ICOT Technical Report: TR-042

TR-042

An Approach to a Parallel Inference Machine Based on Control-Driven and Data-Driven Mechanisms

by
Rikio Onai, Moritoshi Asou and Akikazu Takeuchi

January, 1984

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

An Approach to a Parallel Inference Machine Based on Control-Driven and Data-Driven Mechanisms

January, 1984

Rikio Onai, Moritoshi Asou, and Akikazu Takeuchi

Institute for New Generation Computer Technology

I. Introduction

This paper discusses the design of a distributed Parallel Inference Machine called PIM-F (F meaning Fusion mechanism) based on control-driven and data-driven mechanisms.

The Institute for New Generation Computer Technology(ICOT) is carrying out research and development on a new generation computer system based on a logic language. Along with the major characteristics, such as unification and logical variables,etc., a logic language has other features at program execution level such as high parallelism and "DO NOT CARE/DO NOT KNOW non-determinism"[1].

For construction of a total logic programming system, it is necessary to provide within the logic language concurrent process description facilities[4],[8] for operating systems and I/O device handling.

Therefore the PIM-F machine, which is designed to be a direct execution logic machine, has support for concurrent process functions and parallel "DO NOT KNOW non-determinism" processing function, as indispensable functions. It is worth noticing that the target languages of the machine are parallel Prolog (parallel "DO NOT KNOW non-determinism" processing[2],[3]) and Concurrent Prolog[4],[5](concurrent process control).

ICOT's PIM-F machine possesses the following features:

1. PIM-F is a distributed computer system.

In Prolog and Concurrent Prolog programs there is high parallelism. Therefore each unit in PIM-F is divided into banks to achieve highly parallel processing, and a tagged-packet communication system is adopted to realize highly distributed processing.

2. Partial copy strategy is adopted.

Although there are several goals in a body of a clause, only a reducible goal is copied and is sent to a unification unit in order to shorten the length of a packet going through networks.

3. PIM-F adopts a control-driven mechanism as the basic execution mechanism.

A Prolog program and a Concurrent Prolog program are executed by generating resolvents from parent clauses. A reducible goal (which is one of parents clauses) is indicated by : sequential-AND operator, parallel-AND operator, and commit operator. In other words, the goal reductions are driven by these operators. This is what motivated us to adopt a control-driven mechanism as the basic execution mechanism.

Programs are executed as follows;

(1) In a Prolog program, AND-literals are executed sequentially from left to right. This is because there is a complicated consistency check in solutions of shared variables among AND-literals. OR-clauses, however, are executed in parallel.

(2) In a Concurrent Prolog program, AND-literals connected together by parallel-AND operators are executed in parallel, as are OR-clauses. "DO NOT CARE non-determinism" is realized by commit operators.

4. PIM-F adopts a data-driven mechanism as a synchronization mechanism for concurrent processes.

Concurrent processes can send messages by instantiating shared variables in Concurrent Prolog. We call these shared variables channel variables and the other logical variables non-channel variables hereafter. The busy-wait method, which is one method of implementing channel variables, increases network traffic due to frequent accesses to memories by consumer processes, when realizing concurrent process execution on a distributed parallel machine. Therefore, we introduce a data-driven mechanism as a synchronization mechanism for concurrent processes.

There is a method which treats the channel variables in the same manner as non-channel variables, which are not shared among concurrent processes. However, in this method, even a memory cell which stores a non-channel variable has to possess a field that keeps pointers to suspended consumer processes. Therefore, surplus memory is needed. For this reason we distinguish channel variables from non-channel variables.

For achieving both the data-driven mechanism and the distinction of the above two kinds of variables, we introduce Message Boards in each distributed unit. Message Board is a kind of I-structured storage[12]. As a result, as soon as a producer process binds a value to a channel variable in a Message Board, messages are sent (using the message passing method) to suspended consumer processes which are to be driven to reduction.

In summary, our proposition is that of a Parallel Inference Machine (PIM-F) based on control/data driven mechanisms[6] where a data-driven synchronization of concurrent processes is executed through Message Boards. In addition, we have shown that PIM-F can execute "DO NOT KNOW non-determinism" in parallel.

This paper describes the PIM-F computation model, conceptual configuration, and individual units in PIM-F(particularly the Message Board). It also shows the execution of sample programs in Prolog and in Concurrent Prolog on the PIM-F machine.

II. PIM-F Computation Model

The target languages of the PIM-F machine are parallel Prolog(parallel "DO NOT KNOW non-determinism" processing) and Concurrent Prolog. A Prolog program and a Concurrent Prolog program are executed by generating resolvents from parent clauses. A reducible goal (which is one of parents clauses) is indicated by : sequential-AND operator, parallel-AND operator and commit operator. In other words, the goal reductions are driven by these operators. Therefore we adopt a control-driven mechanism as the basic execution mechanism.

A PROCESS-TREE which is a tree structured network of processes is created when input goals are given, and vanishes when all the input goals are solved. There are two worlds in the computation model.

[1] Non-Guard World

A goal is executed in OR-parallel and AND-sequential. Since there are no parallel-AND operator and no commit operator (no guard) in this world, goal reductions are driven by sequential-AND operator.

* Definitions of GOAL-Process Creation:

(1) A GOAL-Process is created when input goals are first given, and consists of the given input goals.

(2) A GOAL-Process is created when a goal is reduced, and consists of the results of reduction. Consequently, a GOAL-Process is made up of N-number of goals (N>=1). We will refer to these goals as sub-goals hereafter. A sub-goal is the smallest constituent unit of this computation model.

(Sub-goal states)

1) Not-reducible: A state in which a sub-goal is not (yet) subject to reduction (with the exception of "dead" sub-goals, not-reducible sub-goals do not include the left most sub-goal.).

2) Reducible: A state in which a sub-goal is reducible and is waiting to be picked up.

3) Active: A state in which a sub-goal is reducible, has already been picked up and is in the process of being reduced. When a sub-goal is reduced, it creates M-number of new child GOAL-Processes, whose parent is the GOAL-Process containing that sub-goal (Upon implementation, a child GOAL-Process communicates with the reduced sub-goal in its parent GOAL-Process by means of a pointer). The letter M stands for the number of OR forks, and M=0 indicates a sub-goal's failure at OR-reduction.

The reduction consists of OR-reduction and AND-reduction. OR-reduction means unification between a sub-goal and the corresponding head predicate. If OR-reduction succeeds, unification information of OR-reduction is transferred to the body predicate and the resolvent is generated. This is AND-reduction.

These M-number of child GOAL-Processes are called sibling GOAL-Processes. When a sub-goal unifies with a unit clause, a value returns directly to the appropriate sub-goal without creating a new child GOAL-Process. Child GOAL-Processes are also made up of N-number of sub-goals (N>=1).

4)Dead: A state in which a sub-goal has been completely solved. In other words, when OR fork count (described in Section III.[1] ii)) is reduced to zero, the sub-goal state becomes dead. (Upon implementation, the memory cell occupied by this sub-goal becomes garbage.)

* Definition of GOAL-Process Termination: A GOAL-Process terminates whenever all of its sub-goals have been solved (including failures).

* GOAL-Process states

1) Active: A state in which a GOAL-Process has been created and has yet to terminate.

2) Dead: A state in which a GOAL-Process has terminated.

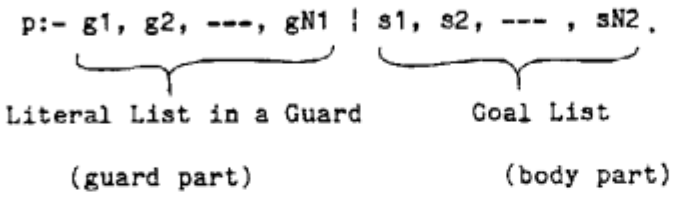
[2] Guard World

GOAL-Processes are executed in OR-parallel. Reduction of AND-literals are driven by sequential-AND operators, parallel-AND operators and commit operators. The user can specify these operators.

* Definitions of GOAL-Process Creation:

(1) A GOAL-Process is created when input goals are first given, and consists of the given input goals. Each individual input goal is a sub-goal of the GOAL-Process.

(2) A GOAL-Process is created when an input goal(a sub-goal) is reduced, and consists of the results of reduction. Consequently, in this case, a GOAL-Process is made up of N1-number of literals in a guard and N2-number of goals (N1>=0; N2>=0). Both literals in a guard and goals are called sub-goals. The syntax of a Guard World clause is given below.



(3) When a literal in a guard or a goal (a sub-goal) is reduced, a GOAL-Process is created which consists of the results of reduction. Consequently, a GOAL-Process is made up of literals in a guard and goals.

(Sub-goal states)

1) Not-reducible: A state in which a sub-goal is not (yet) subject to reduction.

2) Reducible: A state in which a sub-goal is reducible and is waiting to be picked up.

3) Active: A state in which a sub-goal is reducible, has already been picked up and is in the process of being reduced. When OR-reduction is attempted on a reducible sub-goal, that sub-goal happens to be "suspended" due to the limitations of the read-only-annotation (This is described in more detail later in this paper).

When a literal in a guard or a goal (a sub-goal) is reduced, M-number of new child GOAL-Processes are created, which have the GOAL-Process containing the sub-goal as their parent GOAL-Process (Upon implementation, a child GOAL-Process communicates with the reduced sub-goal in the parent GOAL-Process by means of a pointer.). M stands for the number of OR forks, and M=0 indicates that the sub-goal has failed at OR-reduction. These M-number of child GOAL-Processes are called sibling GOAL-Processes.

Child GOAL-Processes are also made up of a guard and goals. In a Guard World, goal reduction is driven by a commit operator. Since sibling GOAL-Processes are killed at the time of commitment, at most only one child GOAL-Process can survive.

4) Suspend: A state in which a sub-goal can unify, except when a read-only variable has unified with a non-variable term in the head of the clause. When another sub-goal binds a value to the appropriate read-only variable, the suspended sub-goal once again becomes reducible.

5) Dead: A state in which a sub-goal has been completely solved. In other words, when OR fork count (described in Section III.[1] ii)) is reduced to zero, the sub-goal state becomes dead. (Upon implementation, the memory cell occupied by the dead sub-goal becomes garbage.)

* Definitions of GOAL-Process Termination:

(1) If either a guard or a goal fails, the GOAL-Process containing the guard or the goal returns a failure message to the parent GOAL-Process and terminates.

(2) A GOAL-Process terminates when its guard and all its goals have been solved (i.e. when a value is returned to the parent GOAL-Process, or when a value is written in the Message Board, or both).

(3) A GOAL-Process terminates when a guard in the GOAL-Process has succeeded and the commit operator has been reached, but the other sibling GOAL-Process has been driven by a commit operator ahead of it.

*GOAL-Process states

1) Active: A state in which a GOAL-Process has been created and has yet to terminate.

2) Dead: A state in which a GOAL-Process has terminated.

The relation among a PROCESS-TREE, its GOAL-Processes and sub-goals is shown in Fig.1.

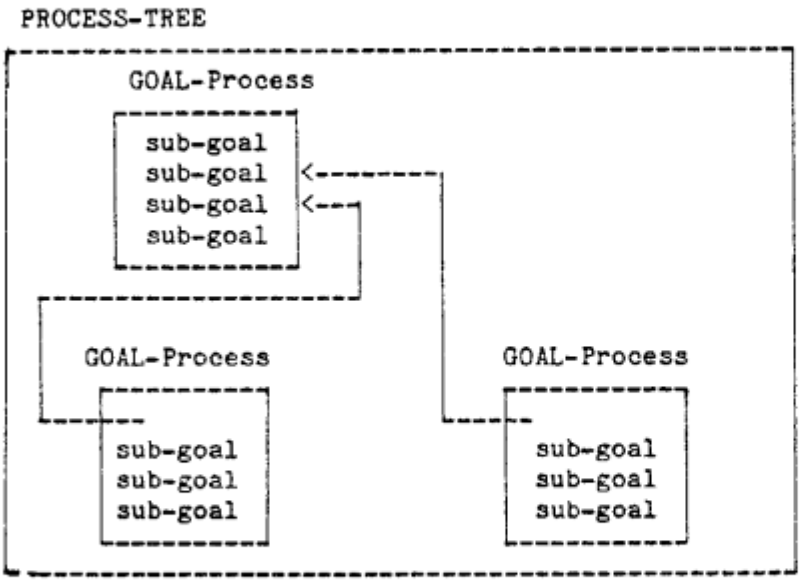


Fig.1

III. PIM-F Architecture(See Fig.2)

This section describes the architecture of PIM-F.

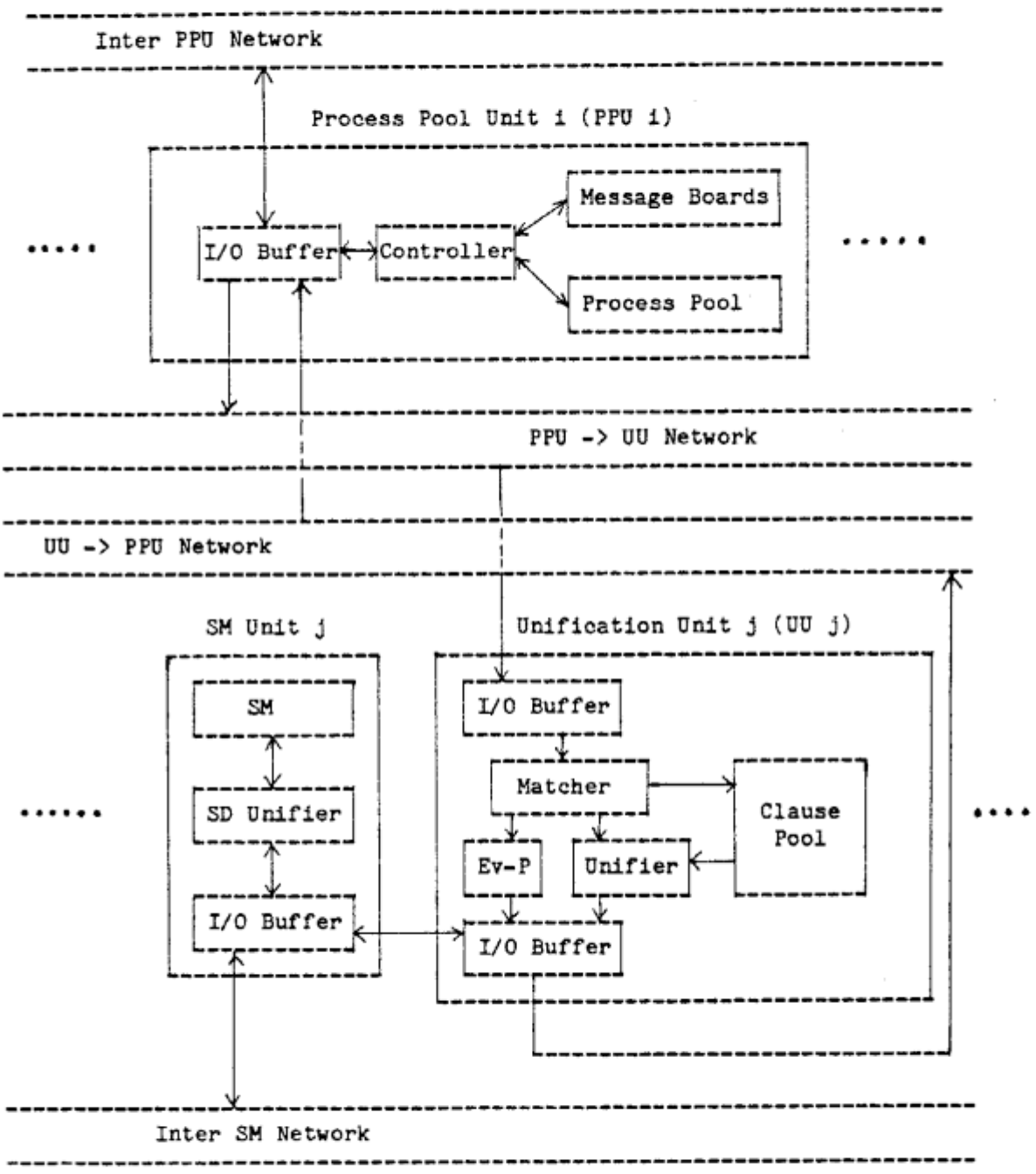


Fig.2 Conceptual Configuration of PIM-F Architecture

(Ev-P is a processor for evaluable predicates and SM is a Structure Memory)

[1] Process Pool Unit (PPU)

This unit consists primarily of a Process Pool and Message Boards. The Process Pool stores GOAL-Processes, and Message Boards are used as synchronization support mechanisms for concurrent processes.

This section describes the Process Pool, Message Boards and the internal format of a Guard World GOAL-Process in a Process Pool.

i) Process Pool (PP)

* A Process Pool stores GOAL-Processes and the relation among GOAL-Processes, i.e. the historical information about reduction. The relation among GOAL-Processes is represented by reversed pointers, which define parent GOAL-Processes.
* Even if there are several sub-goals in a GOAL-Process, in order to shorten the length of a packet going through the network, only reducible sub-goals are picked up by the Controller and sent to Unification Units(UUs). On picking up, the Controller refers to the following tags of the sub-goal.
* Level tag and Node tag:

- Level tag

This tag shows the node level of the GOAL-Process(its depth from the root) in a PROCESS-TREE. Level tag number is increased by one at AND-reduction.

- Node tag

Node tags are numbers given successively by a Unifier in a UU.

These tags are used to identify each sub-goal and to control the pick-up of reducible sub-goals.

ii) Message Boards

A brief explanation of a guard and read-only-annotations in Concurrent Prolog is given prior to our discussion of Message Boards.

(1) A guard and read-only-annotations

A guard consists of AND-literals on the left side of a commit operator which is represented by "|" on the right side of a clause. A read-only-annotation, represented by "?", is added to a variable, such as in the case of "X?". A variable with a read-only-annotation is called a read-only-variable.

Read-only-variables act as channels between processes which correspond to sub-goals in the PIM-F computation model. A process which has a read-only-variable (consumer process with respect to the variable) cannot instantiate the variable (channel variable) and is suspended until another process (producer process) instantiates the channel variable (i.e., sends out a message).

Once a goal has been set, corresponding clauses are searched for in parallel and the first clause that succeeds in unifying the guard is selected. Then the unified information in the guard is released. Its body part (AND-literals on the right side of a commit operator) then becomes a resolvent. As a result, even if another alternative clause should later succeed in unifying its guard, that clause is erased. In this sense, the commit operator functions as a cut symbol.

(2) Introduction of Message Boards

Concurrent Prolog controls concurrent process by means of channel variables. The busy-wait method, which is one method of implementing channel variables, increases network traffic due to frequent accesses to memories by consumer processes, when realizing concurrent process execution on a distributed parallel machine. Therefore, we introduce a data-driven mechanism

as a synchronization mechanism for concurrent processes.

There is a method which treats the channel variables in the same manner as non-channel variables. However, in this method, even a memory cell which stores a non-channel variable has to possess a field that keeps pointers to suspended consumer processes. As a result, surplus memory is needed. Therefore we distinguish channel variables from non-channel variables.

In order to achieve both the data-driven mechanism and the distinction of the above two kinds of variables, we implement Message Boards in each distributed unit.

There are two kinds of channel variables.

1. There are read-only-variables in channel variables that are shared among concurrent processes, which are connected by parallel-AND operators. In this case, concurrent processes which have read-only-variables are consumer processes with respect to this channel variable.

2. There are no read-only-variable in channel variables that are shared among concurrent processes, which are connected by parallel-AND operators. In this case, whether the channel variable is an input or an output is unknown at the time of compilation.

Values sent through channel variables and the channel variables are written into a Message Board (Fig. 3). A channel variable is represented by a channel identifier and a variable identifier(level tag number, node tag number,etc.). A Message Board is a ring-shaped board for each channel identifier. When structure data (usually [a value|next channels]) are sent through a channel variable, variables in the structure data are registered into a Message Board as new channel variables. The sub-goal performs hashing based on the channel identifier and accesses the appropriate Message Board.

When a reducible sub-goal, which has been sent to a Unification Unit, is suspended due to the read-only-annotation and is sent back to the Process Pool, this consumer sub-goal goes to see if a message has already been sent from a producer sub-goal to the Message Board. If a message has arrived, the channel variable of the consumer sub-goal is replaced by the message.

If no message has arrived, the sub-goal links itself to the Suspend Process List. When the producer sub-goal sends a message, i.e. binds a value to the channel variable, the message is written into the appropriate cell of the Message Board. If there is a suspended consumer sub-goal linked to the Suspend Process List, the message is sent to the consumer sub-goal and the consumer sub-goal is to be driven to reduction.

As mentioned above, PIM-F adopts a data-driven mechanism as a synchronization mechanism for concurrent processes.

A head pointer and a tail pointer are used to indicate the area of a ring-shaped Message Board. A reference count of a channel variable controls the head pointer.

Head pointer: When the reference count becomes zero as the result of decrement, the head pointer is increased by one.

Tail pointer: This pointer indicates the Message Board cell into which the next channel variable will be registered. When a channel variable is registered, the tail pointer is increased by one.

The increment and decrement rule of the reference count is as follows (READ-VAR and CHANNEL-VAR are described in the section dealing with the internal format of a GOAL-Process in a Process Pool):

1. In such a case that a channel variable is a READ-VAR.

When the READ-VAR exists in a newly created(committed) sub-goal, plus 1.

When a value is read by a consumer process, minus 1.

When a sub-goal which includes the READ-VAR terminates, minus 1.

2. In such a case that a channel variable is a CHANNEL-VAR.

When the CHANNEL-VAR exists in a newly created(committed) sub-goal,plus 1.

When a value is written or read, minus 1.

When a sub-goal which includes the CHANNEL-VAR terminates, minus 1.

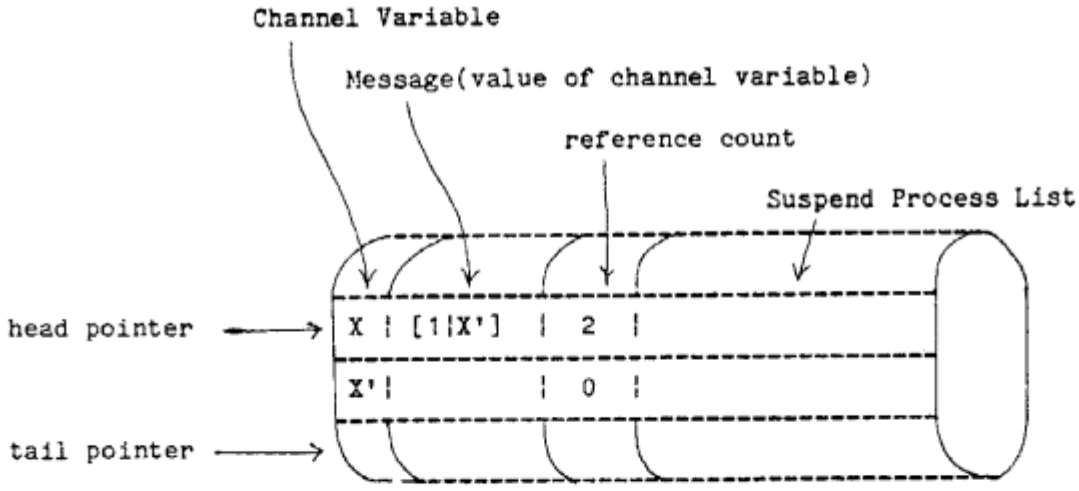


Fig. 3. Message Board

(3)Internal Format of a Guard World GOAL-Process in a Process Pool

This internal format is shown in Fig. 4.

* C-tag: The first child GOAL-Process to reach a commit operator turns this tag ON. If the tag is already ON, the OR fork count is reduced by one and the GOAL-Process becomes dead.

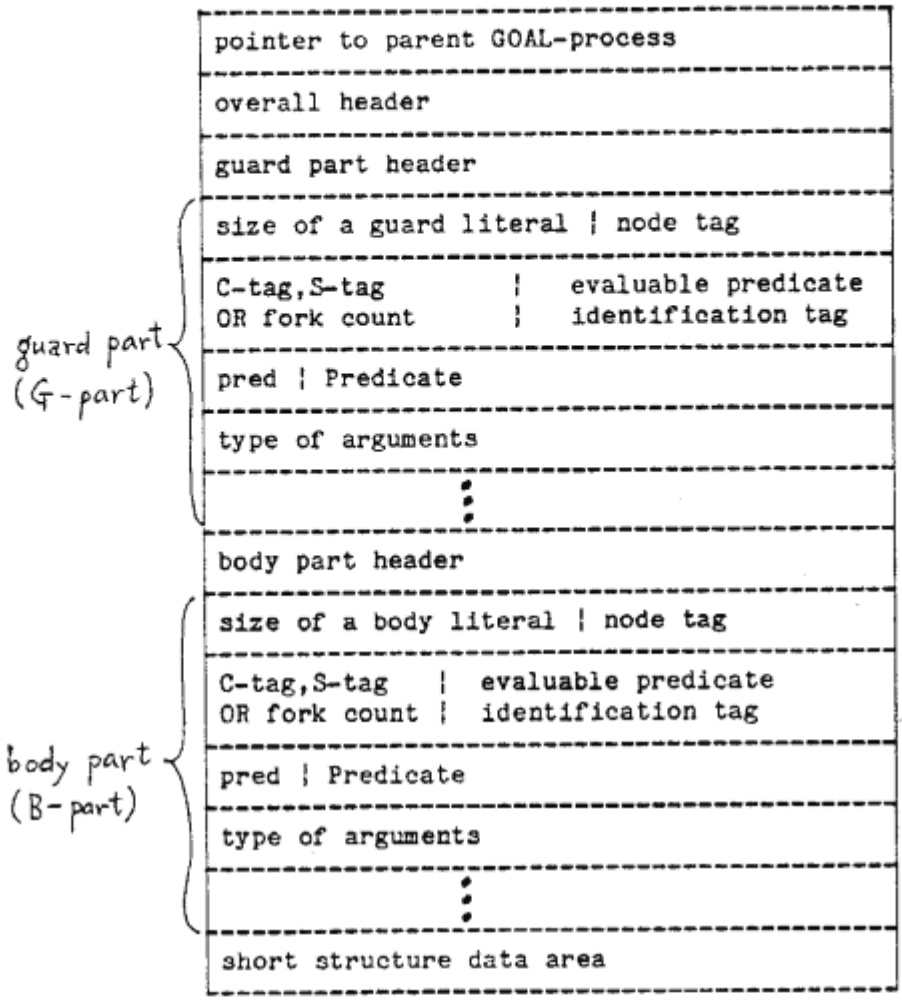


Fig. 4. Internal Format of a Guard World GOAL-Process in a Process Pool

* OR fork count: The OR fork count is made known after OR-reduction, and reduced by one each time a failure occurs in the guard of an OR-forked GOAL-Process or when the guard succeeds (when the commit operator is reached). When the count is reduced to zero, the sub-goal becomes dead (garbage).
* S-tag: State tag. States include not-reducible, reducible, active, suspend and dead(garbage).
* READ-VAR, WRITE-VAR and CHANNEL-VAR are newly established as types of arguments. READ-VAR, WRITE-VAR, and CHANNEL-VAR are channel variables and are used to distinguish channel variables from non-channel

variables. Furthermore, these provide the basis to judge whether a sub-goal is a consumer or a producer with respect to a variable.

READ-VAR: Read-only-variable

WRITE-VAR: Variable for sending out messages.

CHANNEL-VAR: Channel variable. Whether this variable is an input or an output is unknown at the time of compilation.

[2] Unification Unit (UU)

i) Matcher

A Matcher receives a reducible sub-goal from a PPU. If the sub-goal is an evaluable predicate which does not need unification, the Matcher sends it to an evaluable predicate processor. If not, the Matcher searches a Clause Pool for head predicates that match the sub-goal. If the search is successful, then clauses with corresponding heads are transferred to a Unifier.

ii) Unifier

A sub-goal and the corresponding clause are sent to a Unifier. The Unifier, in turn, unifies the arguments. The average number of arguments is approximately three, indicating that no significant benefit could be expected from parallel unification between arguments[5]. Therefore, there is a Unifier and unification processing is performed in sequential.

When unification is completed for a sub-goal and the corresponding clause, the Unifier organizes the resulting packet and sends it back to a PPU. When an argument is a long structure data, the Unifier generates a new packet consisting of the structure data arguments information and transfers it to a Structure Memory Unit.

There is another strategy[6]. A Unification Unit unifies a sub-goal and a head predicate in a Clause Pool(OR-reduction) and AND-reduction is executed in a Process Pool Unit. This strategy shortens the length of a UU->PPU network packet. However, a PPU must have AND-reduction function and may be overloaded. First we will adopt the former strategy and the next investigate the two to compare.

iii) Clause Pool

We adopt the strategy that each Clause Pool stores the same clauses. Therefore, reducible sub-goals in a PPU can be sent to UUs which have low load averages. In this strategy, the corresponding clauses are picked up sequentially from a Clause Pool. However, since the number of OR relation[*] is about three[9] in a program which has rules mainly, there is little possibility that a Matcher and a Unifier are overloaded by many corresponding clauses. We have to consider the support mechanism for executing programs which have facts mainly.

[*] If clauses have the same head predicate symbol and the same number of arguments, they are in OR relation to each other. The number of such clauses is called the number of OR relation. When there is a program described below, the number of OR relation is two.

```
append([],X,X).
append([W|X],Y,[W|Z]) :- append(X,Y,Z).
```

iv) Evaluable Predicate Processor

This processor executes at a high speed evaluable predicates which do not need unification. As the result of static analysis[5] of DEC-10 Prolog programs in ICOT, it is made known that about fifty percent of AND-literals are evaluable predicates. Therefore, an evaluable predicate identification

tag is introduced into a internal format of a sub-goal and a Matcher can detect evaluable predicates.

[3] Structure Memory Unit(SMU)

A Structure Data Unifier (SD Unifier) receives a packet from a UU and unifies the two terms by referring to the relevant memory word (two terms ,one is in a sub-goal and the other is in its corresponding head predicate.) Then the SD Unifier stores the result of unification into the relevant memory words and sends the addresses of the written or modified memory words back to the UU.

IV. Sample program execution

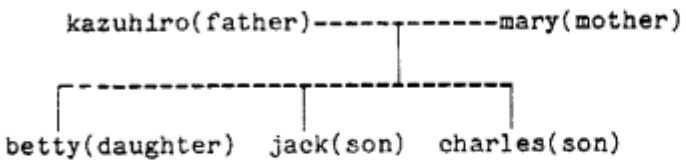
This section shows the execution of sample programs in Prolog and in Concurrent Prolog on the PIM-F machine. The internal formats of processes are shown in abbreviations. An asterisk (*) indicates a reducible state, Not-*, the not-reducible state, Act, the active state, D, the dead(garbage) state and SUS, the suspend state.

[1] An example of Prolog

The program is as follows :

```
son(S,P) :- child(S,P),man(S).
daughter(D,P) :- child(D,P),woman(D).
child(C,P) :-father(P,C).
child(C,P) :- mother(P,C).

father(kazuhiro,charles).
father(kazuhiro,jack).
father(kazuhiro,betty).

mother(mary,charles).
mother(mary,jack).
mother(mary,betty).

man(kazuhiro).
man(charles).
man(jack).

woman(mary).
woman(betty).
```

This program represents the family relation.



The goal is

```
?- son(X,mary).
```

To begin with, the below given GOAL-Process is stored in the Process Pool of Process Pool Unit 0 (abbreviated PP in PPU0) as the summit of the PROCESS-TREE. A node tag is omitted.

PP in PPU0

Total Header ¦ Top GOAL-Process(T G-P) , <X>
Literal Header ¦ level 1 (Lv 1) *
pred ¦ son
var ¦ X
atom ¦ mary

The PPU0 Controller picks up a reducible sub-goal in the GOAL-Process and the packet described below is sent to Unification Unit 0(UU0) (A free UU is selected.).

pointer to the sub-goal in parent GOAL-Process (abbreviated P-sub-goal)
Header ¦ Lv 1, <X>
pred ¦ son
var ¦ X
atom ¦ mary

The result of reduction in UU0 is as follows:

pointer to P-sub-goal
Total Header(T-Header) ¦ <X>
Literal Header(L-Header) ¦ level 2
pred ¦ child
var ¦ X
atom ¦ mary
Literal Header ¦ level 2
pred ¦ man
var ¦ X

The result is then sent to PPU1, for instance. The contents of the PPUs are as follows:

PP in PPU0

T-Header ¦ T G-P,<X>
L-Header ¦ Lv 1,OR 1, Act
pred ¦ son
var ¦ X
atom ¦ mary

PP in PPU1

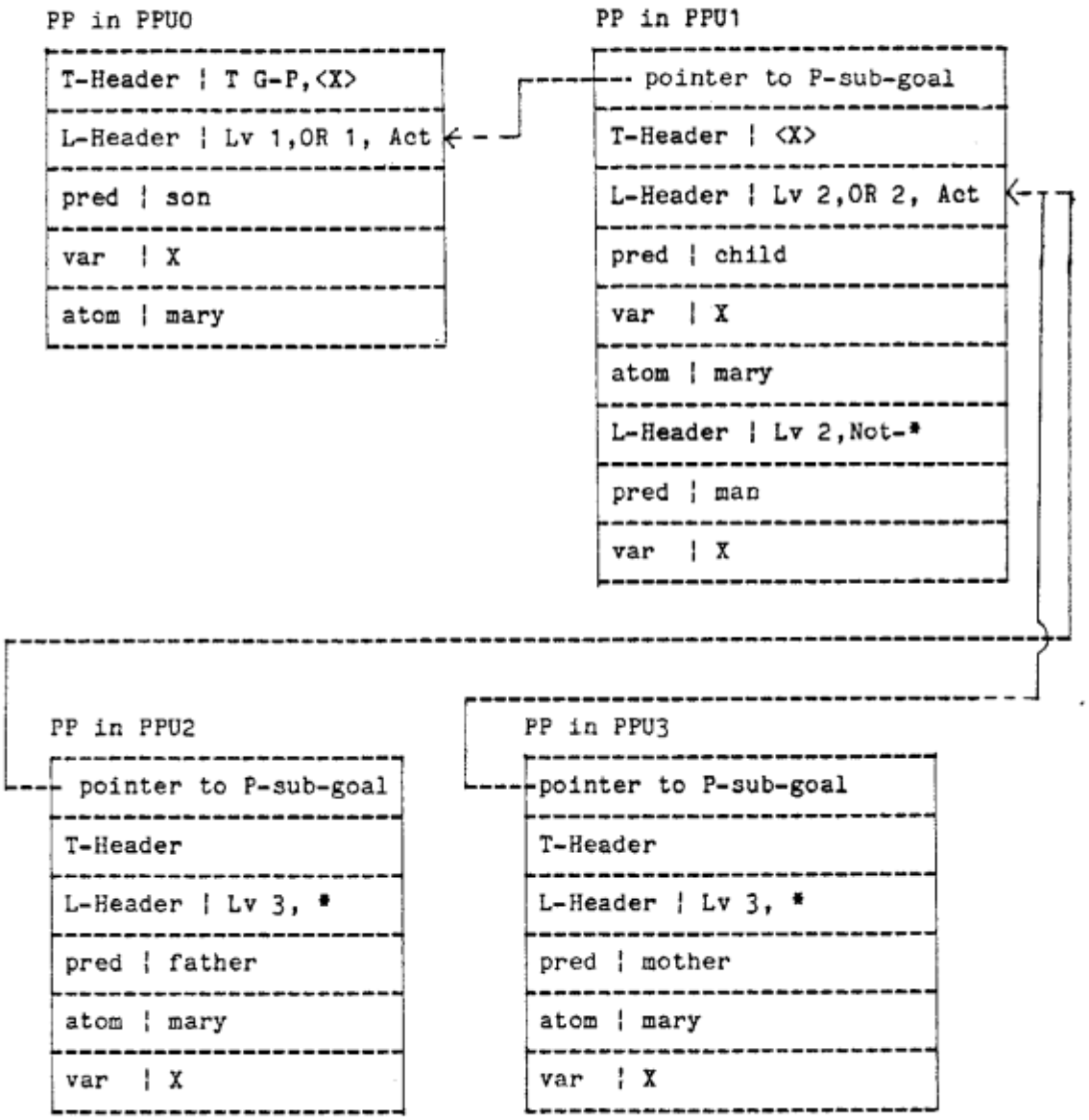
pointer to P-sub-goal
T-Header ¦ <X>
L-Header ¦ Lv 2, *
pred ¦ child
var ¦ X
atom ¦ mary
L-Header ¦ Lv 2,Not-*
pred ¦ man
var ¦ X

The OR fork count (ex. OR 1 in PP in PPU0) is made known after OR-reduction in a UU. Next, a reducible sub-goal "child" is sent to a free UU. The result of reduction in the UU is as follows:

pointer to P-sub-goal
Header ¦ Lv 3
pred ¦ father
atom ¦ mary
var ¦ X

pointer to P-sub-goal
Header ¦ Lv 3
pred ¦ mother
atom ¦ mary
var ¦ X

If these new GOAL-Processes are sent to PPU2 and PPU3, the contents of the PPUs become as follows:



Reducible sub-goals "father" and "mother" are picked up in parallel by Controllers and sent to free UUs.(Rest is omitted.)

As described above, since sibling GOAL-Processes do not kill each other and are executed in parallel, parallel "DO NOT KNOW non-determinism" processing is realized.

[2] An example of Concurrent Prolog

The following program represents the process "integers", which generates integers starting with 0, and the process "outstream", which receives and outputs integers.

The program is as follows (// is parallel-AND operator.) :

```
rule   integers(X,[X|N]) :- Y is X+1 | integers(Y,N).
       outstream([X|N]) :- write(X) | outstream(N?).
goal   integers(0,N)//outstream(N?).
```

The process "integers" takes on the internal format shown in Fig. 5 as a result of reduction. The process "outstream" (which is reducible at first) goes to the Message Board to check the value of N?, but is suspended and linked to the Suspend Process List since the value has not yet arrived. N=[0|N'] is not released since "Y is 0 + 1" has yet to be executed. When "Y is 0 + 1" is executed and becomes "Y is 1," the guard succeeds, turning the C-tag ON and releasing N=[0|N']. Therefore, [0|N'] is written as a value of the channel variable N into the Message Board and the next channel variable N' is registered into the Message Board. As soon as the value of N is written, "outstream", a consumer process linked to the Suspend Process List, is notified that N is [0|N']. Upon receiving of this notification, "outstream" becomes reducible (once again) (for the result of this reduction, see Fig.6). In other words, reduction of "outstream" is driven by the notification. Then, when "write (0)" is executed and "0" is generated as an

output, the outstream reaches the commit operator and N'=N" is released to the body part. Then "integers (1,N')" is reduced and "outstream (N'?)" is linked to the Suspend Process List of the Message Board.

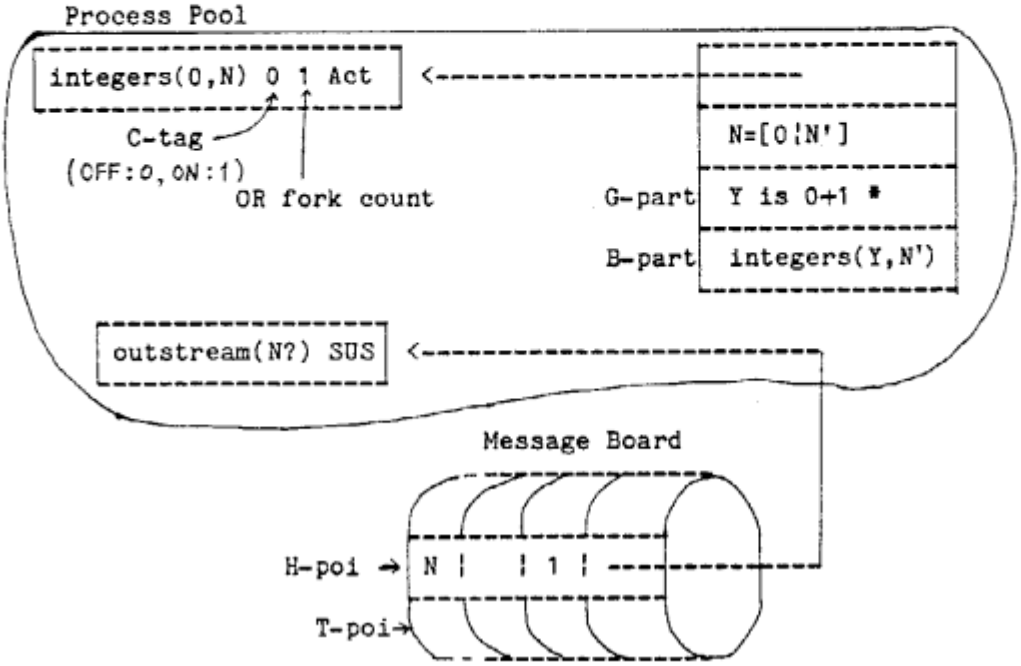


Fig. 5

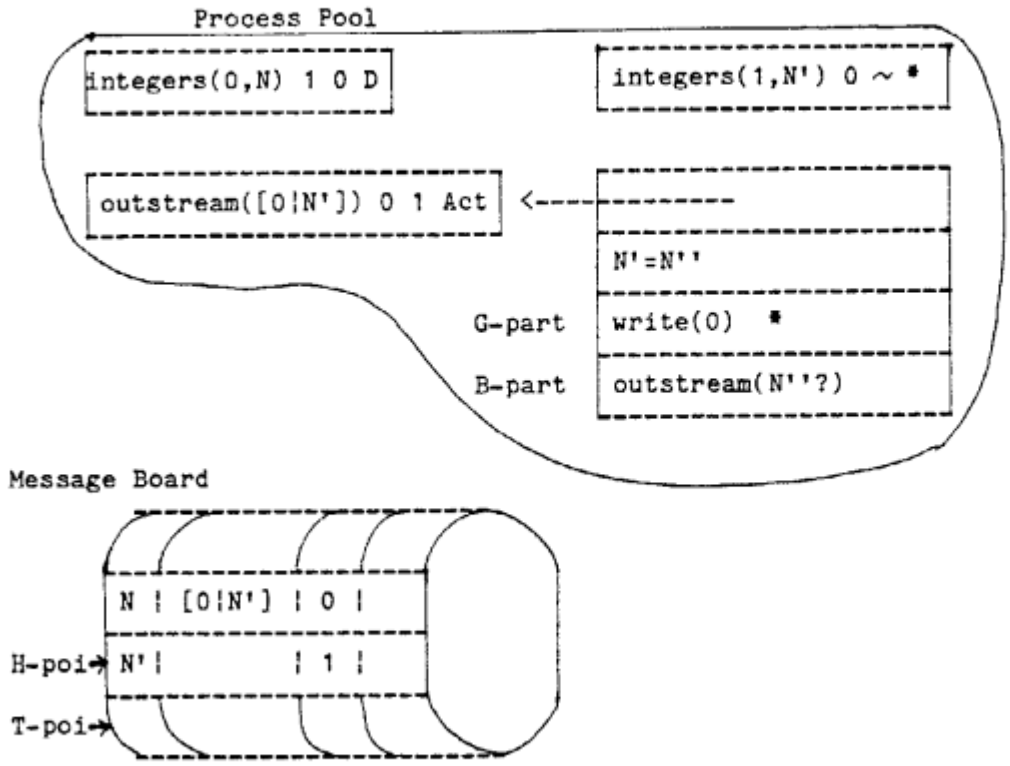


Fig. 6

V. Concluding remarks

This paper describes the design of a Parallel Inference machine (PIM-F) based on control-driven and data-driven mechanisms. We have confirmed that back communication and bounded buffer communication[11] can be executed using Message Boards and their internal formats, and that the Message Boards can function as a data-driven synchronization support mechanism for concurrent processes. Also we have shown that PIM-F can execute "DO NOT KNOW non-determinism" in parallel. Networks are under consideration.

We are developing a software simulator for the PIM-F machine in order to collect data on the status of the unit queue, the operation rate of each unit and the banking effect of the units. Kernel Language version 1 (KL1)[3] will be run on PIM-F. Roughly speaking, Set expression[4] is the bridge between a Guard World and a non-Guard World in KL1. Therefore we will consider a Set expression support mechanism in PIM-F.

Finally, thanks are due to Dr. K. Murakami, chief of the First Research Laboratory for his inspiration. The discussion with and comments of Dr.K.L.Clark, Dr.S.Gregory in Imperial College, Dr.E.Y.Shapiro in Weizmann Institute of Science and Dr.P.C.Treleaven in University of Reading, who were visiting researchers to ICOT in 1983, were of great benefit.

* References *

[1] R.Kowalski : Logic for Problem Solving , North Holland, New York (1979)

[2] S.Haridi and A.Ciepielewski : An OR-parallel Token Machine, Logic Programming Workshop '83 (1983)

[3] J.Darlington and M.Reeve : ALICE and the Parallel Evaluation of Logic Programs, The 10th Annual International Symposium on Computer Architecture (1983)

[4] E.Y.Shapiro : A subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003 (1983)

[5] E.Y.Shapiro : The Bagel : A Systolic Concurrent Prolog Machine, (currently being prepared for publication as an ICOT Technical Report.)

[6] P.C.Treleaven, et al. : Data-Driven and Demand-Driven Computer Architecture, ACM Computing Survey, vol.14, No.1, March (1982)

[7] S.Kunifuji,et al.: Conceptual Specification of the Fifth Generation Kernel Language Version 1 (KL1) , (currently being prepared for publication as an ICOT Technical Memo.)

[8] K.L.Clark and S.Gregory: PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, May (1983)

[9] R.Onai,K.Masuda,M.Asou : Static Analysis of Sequential Prolog Programs, ICOT Technical Report TR—32 (1983)

[10] R.Onai,H.Shimizu,N.Ito,K.Masuda : A Construction Schema of a Prolog

Machine Based on Reduction Mechanism, Proc. of 26th Conference of IPS (Japan) 1983.(In Japanese)

[11] A.Takeuchi and K.Furukawa : Interprocess Communication in Concurrent Prolog, ICOT Technical Report TR-006 (1983)

[12] Arvind and R.E.Thomas : I-structure: An Efficient Data Type for Functional Languages, MIT Lab. for Computer Science Technical Memo TM-178 (1978)