ICOT Technical Report: TR-255

TR-255

Parallel Control Techniques for Retrieval Processes in the Parallel Logic Programming Language and Their Evaluation

by
T. Takewaki and H. Itoh

May, 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

Parallel Control Techniques for Retrieval Processes in the Parallel Logic Programming Language and Their Evaluation

Toshiaki Takewaki and Hidenori Itoh

Institute for New Generation Computer Technology,
Mita Kokusai Building, 21F,
1-4-28 Mita, Minato-ku Tokyo 108 Japan

Abstract

This paper proposes parallel control techniques for retrieval processes in the parallel logic programming language, GHC (guarded Horn clauses), which is used as the basis of KL1 (kernel language version 1) for the parallel inference machine under development at ICOT. It also proposes parallel processing of retrieval operations by division of handled data.

The parallel control strategies are developed from the following parameters: number of available retrieval processors, type of commands, and size of the data to be handled. In this paper, the evaluation of the techniques is a correction between data size of processing and parallelism.

Keywords: *Knowledge Base System, Guarded Horn Clauses, Parallel Control*

1. Introduction

Knowledge information processing systems have been proposed from the viewpoint of logic programming, such as an inference machine for efficient

inference processing and a knowledge base system for efficient retrieval processing.

Advanced knowledge representation languages are developed in logic programming to implement knowledge information processing systems. The advantages of developing these systems in a logic programming environment are development and processing efficiency, and expansion of facilities.

This paper shows that the basic functions of the knowledge base system can be written in the logic programming language, GHC (guarded Horn clauses) [Ueda 85], which is used as the basis of KL1 (kernel language version 1) for parallel inference machines such as the Multi-PSI and PIM.

GHC is a simple, powerful and efficient parallel logic programming language. A GHC program is a finite set of guarded Horn clauses in the following form:

```
H :- G1, ... , Gm | B1, ... , Bn. (m>0, n>0)
```

where H, Gi, and Bi are atomic formulas defined in the usual way. H, Gi and Bi are the *clause head*, *guard goal* and *body goal*. The notation | is the *commitment operator*. The part of a clause preceding | is called the *guard*, and the part succeeding | is the *body*.

This paper uses a subset of GHC, called Flat GHC. Flat GHC is allowed to have only system predicates in guards of clauses.

2. Knowledge Base System

This section describes the knowledge base system model used in this paper. As a first step to the knowledge base system, a deductive database system model (Figure 1) is proposed which stores and manages a finite set of Horn clauses as

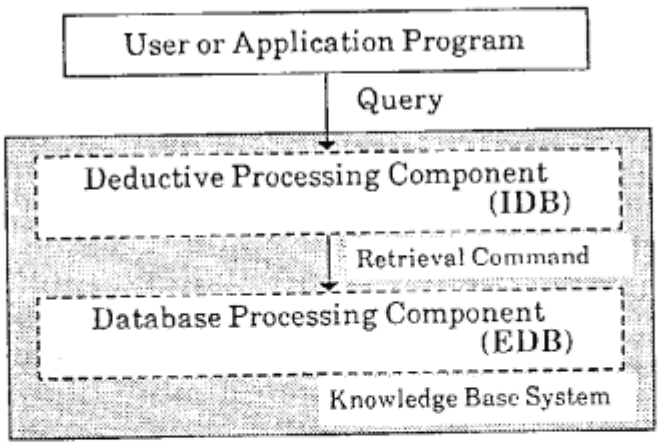


Figure 1 Configuration of Knowledge Base Model

knowledge. This model consists of a *deductive processing component* and a *database processing component.*

The first component manages an intensional database (a set of rules to derive facts or define the viewpoint of facts), and accepts Horn clause queries. These queries are compiled into equivalent programs that include a set of relational queries, and are translated into retrieval commands in search in the EDB. The second component manages and retrieves items from an extensional database (a set of facts: EDB) using their retrieval commands. The IDB and EDB are assumed to be accessible in common by user or application programs, and the size of the EDB is assumed to be very large.

The deductive processing component translates Horn clause queries to equivalent retrieval commands. An important part of this component is

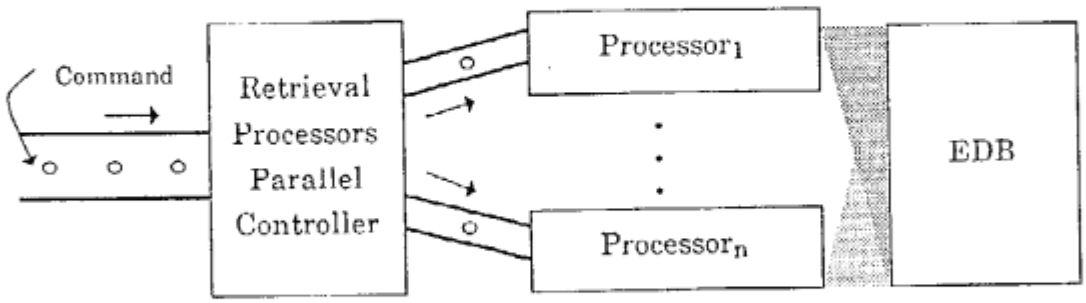


Figure 2 Configuration of Database Processing Component

compiling (optimizing) Horn clause queries to equivalent programs. This is called "knowledge compile". This paper introduces the concept of partial evaluation for its optimization. This method has several advantages: (1) it reduces complex recursions to simple recursions and then handles them by iteration, (2) it converts non-recursive queries to non-iterative programs of relational operations, and (3) evaluation of non-recursive intermediate expressions is not necessary. See [Miyazaki 86b] for details of the method.

The database processing component consists of multiple dedicated retrieval processors, the processors' parallel controllers (including the query analyzer) and EDB storage (shared database) as shown in Figure 2. Retrieval processors perform the relational operations such as sort, join and select operations on the sets of data. To search and handle the EDB efficiently, the database processing component is equipped with multiple retrieval processors controlled in parallel. The retrieval processor parallel controller receives a relational retrieval command from the deductive processing component, analyzes it, and produces a strategy for efficient retrieval from EDB.

The following section proposes parallel control techniques for retrieval processes.

3. Parallel Control of Processes

Parallel control for retrieval processes is influenced by the following parameters.

(a) Number of available retrieval processors,

(b) Type of commands,

(c) Size of data to be handled and searched.

This section describes parallel control techniques by number of processors. The next section describes parallelism from the type of commands and the size of data to be handled.

In the database processing component, three parallel control strategies for retrieval processes are considered. The symbols *n* and *i* indicate the total number of retrieval processors and the number of available retrieval processors when the retrieval command is received from the deductive processing component.

(1) Data non-division method (method 1)

All commands from the deductive processing component are executed by a single processor, without data division. The controller receives a command, looks for an available processor, and allocates the received command to its processor.

(2) Data dynamic division method (method 2)

The controller receives a command, and looks for all available processors. If there are *i* available processors, the controller converts the command to *i* (variable number) sub-commands, and allocates it to *i* processors.

(3) Data static division method (method 3)

This method is a mixture of method 1 and 2. The controller converts the command to *n* (fixed number) sub-commands. The allocation of sub-commands is much the same as method 1.

The parallel controller of method 1 manages only free status for retrieval processors; the parallel controller does not need to know the busy status for retrieval processors. Therefore, the parallel controller can execute by demand-driven command from retrieval processors. Figure 3 shows part of the program for method 1. The predicate 'RPscheduler' checks for retrieval processors, and assigns job to free retrieval processors. The predicate selectRP selects free retrieval processor for execution of command, command is assigned by

```
'RPscheduler'([C|T],StreamSt) :- true | inspect(StreamSt,C,_,List,NewSt),
        selectRP(List), 'RPscheduler'(T,NewSt).
'RPscheduler'([]   ,StreamSt) :- true | closeStream(StreamSt).


inspect([stream(N,St)|Rest],Command,H, List,NewSt) :- true | NewSt=[stream(N,New)|NN],
        checking_state(N,St,Ack,H), accept(OK, ST, Command,New),
        List=[(Ack,OK)|ListR], inspect(Rest,Command,H,ListR,NN).
inspect([]                 ,_      ,H, List,NewSt) :- true | NewSt=[], List=[].


checking_state(N,S    ,Ack,halt) :- true | Ack=busy.
checking_state(N,[S|R],Ack,Halt) :- true | Ack=free, Halt=halt.

accept(ok, [G|R],Exec,NN) :- true | G=Exec, NN=R.
accept(ng, R    ,_   ,NN) :- true | R=NN.

selectRP([(free,A)|Rest]) :- true | A=ok, reject(Rest).
selectRP([(busy,A)|Rest]) :- true | A=ng, selectRP(Rest).
```

Figure 3 Part of Program for Method 1

```
'RPscheduler'([C|T],StreamSt) :- true | inspect(StreamSt,NewStreamSt,Ans),
      checking_free(Ans,FreeList), divide(FreeList,[C|T],NewStreamSt).
'RPscheduler'([]   ,StreamSt) :- true | closeStream(StreamSt).

divide([] ,C    ,St) :- true | 'RPscheduler'(C,St).
divide(List,[C|T],St) :- List\=[] | sendRP(List,C,St,NewSt), RPscheduler'(T,NewSt).

inspect([]                 ,New,Res) :- true | New=[], Res=[].
inspect([stream(N,St1)|Rest],New,Res) :- true | New=[stream(N,St2)|NewR],
       Res=[(N,State)|ResR], St1=[ins(State)|St2], inspect(Rest,NewR,ResR).


'RP'(N,[term  |Command]) :- true | Command=[].
'RP'(N,[ins(C)|Command]) :- true | C=free, 'RP'(N,Command).
'RP'(N,[cmd(C)|Command]) :- true | sendToRP(N,C,Response),
      response(Response,Command,Next), 'RP'(N,Next).

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

Figure 4 Part of Program for Method 2

the predicate accept. The predicate checking_state for the processor's controller waits for judgment of free status for retrieval processor until the cons-list ([_|_]) is instantiated at the second argument (first clause). If the processor's controller finds a free status for retrieval processor, it can halt an inspection of other processors' statuses. It instantiates the atom "halt" in the variable "Halt" (second clause).

Figure 4 shows part of the program for method 2. The predicate 'RPscheduler' checks status for all retrieval processors, and assigns to free retrieval processors. The predicate closeStream informs every retrieval processors that end of command. The predicate inspect inquires at every retrieval processors for its status. The predicate checking_free searches for number of free retrieval processors, and selects free retrieval processors. The predicate sendRP divides job for each free retrieval processors through division of data and sends out commands to free retrieval processors. The predicate 'RP' handles commands from other components, and manages the status for retrieval processors. Arguments of the predicate 'RP' indicate the number of processors and the stream of commands.

The predicate sendToRP sends a command to a retrieval processor. This processor instantiates the third argument to the atom "end" when a process comes to an end. If a processor receives the command ins(C) for an inspection of status when a processor is busy, then the atom "busy" is returned by the predicate "response", otherwise, the atom "free" is returned by the predicate 'RP'.

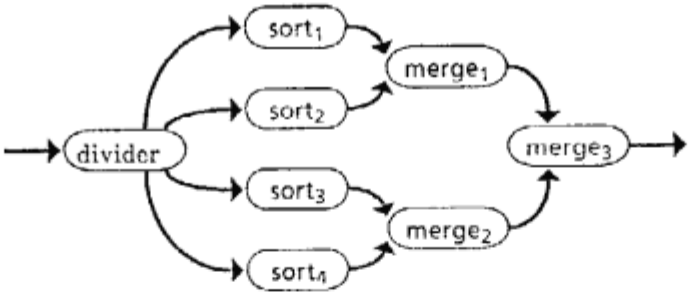
4. Grains of Handled Data and Parallelism

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

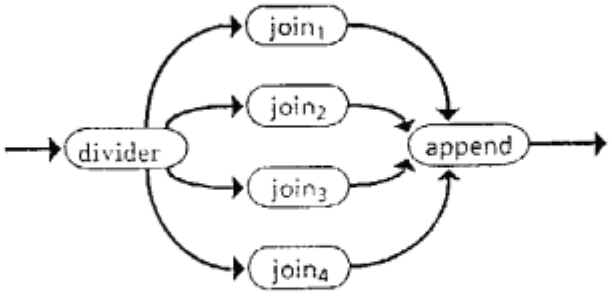
The sort and join operations are very important parts of the relational retrieval processing.

The data division sort and data division join operations are converted into sub-commands and executed. For example, Figure 6 shows the process (oval) and the flow of data (arrow) performed by the data 4 division. The symbol R indicates the size of the data to be handled by one command.

In the sort operation S(R), every sort (sort1-4) corresponds to a sort process in order to handle R/4 data, the first level merge (merge1-2) corresponds to the merge process for pairs of R/4 data streams after it has been sorted, and the last level merge (merge3) corresponds to the merge process for pairs of R/2 data streams. The data N division sort operation needs N sort processes, and $\log_2 N$ level and N (or N-1) merge processes. Total execution cycles for all merge processes are proportional to the length of handled data, and are not influenced by



(a) Sort Operation



(b) Join Operation

Figure 5 Data Division Process of Retrieval Operations

number of divisions. In short, the parts of sort processes are fast when the number of divisions is large, but the communication cost for merge processes has been piling up recently.

In the join operation (R1 |><| R2), every join (join1-4) corresponds to the join process in order to handle R1/4 |><| R2 (where R1 is greater than R2), and an append corresponds to the append process for data streams after it has been joined. The data N division join operation needs N join processes and one append process. To remove the append process, a differential list is used that can partly hold a value. (This is a property of logical variables.) In short, the parts of join 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 from left to right as shown in Figure 5, and parallel processing of every process is possible when every process receives data.

5. Evaluation of Three Strategies and Division Operations

First, three strategies are evaluated to control retrieval processes. Then, parallelism is evaluated by division operations.

Each program is evaluated by the Flat GHC simulator on DEC10-Prolog. This simulator can analyze the number of reducible goals and the number of total goals for each scheduling cycle. It is assumed that a parallel inference machine consists of a large number of processing elements (PEs), and that as many PEs as required by the user are used. A PE and a retrieval processor should not be regarded as the same thing.

Figure 6 shows the reductions and cycles of transactions for 10 commands using each strategy. One command needs 100 reductions and 100 cycles to be

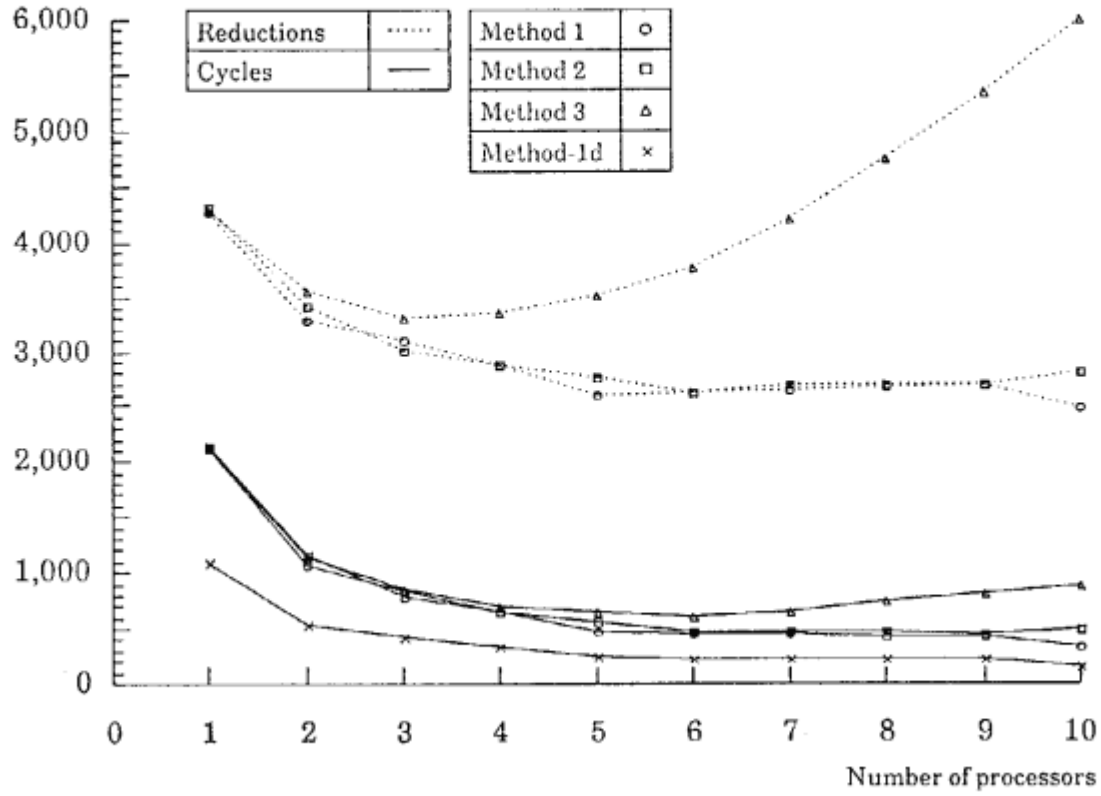


Figure 6 Reductions and Cycles for Each Strategy

resolved. The horizontal axis indicates the number of retrieval processors. All transactions use *1000* reductions and *1000/ [number of retrieval processors]* cycles that are called ideal reductions and ideal cycles. The difference between read reductions and ideal reductions is used by the parallel controller for retrieval processors.

The program of method-1d in Figure 6 is implemented by a parallel controller that is demand-driven by the retrieval processor and receipt of a command from deductive processing components. Otherwise, a parallel controller that periodically investigates the status of retrieval processors is implemented. The parallel controller of method-1d is driven only when required. It requires less labor than the parallel controllers of other methods.

Figure 7 indicates the number of reducible goals for each scheduling cycle.

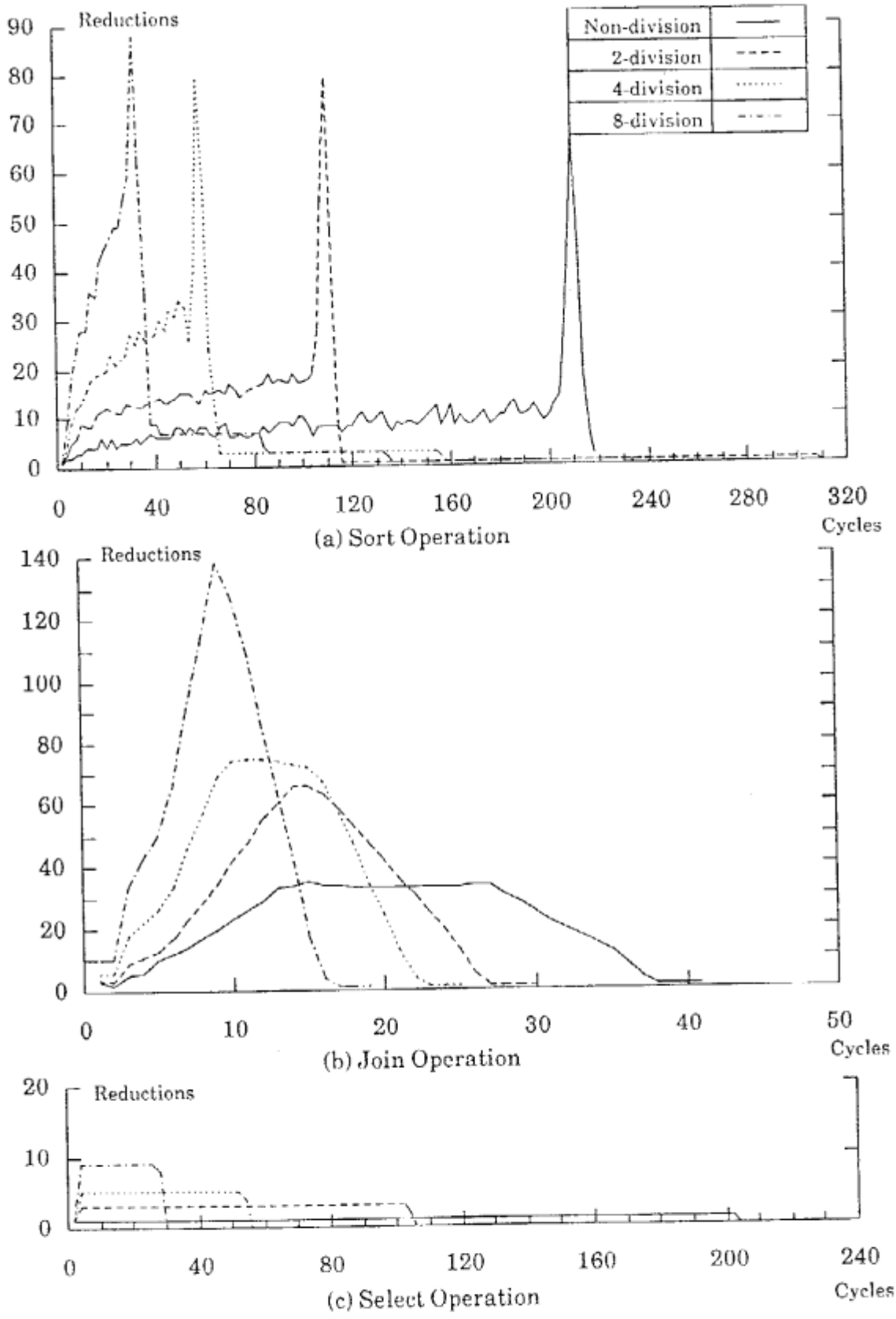


Figure 7 Reductions of Each Cycle for Retrieval Command

Table 1 Result of Division Process for Retrieval Commands

(a) Sort Operation

Number of divisions	Reductions	Suspensions	Cycles	Execution time (sec)
1	2038	1580	218	13.663
2	2050	1349	308	7.987
4	2017	1075	257	4.658
8	2027	893	236	2.960

(b) Join Operetion

Number of divisions	Reductions	Suspensions	Cycles	Execution time (sec)
1	854	253	41	1.120
2	876	255	30	1.043
4	975	260	25	1.108
8	1005	268	19	1.065

(c) Select Operation

Number of divisions	Reductions	Suspensions	Cycles	Execution time (sec)
1	202	1	202	0.345
2	304	3	103	0.326
4	256	5	53	0.269
8	235	9	28	0.252

Table 1 compares the reductions, suspensions, cycles and execution time of various operations. The sort and select operations handle data of 100 length. The join operation handles data of 25 length and 20 length. The parallelism of division operations was as effective as expected.

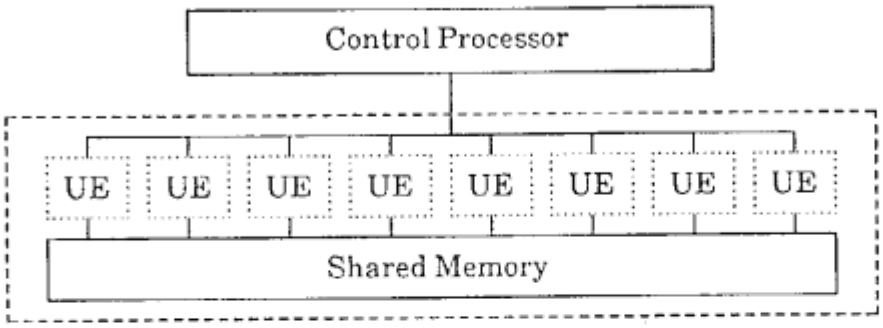
(a) is the result of the sort operation. The execution cycles of merge processes need the length of handled data when handled data is divided. This division sort operation uses more execution cycles than the non-division sort operation if each reduction is executed at the same cost. Reductions of the two operations are much alike, and execution of the sort operation is faster than that of the non-division sort operation. The execution of the division sort operation on parallel machines is expected to be fast.

(b) is the result of the join operation. Handled data are not sorted. The join operations generates a pair of data from two streams, and executes it in parallel. The join operation initially has parallelism. Parallelism is increased by dividing handled data. A data divider in which data are assigned to all join process and reduced until empty is used.

(c) is the result of the select operation. The select operations require less time and labor than the sort or join operations, but they do require execution cycles of the data length. Therefore, parallelism is increased by dividing handled data. A data divider in which data are assigned to all select processes and reduced until empty is used.

6. Conclusion

This paper described the possibility of developing the deductive database system possessing a set of rules and facts using the parallel logic programming language GHC. Three methods were suggested as parallel control strategies for the retrieval process, and evaluated with simple data. Parallelism and execution time were evaluated as data division representative of retrieval commands. Thus, the following advantages are expected from the deductive database system.



UE : Unification Engine

Figure 8 Hardware Simulator for Parallel Control

(a) The database (data and rules), system control program, database management program, and user or application program can be handled with the parallel logic programming in the same way;

(b) Parallel control program can be developed easily by GHC functions, such as differential list, and-parallel mechanism, demand-driven computations, and data-driven computation.

We are now developing a deductive database system on parallel inference machines such as the Multi-PSI and PIM, which is also written in GHC. The parallel inference machine controls queries sent to the common database in the deductive database machine. In this system, we will investigate the relationship between the degree of parallelism and granularity of data to be processed.

The parallel controller of the database processing component and the knowledge compiler of deductive processing component are some of the most important parts in knowledge base system. This paper emphasized the database processing component. The horn clause transformation method used as the knowledge compiler in the deductive processing component was not described here. See [Itoh 86] for details.

We are also developing a hardware simulator for parallel control. This simulator consists of a control processor, eight dedicated unification engines and a shared memory as shown in Figure 8. We are considering an experiment for this simulator based on this result of evaluation by software simulation in GHC.

Building on this research, we will seek to develop many facilities of the knowledge base system in a parallel logic programming language.

Acknowledgments

We wish to express our thanks to Dr. Kazuhiro Fuchi, Director of the ICOT Research Center, who provided us with the opportunity to pursue this research in the FGCS Project at ICOT. We would also like to thank the members of the First Research Laboratory and Third Research Laboratory at ICOT for their valuable comments.

References

[Ueda 86] Ueda, K., "Guarded Horn Clauses", Logic Programming '85, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, pp. 168-179, 1986

[Itoh 86] Itoh, H., Takewaki, T. et al., "A Deductive Database System Written in Guarded Horn Clauses", ICOT Technical Report TR-214, 1986

[Itoh 87] Itoh, H., Abe, M. et al., "Parallel Control Techniques for Dedicated Relational Database Engines", Proc. of Data Engineering '87, 1987

[Miyazaki 86a] Miyazaki, N., Yokota, H. et al. "KBMS PHI(2)", Proc. of the 32nd National Conference on Information Processing Society of Japan, 1986 (in Japanese)

[Miyazaki 86b] Miyazaki, N., Yokota, H. et al. "Compiling Horn Clause Queries in Deductive Databases: A Horn Clause Transformation Approach", ICOT Technical Report TR-183, 1986