(kb1,[1=[an(a,X)],2=[an(a,X)]],[1,3],S1)
  S1 = [[[an(a,X)],[pa(a,X)]],[[an(a,X)],[pa(a,C),an(C,X)]]]
unification_restriction(kb1,[1=[an(a,X)],2=[pa(a,X)]],[1,3],S2)
  S2 = [[[an(a,b)],[]]]
unification_restriction(kb1,[1=[an(a,X)],2=[pa(a,C),an(C,X)]],[1,3],S3)
  S3 = [[[an(a,X)],[an(b,X)]]]
unification_restriction(kb1, [1=[an(a,X)],2=[an(b,X)]],[1,3],S4)
  S4 = [[[an(a,X)],[pa(b,X)]],[[an(a,X)],[pa(b,C),an(C,X)]]]
unification_restriction(kb1,[1=[an(a,X)],2=[pa(b,X)]],[1,3],S5)
  S5 = [[[an(a,c)],[]]]
unification_restriction(kb1,[1=[an(a,X)],2=[pa(b,C),an(C,X)]],[1,3],S6)
  S6 = [[[an(a,X)],[an(c,X)]]]
unification_restriction(kb1,[1=[an(a,X)],2=[an(c,X)]],[1,3],S7)
  S7 = [[[an(a,X)],[pa(c,X)]],[[an(a,X)],[pa(c,C),an(C,X)]]]
unification_restriction(kb1,[1=[an(a,X)],2=[pa(c,X)]],[1,3],S8)
  S8 = []
unification_restriction(kb1,[1=[an(a,X)],2=[pa(c,C),an(C,X)]],[1,3],S9)
  S9 = []
```

Fig. 3. Execution Examples of RBU Commands

implements the OR parallel Horn logic interpreter. This is an example of RBU usage in GHC. The parallel problem solvers and parallel production systems can be built using GHC and RBU.

PARALLEL CONTROL METHODS OF RETRIEVAL PROCESSES IN GHC

This section consists of three parts. First, it describes three parallel control methods. Each method is considered as parallelism depending on the number of retrieval elements and data division. Then, meta-level control uses three parallel control methods according to the processing conditions. The last part describes the implementation of each method in GHC.

The command from inference elements is the RBU command in section 3, and is called the retrieval process. The RBU command is assigned to retrieval elements by the parallel controller and is processed. Parallel control for retrieval processes is affected by the following parameters: number of available retrieval elements, type of retrieval commands, and size of data to be handled by a command.

Parallel Control Methods and Data Division

Three parallel control methods [14] for retrieval processes are considered. The symbols n and i indicate the total number of retrieval elements and the number of available retrieval elements when the retrieval command is received.

(1) Data non-division method (method 1): Each retrieval command is executed by a single element, without data division. The controller receives a command, searches for an available retrieval element, and allocates the received command to its element. This method has a little overhead because of parallel control.

(2) Data dynamic division method (method 2): The controller receives a command, and searches for all available retrieval elements. If there are i available retrieval elements, the controller converts the command to i (variable number) sub-commands (if division of data is possible), and allocates it to i retrieval elements. This method has a lot of overhead because of parallel control, status management of retrieval elements, and data division.

(3) Data static division method (method 3): This method is a combination of methods 1 and 2. The controller converts the command to n (fixed number) sub-commands (if division of data is possible). The allocation of sub-commands is much the same as method 1. This method has overhead because of parallel control and data division. The overhead of this method is between those of methods 1 and 2.

Methods 2 and 3 subdivide the grains of a process according to the division of handled data, cut down the free status of elements by using retrieval elements for details, and increase parallelism.

The data division operation of unification restriction is converted to sub-commands and executed. For example, Fig. 4 shows the process (oval) and the flow of data (arrow) performed by the data n division. The symbol L indicates the size of the data to be handled by one command.

In the unification restriction command (ur(L)), every ur_i corresponds to the unification restriction process in order to handle $ur_i(L/n)$, and an append corresponds to the append process for data streams after it has been restricted. The unification restriction operation of data n division needs n unification restriction processes and one append process. To remove the append process, a differential list that can partly hold a value is used. (This is a property of logical variables.) In short, the parts of unification restriction processes are fast when the number of divisions is large and an append process does not need execution time using a differential list.

The processes in the data division operations are executed along the stream of data from left to right, as shown in Fig. 4, and parallel processing of every process is possible when every process receives data.

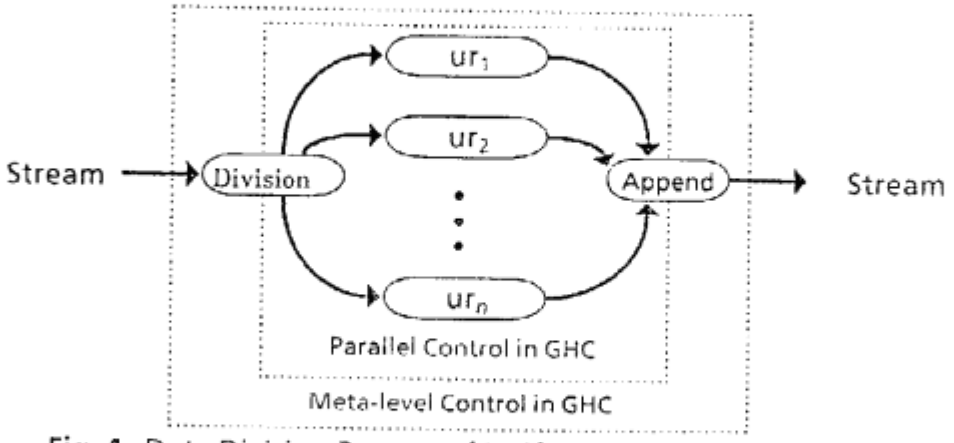

Fig. 4. Data Division Process of Unification Restriction Operation

Meta-level Control

Meta-level control uses three parallel control methods according to the retrieval processing conditions. The meta-level controller is controlled by meta-knowledge which guides the use of retrieval elements toward the best way which is usually synonymous with fast processing. Meta-level control uses meta-knowledge that satisfies the following conditions: state of the waiting queue of commands and type of commands.

First, control according to the state of the waiting queue of commands is described. If there is little traffic in the waiting queue, the meta-level controller will use method 3 because many retrieval elements are available. If there is heavy traffic in the waiting queue and the queue contains unprocessed commands, the controller will use method 1 because few retrieval elements are available. Otherwise, the controller will use method 2 for fine control.

Next, control method by command type is described. Use of the retrieval elements in parallel depends on the command type. For example, term sorting is completed during n phases. In the $i+1$ phase, $k/2$ retrieval elements are used, whereas in the i phase, k retrieval elements are used. In the next example, unification restriction is completed in only one phase. In this case, all the idle retrieval elements can be used. When the retrieval element controller receives several types of commands at the same time, it is important to use the retrieval elements in high efficiency. All of the above items are also controlled by meta-level control.

Implementation of Methods in GHC

The implementation of each method uses the properties of logic programming, and parallel semantics of GHC and its synchronization mechanism. Each method is written in Flat GHC [9].

Fig. 5 shows the parallel controller with status management for retrieval elements under method 2. The top level predicate `'REscheduler'` of the parallel controller receives a stream of RBU commands from inference elements, selects all available retrieval elements, and assigns commands to available retrieval elements. The predicate `closeStream` informs every retrieval element of the end of a command when a stream of commands is empty. The predicate `inspect` asks the free or busy status of

```
% Top level
'REscheduler'([C|T],Stream) :- true |
   availableREs(Stream, NS,Free), divide(Free,[C|T],NS).
'REscheduler'([]   ,Stream) :- true | closeStream(Stream).

availableREs(St, NS,Free) :- true |
   inspect(St,NS,Ins), checkingFree(Ins,Free).

% Inspection of REs status
inspect([]                  ,New,Ins) :- true | New=[], Ins=[].
inspect([stream(N,St)|Rest],New,Ins) :- true |
   New=[stream(N,SR)|NR], Ins=[(N,State)|IR],
   St=[ins(State)|SR], inspect(Rest,NR,IR).

divide([] ,C    ,St) :- true    |
   'REscheduler'(C,St).                          % All REs are busy.
divide(REs,[C|T],St) :- REs\=[] |                % send out C to free REs
   division(C, REs, SubC), sendOut(REs,SubC,St,NS),
   'REscheduler'(T,NS).

% 'RE' manages status of REs, SendToRE sends C to Nth RE.
'RE'(N,[term  |Cmd]) :- true | Cmd=[].          % termination
'RE'(N,[ins(C)|Cmd]) :- true |
   C=free, 'RE'(N,Cmd).                          % inspection of status
'RE'(N,[cmd(C)|Cmd]) :- true |                   % retrieval command
   sendToRE(N,C,Res), response(Res,Cmd,Next), 'RE'(N,Next).

response(end,Cmd         ,N) :- true | Cmd=N.   % Process ends.
response(R  ,[ins(C)|Cmd],N) :- true | C=busy,response(R,Cmd,N).
```

Fig. 5. Parallel Controller with Status Management

every retrieval element. The predicate checkingFree searches for the number of available retrieval elements, and selects the available retrieval elements.

The predicate **division** generates sub-commands (third argument) by dividing the data of the command (first argument) by the number of available retrieval elements (the second argument indicates the list of retrieval elements). The predicate sendOut assigns sub-commands (first argument) to retrieval elements (second argument). The predicate 'RE' handles commands from inference elements, and manages the status for retrieval elements. Arguments of the predicate 'RE' indicate the number of elements and the stream of commands.

The predicate sendToRE issues a command to a retrieval element which is appointed by the first argument. This predicate instantiates the third argument to the atom end when a process comes to an end. If the predicate response receives the command ins(C) for an inspection of

```
metaControl([] ,                          ST, REs) :- true |
      closeStream(ST).
metaControl([cmd(C,Type,Size)|Rest], ST, REs) :- true |
      availableREs(ST, IS, Free),
      strategy(C,Type,Size,Free, Method),
      solve(Method, REs, Free, cmd(C), IS, NS),
      metaControl(REs, Rest,NewST).

solve(method1, REs, Free, C, ST, NS) :- true |
      selectRE(Free,RE), sendOut([C],[RE],ST,NS).
solve(method2, REs, Free, C, ST, NS) :- true |
      division(C,Free,SubC), sendOut(SubC,Free,ST,NS).
solve(method3, REs, Free, C, ST, NS) :- true |
      division(C,REs, SubC), sendOut(SubC,Free,ST,NS).
```

Fig. 6. Part of Program for Meta-level Control

status when an element is busy, then the atom **busy** is returned, otherwise, the atom **free** is returned by the predicate **'RE'**.

Fig. 6 shows the part of the program for meta-level control. The top level of the meta-level controller uses the predicate **metaControl** instead of the predicate **'REscheduler'** in Fig. 5. The predicate **metaControl** uses three control methods according to retrieval element conditions, command type, and the size of the data to be handled by the command. This predicate receives commands with several items of information, searches for any available retrieval elements (**availableREs**), selects the best control method from among all the methods (**strategy**), processes the command by this method (**solve**), and repeats this processing.

KNOWLEDGE BASE MACHINE MODEL

The knowledge base machine model is shown in Figs. 7 and 8.

The upper network and clusters (CLs) are also being researched and developed for a parallel inference mechanism (PIM). The PIM will be composed of about 100 CLs, each CL with about 10 processing elements.

The KBM is composed of a PIM, retrieval elements (REs), and a global shared memory (GSM). The number of REs is about one tenth of the number of CLs. Each CL has a local shared memory (LSM) for its PEs, and has a GSM connected with the lower network and REs for PEs in other CLs. The KBM has a hierarchical shared memory composed of LSM and GSM.

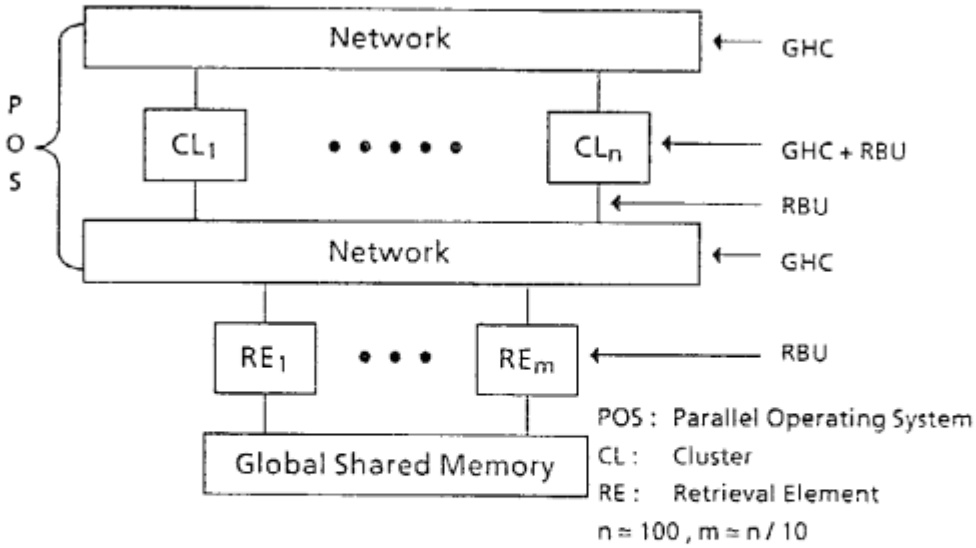

Fig. 7. Ideal Model of Knowledge Base Machine

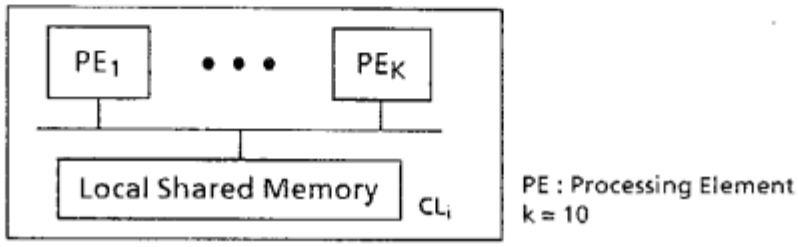

Fig. 8. Ideal Model of Inference Element Cluster

The RE receives the retrieval commands given in section 4 from CLs through the lower network. In the lower network, the destination REs of retrieval commands should be dynamically determined, as described in section 5. For this purpose, the RE parallel control method in GHC needs to be integrated into the lower network. Physically, the upper and lower networks are the same. These networks and CLs are controlled under a single parallel operating system (POS). REs are also controlled in parallel as KBM classes under the POS.

When the RE receives a retrieval command, it starts to access the global shared memory. The sets of retrieved knowledge stream into the RE. While the pairs of sets of knowledge flow into the RE, the RE sorts the knowledge in the generality defined in unifiability and detects any pairs that are unifiable. The unifiable sets of pairs are unified in a pipelined way and are sent to CLs as the response of retrieval commands [8]. The stream flowing into the RE has an affinity with the definition of the data stream in GHC.

The GSM is composed of a number of memory banks. Each bank has ports for reading and writing knowledge stored in logical page size. A logical page lies across all the banks. Each port connected to the bank can access the same page. Then the GSM mechanism permits REs to be accessed in the same logical domain at the same time [6, 15].

CONCLUSION

This paper described a knowledge base machine model based on a parallel kernel language(PKL). For the PKL, GHC has been introduced and for the knowledge base model, a term relation and its operations, named RBU, has also been introduced. We combined RBU and GHC with built-in predicates so that all solution collecting feature required in RBU were realized in GHC, which has only a don't care nondeterministic feature. The parallel control feature of GHC is effectively used for the retrieval elements, and the data-stream manipulation method has an affinity with the definition of data-stream in GHC introduced in the retrieval element. The knowledge base machine and parallel inference machine will be integrated into an FGCS prototype in the final stage of the project. To realize this aim, we accumulated technologies through research and development of a relational database machine (DELTA) in the initial stage, and a deductive database machine (PHI) based on the PSI and an experimental multi-port page shared memory mechanism in the intermediate stage. Applying these technologies, we have started to develop an experimental KBM in a parallel kernel language.

The basic functions introduced in this FGCS have been evaluated by developing some applications on it.

ACKNOWLEDGEMENTS

We would like to thank Dr. Fuchi, the director of ICOT, who contributed many useful suggestions for this research and development. We also wish to thank the members of the KBM working group and the people in industry who participated in joint research programs for their helpful discussions. We also extend our thanks to Dr. Furukawa and Dr. Uchida for their valuable comments on the KBM project.

REFERENCES

[1] K. Fuchi, "Revisiting Original Philosophy of Fifth Generation Computer Systems Project," in *Proc. of the International Conference on Fifth Generation Computer Systems*, ICOT, 1984

[2] T. Chikayama, "Unique Features of ESP," in *Proc. of the International Conference on Fifth Generation Computer Systems*, pp.292-298, ICOT, 1984

[3] M. Yokota, A. Yamamoto, et al., "The Design and Implementation of a Personal Sequential Inference Machine: PSI," *New Generation Computing*, Vol.1, pp.125-144, Ohmsha, 1984

[4] K. Murakami, et al., "A Relational Database Machine: First Step to Knowledge Base Machine," in *Proc. of 10th Annual International Symposium on Computer Architecture*, 1983

[5] M. Wada, Y. Morita, et al., "A Superimposed Code Scheme for Deductive Databases," in *Proc. of the 5th International Workshop on Database Machines*, Oct. 1987

[6] Y. Tanaka, "A Multiport Page-Memory Architecture and a Multiport Disk-Cache System," *New Generation Computing*, Vol. 2, pp. 241-260, Ohmsha, Feb. 1984

[7] H. Sakai, et al., "A Simulation Study of a Knowledge Base Machine Architecture," in *Proc. of the 5th International Workshop on Database Machines*, Oct. 1987

[8] Y. Morita, H. Yokota, et al., "Retrieval-by-Unification on a Relational Knowledge Base Model," in *Proc. of the 12th International Conference on VLDB*, Aug. 1986

[9] K. Ueda, "Guarded Horn Clauses," *Logic Programming '85*, E. Wada (ed). Lecture Notes in Computer Science 221, Springer-Verlag, 1986

[10] K. Clark, S. Gregory, "A Relational Language for Parallel Programming," in *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pp.171-178, ACM, 1981

[11] E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," Technical Report TR-003, ICOT, 1983

[12] K. Clark, S. Gregory, "PARLOG: Parallel Logic Programming in Logic," Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, 1984

[13] H. Yokota, H. Itoh, "A Model and an Architecture for a Relational Knowledge Base," in *Proc. of the 13th International Symposium on Computer Architecture*, pp.2-9, June 1986

[14] H. Itoh, C. Sakama, et al., "Parallel Control Techniques for Dedicated Relational Database Engines," in *Proc. of 3rd International Conference on Data Engineering*, 1987

[15] H. Monoi, et al., "Parallel Control Technique and Performance of an MPPM Knowledge Base Machine," Technical Report TR-284, ICOT, 1987