TR-615

Knowledge-Based Parallel Inference System

by
H. Kitakami & H. Yokota (Fujitsu)

February, 1991

© 1991, ICOT

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

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Knowledge-Based Parallel Inference System

Hajime Kitakami and Haruo Yokota

Fujitsu Limited

1015 Kamikodanaka, Nakahara-Ku, Kawasaki 211, Japan

Abstract

We need a knowledge-based parallel inference system to support problem solving by those using computer systems. Today's systems tend to produce combinatorial growth and large-scale knowledge bases.

This paper describes parallel programming using metaprogramming based on the parallel logic programming language guarded Horn clauses(GHC). Since GHC does not include OR parallel functions for knowledge retrieval systems, we selected a simple way than this perpetual process for representing knowledge bases. Data is represented in the knowledge base using $-variables, which are specific constants, to distinguish them from any other GHC variable in GHC's interpretation system. Since we found no advantage in implementing $-variable processing in GHC, we wrote it in C. The system includes a knowledge retrieval system which supports a large-scale knowledge base and an interface between it and GHC's interpretation system.

We present the performance of the parallel constraint logic programming system, including the kernel system, used for knowledge-base-oriented parallel inference.

Keywords

Knowledge base, parallel inference, GHC, OR-parallel, AND-parallel

1. Introduction

We need a knowledge-based parallel inference system to support problem solving by users of computer systems [Fuchi '78]. Knowledge information systems using this concept include medical diagnosis, electrodynamic system fault diagnosis, electric power system planning, production and transportation scheduling, and physical phenomena simulation.

New knowledge information systems require faster processing to cope with combinatorial growth in inference and large-scale knowledge base access. Detailed examples include genome analysis [Lander '88], transportation scheduling, LSI design, and particle simulation [Sato '90]. We believe that the parallel inference machine PIM [Uchida '88, Hattori '89] effectively speeds up execution times for these symbolic processing applications.

This paper presents a parallel inference system including a knowledge base mechanism. We implemented the system using metaprogramming [Bowen & Kowalski '81, Kitakami '84] using GHC on a SEQUENT parallel computer.

The basic actions of the system are OR-parallel knowledge base search and AND-parallel knowledge inference. However, prior research on OR- and AND-parallel processing has been separate. For example, PRISM [Kasif '83], and P-Prolog[Yang '87] were researched in OR-parallel processing. GHC [Ueda '85, Ozawa '90], Parlog [Clark '84] and Concurrent Prolog [Shapiro '84] were researched in AND-parallel processing. Since the parallel inference system needs both OR- and AND-parallel processing, existing programming languages are inadequate. One research paper [Takeuchi '87] discusses translating OR- into AND-parallel processing, but the method is incomplete.

Since the kernel language of the parallel inference machine is GHC, we implement knowledge-base-oriented parallel inference by adding AND-parallel GHC to OR-parallel knowledge base search. We propose a way to separate knowledge base variables from the restrictions of single assignment of GHC variables. We developed a simple, extendable metaprogramming technique using GHC. With it, cut symbol implementation becomes easy. Knowledge base variable management is implemented in C instead of GHC to minimize memory use. We also describe a way to interface GHC and knowledge retrieval system processes [Yokota '89] to support large-scale knowledge base access using shared memory.

Finally, we evaluate system performance with a simple constraint programming system. It extends the parallel inference system and adds a simple constraint solver using Gaussian elimination. Our example is of electrostatic fields [Heintze '87]. In general, constraint logic programming systems are implemented by extracting and computing constraint equations from the knowledge bases [Lassez '87]. Extraction is implemented by extending the parallel inference system. Computation is implemented by AND-parallel computing for a Gaussian elimination algorithm. GHC runs about 9 times faster with 9 processors than with one processor. We also assigned the knowledge retrieval system to 1 or 6 processors.

The parallel inference system was developed by researchers at Fujitsu Laboratories Ltd. sponsored by the Institute for New Generation Computer Technology.

2. Inference Using Metaprogramming

Our parallel inference system uses metaprogramming techniques to apply Prolog logic to GHC parallelism. Most programming on

metalevel language system is object-level; metaprogramming interprets user programs. Since metaprogramming is an interpreter, the language involved can easily be extended without revising the original language system.

For example, a knowledge information system has the knowledge base (Fig. 1) as an object-level program, and has the inference rules (Fig. 2) as a metalevel program. Fig.1 shows a knowledge base defining ancestry. The first two lines show the rules of the ancestor concept. The term "ancestor(X,Y)" shows that "X" is the ancestor of "Y". The ancestor program also defines the term "parent(X,Y)" which means that "X" is the father or mother of "Y".

The program in Fig. 2 is implemented by metaprogramming and represents the simplest Prolog interpreter. This program finds one solution on the search tree. If a user defines a problem "?-solve(Goals).", the second rule in Fig. 2 resolves the goal sequence "Goals". The goal is analyzed by searching the knowledge base and resolving retrieved knowledge with the third rule in Fig.2. Analysis continues until the goal is evaluated as true or false. A goal is true if it agrees with by the first rule in Fig. 2. If it is not retrieved by the knowledge base search "clause(P, Q)" of the third rule, the system backtracks on the search tree. If backtracking cannot be done, the goal is false.

Since metaprogramming does not distinguish between the knowledge base as input data and inference processing as procedure, we can use the input data as Prolog variables.

The metaprogramming in Fig. 2 gives one solution on serial computers. The process is searching a path on the search tree, which is difficult to do in parallel. If we execute the problem "?-solve(ancestor(X,Y)).", metaprogramming outputs "X=f1, Y=f2". It is

difficult to get ideal parallel performance in AND-parallel. The system can be extended to search all possible paths in parallel. Exhaustive searches have many applications such as constraint logic programming, knowledge base management, and so on. Knowledge base management can check consistency and eliminate redundancy in the knowledge base. We will introduce a simple constraint logic program and describe a way to implement exhaustive searches using metaprogramming on parallel computers. It is important to know how knowledge base access is implemented in GHC. The inference system including knowledge base access is summarized in the following procedures:

(1) Accessing knowledge bases

This procedure retrieves knowledge which relates to the goal. If the procedure does not find any, the system executes procedure (4). Otherwise, the system executes procedure (2) for each possibility.

(2) Changing variable names

Since retrieved knowledge (Horn clauses) are copies of existing knowledge in the knowledge base, the retrieved knowledge must use different variable names. The procedure assigns variables the retrieved Horn clause variables and executes procedure (3).

(3) Analyzing retrieved Horn clauses

If the retrieved Horn clause has a conditional part, the procedure defines it as a new goal(s). If not, the Horn clause is true. The procedure outputs the result and executes procedure (4).

(4) Backtracking

Backtracking removes instances assigned to goal variables and assigns instances for previous procedures (1). If the previous instances are exhausted, the procedure repeatedly executes

ascendant nodes (goals) in the proof tree. If there are no more ascendant nodes, the procedure stops.

3. Parallel Implementation in GHC

The previous section showed that exhaustive parallel inference is executed by (1) searching the knowledge base, (2) duplicating and renaming variables, (3) analyzing retrieved Horn clauses, and (4) backtracking.

It is possible to implement procedure (3) by metaprogramming techniques, but procedure (1) is difficult because it is implemented in GHC, which lacks the OR-parallel search. Since GHC cannot assign multiple values to a variable, not all solutions of a knowledge base search can be stored in one variable. Also, since GHC cannot distinguish between variables and constants, we cannot implement procedure (2) without adding to GHC. If we implement parallel backtracking in procedure (4), it results in searching all paths on the search tree in parallel. Parallel searching is executed by defining goals for each path. It is awkward to assign goals for each path by copying the original goals.

3.1 New Data Type $-Variable

Our idea is to distinguish between GHC variables and knowledge base variables. To do so, we add a new data type, the "$-variable," to GHC to represent knowledge base variables (Fig. 3). We represent $-variables using parentheses ($(1), $(2), $(3)) in GHC programming, but a simplified representation of $-variables uses none ($1, $2, $3). A knowledge base search is done using the following expression:

```
?- clause_stream( ancestor($10,$11), Out_Stream).
   Out_Stream=[ [(ancestor($12,$13):- parent($12,$13)), Binf1],
   [ (ancestor($14,$15):- parent($14,$16), ancestor($16,$15)), Binf2]].
```

The knowledge base representation using the $-variables in Fig. 3 could not be managed by the GHC interpreter. Metaprogramming needs unification and substitution mechanisms to manage $-terms.

(1) unify(Term1, Term2, NewTerm, OutputBinf).

This predicate unifies "Term1" and "Term2" including $-variables. The unification result is returned in the third argument "NewTerm" and the binding information of $-variables is returned in the fourth argument "OutputBinf". "Term1" and "Term2" are left unchanged. For example, if the user gives the goal "unify(p(a,$1),p($2,b),Result,OutputBinf)", the predicate outputs "Result=p(a,b)" and "OutputBinf=[bind($1,b),bind($2,a)]".

(2) substitute(InputBinf, Term, Result, OutputBinf).

This predicate rewrites "Term" using binding information for $-variables and outputs "Result=p(a,b)" and "OutputBinf=[bind($1,b),bind($2,a)]". For example, if the user gives the goal "substitute([bind($1,b), bind($2,a)], p($1, $2), Result, Binf)", the predicate outputs "Result=p(b,$3)" and "OutputBinf=[bind($1,b)]"

3.2 Parallel Programming

Exhaustive parallel inference can be implemented by meta-programming using the previous unification and substitution for GHC $-variables.

The inference processing in Fig. 2 obtains all solutions by backtracking. Since parallel inference searches for every solution on the search tree in parallel, the predicate "clause(P,Q)" must be extended to search the knowledge base. The predicate is extended

to receive a stream of retrieved Horn clauses. The set of Horn clauses received by the previous "clause_stream" predicate is analyzed by the following parallel program:

```
bagof_solve([ ], VL, Binf, His, Result):-true|
    Define "His1" as the result of assigning the set of terms "His"
    into binding information "Binf" by substitution predicate,
    Result=[ [VL,Binf,His1] ].
bagof_solve([true], VL, Binf, His, Result):-true|
    Define "His1" as the result of assigning the set of terms "His"
    into binding information "Binf" by substitution predicate,
    Result=[ [VL,Binf,His1] ].
bagof_solve([ P|Q], VL, Binf, His, Result):- Q \=[ ] |
    bagof_solve([P], VL, Binf, His, Result1),
    and_solve( Q, Binf, Result1, Result).
bagof_solve([ P], VL, His, Result):-otherwise|
    clause_stream( P, ClauseStream),
    or_solve( ClauseStream, VL, Binf, His, Result).
```

The first argument of the predicate "bagof_solve" specifies goals to be solved by parallel inference. The program represents the goals using list expressions. The second argument specifies output variables. For the user's goals, each variable is assigned a value obtained by parallel inference. The third argument specifies binding information for the parallel inference, represented by list expressions. The fourth argument specifies the proof tree to be generated by parallel inference. The fifth argument specifies the set of solutions obtained by parallel inference.

The first and second clauses of the program are procedures for the first clause in Fig.2. The third and fourth clauses are procedures for the second and third clauses in Fig. 3. The program is the parallel procedure for searching the knowledge base, analyzing retrieved Horn clauses, and finding all solutions. The third clause is

the procedure for analyzing from the left side of AND-goals. Solutions for the variable "Result1" are transferred from the predicate "bagof_solve" to the predicate "and_solve" to maintain the consistency of shared $-variables for the AND-goals. The "and_solve" predicate in the third clause is implemented by following parallel program:

```
and_solve( Q, Binf, [ [ VL, BinfHis, His ] | R ], Result):-wait(Q) |
    Define "Q1" as the result of assigning Goals "Q" into binding
    information "BinfHis" by substitution predicate,
    bagof_solve( Q1, VL, BinfHis, His, Result1),
    and_solve( Q, Binf, R, Result2),
    merge( Result1, Result2, Result).
and_solve( Q, Binf, [ ], Result):- wait(Q) | Result = [ ].
```

The first argument of the "and_solve" predicate specifies goals to be solved. The second argument specifies information bound to $-variables during inferencing. The third argument specifies execution results of AND-goals connected before AND-goal "Q". This includes the output variables ("VL") of the goals, information bounded ("BinfHis") $-variables, and a path of the generated proof tree ("His") during inferencing. The fourth argument specifies execution results for "Q".

If the AND-goals connected before "Q" have solutions, the first clause of the program is executed to assign $-binding information for one of the solutions to "Q" to give duplicate values for terms with shared $-variables and solves goal "Q1". After that, the clause solves other goals with the "and_solve" predicate. Execution results "Result1 and Result2" are merged by the "merge" predicate. If the AND-goals connected before AND-goals "Q" have no solutions, the

second clause is executed. In this case, the clause outputs nil ("[ ]") for the execution result in the fourth argument.

This "or_solve" predicate implements the following parallel program:

```
or_solve( [[(Head:-Body),Inf] | ClauseList], VL, Binf, His, Result):-true|
    Define "NewVL" as the result of assigning variables" VL" into
    binding information "Inf" by substitution predicate,
    append( Inf, Binf, NewBinf),
    append( His, [ (Head:-Body) ], NewHis),
    bagof_solve( Body, NewVL, NewBinf, NewHis, Result1),
    or_solve( ClauseList, VL, Binf, His, Result2),
    merge( Result1, Result2, Result).
or_solve( [ ], VL, Binf, His, Result):-true | Result = [ ].
```

The first argument of the "or_solve" predicate specifies clauses retrieved from the knowledge base by the "clause_stream" predicate. The second argument specifies output variables ("VL") of the user defined goals. The third argument specifies a path of generated proof tree ("His"). The fourth argument specifies the proof tree to be generated by parallel inference. The fifth argument specifies execution results.

The first clause executes retrieved clauses with the "clause_stream" predicate in the recursive structure. Since information about $-variables replaced by the "clause_stream" predicate is stored in binding information ("Inf"), the first clause assigns the information to "VL". After that, it adds the old proof history to a "Head:-Body" clause to make a new proof history and analyzes the condition part ("Body") of the clause using the "bagof_solve" predicate. Execution results "Result1" and "Result2" are merged by the "merge" predicate. If the knowledge base search

fails, the second clause is executed and the execution result in the fifth argument is nil "[ ]".

3.3 Processing Cut Symbols

Cut symbols are processed by an extension of the parallel inference system. We assume that the cut symbol includes only one conditional part, and represent it using an "ifthen(P, Q)" predicate. An example including the predicate is shown in the following knowledge base:

```
children( $1, $2):-
    children( $1, [ ], $2).
children( $1, $2, $3):-
    parent( father( $1, $4), children( $1, [$4|$2], $3)).
children( $1, $2, $2).
```

The predicate "children($1, $2)" specifies that children of father "$1" are "$2"; "$2" is represented by a list structure. The parallel programming to execute the cut symbol adds the following clause between the third and fourth clauses of the program "bagof_solve":

```
bagof_solve( [ ifthen(P,Q) ], VL, Binf, His, Result):-true |
    bagof_solve( P, VL, Binf, His, result1),
    then_part( Q, Binf, Result1, Result).
```

The "then_part" predicate is a parallel program to analyze the goals "Q" of the metapredicate "ifthen(P,Q)":

```
then_part( Q, BindInf, [ Result1| Result2 ], Result):-true
    and_solve( Q, Binf, [ Result1], Result2),
   If "Result2" is null "[ ]", assigns the result "Result" into a value
   " [fail]". If not, assigns it into a value of "Rsult2".
```

```
then_part( Q, BindInf, [ ], Result):-true | Result=[ ].
```

Parallel programming also needs to add the "or_solve" program of the previous section to the following clause:

```
or_solve( [ [( Head:- [ifthen(Cond, Body)]), Inf] | ClauseList], VL,
        Binf, His, Result ):-true |
    Define "NewVL" as the result assigned from "VL" to
    "Inf" for variable bindings by the substitution predicate
    of section 3.1,
    append( Inf, Binf, NewBinf),
    append( His, [(Head:- [ifthen( Cond, Body) ] ) ], NewHis),
    bagof_solve( [ ifthen( Cond, Body) ], NewVL, NewBinf,
                 NewHis, Result1),
    next_solve( ClauseList, VL, Binf, His, Result1, Result).
```

If the execution result ("Result1") of the "bagof_solve" predicate is null ("[ ]") after "ifthen(Cond, Body)" is analyzed in the "or_solve" predicate, the "next_solve" predicate analyzes the next clause list ("ClauseList") using the OR-parallel program "or_solve". If the result is not null, it prunes the clause list.

4. Large-Scale Knowledge Base Search

The previous section described how the knowledge retrieval function ("clause_stream") of the parallel inference system was implemented in GHC. An issue involved in this approach is that $-unification ("unify") and $-substitution ("substitute") implemented in GHC do not have enough parallel performance and require more memory than when implemented in C. This issue becomes important when we use large-scale knowledge bases.

We programmed the knowledge retrieval, $-unification processing, and $-substitution in C. It is easy to interface GHC and

C. The knowledge base must be shared by multiple GHC processes. We implemented the knowledge retrieval with the knowledge retrieval system retrieval by unification (RBU) to speed up access. We call the knowledge retrieval system interface between GHC and RBU [Yokota '89] a "parallel knowledge retrieval subsystem," described as follows:

The system configuration for the parallel knowledge retrieval subsystem is shown in Fig. 4. The system uses shared memory and a parallel inference machine made by the SEQUENT Corporation. We implemented the system on a parallel operating system (DYNIX) with UNIX. Hash and tree structures speed up knowledge base access. Horn clauses stored in shared memory are accessed from RBUs in parallel. Each RBU is assigned one UNIX process. Concurrency control for parallel access applies to each set of Horn clauses with the same relation name. Concurrent retrieval is allowed (shared mode), but concurrent updates are not (exclusive mode). The parallel knowledge base is searched by interaction between GHC and RBU processes.

The system runs 25 times faster for 6 RBU processors and 10 GHC processors than for Quintus-Prolog on one processor.

5. System Evaluation

The system can be used as the kernel of several knowledge information systems. We obtain good parallel performance for problems such as in Fig. 3 and problems such as "append(X,Y,[a,b,c,...])" without an included database, for example. We created a simple constraint logic programming system to evaluate the performance of the system. Fig. 5 shows the system's parallel action. Given a problem, the system searches the knowledge

base using exhaustive parallel inferencing. Parallel inferencing combines OR- and AND-parallel execution. The parallel inference system is the kernel of the constraint logic program. We use a two-dimensional electrostatic field design problem to evaluate performance. The constraint logic programming system is implemented by extending the "bagof_solve" program (Section 3.2) and adding it to the constraint solver. The extension collects all sets of linear equations in parallel inferences.

5.1 Example of Design Problem

Fig. 6 shows the design problem for a two-dimensional electrostatic field. Phenomena in electrostatic fields are characterized by Laplace's equation, represented by Liebman's 5-point approximation (Fig. 7). The knowledge base has two constraints: the space potential of the center, restricted to the range "74.9 volts to 75.1 volts," and the space potential approximated by Liebmann's equation. The problem is to compute boundary potential X. The other three boundaries are defined as 100 volts. Fig. 8 shows a knowledge base for the design problem.

5.2 Performance

Given the following problem,

```
?- constraint_solve( design( [ [ $900, $900, $900, $900, $900],
                               [  100, $911, $912, $913,  100],
                               [  100, $921, $999, $923,  100],
                               [  100, $931, $932, $933,  100],
                               [  100,  100,  100,  100,  100] ],
                               $900, $999 ) ),
```

the system solves the problem using "bagof_solve". The system generates linear equations for each element of the member in Fig. 8. The system then calls the constraint solver with linear equations. The constraint solver converts these equations to matrices for parallel computation. Computation is implemented by AND-stream parallel programming using GHC.

Fig. 8 shows an example of a 4 x 4 matrix. Each matrix is computed by assigning each element to a GHC process and applying Gaussian elimination from the upper elements of the diagonal to the lower elements in parallel.

The second of the diagonal elements, for example, is computed as follows:

(1) The GHC process of the second element sends a message to its vertical and horizontal neighbors.

(2) Each element which receives a massage sends a new message to these neighbors.

(3) Each element receiving a message from these neighbors computes based on Gaussian elimination.

Eight processors were 6 times faster than 1 for parallel matrix computation.

We measured the performance of the parallel inference system with the constraint solver connected to 15 processors assigned to GHC and RBU. Hardware performance of the parallel computer using shared memory corresponds to the number of processors. The maximum number of GHC processors was 9, and the maximum number of RBU processors was 6. Fig. 9 shows parallel performance measured by assigning RBU to 1 or 6 processors, and GHC to 1, 3, 5, 7, or 9 processors. The black circles in Fig. 9 denote 1 RBU processor and the white circles 6 RBU processors.

Six RBUs are 6 percent faster than one when there is 1 GHC processor. As the number of GHC processors increases, the parallel performance ratio decreases. However 9 GHC processors are roughly 9 times faster than 1 processor for each case.

The following parallel performance summarizes the previous results:

(1) All sets of linear equations are collected.

(2) Equations are converted to matrices and processed in parallel.

If the number of elements in the example increases, the number of GHC processes to be executed in parallel is increased by the number of OR-parallel executions collecting linear equations. If the number of meshes dividing the electrostatic field increases, the number of GHC processes needed to compute the matrix is increased by the increase of matrix size.

The constraint problem I have described for reducing linear equations including equality can be extended to reducing linear equations including inequality.

6. Conclusions

I have described a knowledge-based parallel inference system using metaprograming techniques in GHC. I showed how a method to add OR-parallel functions to GHC, which has only AND-parallel functions. I also described functions to manage $-variables representing the knowledge base. We implemented these in C to avoid excessive GHC garbage collection. The system can be extended to handle knowledge bases including cut symbols. We also interfaced GHC and the knowledge base retrieval subsystem to speed up large-scale knowledge base access.

Lastly, I have described a simple constraint logic programming system for evaluating performance which is implemented by extending the parallel inference system and adding a simple constraint solver. The system performed well.

This research demonstrates the basic techniques for implementing parallel applications efficient from the symbolic point of view, and related to partial evaluation [Fujita '88, Furukawa '88] and reflective programming [Tanaka '90].

7. Acknowledgements

We thank Messrs. H. Hayashi and A. Hattori of Fujitsu Laboratories Ltd. for their valuable suggestions. We also thank the researchers in ICOT's KBM working group for their very useful suggestions.

[References]

[Bowen & Kowalski '81] K. A. Bowen and R. A. Kowalski: Amalgamating Language and Meta-Language in Logic Programming, TR 4/81, Syracuse University, 1981.

[Clark '84] K. L. Clark and S. Gregory: Notes on Systems Programming in Parlog, Proc. of the International Conference on Fifth Generation Computer Systems '84, 1984.

[Fuchi '78] K. Fuchi: Problem Solving and Inference mechanism, Journal of Information Processing of Japan (in Japanese), Vol. 11, No. 10, 1978.

[Fujita '88] H. Fujita, A. Okumura, and K. Furukawa: Partial Evaluation of GHC Programs Based on the UR-set with Constraints, Proc. of the Fifth International Conference and Symposium on Logic Programming, 1988.

[Furukawa '88] K. Furukawa, A. Okumura, and M. Murakami: Unfolding Rules for GHC Programs, New Generation Computing, Ohmusha LTD., 1988.

[Hattori '89] A. Hattori et al: A Hierarchical Structured Parallel Inference Machine, Parallel Computing '89, 1989.

[Heintz '87] N. Heintze, S.Michaylov, P.Stuckey: CLP(R) and Some Electrical Engineering Problems, Fourth IEEE Symposium on Logic Programming, 1987.

[Kasif '83] S. Kasif, M Kohli, and J. Minker: 'PRISM: A Parallel Inference System for Problem Solving', Proc. of the Logic Programming Workshop '83, 1983.

[Kitakami '84] H. Kitakami, S. Kunifuji, T. Miyachi, and K. Furukawa: A Methodology for Implementation of a Knowledge Acquisition System, Proc. of the 1984 International Symposium on Logic Programming, 1984.

[Lander '88] E. Lander and P. Mesirov: Protein Sequence Comparison on a Data Parallel Computer, Proc. Int. Conf. Parallel Processing, 1988.

[Lassez '87] C. Lassez: Constraint Logic Programming, Fourth IEEE Symposium on Logic Programming, 1987.

[Ozawa '90] T. Ozawa, A. Hosoi, and A. Hattori: Generation-Type Garbage Collection for Parallel Logic Languages, NACLP90, 1990.

[Sato '90] H. Sato and M. Ikesaka: Particle Simulation on a Distributed Memory Highly Parallel Processor, Supercomputing in Nuclear Applications '90, 1990.

[Shapiro '84] E. Shapiro: System Programming in Concurrent Prolog, Proc. 11th Annual ACM Symp. on Principles of Programming Languages, ACM, 1984.

[Takeuchi '87] A. Takeuchi: Parallel Problem Solving Language ANDOR-II, Proc. of the 2nd Program Symposium, Japan Society for Software Science and Technology(in Japanese), 1986.

[Tanaka '90] J. Tanaka, Y. Ohta, and F. Matono: Overview of an Experimental Reflective Programming System: ExReps, Fujitsu Scientific and Technical Journal, Vol. 26 No. 1, 1990.

[Uchida '88] S. Uchida, K. Taki, K. Nakajima, A. Goto, and T. Chikayama: Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project, Proc. of the International Conference on FGCS '88, 1988.

[Ueda '85] K. Ueda: Guarded Horn Clauses, Technical Report TR-103, ICOT, 1985.

[Yang '87] R. Yang: P-Prolog: A Parallel Logic Programming Language, Series in Computer Science - Vol. 9, World Scientific Publishing Co Pte LTD., 1987.

[Yokota '89] H. Yokota, H. Kitakami, and A. Hattori: Term Indexing for Retrieval by Unification, Proc. of 5th Int'l Conf. on Data Engineering, 1989.

```
ancestor( X, Y):- parent( X, Y).
ancestor( X, Y):- parent( X, Z),ancestor( Z, Y).

parent( f1, f2).
parent( f2, f3).
parent( f3, f4).
```

Figure 1 Example of a knowledge base

```
solve( true):- !.
solve( ( P, Q)):- solve( P),solve( Q).
solve( P):- clause( P, Q), solve( Q).
```

Figure 2 Example of inference processing

```
ancestor( $1, $2):- parent( $1, $2).
ancestor( $1, $2):- parent( $1, $3), ancesor( $3, $2).

parent( f1, f2).
parent( f2, f3).
parent( f3, f4).
```

Figure 3 Example of a knowledge base representing $-variable

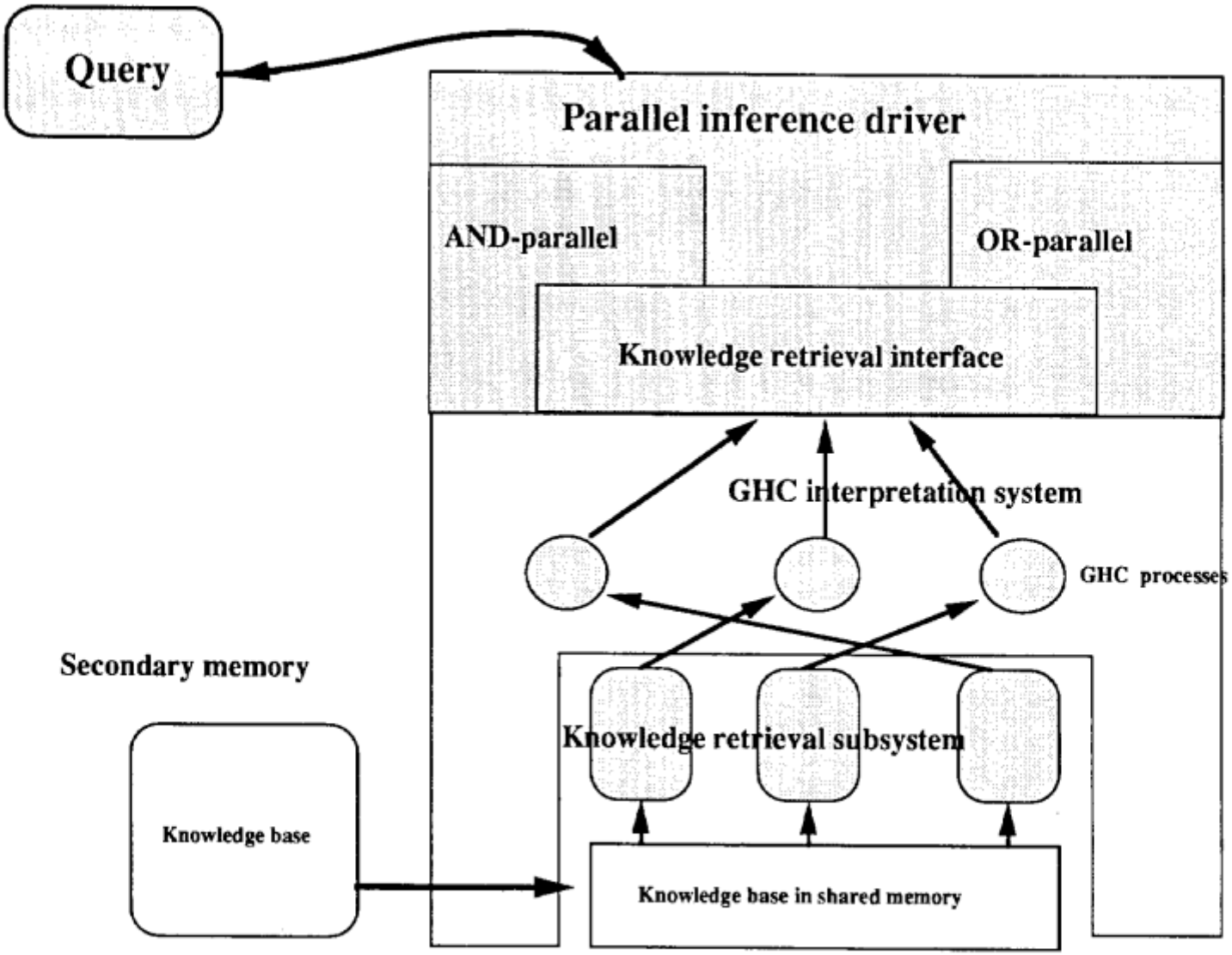

Figure 4 System configuration

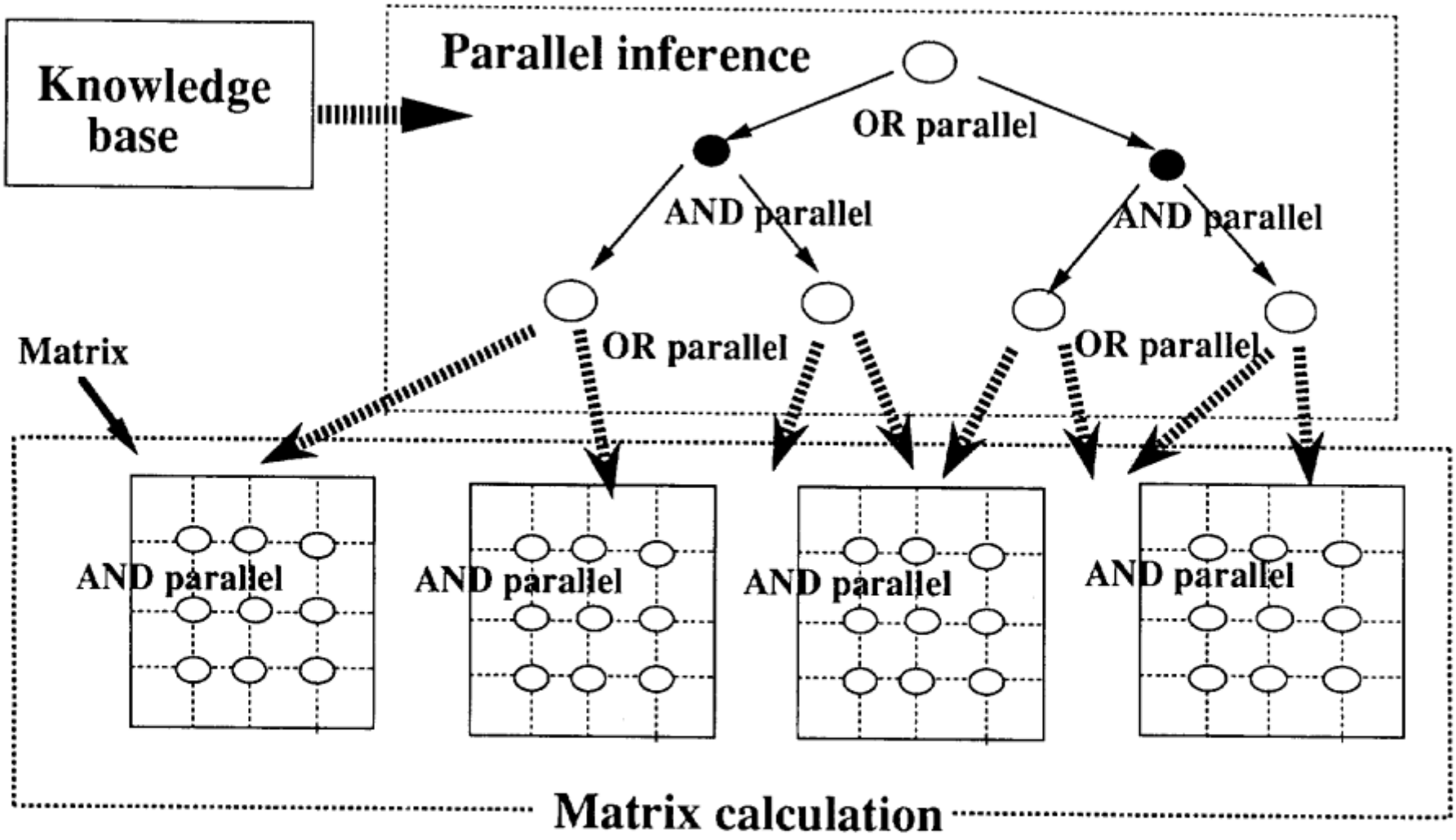

Figure 5 Parallel action

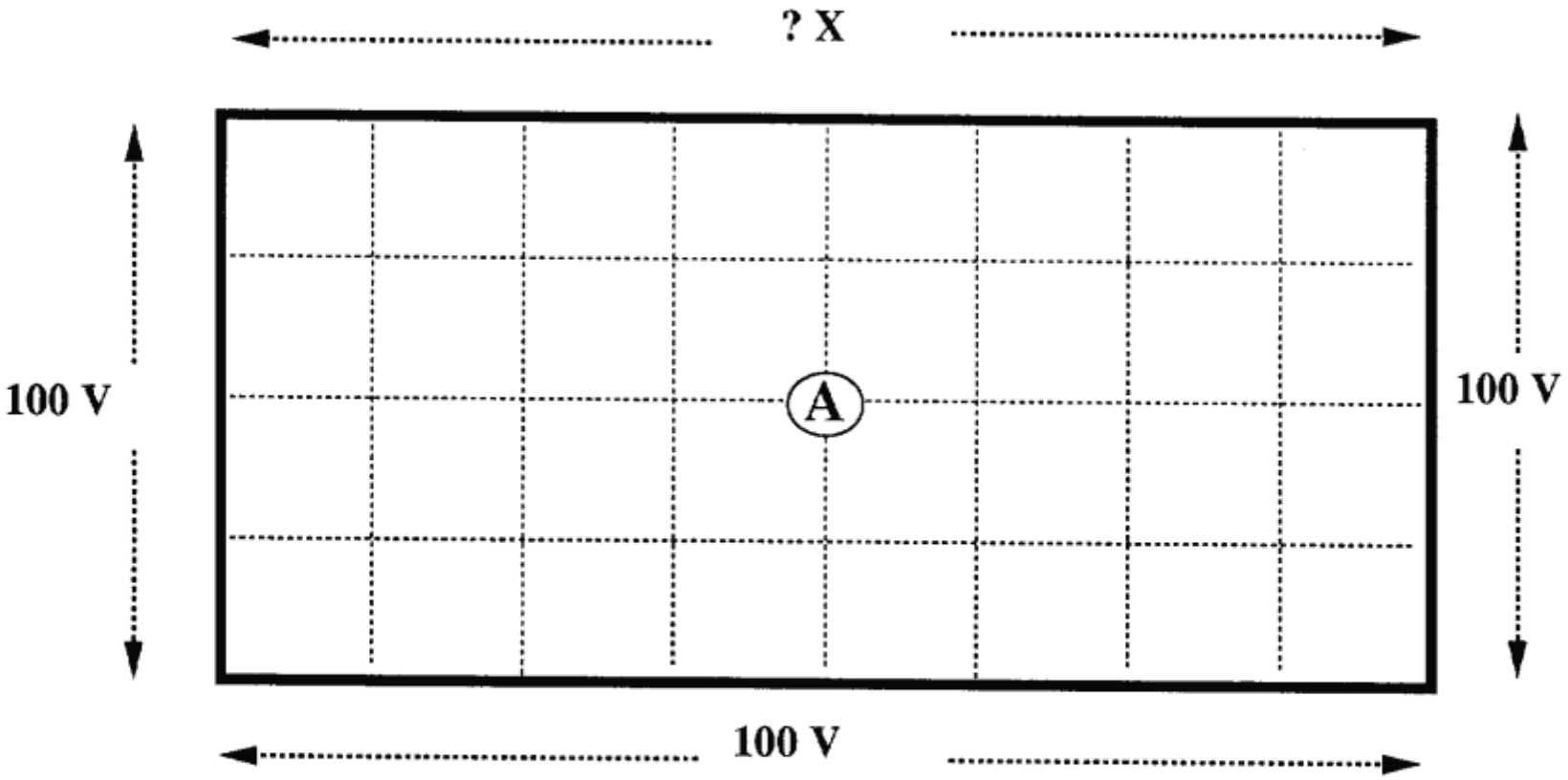

Figure 6 Enclosed 2-dimensional region
(74.9 <= A<= 75.1)

```
design([$1|$2],$3,$4):-
      member($3,[-1,0,1]),
      laplace([$1|$2]),74.9<=$4,$4<=75.1 .

member($1,[$1|$2]).
member($1,[$2|$3]):- member($1,$3).

laplace([$1,$2,$3|$4]):-   liebmann( $1,$2,$3), laplace([$2,$3|$4]).
laplace([$1,$2]).
liebmann([$1,$1,$3|$4],[$5,$6,$7|$8],[$9,$10,$11|$12]):-
    $10+$2+$5+$7-4*$6=0,
    liebmann([$2,$3|$4],[$6,$7|$8],[$10,$11|$12]).
liebmann( [$1,$2], [$3,$4], [$5,$6]).
```

Figure 7 Knowledge base for analyzing electrostatic fields

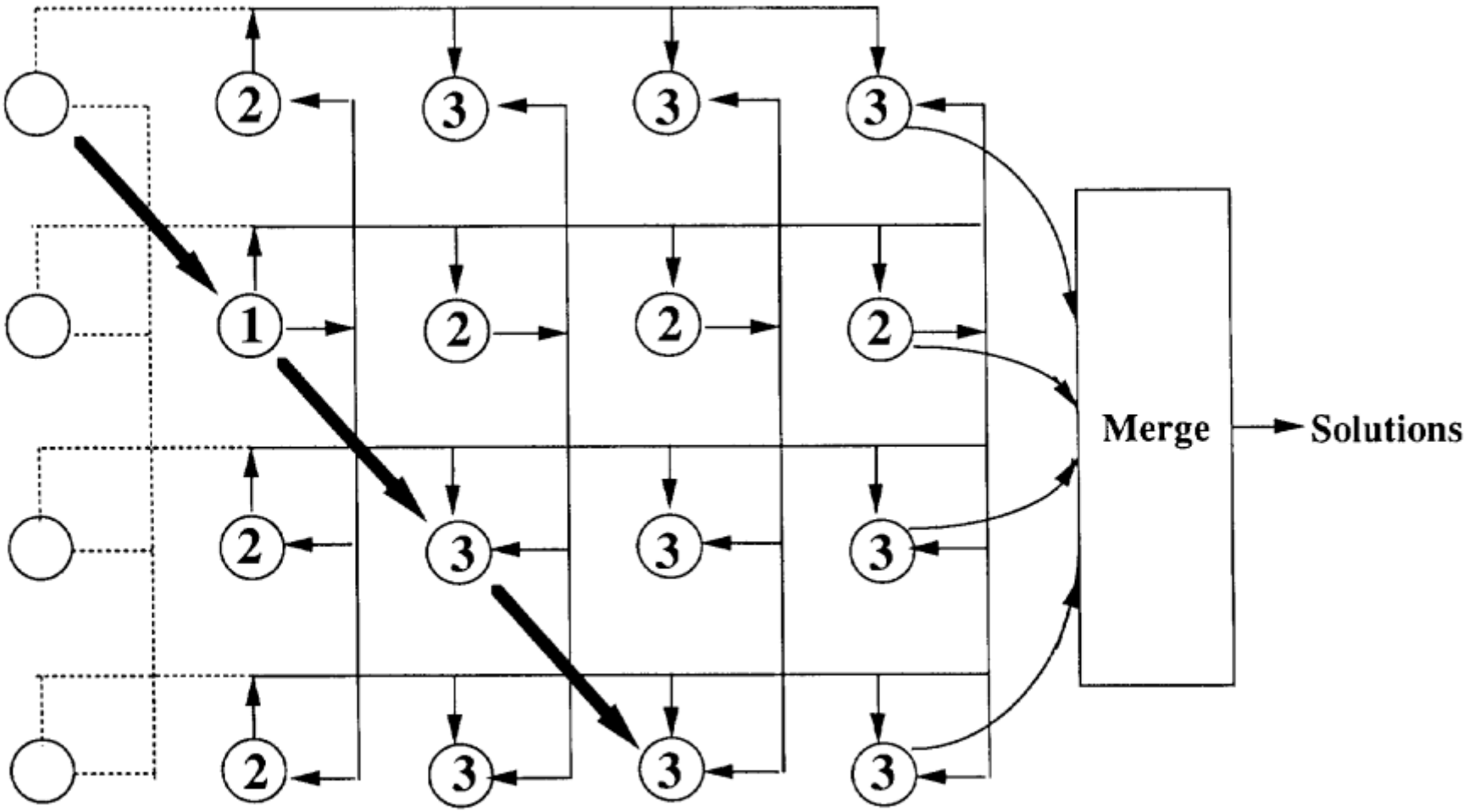

Figure 8 Parallel matrix solution

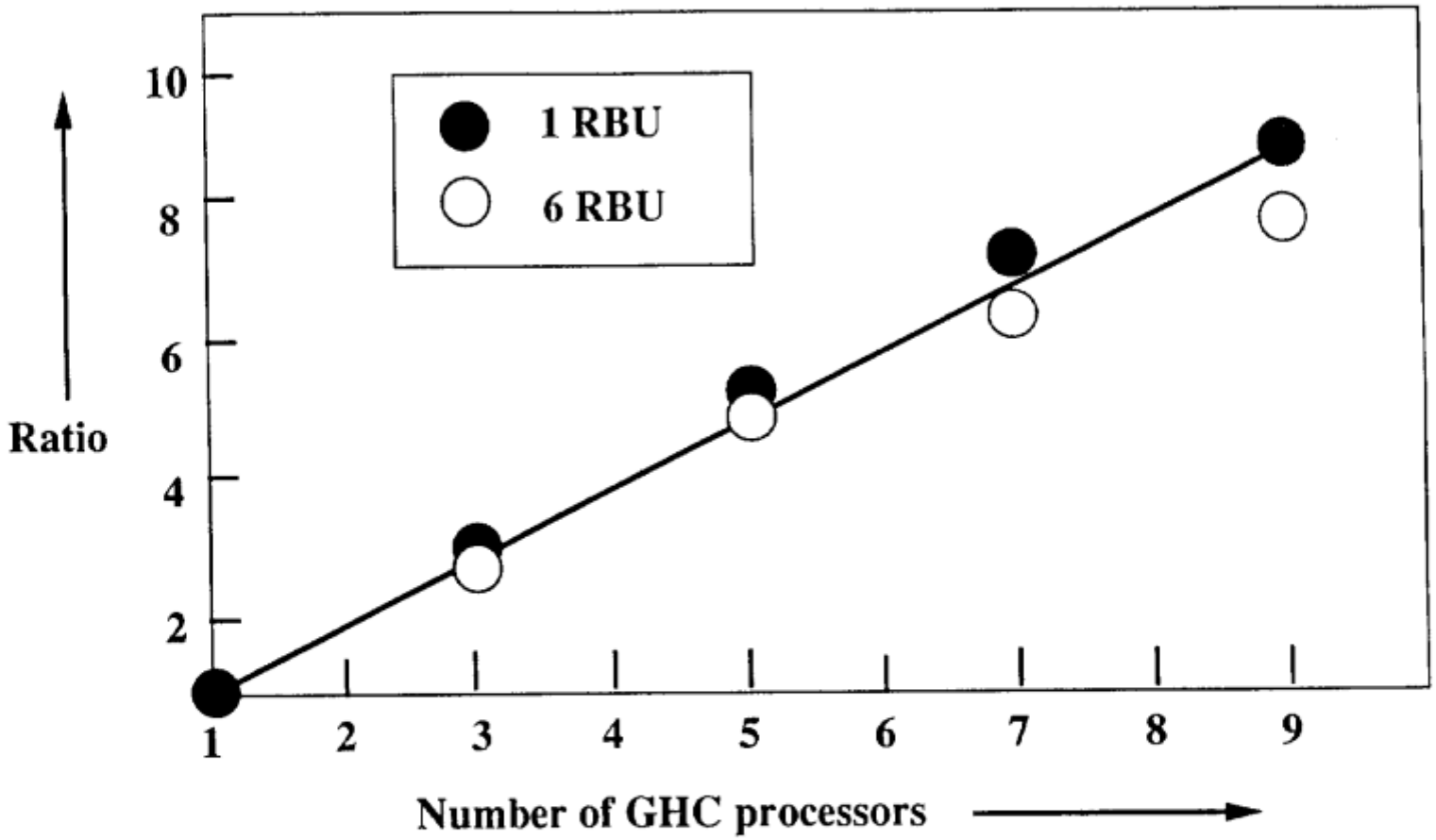

Figure 9 Performance ratio for multiple processors