ICOT Technical Report: TR-380

TR-380

Knowledge Retrieval and Updating for Parallel Problem Solving

by
H. Yokota, H. Kitakami
and A. Hattori(Fujitsu)

June, 1988

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 Retrieval and Updating for Parallel Problem Solving

Haruo Yokota †
Hajime Kitakami††
Akira Hattori†††

FUJITSU LIMITED
Kawasaki, Japan

ABSTRACT

Parallel problem solving systems capable of updating and retrieving information from a knowledge base are being studied as applications of the Fifth Generation Computer Systems project in Japan. This paper describes one such production system that traverses a search tree in parallel. We propose a new search strategy called the "Better First Search." Communications among processes to control execution priority is localized by the search strategy, with a tree structure used for the process configuration. We use the Guarded Horn Clauses (GHC) parallel logic programming language and the Retrieval By Unification (RBU) knowledge base handling system to implement this system. Each node of the process tree is implemented as a perpetual GHC process. Elements of knowledge are stored in knowledge bases as sets of RBU terms. GHC and RBU are tightly coupled to minimize communication overhead. Indexing using hashing and a trie structure is used to retrieve terms from the knowledge base using unification and backtracking. The retrieval and update speeds of a prototype implemented using this method are compared with those of a Prolog system.

---

e-mail address:
† hyoko%motoko.stars.flab.fujitsu.junet@uunet.UU.NET
†† kami%motoko.stars.flab.fujitsu.junet@uunet.UU.NET
††† hattori%ayumi.stars.flab.fujitsu.junet@uunet.UU.NET

1. Introduction

Parallel problem solving systems show promise in applications of the Fifth Generation Computer Systems (FGCS) project in Japan. The goal of the FGCS project is to build a knowledge information processing system using logic programming paradigms [Fuchi 84]. Parallel logic programming languages are used to develop applications in the FGCS project. Knowledge bases must be used efficiently, which is difficult in a parallel logic programming language. Combining a parallel logic programming language and a dedicated system for operating a knowledge base seems to be one possible solution to implement a logic programming language based parallel problem solving system with reasonable efficiency [Itoh 87].

Guarded Horn Clauses (GHC) [Ueda 85], a parallel logic programming language with committed choice semantics, is the kernel language of the FGCS project. It handles parallel processes and streams for communicating among processes efficiently, but is inadequate in searching for alternate knowledge elements used in problem solving, since a variable of GHC can only be assigned once. GHC also has trouble handling global information such as that in knowledge bases. GHC has no appropriate means of guaranteeing the consistency of knowledge bases during parallel updating.

[Yokota 86] proposed a system called by Retrieval By Unification (RBU) in which the knowledge element is a term, a well defined structure capable of handling variables. A set of terms is stored as a table and called a term relation. A term relation is used to control consistency in parallel operations. This system retrieves terms from term relations using unification and backtracking and updates each element. RBU commands for retrieving and updating term relations are issued from a parallel problem solving system written in GHC.

Operations for retrieving and updating knowledge elements are very influential in determining the speed of each problem solving step. The data structures for storing knowledge elements, the mechanisms for retrieving and updating these elements, and the associated indexing methods are therefore important issues and deserve careful consideration in the design of this type of system.

Traversing a search tree is a commonly used method for solving Artificial Intelligence (AI) problems. Since the search space for a problem can grow quickly, several strategies have been proposed to solve actual problems [Barr 81]. These strategies are for use in sequential problem solving systems, however, and new strategies are needed for parallel problem solving.

In Section 2 of this paper, we first outline a parallel problem solving system using a new search strategy called the Better First Search. We use a tree structure as the process configuration. Section 3 discusses the stream oriented use of RBU by GHC processes and communication between RBU and GHC. Knowledge bases and intermediate process states are manipulated by RBU. Section 4 presents a prototype system of RBU with a new indexing method. Search and update speeds are compared with those of a Prolog system.

2. Overview of a parallel problem solving system

Production (rule-based) systems are used as tools for problem solving. The basic concept involves applying state transition production rules from an initial state to reach a goal state that satisfies termination conditions. Several states can be generated from a single state by applying the production rules and the state transitions make a search tree (Figure 1). Each node of the tree is a transitive state and the root node is the initial state. The goal of a production system is to derive a path from the initial state to a goal state by traversing the search tree.

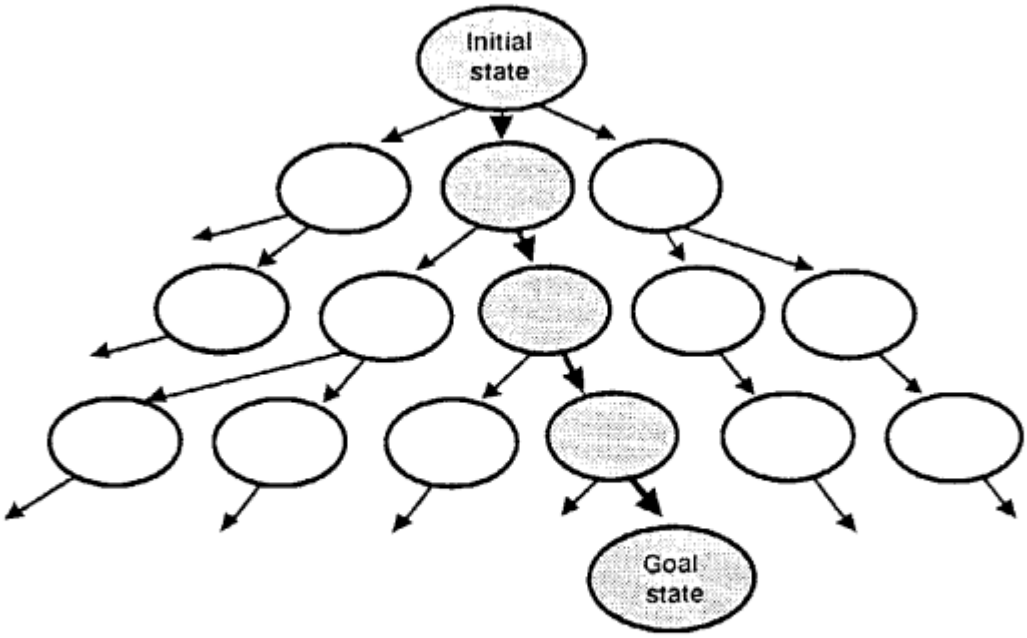

Figure 1. Traversing a search tree

2.1 Search strategy for parallel production systems

Parallel processing is viewed as a way of reducing the large amounts of time consumed by production systems [Gupta 87]. One implementation is the parallel traversal of a search tree in which new states are generated from different states in parallel. Limits on memory and the number of processors require the use of special search strategies. The best first search [Barr 81] is one such strategy. It selects a state from a search tree using state evaluation of the current state to generate new states. The state selected has the best evaluation value in the tree at a given time.

The centralized control of this strategy makes finding the best value a bottleneck, however. Control must be localized for efficient parallel processing. We propose a new search strategy called the Better First Search. The strategy looks for a state having the best evaluation value only in a subtree of the search tree. Although this value is good, it may not be the best in the entire tree; therefore we call it a "better" value.

2.2 Process configuration

We use a tree structure as the process configuration to implement the Better First Search in parallel. The tree configuration is not directly related to the search tree traversed by the production system. The three types of nodes (processes) in the process tree are the root node, leaf nodes, and other branch nodes. Productions are performed at the leaf nodes. Production priorities are controlled at the branch nodes based on their evaluation values. System control such as that of the user interface is performed at the root node. Figure 2 shows the process configuration and a search tree.

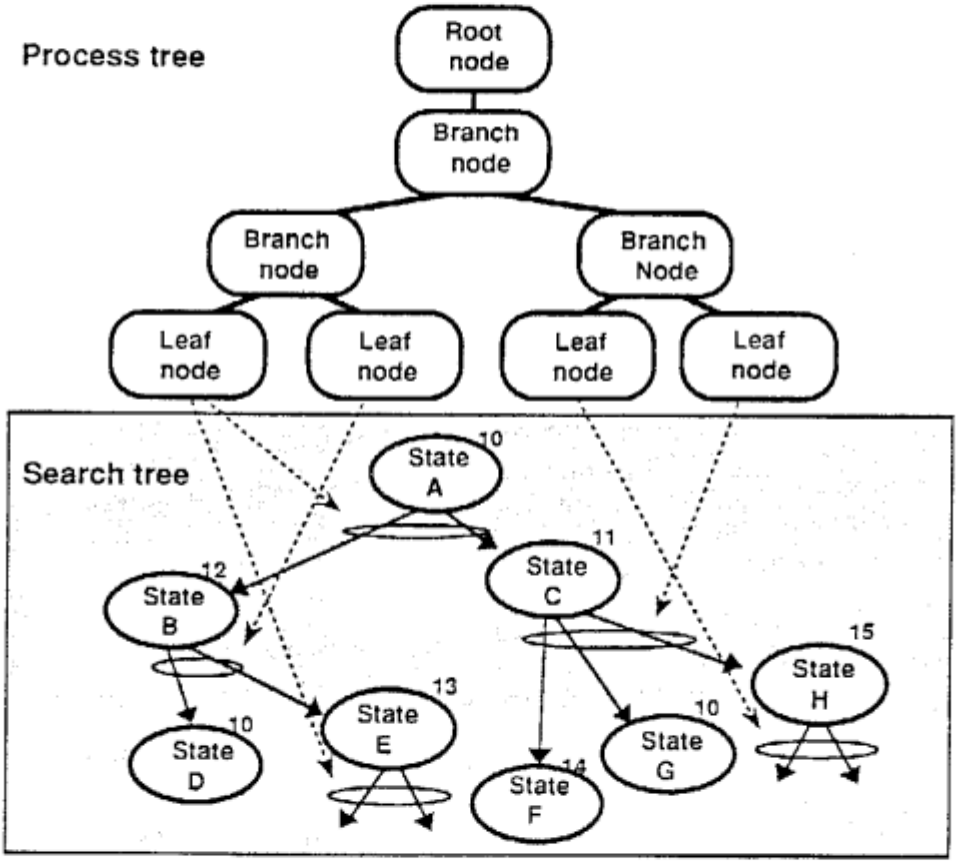

Figure 2. Process configuration and a search tree

2.3 Process tree nodes

A process tree having four leaf nodes is constructed in GHC as follows:

```
four_leaf_tree :- true |
        root(B1u,B1d),
        branch(B1d,B2u,B3u,[],[],B1u,B2d,B3d),
        branch(B2d,L1u,L2u,[],[],B2u,L1d,L2d),
        branch(B3d,L3u,L4u,[],[],B3u,L3d,L4d),
        leaf(L1d,L1u),
        leaf(L2d,L2u),
        leaf(L3d,L3u),
        leaf(L4d,L4u).
```

Nodes in the process tree are implemented using perpetual processes generated from recursively called GHC clauses. Process behavior is controlled by streams bound to variables in arguments in the clauses. The streams are treated as messages for the process. A leaf node is a perpetual process generated by the following two clauses:

```
leaf([state(E,S)|I],O1) :- true |
        production(state(E,S),O1,O2),
        O2 = [request|O3],
        leaf(I,O3).
leaf([],O) :- true |
        O=[].
```

The first argument is used as a receiver of messages and the second as a sender. Where the top element of the first argument is a state indicator, the leaf node receives a state from another node. The leaf node then invokes a production process for the state. The results and a request message for the next production are sent using the second argument. The second clause indicates that the leaf node returns a nil when it receives a nil.

Branch node behavior is also determined by node status. Two nodes receiving the same message execute different operations based on whether

they are holding states, or waiting for states. A branch node that has no states and receives a request message must keep the message in a queue to indicate that the node is waiting for states and must send another request message to the upper node. The direction of a request stored in the queue indicates which child node is to be sent a state when the branch node receives a state.

```
branch(IU,[request|IL],IR,RQ,SQ,OU,OL,OR) :- SQ = [] |
        OU = [request|OUx],
        branch(IU,IL,IR,[req(left)|RQ],SQ,OUx,OL,OR).
branch(IU,IL,[request|IR],RQ,SQ,OU,OL,OR) :- SQ = [] |
        OU = [request|OUx],
        branch(IU,IL,IR,[req(right)|RQ],SQ,OUx,OL,OR).
```

IU, *IL*, and *IR* indicate streams of input messages from the upper, left, and right child nodes. *OU*, *OL*, *and OR* indicate output messages to these nodes. *RQ* and *SQ* are queues for requests and states.

If state queue *SQ* contains states when the node receives a request message, it selects a state from the queue based on evaluation values and sends it to the node that sent the request message.

```
branch(IU,[request|IL],IR,RQ,SQ,OU,OL,OR) :- SQ = [_|_] |
        select(SQ,State,SQx),
        OL = [State|OLx],
        branch(IU,IL,IR,RQ,SQx,OU,OLx,OR).
branch(IU,IL,[request|IR],RQ,SQ,OU,OL,OR) :- SQ = [_|_] |
        select(SQ,State,SQx),
        OR = [State|ORx],
        branch(IU,IL,IR,RQ,SQx,OU,OL,ORx).
```

When request queue *RQ* contains requests, the node sends a state received from another node to the node that generated the request message. In the following example, a state received from an upper node is sent to the left child node:

```
branch([state(E,S)|IU],IL,IR,[req(left)|RQ],[],OU,OL,OR) :- true|
        OL = [state(E,S)|OLx],
        branch(IU,IL,IR,RQ,[],OU,OLx,OR).
```

When the request queue RQ is empty in the above situation, the node adds a state received from another node to the SQ:

```
branch([state(E,S)|IU],IL,IR,RQ,SQ,OU,OL,OR) :- RQ = [] |
        branch(IU,IL,IR,RQ,[state(E,S)|SQ],OU,OL,OR).
```

3. Use of RBU from GHC processes

A production is performed by retrieving knowledge elements from a knowledge base and updating the knowledge base based on production rules. The knowledge base is a global state for parallel production processes. GHC cannot handle global states among perpetual processes, nor effectively retrieve and update the knowledge base, even if a common stream is prepared as an argument of every clause to implement a global state in GHC. The unification implemented in GHC cannot be used to search for multiple knowledge elements, because a GHC variable can only be assigned a value once. Once bound with a knowledge element, the GHC variable's binding cannot be changed.

Connecting GHC to a dedicated system that processes knowledge bases enables a parallel production system to be built. RBU knowledge elements are terms defined in the same first order logic as GHC thus eliminating syntactical transformation. RBU stores a set of terms as a term relation which is used to guarantee the consistency in knowledge bases during parallel updating.

3.1 *Rbu* predicate

The special predicate $rbu(C)$ is provided in GHC to use RBU. Commands for retrieving and updating knowledge bases are bound to the stream argument C.

For example:

$$C = [urs(tr1,[1],p(a,\$(1)),[1],X),ujs(tr1,[2],tr2,[1],[3],Y),...].$$

Urs represents a unification restriction stream operation and *ujs* a unification join stream operation [Yokota 86]. These operations are an extension of relational algebra operations with unification. The first command sentence, *urs(tr1,[1],p(a,$(1)),[1],X)*, dictates a search of the first attribute of the term relation *tr1* for terms unifiable with the condition *p(a,$(1))*, yielding the derivation of the first attribute as a result. Results are returned as a stream bound to the variable *X* in the command sentence.

$$X = [p(a,g(\$(2))),p(a,g(b)),..],$$

The second command sentence, *ujs(tr1,[2],tr2,[1],[3],Y)*, is used to derive the third attribute of a result relation generated by a unification join operation which searches the second attribute of *tr1* and the first attribute of *tr2* for unifiable terms. Results are returned bound to the variable *Y*.

$$Y = [q(\$(10),c),..]$$

The special function symbol *$* is used to indicate a variable in command sentences and in results. GHC variables cannot be used for knowledge retrieval, so other symbols are needed to indicate variables for retrieval. These variables are bound to knowledge elements in RBU, but unbound in GHC. This corresponds to unbound variables appearing in a template predicate of the *setof* predicate in Prolog systems.

3.2 Production in GHC with RBU

A production system is implemented in GHC using the *rbu* predicate. A term relation for production rules and another term relation for an initial state are given and transitive states generated during production stored as term relations.

As an example, consider the production rules and the initial state of the "Monkey and Banana" problem in Tables 1 and 2. The first attribute of

Table 1 contains the matching part and the second attribute contains the execution part of the production rules. The first attribute of Table 2 is the contents of the initial state. The second and third attributes of the table are used at execution time as working spaces.

Table 1. Example of production rules

[monkey(on(floor),hold($(1)),$(2),$(3)). object($(4),$(5),$(6),$(7)). history($(8))]	[[$(2)\=$(6),$(3)\=$(7)], [monkey(on(floor),hold($(1)),$(2),$(3)). history($(8))]. [monkey(on(floor),hold($(1)),$(6),$(7)). history([move($(6),$(7))\|$(8)])]]
[monkey(on(ladder),hold(nil),$(1),$(2)). object($(3),on(ceiling),$(1),$(2)). history($(4))]	[[]. [monkey(on(ladder),hold(nil),$(1),$(2)). object($(3),on(ceiling),$(1),$(2)). history($(4))]. [monkey(on(ladder),hold($(3)),$(1),$(2)). history([hold($(3))|$(4)])]]
[monkey(on(floor),hold(nil),$(1),$(2)). object($(3),on(floor),$(1),$(2)). history($(4))]	[[]. [monkey(on(floor),hold(nil),$(1),$(2)). object($(3),on(floor),$(1),$(2)). history($(4))]. [monkey(on(floor),hold($(3)),$(1),$(2)). history([hold($(3))|$(4)])]]
[monkey(on(floor),hold($(1)),$(2),$(3)). history($(4))]	[[$(1)\=nil], [monkey(on(floor),hold($(1)),$(2),$(3)). history($(4))]. [monkey(on(floor),hold(nil),$(2),$(3)). object($(1),on(floor),$(2),$(3)). history([drop($(1))\|$(4)])]]
[monkey(on(floor),hold(nil),$(1),$(2)). object($(3),on(floor),$(1),$(2)). history($(4))]	[[]. [monkey(on(floor),hold(nil),$(1),$(2)). history($(4))]. [monkey(on($(3)),hold(nil),$(1),$(2)). history([climb($(3))\|$(4)])]]
[monkey(on($(1)),hold($(2)),$(3),$(4)). history($(5))]	[[$(1)\=floor], [monkey(on($(1)),hold($(2)),$(3),$(4)). history($(5))]. [monkey(on(floor),hold($(2)),$(3),$(4)). history([down($(1))|$(5)])]]

Table 2. Example of initial state

[monkey(on(chair),hold(nil),5,7)|$(1)]	$(1)	$(2)
[object(chair,on(floor),5,7)|$(1)]	$(1)	$(2)
[object(banana,on(ceiling),2,2)|$(1)]	$(1)	$(2)
[object(ladder,on(floor),9,5)|$(1)]	$(1)	$(2)
[history([])|$(1)]	$(1)	$(2)

The following GHC clauses use RBU commands to produce new states from a given state using production rules:

```
production(state(Eval,State),O1,O2,C1,C3) :- true |
        C1 = [ujs(rule,[1],State,[1],[4,2],X)|C2],
        X = [_|_],
        loop(X,Eval,State,O1,O2,C2,C3).

loop([finish],Eval,State,O1,O2,C1,C2) :- true |
        O1 = O2,
        C1 = C2.
loop([[Rule,[Cond,Del,Add]]|L],Eval,State,O1,O3,C1,C3) :- Rule = [] |
        check(Cond,true,X),
        update(X,Eval,State,Del,Add,O1,O2,C1,C2),
        L = [_|_],
        loop(L,Eval,State,O2,O3,C2,C3).
loop([[Rule,[Cond,Del,Add]]|L],Eval,State,O1,O3,C1,C4) :- Rule \= [] |
        C1 = [urs(State,[1=Rule,3=[Cond,Del,Add]],[2,3],X)|C2],
        X = [_|_],
        loop(X,Eval,State,O1,O2,C2,C3),
        L = [_|_],
        loop(L,Eval,State,O2,O3,C3,C4).
```

The *ujs* command in the *production* predicate is used for pattern matching by unifying matching parts of rules with a state. Unifying a rule with an element of a state causes the top of the matching part to be removed. The *ujs* command returns modified rules as a stream bound to *X*.

X=[_|_] indicates a request for send a result from RBU. Modified rules are processed by the *loop* predicate, which invokes the *urs* command to unify matching parts of modified rules with the state. Once the matching part of a rule becomes empty, conditions are checked and the state is updated using information stored in the execution part of the rule.

The GHC program for leaf nodes must be changed to use RBU commands. The third argument is used to transfer RBU commands.

```
leaf([state(E,S)|I],O1,Cmd1) :- true |
        production(state(E,S),O1,O2,Cmd1,Cmd2),
        O2 = [request|O3],
        leaf(I,O3,Cmd2).
```

States and rules are best retrieved and updated in parallel. Each time the rbu predicate is invoked, a new RBU process is created which is able to independently retrieve and update term relations. An *rbu* predicate is prepared for each leaf node.

```
four_leaf_tree :- true |
        root(B1u,B1d),
        branch(B1d,B2u,B3u,[],[],B1u,B2d,B3d),
        branch(B2d,L1u,L2u,[],[],B2u,L1d,L2d),
        branch(B3d,L3u,L4u,[],[],B3u,L3d,L4d),
        leaf(L1d,L1u,C1), rbu(C1),
        leaf(L2d,L2u,C2), rbu(C2),
        leaf(L3d,L3u,C3), rbu(C3),
        leaf(L4d,L4u,C4), rbu(C4).
```

Figure 3 illustrates state transitions in RBU. Parallel updates do not cause inconsistency in the knowledge base, because a new term relation is generated for each update. Artifice is needed to minimize copying between transitive states. Most state elements are unchanged by a state transition, and must be shared among states.

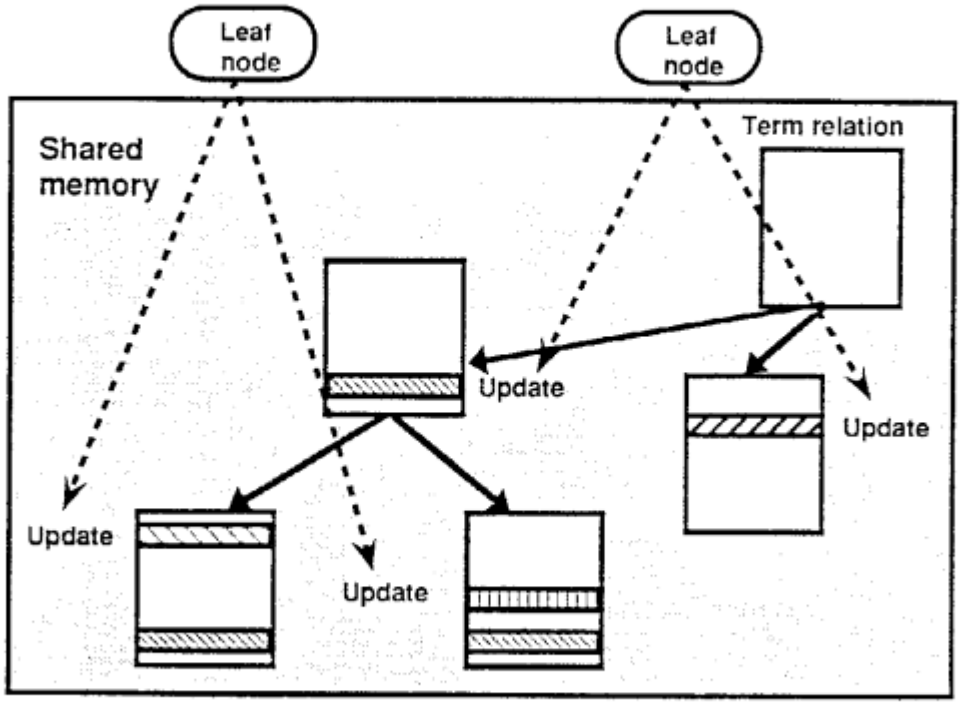

Figure 3. State transitions in RBU

3.3 Communication between GHC and RBU

The RBU system is independent of the GHC system because they have different execution mechanisms. In actual use, they must be tightly coupled. The two systems should share an atom table to minimize the translation overhead and storage space. Predicate names and function symbols are stored in the atom table and referenced by both systems using table entry addresses. It is redundant both to translate reference addresses into character strings for communicating, and to prepare storage for two atom tables. It is much more efficient to use the reference addresses themselves for communication.

Buffers are needed to transfer retrieved results, because retrieval and GHC processes are executed concurrently. A demand-driven protocol is used to transfer results. A buffer is used for each output stream (variable) in the RBU system. On demand, GHC can request the RBU system to transfer a

knowledge element (tuple) from the buffer. Figure 4 shows this communications protocol.

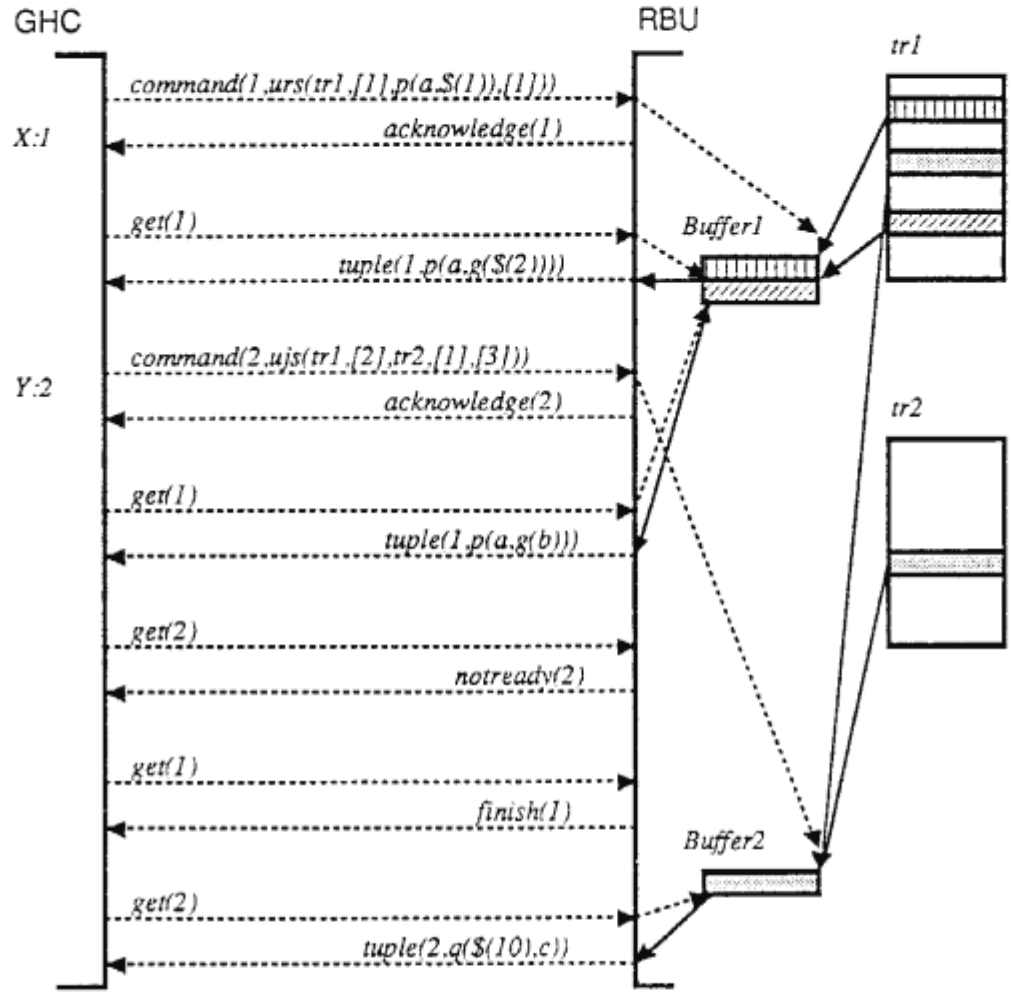

Figure 4. Example of GHC and RBU communications

Each command sentence has an identifier corresponding to the buffer. In Figure 4, variable X in the command sentence $urs(tr1,[1],p(a,\$(1)),[1],X)$ corresponds to a buffer identified by the number 1, and the variable Y in $ujs(tr1,[2],tr2,[1],[3],Y)$ corresponds to buffer number 2.

RBU returns an acknowledgment when it accepts a command sentence and prepares a buffer for an output stream. GHC sends *get* commands with stream identifiers on demand. RBU returns a tuple for each *get* command from a corresponding buffer. If results are not yet ready for the *get*

command, a status indicating "not ready" is returned with the identifier. Once all results have been sent to GHC, a status indicating "finish" is returned. *Get* commands have identifiers and do not have to be sequences in RBU commands.

4. RBU prototype

We implemented an RBU prototype to study this production system. As yet it can only sequentially retrieve and update knowledge bases, but is a first step to building a parallel knowledge handling system. A parallel version of RBU will be implemented soon. Data structures are assumed to be manipulated in parallel. A term relation is an individual unit that provides a lock for updating. A chunk of storage is treated as a page to manage storage allocations. Storage management is simplified by restricting page use, i.e., a page is never used by more than two relations.

The prototype is also being used to evaluate indexing for term relations. Our indexing uses data structures called hashing along with a trie structure, and is suitable for frequent updating and retrieval of terms by unification with a condition using backtracking.

4.1 Indexing using hashing and a trie structure

Different approaches have been proposed to improve retrieval speed. One is dedicated hardware, e.g., a unification engine [Morita 86, Yokota 86]. [Ohmori 87] proposes a hash vector for indexing clauses. Superimposed code words for terms and a dedicated engine for manipulating the words has also been proposed by [Wada 87]. We use indexing that retrieves a set of terms using unification and backtracking.

Retrieved terms resemble each other somewhat because they are unifiable with the search condition. For efficient backtracking, these terms need to be located near an index. The trie is a type of structure that shares identical elements and meets this requirement [Knuth 73]. Figure 5 gives an example of a trie for a set of terms.

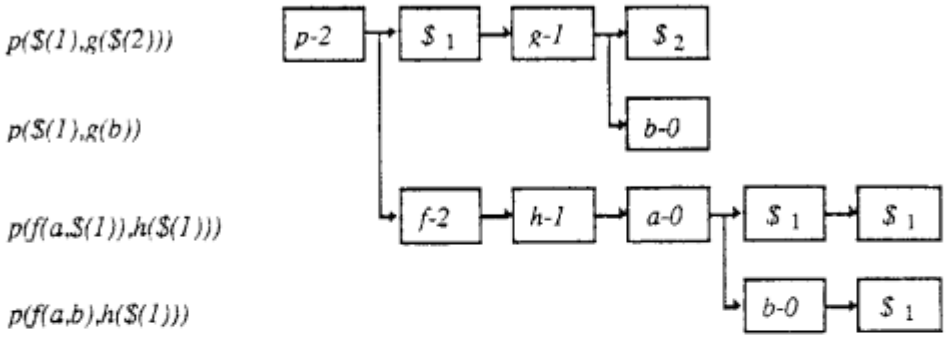

Figure 5. Trie structure for a set of terms

The costs of unification are proportional to the count of comparisons between components of the object terms. A trie reduces the number of comparisons when unification is performed. For example, consider the situation of searching the set of terms in Figure 5 for terms unifiable with the condition $p(f(a,b),h(c))$. Using the trie structure, the component p is compared only once, whereas four comparisons are necessary if the trie structure is not used. The number of comparisons needed to search for all terms unifiable with the condition is 10 using the trie structure as opposed to 18 not using the trie structure.

A hash table is used before the trie structure when storing many types of terms in a term relation (Figure 6). The first component of terms are used as hash entries. The trie structure is combined with hash collision resolution.

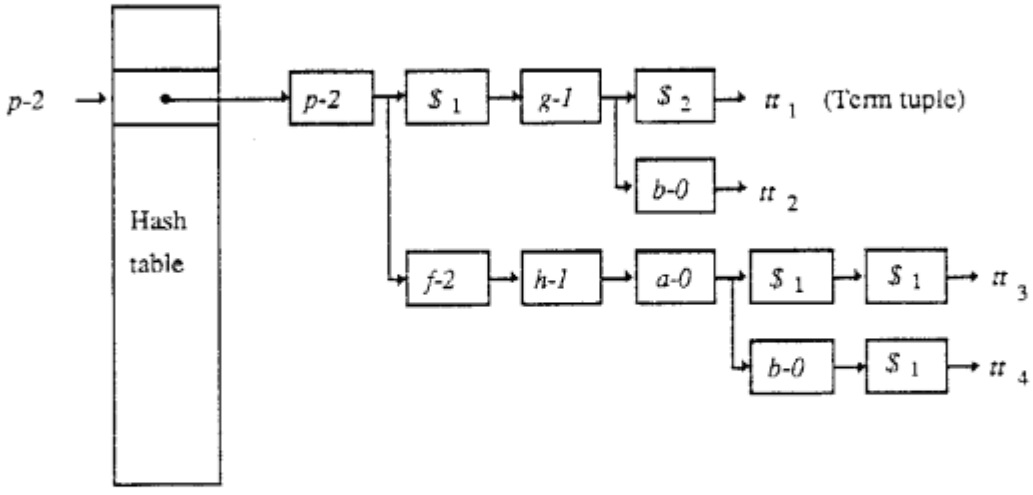

Figure 6. Tuple index with hashing and trie structure

4.2 Search and updating speed

This section compares the search and updating speeds of the RBU prototype with those of the Quintus-Prolog interpreter. Prolog compilers do not support *assert* and *retrace* predicates, i.e., they cannot update knowledge bases, and therefore the compiler has not been examined.

Figure 7 compares the search speeds of the prolog interpreter and *urs* with and without indexing. The *urs* without indexing is about four time slower then the prolog clause search. This search time increases with tuple count in both prolog and urs without indexing. However, the search speed of urs with indexing is scarcely changes regardless of the number of tuples. For 1000 tuples, it is about four times *faster* than the corresponding prolog clause search. This speed up is a result of the indexing.

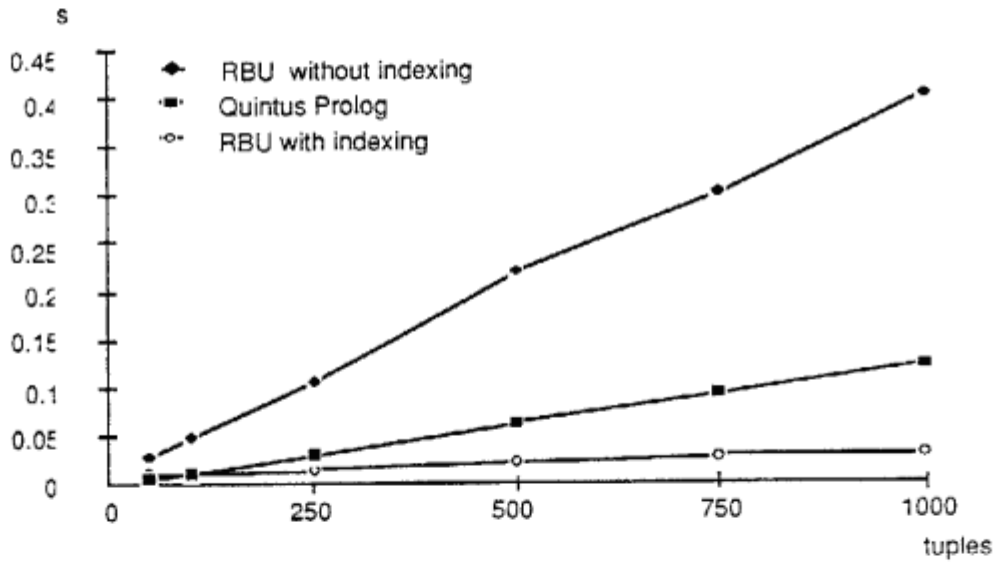

Figure 7. Comparison of search speeds

Figure 8 compares the tuple insertion speeds of the two systems. Tuple insertion using RBU takes only about one sixth the time of a Prolog *consult* operation. The overhead for making an index for a term relation is about one tenth of the insertion time.

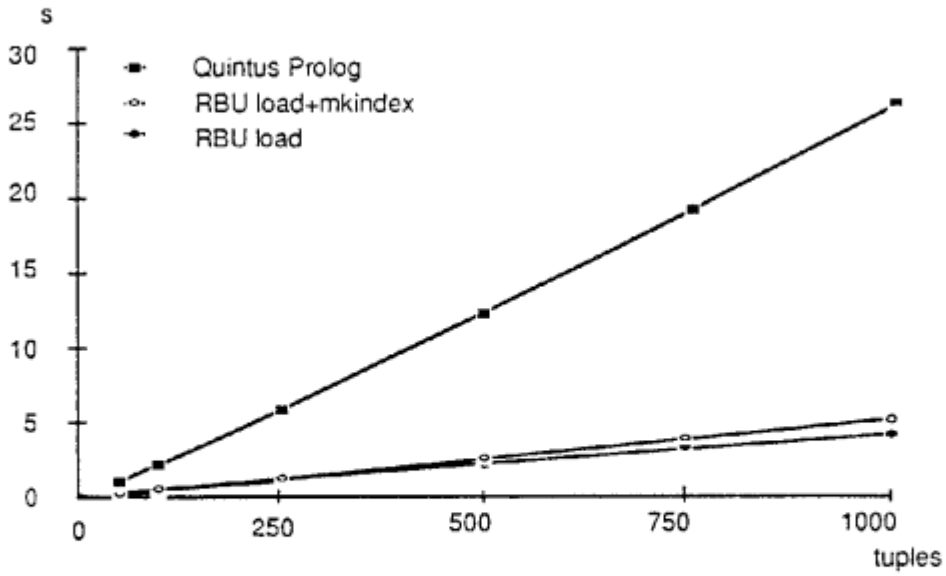

Figure 8. Insert speed comparison

The unification algorithm for a trie structure and a more detailed evaluation of the RBU prototype will be given in another paper.

5. Concluding remarks

We have presented one way of implementing a parallel problem solving system using GHC and RBU. The Better First Search localizes communications among parallel processes. This configuration is suitable for the Parallel Inference Machine (PIM) [Goto 87] being developed in the FGCS project. A number of processors and shared storage compose a cluster in this machine, making it important to localize processors communications. We plan to locate each leaf process in a processor (Figure 9).

RBU enables GHC to process knowledge bases. The combination is useful both in parallel production systems and other knowledge processing systems. Indexing method using hashing and a trie structure effectively speeds up retrieval and keeps overhead low in RBU updating. We plan to implement a parallel version of RBU on the PIM.

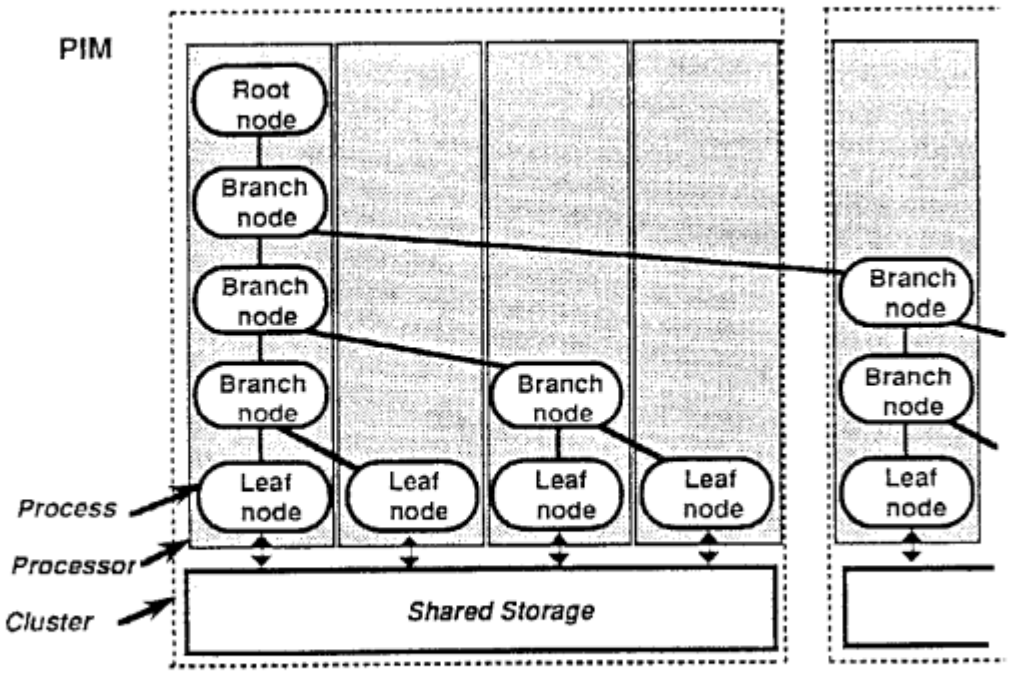

Figure 9. Implementation on the PIM

Acknowledgments

We thank Dr. Hidenori Itoh, manager of the third laboratory of ICOT, and the laboratory staff for their useful discussion and Mr. Hiromu Hayashi, manager of the artificial intelligence laboratory, Fujitsu Laboratories Ltd., for his helpful suggestions. We also thank Mr. Mark Feldman of the artificial intelligence laboratory for his helpful advice on the refinement of the paper.

References

[Barr 81] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*,1, William Kaufmann, Inc. 1981.

[Fuchi 84] K. Fuchi, "Revisiting Original Philosophy of Fifth Generation Computer Systems Project," *Proc. of the International Conference on Fifth Generation Computer Systems*, 1984.

[Goto 87] A. Goto, "Parallel Inference Machine Research in FGCS Project," *Proc. of the US-Japan AI Symposium 87*, pp. 21-36, 1987.

[Gupta 87] A. Gupta, *Parallelism in Production Systems*, Morgan Kaufmann Publishers, Inc.,1987.

[Itoh 87] H. Itoh, T. Takewaki, and H. Yokota, "Knowledge Base Machine Based on Parallel Kernel Language," *Proc. of 5th International Workshop on Database Machines*, pp. 15-28, 1987

[Knuth 73] D. E. Knuth, *The Art of Computer Programming*, 3, Sorting and Searching, Addison-Wesley, 1973.

[Morita 86] Y. Morita, H. Yokota, K. Nishida and H. Itoh, "Retrieval-By-Unification Operation on a Relational Knowledge Base," *Proc. of 12th International Conference on VLDB*, pp. 52-59, 1986.

[Ohmori 87] T. Ohmori and H. Tanaka, "An Algebraic Deductive Database Managing a Mass of Rule Clauses", *Proc. of 5th International Workshop on Database Machines*, pp. 291-304, 1987.

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

[Wada 87] M. Wada, Y. Morita, H. Yamazaki, S. Yamashita, N. Miyazaki, and H. Itoh, "A Superimposed Code Scheme for Deductive Databases", *Proc. of 5th International Workshop on Database Machines*, pp. 569-582, 1987.

[Yokota 86] H. Yokota and H. Itoh, "A Model and Architecture for a Relational Knowledge Base," *Proc. of the 13th International Symposium on Computer Architecture*, pp. 2-9, Tokyo, 1986.