ICOT Technical Report: TR-159

TR-159

Distributed Implementation of FGHC —Toward the realization of Multi-PSI system—

by
Makoto Kishishita (Fujitsu Ltd.)
Jiro Tanaka, Toshihiko Miyazaki, Kazuo Taki
and Takashi Chikayama (ICOT)

March, 1986

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

Distributed Implementation of FGHC
--Toward the realization of Multi-PSI system --

Jiro Tanaka, Toshihiko Miyazaki, Kazuo Taki, Takashi Chikayama
ICOT Research Center, Mita-kokusai-build. 21F
1-4-28, Mita, Minato-ku, Tokyo 108, JapaN

Makoto Kishishita
International Institute for Advanced Study of Social
Information Science (IIAS-SIS), Fujitsu Limited
1-17-25, Shinkamata, Ohta-ku, Tokyo 144, Japan

[Abstract]

Various technical problems arise when we consider the distributed implementation of a parallel logic language. This paper tries to describes our solutions for such problems.

The model we assumed is the multi-PE system where dozens of PEs are grid-network connected. Each Processing Element (PE) has local memory. It has no shared memory nor global address space.

We assumed Flat Guarded Horn Clauses (FGHC), which is the simplification of Guarded Horn Clauses (GHC) [Ueda 85], as our underlying parallel logic language. Each PE can execute FGHC programs.

We have implemented the software simulator of the multi-PE system. Our simulator is based on the work of Murakami and Miyazaki [Murakami 85b]. It simulates the execution of pre-processed FGHC program in a multi-PE environment.

1. Introduction

The development of logic-based high-speed parallel-computing-system is the final goal of Fifth Generation Computer Project at ICOT. This project is the ten year project and we are now on the beginning of the fifth year.

Various researches have been carried out for the development of logic-based high-speed parallel machine from the "architectural" view point [Murakami 85a]. The prototype developments of PIM-D (Parallel Inference Machine based on Dataflow) and PIM-R (Parallel Inference Machine based on Reduction) are the examples of such activities.

However, in accordance with the development of our research, we noticed that there exists lots of problems to be solved at "software" or "firmware" level, such as (1) how to boot the system, (2) how deliver object program to each PE, (3) how to handle input/output and interrupt, (4) how to balance the load between PEs, etc.

These problems are not the hardware problem in itself. These problems must be solved at the software or firmware level. Therefore, ICOT decided to start "Multi-PSI" project. There is not so much new in hardware. ICOT has already developed Personal Sequential Inference (PSI) machine [Taki 84]. This machine is a personal workstation and its designing concept is very similar to LISP machine. Conventional techniques have been adopted and ESP [Chikayama 84], which is the object-oriented extension of Prolog, is firmware supported. Multi-PSI hardware is simply built up by connecting 6 - 16 Personal Sequential Inference (PSI) machines with high speed grid-hardware. Most of problems exists in "software".

2. The multi-PE model

The model we assumed is the multi-PE system where dozens of PEs are grid-network connected. There exist lots of designing choices. Lots of intensive discussions had been done inside ICOT. The decisions we have made are as follows:

(1) PE must be connected in a way which allows the increase of PE number. Our system must work even if we have hundreds of PE. Therefore, we did not adopt the common bus. Instead, we adopted grid-network.

(2) Each Processing Element (PE) has its local memory. It has no shared memory nor global address space. Global address schema has not adopted because we have thought that the distributed garbage collection is extremely difficult. In our system, PE communicates each other only via message exchange.

(3) We assume and-parallel logic language as our underlying logic language. And-parallel execution of a program is also assumed. Choosing "pure" prolog and "or-parallel" execution may be the other possibility, which was not our choice. Or-parallel execution accompanies the copying of variables. And-parallel execution does not. Our claim is that and-parallelism plays more basic roles than or-parallelism in distributed environment. Each PE executes a program independently or cooperatively.

(4) "Distributed" logic languages has not been adopted as our base language. D-prolog [Pereira 84] or MENDEL [Honiden 86] is the examples of such distributed language. In such language, each

"process" is a sequential prolog program and "processes" communicate each other via sending or receiving messages. These languages has not been selected since we must write "process" explicitly in such languages and we did not like such coarse granularity.

Figure 1 shows how the multi-PE model looks like.

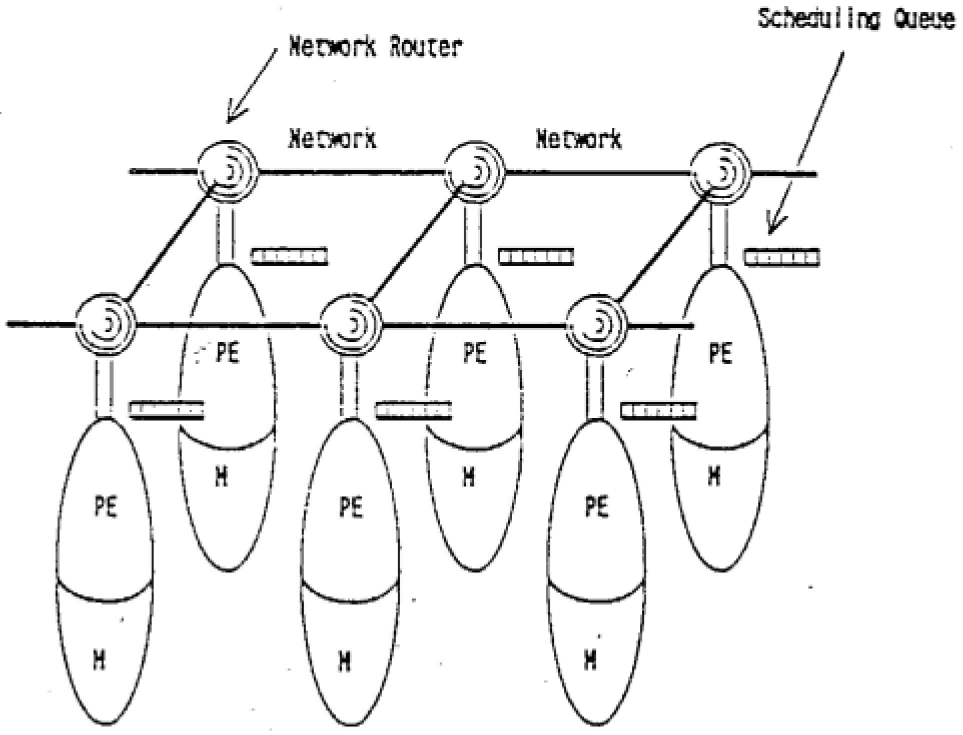

Figure 1 Multi-PE Model

This multi-PE model has the following features.

(1) We assume that the program, i.e., the definition of clauses, has been loaded to every PE from the beginning for simplicity. First, a goal is put into one PE, this automatically starts the computation. When there is no goals to be processed on all PEs, it means the end of computation.

(2) Each PE has one scheduling queue. Each PE repeats to dequeue a goal from the scheduling queue and reduces it to the resulting goals. These goals are enqueued to the scheduling queue or thrown to other PE.

(3) Each PE executes the goals which are thrown from other PEs besides processing local goals.

(4) Since each PE has independent address space, the unification of two variables existing on different PEs invokes the necessity to span inter-PE reference chains. Each PE has the variable management table for that purpose.

(5) Unification sometimes invokes various messages to other PEs. The examples of such messages are "get_value," "unify," "unify_channels," etc. The meaning of these messages is explained in section 6.

3. Underlying language

As mentioned in section 2, the language we are interested in is "parallel logic language", which allows and-parallel execution of a program. Parlog [Clark 85] and Concurrent Prolog (CP) [Shapiro 83a] are the examples of such "Parallel Logic Languages." Although there are differences, the basic computation mechanisms of these languages are quite similar. Horn clauses with guards are used to define predicates, goals are executed in parallel, and they have some synchronization mechanisms between goals.

After examining various trade-offs, we have decided to adopt Flat Guarded Horn Clauses (FGHC) as our underlying language. FGHC is the simplified version of Guarded Horn Clauses (GHC) which is originally proposed by Ueda [Ueda 85]. The FGHC clause has the following format.

```
H(Arg1, Arg2, .... Argn) :- G1, G2, ... , Gm | B1, B2, ,,, Bk .
       Head Part                Guard Part       Body Part
                 passive part                    active part
```

Similar to other parallel logic languages, a FGHC clause consists of three parts: Head Part, Guard Part and Body Part. Head Part and Guard Part are called "passive part" because these parts do not instantiate variables. On the other hand, Body Part is called "active part."

The features of FGHC and its evaluation rules are summarized as follows:

(1) Only system predicates are allowed as goals in Guard Part. This is the reason that we call our language "Flat" Guarded Horn Clauses.

(2) Passive part is executed sequentially. First, head unification is executed sequentially from right to left. Second, Guard Part is executed sequentially.

(3) Suspension occurs when the unification of passive part wants to instantiate global variables, i.e., passive part never exports bindings to the outer world. This provides the basic mechanism of "suspension" in FGHC. The principle of GHC can be understood as follows: We never "assume" in the condition part. If there is not enough information to judge, we simply wait until we get the enough information.

(4) The passive part of candidate clauses are tested sequentially. If a passive part of one candidate clause succeeds, that clause wil be committed. If the candidate clause fails or suspended, the next candidate clause will be tested. When none of candidate clauses succeed and there exists more than one suspended clauses, the goal is suspended.

(5) The suspended goal is hooked to all variables which have caused suspension. The suspended goals are kept in the form of "goal($(Arg1, ..., Argn), [R1, ..., Rm])," where "goal" is the predicate name and [R1, ..., Rm] is the result list whose each element shows the execution result of each candidate clause. A suspended goal is resumed when that variable is instantiated. Notice that suspension occurs to "goals" not to "clauses." Our implementation experiences [Tanaka 85a, Tanaka 85b] show that the or-parallel execution of a parallel logic language accompanies too much overhead. Therefore, we decided not to support or-parallelism.

We show an simple example of FGHC program here. This example is an "stream-merge" example [Ueda 85].

```
merge([A|X], Y, Z) :- true | Z = [A|W], merge (X, Y, W).
merge(X, [A|Y], Z) :- true | Z = [A|W], merge (X, Y, W).
merge([], Y, Z) :- true | Z = Y.
merge(X, [], Z) :- true | Z = X.
```

The meaning of this example is self-explanatory. Notice that the unification which exports bindings to the global world must be written explicitly in the active part. Also notice that "merge" process is hooked on both variables X and Y, in the case no input data is available. In such case, only one of these goals will be resumed and other goals are disabled at that time.

In short, FGHC is the subset of GHC which is fully tuned for the implementation easiness. Our assumption is, even if it is "flat," it is still powerful enough for the practical application.

4. Multi-PE Simulator

We have implemented the software simulator of the multi-PE system. In relate to this simulator, our system is based on the work by Murakami [Murakami 85b]. Our system is written in prolog and simulates the execution of pre-processed FGHC program in a multi-PE environment. We assum 9 PE system in this paper.

Figure 2 shows the top-level program for Multi-PE Simulator. Here, the top level goal is "dg/1," which stands for distributed_ghc. "dg/1" calls "scheduler" and it puts the given "Goals" to PE#1. Then "dg/4" is invoked. The first and the second arguments are the D-list which schedules PE execution. The third and the fourth arguments are the lists of input channels, the list of output channels, respectively. The "dg/4" executes the first element of D-list, append the computation result "NPE" to the tail of D-list, invokes "nm" to take care of message exchange between PEs, and calls "dg/4" recursively.

```
   /* Top Level of Interpreter */
dg(Goals) :-
           /* Call SCHEDULER */
    schedule(Goals,X,X,QH1,[$$$|QT1]),

           /* Call DISTRIBUTED FGHC Interpreter */
    dg([pe(#1,    QH1,    QT1, C_Tbl1, IN1, OUT1, R1),
        pe(#2, [$$$|QT2], QT2, C_Tbl2, IN2, OUT2, R2),
        pe(#3, [$$$|QT3], QT3, C_Tbl3, IN3, OUT3, R3),
        pe(#4, [$$$|QT4], QT4, C_Tbl4, IN4, OUT4, R4),
        pe(#5, [$$$|QT5], QT5, C_Tbl5, IN5, OUT5, R5),
        pe(#6, [$$$|QT6], QT6, C_Tbl6, IN6, OUT6, R6),
        pe(#7, [$$$|QT7], QT7, C_Tbl7, IN7, OUT7, R7),
        pe(#8, [$$$|QT8], QT8, C_Tbl8, IN8, OUT8, R8),
        pe(#9, [$$$|QT9], QT9, C_Tbl9, IN9, OUT9, R9) |T], T,
                                                /* Scheduling Queue */

       [IN1,  IN2,  IN3,  IN4,  IN5,
        IN6,  IN7,  IN8,  IN9],
                                                /* Input-channels */

       [OUT1, OUT2, OUT3, OUT4, OUT5,
        OUT6, OUT7, OUT8, OUT9]                 ).
                                                /* Output-channels */


          /* DISTRIBUTED FGHC Interpreter */
dg([PE|PEs], [NPE|T], C_IN, C_OUT) :-
                                    /* Pop Next Processing-Element */
   call( PE ),                             /* Drive Processing-Element */
   arg(7,PE,NPE),                          /* Get Returned PE-Status */
   nm(C_IN, C_OUT, NC_IN, NC_OUT),    /* Call Network-Manager */
   dg(PEs, T, NC_IN, NC_OUT).
```

Figure 2 Top Level Program for Multi-PE Simulator

Figure 3 shows the program for PE.

```
     /* Processing Element */

  pe(ID, H, T, C_Tbl, C_IN, C_OUT,
                                    /* Input PE-Status */

             pe(ID, RH, RT, RC_Tbl, NC_IN, RC_OUT, NR) ) :-
                                    /* Return New PE-Status */

     recieving(C_IN, T, C_Tbl, NC_IN, NT, NC_Tbl),
     pe(H, NT, NC_Tbl, C_OUT, (H, T, C_Tbl, C_OUT) ).

  pe([$$$|H],[$$$|T], C_Tbl, C_OUT,
             (H, T, C_Tbl, C_OUT) ) :- !.

  pe([$(Goal, G, Gt)|H], T, C_Tbl, C_OUT, R) :-
     reduce(Goal, T, C_Tbl, C_OUT, NT, NC_Tbl, NC_OUT),
     pe(H, NT, NC_Tbl, NC_OUT, R).

                 .
                 .
                 .
```

Figure 3 Program for PE

"pe/7" is set in motion by "call(PE)" in "dg/4." The first argument of "pe/7" is the ID number of PE, the second and the third is the D-list which corresponds to the scheduling queue inside the PE, the forth is the variable management table which is used to span inter-PE reference chains, the fifth and the sixth is the input and output channels to network manager "nm," and the seventh is the new PE status "NPE" which will be enqueued in "dg/4."

"pe/7" receives the message from C_IN, append this message to the tail of its scheduling queue, and starts up "pe/5." "pe/5" reduces a goal inside the scheduling queue. "pe/7" returns its state after it has finished the 1 cycle computation of the scheduling queue.

In short, PE consists of input/output channels, scheduling queue and the variable management table. The conceptual structure for PE is shown in Figure 4.

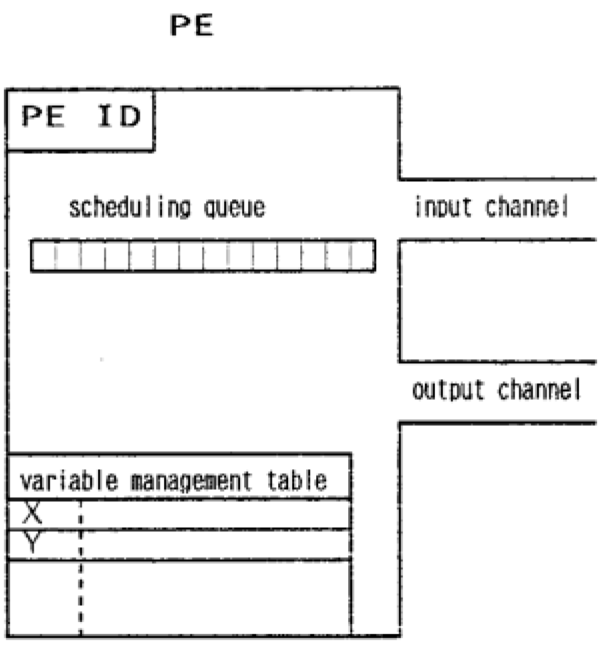

Figure 4 Conceptual Structure for PE

Network manager takes care of message exchange between PEs. Messages are sent including the address information to be sent from output channel. Network manager delivers the message to the addressed input channel. Figure 5 shows the conceptual image how input/output channels are connected in our simulator.

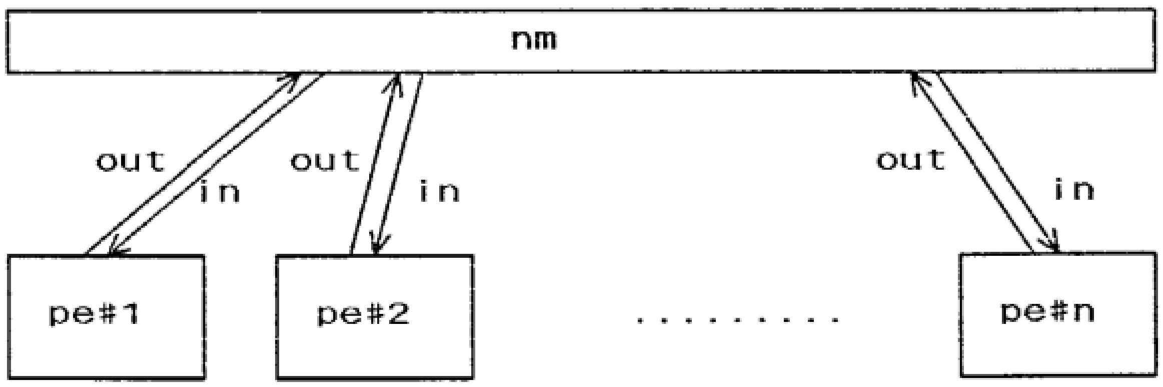

Figure 5 Channel Connection in Multi PE Simulator

The simulation of the grid-network behavior is fairy easy. We just need to specify "nm" more realistically.

5. Executing FGHC code

We have already mentioned that pre-processed FGHC program is executed inside each PE in our simulator. We show how these pre-processing will take place by a simple example. We consider the "merge" example in section 3 again.

The first thing the user must do is to add "pragma" to source program. "Pragma" specifies how the program should be executed in a multi-PE environment. Pragma shows just the execution control information, and does not effect the semantics of the original source program.

In relate to "pragma" specification strategy, we do not have enough experiences yet. Therefore, we are currently testing various "pragma" methods. Currently, the most dominant "pragma" specification method is the one proposed by Chikayama [Chikayama 85]. However, we assume Shapiro-like "pragma" [Shapiro 83b] for explanation simplicity in this paper.

The source code which are added "pragma" must be pre-processed to the executable code. We have adopted prolog code as our target code. Figure 6 shows the pre-processing example.

```
/* merge in FGHC  */

merge([A|X], Y, Z) :- true | Z = [A|W], merge (X, Y, W).

merge(X, [A|Y], Z) :- true | Z = [A|W], merge (X, Y, W).

merge([], Y, Z) :- true | Z = Y.

merge(X, [], Z) :- true | Z = X.


/* merge in FGHC with Pragma */

merge([A|X], Y, Z) :- true | Z = [A|W], merge (X, Y, W)@up.

merge(X, [A|Y], Z) :- true | Z = [A|W], merge (X, Y, W)@down.

merge([], Y, Z) :- true | Z = Y.

merge(X, [], Z) :- true | Z = X.


/* compiled merge program */

merge(1, $(A1,A2,A3), Result1,

        [$(ulist(A3, A, W)),

         throw( merge(X, A2, W), up) |T], T) :-

                                    ulist(guard, A1, A, X, Result1), !.

merge(2, $(A1,A2,A3), Result2,

        [ulist(A3, A, W),

         throw( merge(A1, Y, W), down) |T], T) :-

                                    ulist(guard, A2, A, Y, Result2), !.

merge(3, $(A1,A2,A3), Result3,

        [unify(A2, A3)|T], T) :-

                                    unil(guard, A1, Result3), !.

merge(4, $(A1,A2,A3), Result4,

        [unify(A1, A3)|T], T) :-

                                    unil(guard, A2, Result4), !.
```

Figure 6   FGHC Code Pre-processing Example

The prolog compiled code may hard to understand. However, these compilation techniques are already "established" implementation techniques which can be seen in [Ueda 85a, Murakami 85, Tanaka 85b, Tanaka 86]. You may notice that every compiled predicate carries scheduling queue as its argument.

6. Inter-PE communication

One of the most important operation in distributed computing is the communication between PEs. We have prepared the following 5 predicates as system predicates:

(1) send_goal (PE_ID, Goal)

Send goal to the PE specified by PE_ID.

(2) get_value(C_Var, Result)

Request the value of C_Var. The value is returned to "Result" by "reply_result" message. Usually used executing passive part of the source program.

(3) unify(C_Var, Value, Result)

Request the unification of C_var and Value to PE which C_Var belongs to. Value must be instantiated at run time. The computation result is returned to "Result" by "reply_result" message. Usually used executing active part of the source program.

(4) unify_channels(C_Var1, C_Var2, Result)

Request the unification of C_var1 and C_Var2 to PE which C_Var1 belongs to. The computation result is returned to "Result" by "reply_result" message. Usually used executing active part of the source program.

(5) reply_result(Result, Value)

Returns the unification result or computation result to "Result." Value must be instantiated at run time.

7. Sending/Receiving goals

Since our system does not have global address space, sending of a goal which includes variables to other PE needs a little complicated mechanism.

The sender "PE" transforms the variable X which is included in the sending goal to "$VAR(ID, N)," where ID denotes ID number of sender PE and N is a newly generated number. This corresponding information is registered in the variable management table with the form [X, $VAR(ID, N), _].

The receiver "PE" generates a new variable X' from the form "$VAR(ID, N)" and contain this correspondence to variable management table in form [X', $VAR(ID, N), "on-asking flag"]. The variable in the received goal is also replaced by $cha(X', "on-asking flag"). Here $cha(X', "on-asking flag") is called "channel-variable."

Figure 7 shows the case where a goal f(X,Y) is thrown from PE#1 to PE#2.

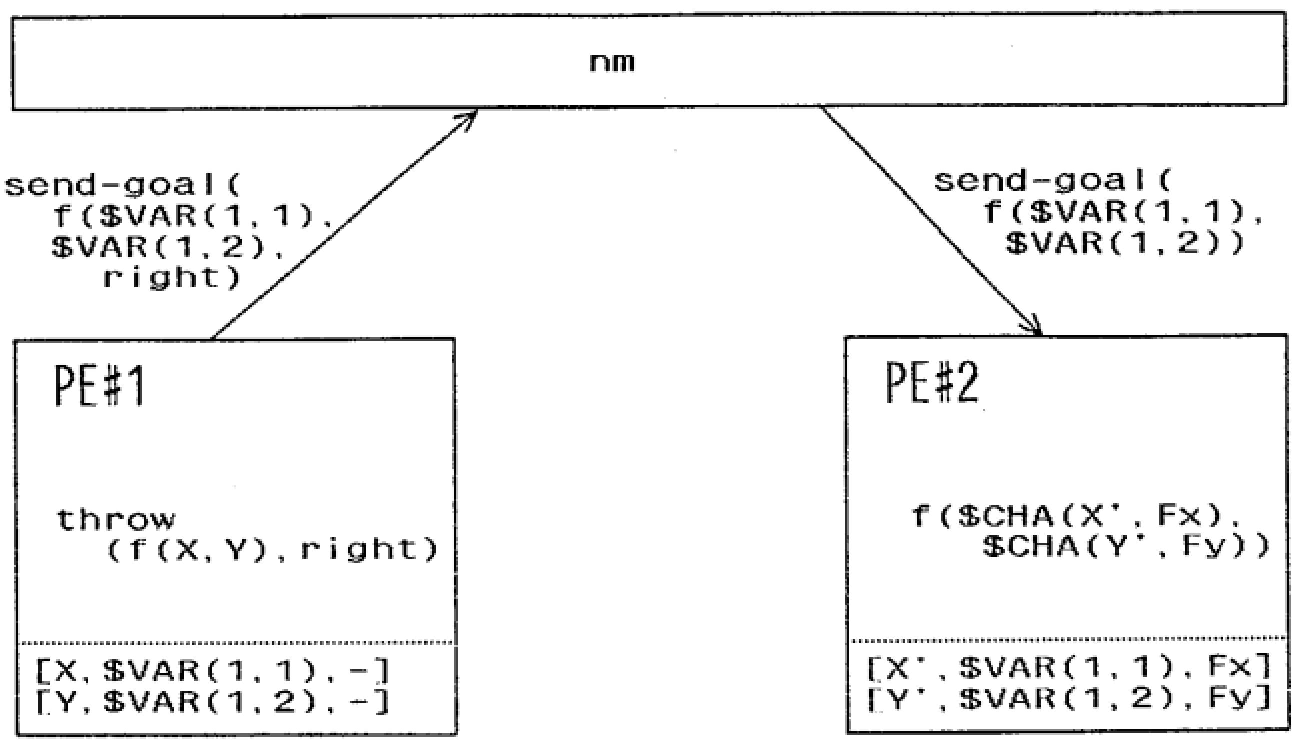
nm
send-goal(f($VAR(1,1), $VAR(1,2), right)
send-goal(f($VAR(1,1), $VAR(1,2))
PE#1
throw (f(X,Y),right)
[X,$VAR(1,1),-]
[Y,$VAR(1,2),-]
PE#2
f($CHA(X',Fx), $CHA(Y',Fy))
[X',$VAR(1,1),Fx]
[Y',$VAR(1,2),Fy]

Figure 7 Sending/Receiving Goals

8. Distributed unification

We have introduced "channel-variable" in the previous section. This "channel-variable" is used to carry out the distributed unification. When program execution gets across the unification with channel variables, messages to other PE must be generated.

The distributed unification algorithm can be summarized as follows:

(1) Unification in the passive part.
In FGHC, we cannot instantiate global variable before the clause is committed. Only the reference of the value is possible for the global variable. Therefore, when unification operation with the "channel-variable" is broke out in the passive part, it issues the

"get_value" message and this unification is suspended until the value is returned by "reply_result" message.

(2) Unification in the active part.
In the active part, there is no regulation as in the passive part. We can instantiate a value to the global variable.

(a) Unification between a variable and a channel-variable
If the variable is undefined, the channel-variable itself is substituted as its value. If the variable has already been instantiated, "unify" message is issued. This unification is resumed when the value is returned by "reply_result" message.

(b) Unification between two channel-variables.
Unification between two channel-variables issues the "unify_channels" message. This message is sent to the PE which the first channel_variables belongs to. Then this message is processed as exactly the same as in (a).

Figure 8 shows the typical case of unification between PE#1 and PE#2.

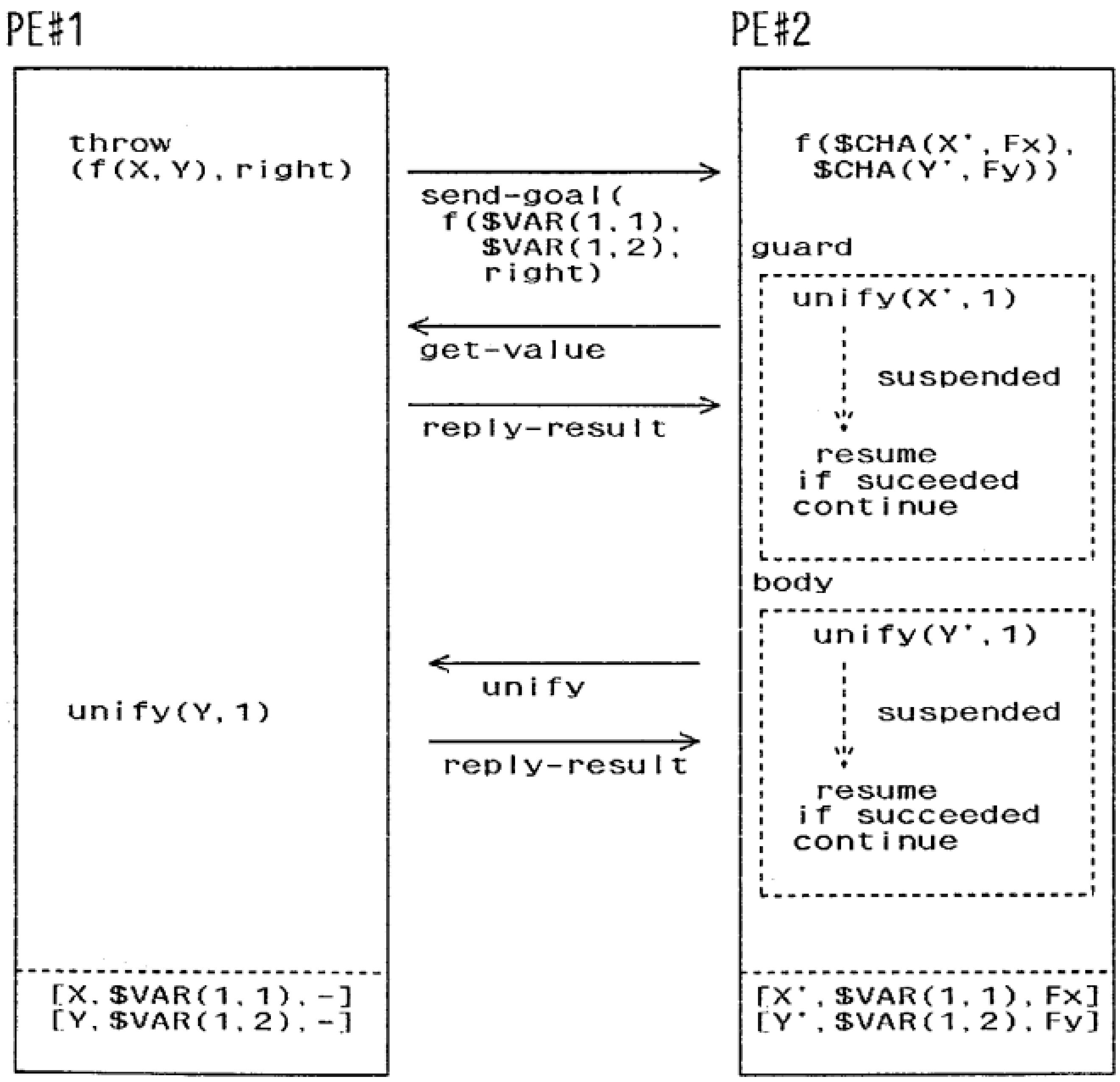

Figure 8 Typical Case of Distributed Unification

9. Coding and implementation details

We asked the actual coding of the simulator to Etsuo Ono, Satoru Torii and Yuri Ohara at Software Laboratory, Fujitsu Laboratories. They took approximately 3 months to finish up the simulator, including the design of detailed specification and the debugging. The simulator is approximately 1000 lines long in prolog, except the pre-processing compiler part. We have tested several sample programs on this simulator, such as merge, primes, etc.

As far as we know, there was not much difficulty in coding. We imagine this comes from the language simpleness of FGHC. However, unification algorithm we reached was pretty complex, since it includes "channel-variables" and "bind-hook" and "resume" mechanism.

10. Concluding remarks

We described the outline of our Multi-PE simulator. There are two things that must be done. One is the actual construction of Multi-PSI system. Currently, the enthusiastic efforts for Multi-PSI system are in progress by Toshihiko Miyazaki, Kazuo Taki, Takashi Chikayama, and other members of ICOT. ICOT has already finished up the design of connecting hardware. The actual hardware of version 1 will be completed by April 1986.

The other direction is the research of operating system written in FGHC. The operating system is called "PIMOS" and it operates on Multi-PSI system. Our claim is that most problems in parallel systems exist in "software". PIMOS is written in FGHC and aims to examine such software problems. PIMOS , for example, tries to examine process allocation, load balancing and scheduling problems. Chikayama's Computing Power Uniform Distribution Model [Chikayama 85] is one such example. In that sense, our direction is quite opposite to Shapiro's Logix [Shapiro 85], which is concentrated on user "services" based on sequential implementation.

11. Acknowledgments

This research was carried out as a part of the Fifth Generation Computer Project. The authors would like thank Kenichiro Murakami, NTT, and Kazunori Ueda, ICOT, for their useful comments and suggestions. We also would like to thank Etsuo Ono, Satoru Torii and Yuri Ohara at Software Laboratory, Fujitsu Laboratories for their implementation efforts.

[References]

[Chikayama 84] Chikayama, T. : ESP Reference Manual. ICOT Technical Report TR-044, ICOT, 1984.

[Chikayama 85] Chikayama, T. : Computing Power Uniform Distribution Model. ICOT Internal Memo, 1985.

[Clark 85] Clark, K., Gregory, S.: PARLOG: Parallel Programming in Logic. Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, Revised June 1985.

[Honiden 86] Honiden, S. et al.: MENDEL: Prolog based concurrent object-oriented language. Compcon 86 spring, San Francisco, March 1986.

[Murakami 85a] Murakami, Kunio et al.: Research on Parallel Machine Architecture for F.G.C.S.. Computer, vol.18, No.6, June 1985.

[Murakami 85b] Murakami, Kenichiro: The study of unifier implementation in multi-processor environment. Multi-SIM study group internal document, ICOT, 1985, in Japanese.

[Pereira 84] Pereira, L.M. et al.: Delta-prolog : A Distributed Logic Programming Language. Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT, pp.283-291.

[Shapiro 83a] Shapiro, E.: A Subset of Concurrent Prolog and its Interpreter. ICOT Technical Report TR-003, ICOT, 1983.

[Shapiro 83b] Shapiro, E.: Lecture Notes on The Bagel: a Systolic Concurrent Prolog Machine. ICOT Technical Memorandum TM-031, ICOT, 1983.

[Shapiro 85] Shapiro E et al: Logix User Manual for Release 1.1. Weizmann Institute, Israel, 1985.

[Taki 84] Taki, K. et al. : Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). Proc. International Conference on Fifth Generation Computer Systems 1984, ICOT, pp.398-409.

[Tanaka 85a] Tanaka, J. et al.: AND-OR Queuing in Extended Concurrent Prolog. Proc. the Logic Programming Conference '85, ICOT, pp.215-224, in Japanese. English version is to appear in Lecture Notes in Computer Science, Springer-Verlag. Also available as ICOT Technical Memorandum TM-120, ICOT, 1985.

[Tanaka 85b] Tanaka, J. et al.: Single Queue Compilation in Extended Concurrent Prolog. Mathematical Methods in Software Science and Engineering, RIMS Kokyuroku, Research Institute for Mathematical Science, Kyoto University. Also available as ICOT Technical Report TR-139, ICOT, 1985.

[Tanaka 86] Tanaka J et al.: Compiling Extended Concurrent Prolog -Single Queue Compilation-. Proc. European Symposium on Programming 86, March, 1986, University of Saarlandes, West Germany.

[Ueda 85a] Ueda, K., Chikayama, T.: Concurrent Prolog Compiler on Top of Prolog. Proc. of 1985 Symposium on Logic Programming, pp.119-126, 1985.

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