TR-114

The Dataflow-based Parallel Inference Machine To Support Two Basic Languages in KL1

by
M. Kishi, E. Kuno, K. Rokusawa
(Oki Electric Industry Co.,)
and N. Ito (ICOT)

July, 1985

The Dataflow-based Parallel Inference Machine To Support Two Basic Languages in KL1

Noriyoshi Ito
Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo, Japan

Masasuke Kishi, Eiji Kuno, and Kazuaki Rokusawa
Oki Electric Industry Co.,
4-10-12 Shibaura, Minato-ku, Tokyo, Japan

ABSTRACT

Dataflow-based implementation models to support two types of logic programming languages and the machine architecture are presented. The logic programs are compiled into data flow graphs using primitive operators as their nodes and interpreted by the machine to exploit parallelism included in these languages. Basic unification primitives and nondeterminate control primitives are discussed. The simulation results of the machine and its performance considerations are also presented.

1. INTRODUCTION

The logic programming language inherently possesses unification and nondeterminate control functions required for inference processing. In addition, it also has the potential for parallel processing. In the knowledge information processing areas at which the fifth generation computer systems are aiming, parallel processing may be unavoidable in order to solve complicated problems within practical time constraints. The logic programming language, therefore, has been selected as the kernel language for ICOT's fifth generation computers.

KL1 (Kernel Language version 1) is a parallel version of the kernel language including two types of basic languages: AND-parallel Prolog and OR-parallel Prolog. AND-parallel Prolog facilitates the implementation of AND parallelism by introducing a guard mechanism. OR-parallel Prolog focuses on OR parallelism, where each process can solve goals independently.

Two execution models of these languages are introduced: one is an interactive resolution model for AND-parallel Prolog and the other is an independent resolution model for OR-parallel Prolog. In the interactive resolution model, processes interactively communicate variable bindings (messages) by using the guard mechanism. In the independent resolution model, processes invoked search independent solution sets for given goals and the solutions obtained are merged into streams.

The machine is based on the data flow model, where programs are represented by data flow graphs. Nodes and directed arcs in the graphs correspond to operators and data paths along which operands are sent, respectively. Execution of the graphs is performed in a data driven manner. That is, each node becomes executable only when all the operands are arrived on its input arcs; it performs the operation and puts the results on its output arcs without side-effect. This functionality of the operators has close similarity to the functional languages and well suited to parallel processing [7] [3] [9] [2].

The data flow model has also similarity to the logic programming languages described above. Execution of logic programs is performed in a goal driven manner; a clause in the programs is initiated when a goal is given and returns the results to the goal.

The programs written in these languages are compiled into data flow graphs corresponding to the machine language codes of the parallel inference machine. The machine is based on the unfolding interpreter [3], in which unique identifiers are assigned to all the active procedure instances and, thus, all the active operators in these procedures can be executed in parallel. Several primitive operators are introduced to implement unification and nondeterminate control functions included in these languages [13].

The machine is constructed from multiple processing elements and structure memories interconnected by a network. Each processing element interprets the data flow graphs in parallel and transfers packets to/from other processing elements or the structure memories, which store the structured data and are responsible for the structure accessing command packets from the processing elements. Performance evaluation results from the software simulator show that about one million HUPS (Head Unifications Per Second) can be achieved by exploiting parallelism. Considerations for increasing performance are also discussed.

The target languages are outlined in Section 2 while Section 3 describes implementation schemes for these languages. The machine architecture and its simulation results are shown in Sections 4 and 5, respectively.

2. PARALLEL PROLOG

The target languages of the machine include two types of logic programming languages in KL1: OR-parallel Prolog and AND-parallel Prolog. Both languages are based on Horn logic, a subset of first-order predicate logic. Informal descriptions of these languages are given in this section.

2.1 OR-parallel Prolog

OR-parallel Prolog programs consist of Horn clauses with additional sequence control operators, such as AND-sequential or AND-parallel control operators. Each clause has a following form:

H :- B1 & B2 & ... & Bn.

where, symbol ':-' means implication, and the left and right sides of this symbol are called the head and body respectively. H is a head literal, and Bi (1<=i<=n) are body literals. The body consists of an arbitrary number of body literals connected by logical AND operators. Two types of logical AND operators are introduced to specify whether sequential or parallel execution is performed. The operator '&' specifies the AND-sequential control of the body literals from left to right sides and the operator ',' specifies AND-parallel control of the body literals.

The program is initiated when a goal statement (a clause consisting only of body literals) is given. If the parallel AND operators are specified in the goal statement, the body literals (called goal literals) on both sides of the operators will be executed in parallel. Otherwise, the goal literals are executed sequentially from left to right. Unification is attempted between each goal literal and the head literals in the program; if multiple clauses exist, their head unification with a goal literal may be performed in parallel (OR-parallel). But the candidate clauses unifiable with the goal literal are limited to the clauses whose head predicates are the same as that of the goal literal. Such a subset of clauses is called a definition of the goal predicate.

The clauses whose head unification succeeds proceed to further unification taking their bodies as new goals, or they return the solutions to the goal if their bodies are empty.

2.2 AND-parallel Prolog

Several languages have been proposed to realize AND parallelism. They includes PARLOG [5], Concurrent Prolog [16], GHC (Guarded Horn Clauses) [17], and so on. Common features of these languages are that they emphasize AND parallelism rather than OR parallelism, and that they provide an inter-process communication facility by sharing logical variables among AND processes (goals). The instances of these variables (the messages) are sent from processes to other processes using the guard mechanism.

GHC was selected as a basic language of KL1 because it has clearer semantics and provides more efficient implementation than Concurrent Prolog, it and has more powerful descriptive power than PARLOG [17].

AND-parallel Prolog programs consist of guarded clauses such as:

```
H :- G1, G2, ..., Gm | B1, B2, ..., Bn.
```

where, H and Bi (1<=i<=n) are a head and body literals, respectively, as in OR-parallel Prolog. Gj (1<=j<=m) are called guard literals, and '|' is called a commit operator. The left side of the commit operator is called a guard and the right side of the commit operator is called a body.

If a goal literal is given, only one clause whose guard succeeds can proceed to its body execution. The commit operator performs mutually exclusive control among the clauses in the invoked definition as in the guarded command [8]. In these languages, the guard or body literals are executed in parallel. In order to implement the interactive communication among the AND processes invoked from these guard or body literals, read-only variables are introduced, or unification directions are specified. A variable shared among AND processes can be changed to a read-only variable if a process wants to receive the variable instance (message) from the other process. If a read-only variable is unified with a non-variable term, the variable instance is read before unification. If the variable is not instantiated, the unification operation is suspended until the variable is bound to another non-variable term by other unification operation. Thus, the interactive message passing is achieved among processes.

2.3 Invisible and Visible Streams

In OR-parallel Prolog, each goal literal execution will produce a set of solutions. These solutions may be returned to the goal in a nondeterminate manner. OR processes will return the solutions to the goal in the order in which the solutions are obtained. These solutions are merged into a stream by stream merging primitives. The stream here is called an invisible stream, because the structure of the stream is not 'visible' to the program directly. Figure 2.1 shows an example of a goal where the instances of variable X obtained by the execution of the literal p(X) are sent to the next literal q(X).

In AND-parallel Prolog, however, the streams are 'visible' to the programmers. The programmers may code the programs that explicitly generate or consume the streams, which are implemented as structured data containing unbound variables as their elements. Such streams are called visible streams. Figure 2.2 shows an example, where a producer invoked by the literal p(X) generates a visible stream represented by a list and a consumer invoked by the literal q(X) reads its elements.

In contrast to OR-parallel Prolog, at most one instance is bound to the variable X and it may be a structure representing a visible stream.

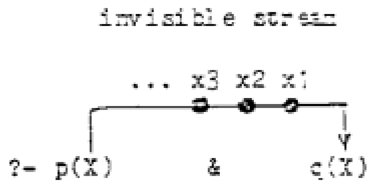


Fig. 2.1 Communication on an Invisible Stream

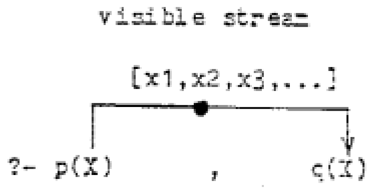


Fig. 2.2 Communication on a Visible Stream

3. IMPLEMENTATION OF PARALLEL PROLOG

In this section, the implementation schemes to exploit three types of parallelism (OR-parallelism, AND-parallelism and low-level parallelism in unification) included in OR-parallel Prolog and GHC are described.

3.1 Implementation of OR-parallel Prolog

The independent resolution model for OR-parallel Prolog and its implementation are described below.

(1) Nondeterminate Stream Merge

Multiple solutions may exist for a given goal in OR-parallel Prolog. The order in which these solutions are returned to the goal is nondeterminate. The solutions found may be returned to the goal in the order they are obtained. This nondeterminism is called "don't know nondeterminism." To implement this nondeterminism, the definition is compiled into a data flow graph as shown in Fig. 3.1. If a goal literal is given, the multiple clauses in the definition are invoked and unification with the goal is attempted. The solutions, if found, are merged into a invisible stream in a nondeterminate manner.

Stream merging is performed by a "create-stream" primitive and "append-stream" primitives. The create-stream primitive is initiated by the arrived goal literal and creates an empty stream. The stream is represented by a stream head pointer (SHP) and a stream tail pointer (STP) to the current tail of the stream. The contents of STP are initialized to the address of the SHP cell as shown in Fig. 3.2 (a).

Each append-stream primitive receives the address of the STP cell and a solution of the clause. If the solution is not "fail" (i.e., unification succeeds), the append-stream primitive appends a new solution to the current tail pointed to by STP and updates the contents of STP as shown in Fig. 3.2 (b); failed unification does not affect the stream.

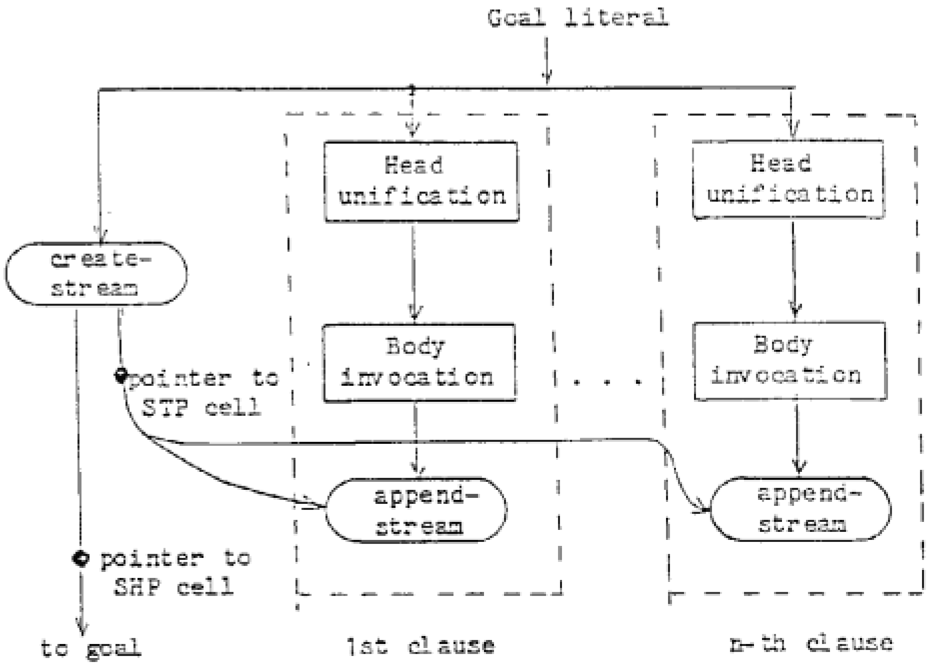


Fig. 3.1 Stream Merging Schema in OR-parallel Prolog

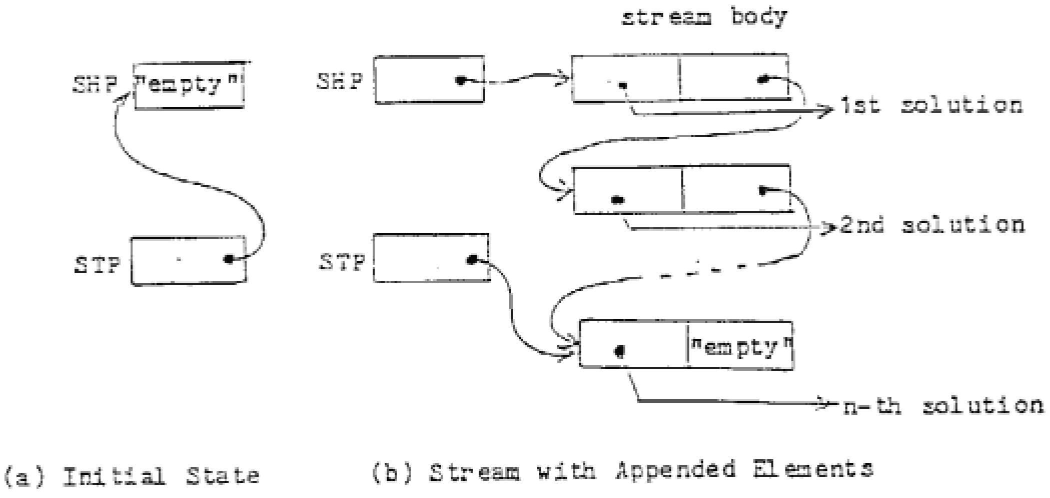


Fig. 3.2 The Structure of an Invisible Stream

The consumer of the stream (i.e., the goal waiting for the solutions from the definition) will receive the address of STP immediately after the create-stream primitive is executed and try to read the contents of STP. Each memory cell word in the stream has a tag field specifying whether the contents of the memory word are valid or not. The "empty" tag indicates that the word is empty (i.e., no write operation to the word has been performed yet). The "pending" tag indicates that some read operations are performed to the empty word (the read operations are suspended and the suspended read requests are chained into the memory word until the write operation to the word is performed). Other tag values indicate the data type of the data written into the word.

The read operation of the SHP will get a stream head address if some solutions have been appended to the stream. It is suspended otherwise. Suspended operations are activated when a write operation is performed by the append-stream primitive. The consumer process, then, traverses the stream and gets the solutions from the invoked definition.

As the stream structures are stored in and distributed to the structure memories, the stream merging primitives are implemented as the structure manipulation commands to the structure memories. In order to manage the stream structure, the reference count method [1],[2] can be adopted. It is also used to reclaim the structure memories in the machine [12]. A reference count field is appended to each STP cell, which is shared among the clauses in the invoked definition. The reference count is initialized to the number of the clauses invoked. It is decremented by one each time an append-stream operator is executed, and is incremented by the number of the child clauses invoked from the body literals.

If the reference count reaches to zero by decrementing, the append-stream operator writes "fail", which indicates the end-of-stream, into the cell pointed by STP, and the STP cell becomes garbage. If all the clauses fails (i.e., if the stream is still empty when the reference count is zero), the contents of the STP cell are set to "fail"; otherwise, it is set to the first cell address of the stream body.

(2) Compilation of the Clauses

Variables in each clause may be assumed as the data path names along which the instances of the variables are sent. A data flow graph is obtained by connecting the same variables with directed arcs. Figure 3.3 shows the graph of the flllowing clause:

p(X,Y) :- q(Y,Z) & r(Z,X).

The instance of the variable X in the example is used as the first argument of the first body literal q(X,Z) and the instance of Y is used as the second argument of the body literal r(Z,Y). Because the two body literals are connected by a sequential AND operator, the instance of Z, which is obtained as the second argument of the literal q(X,Z), is sent to the next literal r(Z,Y).

If the parallel AND operator, on the other hand, is used in the above clause, the unbound variable Z is shared by two parallel body literals as shown in Fig. 3.4. In order to assure that the two occurrences of Z are bound to the same instance, the unbound variable Z is changed to a shared variable Zs before two body literals are called (the subscript 's' indicates that the variable is shared). The shared variable is distinguished from a non-shared variable by its data type and dynamically created by a "share" primitive.

The two goal literals, thus, have the shared variable Zs as their arguments. The invoked definition will generates bindings for the shared variables if they are bound to the terms other than non-shared variables. These bindings are returned to the goal as the stream elements and are used to checking of the shared variables in the goal for consistency.

A solution to be returned to the goal (or to be appended to the stream) is a list constructed of the final instances of the given goal literal arguments followed by a binding environment for the shared variables included in the goal arguments. The goal gets the solutions from the stream, decomposes them into the instances and the binding environment, and then performs consistency checking or next goal literal invocation, followed by constructing new solutions and returning them to the parent goal.

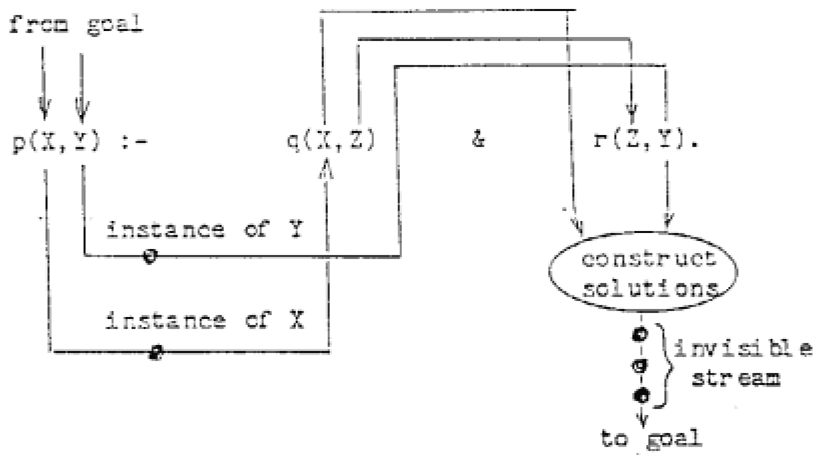


Fig. 3.3 Connection Graph of a Clause with Sequential Body Literals

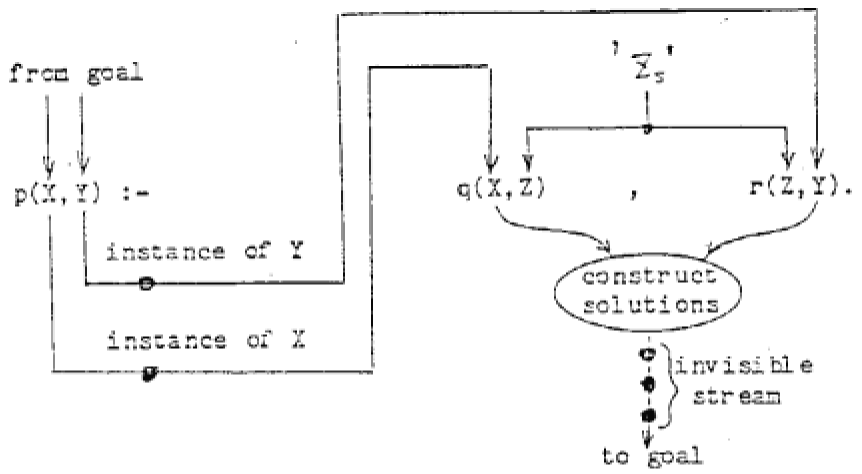


Fig. 3.4 Connection Graph of a Clause with Parallel Body Literals

(3) Head Unification

Parallelism included in head unification can also be exploited. If a goal literal have multiple arguments, their unification with corresponding head arguments can be achieved in parallel, and if a pair of arguments to be unified are structures, their substructures can be also unified in parallel. If the arguments passed from the goal are the structured data, such as lists or vectors, these structures are shared instead of beeing eagerly copied; they are lazily copied when copying is necessary. Only when the unification is performed between two structured data, the contents of these structures are accessed. Therefore, copying overhead via the network or redundant storage for copied structures can be eliminated.

A problem in this scheme is latency for structure accesses. Latency may be increased as the number of processing elements increases. In order to exploit parallelism, the processing element must issue multiple remote access requests without waiting for their responses. In such an environment, the requests and responses are managed by their identifiers since responses may not be returned to the processing element in the order in which the requests were issued. The data flow model implements this type of control in a natural way, and can also exploit such low-level parallelism in head unification because it is assured of independence of operations or instructions [2] [4].

A clause is compiled into a data flow graph, where head unification is performed by multiple unify primitives. Figure 3.5 shows a data flow graph when the following clause is given:

```
p([X,a],b,Y) :- ...
```

A unify operator is provided for each of the three arguments of the head literal; they are executed in parallel. Each unify operator has an I (instance) port and E (environment) port; the I-port for output of a common instance of the input operands, and the E-port for output of a binding environment of shared variables included in the input operands. The binding environment is created only when shared variables are bound to non-variable terms or to other shared variables; it is represented by a list, whose elements are pair lists constructed of the shared variables and their instances. For example, if the given goal literal is p([c,Us],V,Us), where Us is a shared variable, then the common instance and the binding environment produced by the unify operator of the first argument are [c,a] and [[Us|a]], respectively. The environment shows that the shared variable Us is bound to a constant instance 'a'. The failed unify operators generate "fail" symbols to their output ports.

A non-shared variable can be unified with any team, and their common instance is always the term bound to the variable. Therefore, when the head argument is a variable as in the third argument in the example, the unify operator can be omitted, as shown by the broken line in the figure. The simpler example shown in the Fig. 3.3, where all the head arguments are non-shared variables, does no head unification; the arguments passed from the goal literals are sent to the body directly.

The check-consistency primitive in Fig. 3.5 receives the E-port outputs of the unify primitives, and performs consistency checking for the bindings of the shared variables; it searches two environments whether they have bindings for same shared variables, and, if so, tries to unify both instances for the shared variables and create a new environment. The consistency checking is easy if at least one of the environments is nil or fail.

The variable instances obtained by the unify primitives are substituted for the shared variable included in the binding environment. This substitution is executed by "substitute" primitives. A substitute primitive replaces the shared variables in the instance with the binding environment, if the instance includes shared variables and if the environmnts is a list containig their binding; it outputs "fail" if the environment is "fail" (i.e., head unification fails), or outputs the instance directly, otherwise. Their outputs are sent to the next stage: execution of the body if the body exists, or returning of the solution if the body is empty. Body execution is suppressed if head unification fails. The procedure call primitives invoke the definitions only when the final environment obtained by the check-consistency primitive is not "fail".

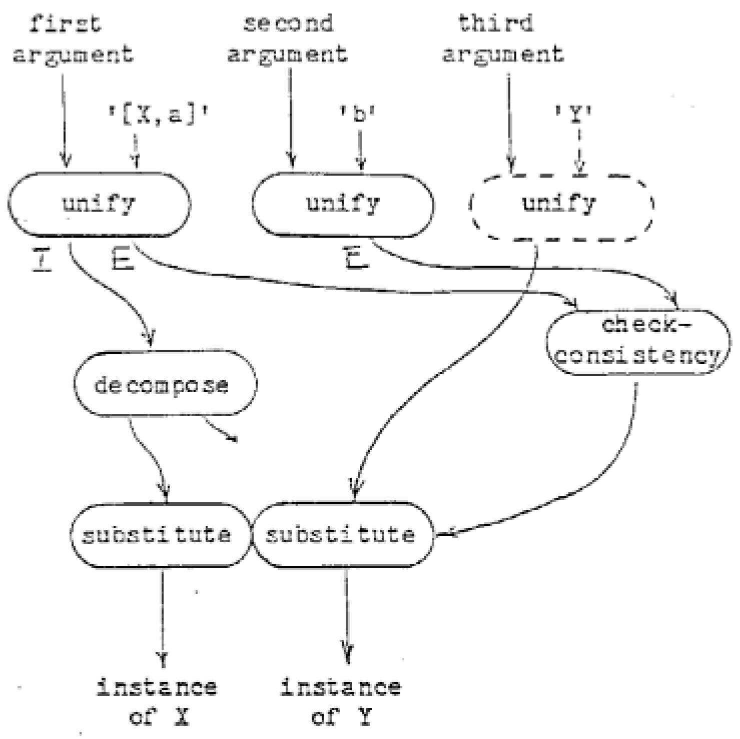


Fig. 3.5 Data Flow Graph Representation of Head Unification

3.2 Implementation of GHC

The interactive resolution model for GHC and its implementation are described below.

(1) Read-only Tagging Scheme

In GHC programs, only a clause whose guard succeeds first for the given goal can proceeds its body execution. This nondeterminism is called "don't care nondeterminism." When the definition is invoked, a semaphore flag shared among the clauses in the definition is created. The clause whose guard succeeds (i.e., its head unification with the goal succeeds and the invocations of all its guard literals are successfully terminated) performs a test-and-set operation to the shared semaphore flag. If the result of this operation is also successful (i.e., if the clause executed the operation is a first one whose guard succeeds), the clause can execute its body; processing of the other clauses is terminated. The number of instances bound to a variable is restricted to one, because the other candidate clauses are excluded by the guard mechanism. The variable in AND-parallel languages, therefore, can be represented by (a pointer to) a memory cell.

The main difference between Concurrent Prolog and GHC is that Concurrent Prolog uses read-only annotation, while GHC uses a guard mechanism to control the direction of unification. That is, in Concurrent Prolog, if a variable with read-only annotation (read-only variable) is unified with a non-variable term, the instance bound to the variable is read before unification is performed. Thus, this unification is suspended until the variable is instantiated to a non-variable term. If a variable without read-only annotation is unified with a term, the term is written to the variable cell. In GHC, on the other hand, a variable in the goal literal cannot be bound to a non-variable term, or to another variable contained on the goal literal, in the guard of the invoked clause. Such unification is suspended until the variable in the goal literal is bound. Synchronization between the read and write operations is realized by the memory tagging scheme described in Subsection 3.1.

There are several implementation schemes to support the guard mechanism in GHC [14]. One of them is a complete compilation scheme, where all the unification directions are analyzed in compilation time, and codes are generated using unidirectional unification primitives as in Kernel PARLOG [6]. In this scheme the compiler is complicated because it must analyze guard nesting levels of all clauses by traversing the whole predicate invocation tree and determine all the directions of unification. The separate compilation of programs required to develop large programs may be difficult.

The next is a guard system number scheme, where all the environments are managed by guard system numbers. A new guard system number is allocated when a new definition is invoked and is restored to its parent number when the commit operator is executed. The guard system numbers are associated with all the variables included in the invoked clauses and the envirorment to which each variable belongs is compared with the current envirorment when unification to the variable is attempted.

The pointer coloring scheme distinguishes variables belonging to the goal literals from those belonging to the current guard by coloring. If unification is attempted between a goal variable and a variable in the invoked clause, the callee's variable is changed to a colored variable, which points to the original variables. The colored variables are treated like the read-only variables in Concurrent Prolog. If a colored variable is unified with a term, the instance bound to the variable is read before unification. The commit operator restores the colored variables to their original variables.

The latter two schemes have the drawback of overhead caused by allocation and deallocation of variable cells and by extra memory accesses to get the guard system numbers or the original variables. The last method is one we propose to improve on by reducing overhead. It is an extension of the pointer coloring scheme and is called a read-only tagging scheme, in which every variable has a tag specifying its read-only level.

In the tagging scheme, each variable occurrence has its read-only level. As described above, Concurrent Prolog gives only one level read-only control by using read-only annotation, while GHC permits arbitrary read-only level as shown in the following table:

Table 3.1 Read-only Levels in AND-parallel Prolog

Read-only level	Semantics in Concurrent Prolog	Semantics in GHC	Notation in this paper
0	X	X	X
1	X?	X?	X'
2	X?	X??	X''
3	X?	X???	X'''
.	.	.	.
.	.	.	.
.	.	.	.

(The symbol '?' indicates a read-only tag)

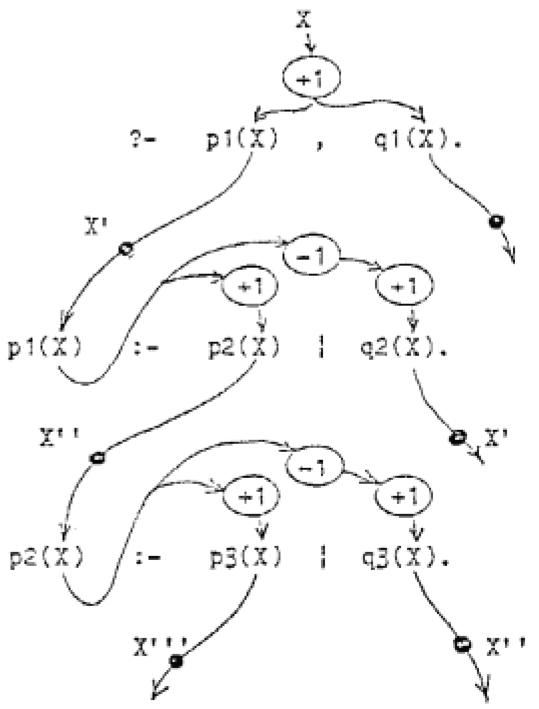


Fig. 3.6 Read-only Levels of Variables

The read-only levels of the goal variables are incremented by one before the definitions are invoked, and decremented by one when the commit operator of each invoked clause succeeds. Figure 3.6 shows an example of the data flow graph and the literal arguments of the following goal and clauses:

```
?- p1(X), q1(X).
p1(X) :- p2(X) ¦ q2(X).
p2(X) :- p3(X) ¦ q3(X).
      . . .
```

In the figure, '+1' and '-1' specifie the increment and decrement operators of the read-only levels respectively. In this example, the read-only level of each variable instance sent from the head literal to the body literal is decremented, followed by being incremented just after the decrement operation. These sequences of the decrement and increment operators, therefore, can be eliminated by the compiler.

It should be noted here that no variable cell allocation is necessary for the callee's variables unified with the caller's variables. Memory cells are dynamically allocated only for variables not included in the head literals while appeared as the arguments of the guard or goal literals. These variables are represented by the pointers to the cells, whose read-only levels are zero. They are called writable variables.

There is a problem of the decrement operation overhead in this scheme. If a variable instance sent from the guard to the body is a structure whose components include read-only variables, the decrement operation must traverse whole the structure and decrement all the read-only levels of these variables. This operation may degrade the performance. For example, if the following goal and clause is given:

```
?- p(X).
p(X) :- Y = f(Z), Z = X | r(Y).
```

In the guard of the clause, the writable variable Y is bound to the structure f(Z) while the variable Z is bound to the read-only variable X'. Thus, the decrement operator should change the structure f(X') to the new structure f(X). In order to execute this operation faster, if the writable variables are bound to the read-only variables, the unify primitive generates a decrement variable list including the writable variables as its elements. The decrement operator receives this list and performs decrement operation only for these variables. In the actual programs, unification such as described above may be rare cases. Therefore, almost all the decrement variable lists are nil and the decrement operators are simply executed.

It is difficult to represent arbitrary read-only level in the tag, because the tag field length to specify the level may be limited. So, the pointer coloring scheme can be used as well. If the machine permits N levels in the tag field, a variable with read-only level less than N+1 can be represented in a straightforward manner. If the variable's read-only level is incremented to N+1, then a memory cell is allocated and the new variable is represented by the pointer to the allocated memory cell into which the original variable is written. The new variable is tagged to show that indirect memory access is necessary to get the original variable. This type of variable is called an extended variable.

In actual programs written in GHC, almost all the guard nesting levels are zero or one; few clauses invoke the user-defined clauses from their guards. The number of maximum read-only levels represented by the tag can be restricted to one. Thus, three different tags are sufficient to represent variables: a writable variable (a variable with read-only level zero), a variable with read-only level one, and an extended variable.

(2) Optimization of GHC Implementation

Many stream parallel programs in GHC can be considered as producer-consumer problems, where producer processes produce a sequence (stream) of messages and consumer processes receive the messages from them. GHC and the other AND-parallel languages use logical variables for message passing. Each logical variable can be bound to only one instance and may be immediately discarded after the consumers read the instance from it. That is, the life span of each logical variable is very short and frequent memory allocation and deallocation for the logical variables are necessary.

The main idea of this subsection is to invisualize the detailed structure of the stream from the programmer's point of view and to develop a more efficient stream structure and primitives instead of using logical variables directly.

(a) A Simple Producer-Consumer Program

The example shown below is a GHC program of a producer-consumer problem, where the producer invoked by the literal p(0,X) generates a sequence of integers and the consumer invoked by the literal q(X) consumes them:

```
?- p(0,X), q(X).
p(N,X)    :- N1 := N + 1 | X = [N|Y], p(N1,Y).
q([N|Y]) :- print(N)    | q(Y).
```

The stream in this program is represented by a list whose first element is a number and the second element is the rest of the stream. The clause 'p' will allocate a new memory cell for logical variable 'Y' each time it is invoked and the clause 'q' will discard the memory cell after it reads the instance of the variable.

In order to hide the actual representation and implementation of the stream from the program, we can rewrite the above program as follows:

```
?- create_stream(Head,Tail), p(0,Tail), q(Head).
p(N,Tail) :- N1 := N + 1 | stream(Tail,N,NewTail), p(N1,NewTail).
q(Head)   :- stream(Head,N,NewHead), print(N) | q(NewHead).
```

where the predicates 'create_stream' and 'stream' can be defined as follows in the 'pure' logical interpreter:

```
create_stream(X,Y) :- true | X = Y.
stream(X,V,Y) :- true | X = [V|Y].
```

The 'create_stream' clause succeeds when it is invoked, while the 'stream' clause tries to unify its first head argument with a list constructed from its second and third head arguments. The execution of the 'stream' predicate in the body (as in the clause 'p') will append a new element 'N' to the stream and return a new stream pointer. The execution of the 'stream' predicate in the guard (as in the clause 'q') will read and decompose the stream into its first element 'N' and the rest of the stream.

We can use more efficient, built-in predicates embedded in the system instead of these user-defined stream handling predicates. They can be implemented as stream merging primitives described in Section 3. The 'create_stream' predicate creates an empty stream. The steam can be represented by a structure as shown in Fig. 3.1. In order to reduce memory allocation overhead further, the predicate can allocate a contiguous memory block for the stream buffer as shown in Fig. 3.7 (a) [15]. The pointer of the block may be bound to the variables 'Head' and 'Tail'. These pointers are then sent to the clauses 'q' and 'p' respectively. The 'stream' predicate simply increments these pointers and returns them when it is invoked.

(b) The Multiple Producer Problem

This stream implementation can be proved to be more efficient for multiple producer problems. In the following goal, the producers 'p1' and 'p2' generate streams which are merged into a single stream consumed by the predicate 'q':

```
?- merge(X,Y,Z), p1(X), p2(Y), q(Z).
```

The predicate 'merge' merges two streams 'X' and 'Y' into a stream 'Z' in a nondeterminate manner defined as follows [17]:

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

In the 'pure' logical interpreter, the goal literals p1(X) and p2(X) will produce the independent streams 'X' and 'Y' and the goal literal merge(X,Y,Z) will copy their elements into a new stream 'Z' while waiting for the arrival of the elements.

This merge predicate can be implemented using an extended version of the built-in predicate 'create_stream':

```
merge(X,Y,Z) :- true | create_shared_stream(Z,X), Y = X.
```

The predicate 'create_shared_stream' creates a shared stream as shown in Fig. 3.7 (b).

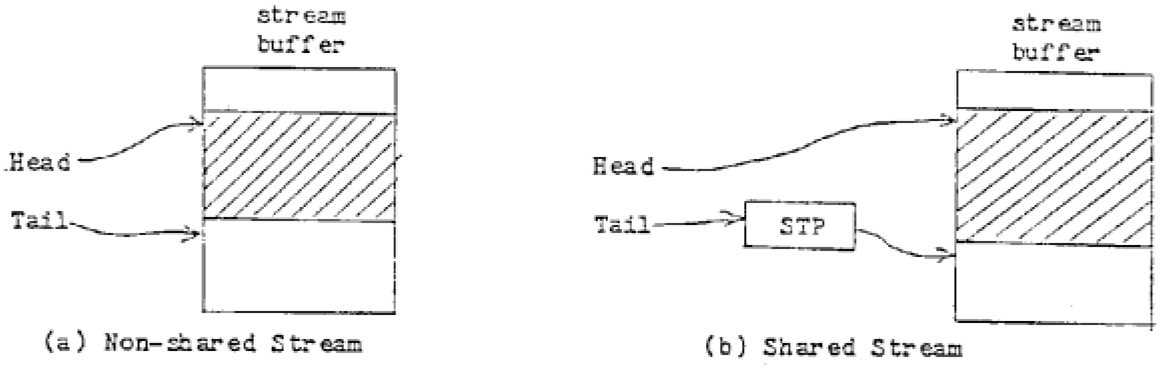


Fig. 3.7 Representation of a Stream

The producers 'p1' and 'p2' may share the current stream tail pointer (STP). Each 'stream' predicate call in these producers will append a new element to the memory word pointed to by STP and increment the contents of STP.

(c) The Bounded Buffer Communication Program

We can extend the stream handling predicates to control bounded buffer communication. The 'create_stream' predicate in the following goal creates a bounded buffer and buffer size counter initialized to 10.

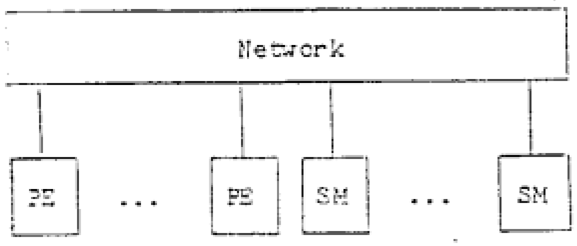
```
?- create_stream(Head,Tail,10), p(Tail), q(Head).
```

The buffer size counter is shared among all the processes invoked by this goal. The created processes execute semaphore operations on the counter. The producer processes invoked by the predicate 'p' will test and decrement the current buffer size counter when they try to append new elements to the stream. They may be suspended if the counter is zero. The consumer process invoked by the predicate 'q' will test and increment the counter after it reads a stream element.

4. MACHINE ARCHITECTURE

The conventional processor based on the von-Neumann model, in which both program codes and data are stored in the memory unit, fetches and interprets program codes sequentially. If such processors are used as the processing elements of the distributed, parallel inference machine, the context switching of processes or packet communication overhead, caused by frequent remote accesses of shared structures or shared variables, will significantly reduce the system performance.

As described in Section 3, the data flow machine provides independence of operations executing in parallel. The machine can exploit parallelism easily in such a distributed processing environment. The abstract machine architecture is shown by Fig. 4.1. The machine is constructed of processing elements (PEs) and structure memories (SMs) interconnected by networks. Each PE has several stages. The packets transferred between these stages include result packets and executable instruction packets. A result packet consists of three fields: activity identifier, destination, and data fields. The activity identifier specifies the invoked procedure instance to which the result packet belongs. The destination specifies the destination instruction address of the result packet. It also specifies whether the destined instruction receives a single operand or two operands (i.e., the instruction is executable on arrival of a single operand or two operands). The data field contains the operand data to be sent to the instruction.

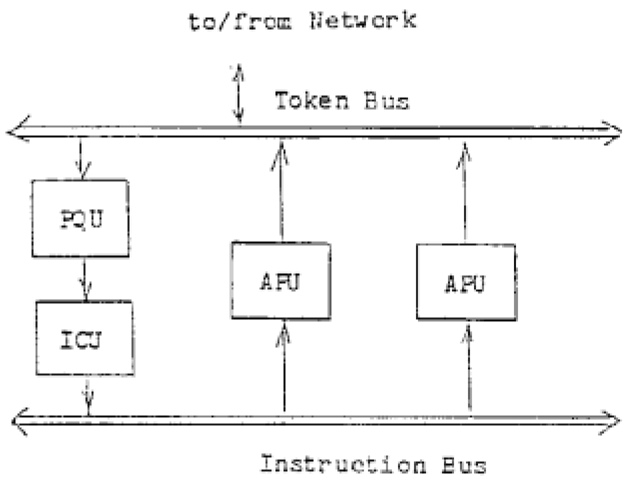


PE: Processing Element
SM: Structure Memory

Fig. 4.1 Abstact Machine Architecture

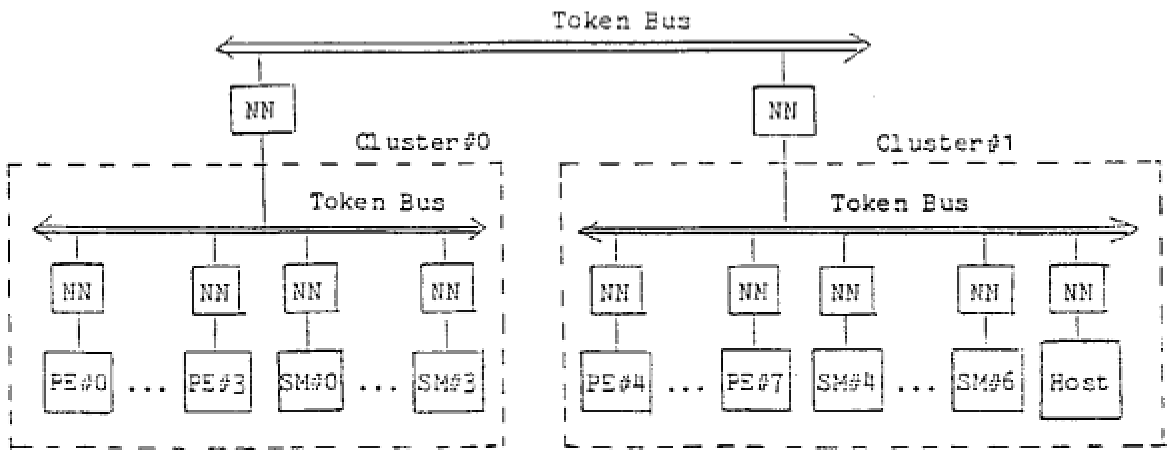
Figure 4.2 depicts the configuration of each PE. PQU (Packet Queue Unit) is a first-in first-out queue memory to store the result packets form the Token Bus. ICU (Instruction Control Unit) receives the result packets from PQU and checks if the destination instructions are executable or not (i.e., if their operands are ready or not). An instruction is executable if it receives a single operand, or if the partner operand is already in the operand memory in the ICU when it receives two operands. In the latter case, the ICU searches in its operand memory whether the partner operand with the same activity identifier and destination as the result packet exists or not. If it does, the partner is removed from the operand memory; otherwise, the result packet is stored in the operand memory. This searching is performed associatively by hardware hash assuming the activity identifier and destination address as the key field. If the instruction is executable, the ICU fetches the instruction code in its instruction memory and constructs an executable instruction packet and sends the packet to the next stage, one of APUs (Atomic Processing Units), via the Instruction Bus. The APU interprets the instruction packets and sends result packets to the PQU in its PE or other PEs, or sends structure access command packets to SMs via the Token Bus. The SMs are responsible for the structure access commands, perform structure manipulation operations, and return results to the destination specified by the commands.

Actual implementation of the experimental machine is currently being developed using the hierarchical network shown in Fig.4.3 [10]. The machine includes eight PEs, seven SMs, and one host computer used to monitor or debug the system. Packets are transferred between the modules (PEs and SMs) via Token Buses controlled by NNs (Network Nodes) that arbitrate the packet sending requests from the modules and send the packets to the other modules. The APUs and SMs are implemented as microprogram control units using bit-slice microprocessors or special hardware to recognize the data tag. The ICUs are also microprogram controlled to implement hashing hardware.



PQU: Packet Queue Unit
ICU: Instruction Control Unit
APU: Atomic Processing Unit

Fig. 4.2 Configuration of a Processing Element



NN: Network Node
PE: Processing Element
SM: Structure Memory

Fig. 4.3 Configuration of the Experimental Machine

5. PERFORMANCE CONSIDERATIONS

5.1 Simulation Results

A software simulator for OR-parallel and Concurrent Prolog was developed and machine performance evaluated [11]. In the software simulator three types of networks are assumed. An Inter-PE network is used to transfer result packets between processes being executed in PEs. A PE-SM Network is used to transfer the structure commands or their result packets between PEs and SMs. Finally, an Inter-SM network is used to transfer the structure commands between SMs. The Inter-PE and Inter-SM Networks are two-dimensional mesh networks and the PE-SM

Network is a multi-stage network.

The simulator is an event-driven simulator implementing the functional units described above and network nodes as independent processes. Their typical processing times assumed are given in Table 5.1. It is also assumed that all the ICUs in the system contain the complete data flow graphs in their instruction memories, and the allocation of new procedure instances is performed with no dynamic program loading overhead (but the control overhead to allocate new procedures to PEs is taken into account).

Table 5.1 Typical Processing Times of the Units

Unit	Item	Time(Machine Cycles)
PQU	Packet Receive	2
	Delay in queue	8
ICU	Single operand instruction	2
	Two operand instruction (on arrival of 1st operand)	3
	Two operand instruction (on arrival of 2nd operand)	5
APU	"copy" instruction	3
	Packet creation	2
NN	Packet receive	2
	Packet send	8
SM	SM-read operation	2
	SM-write operation	2

The sample programs include:

- DCG (Definite Clause Grammar) and BUP (Bottom Up Parser) programs that generate parsing trees by analyzing natural language sentences (written in OR-parallel Prolog).

- a compact program to eliminate duplication of list elements (written in Concurrent Prolog).

- an editor, which interprets the sequence (stream) of database commands and updates the internal database (written in Concurrent Prolog).

- quick-sort programs that sorts the list elements in ascending order (written in Concurrent Prolog).

- N-queens programs, which places the N queens on the N X N chess board so that no queen capture any other queen (written in both languages).

Figure 5.1 and 5.2 shows performance curves for OR-parallel Prolog and Concurrent Prolog Programs, respectively, when the number of modules (PEs and SMs) are increased. Performance is represented by HUPS (Head Unifications Per Second) assuming that one machine cycle is 250 nanosecond.

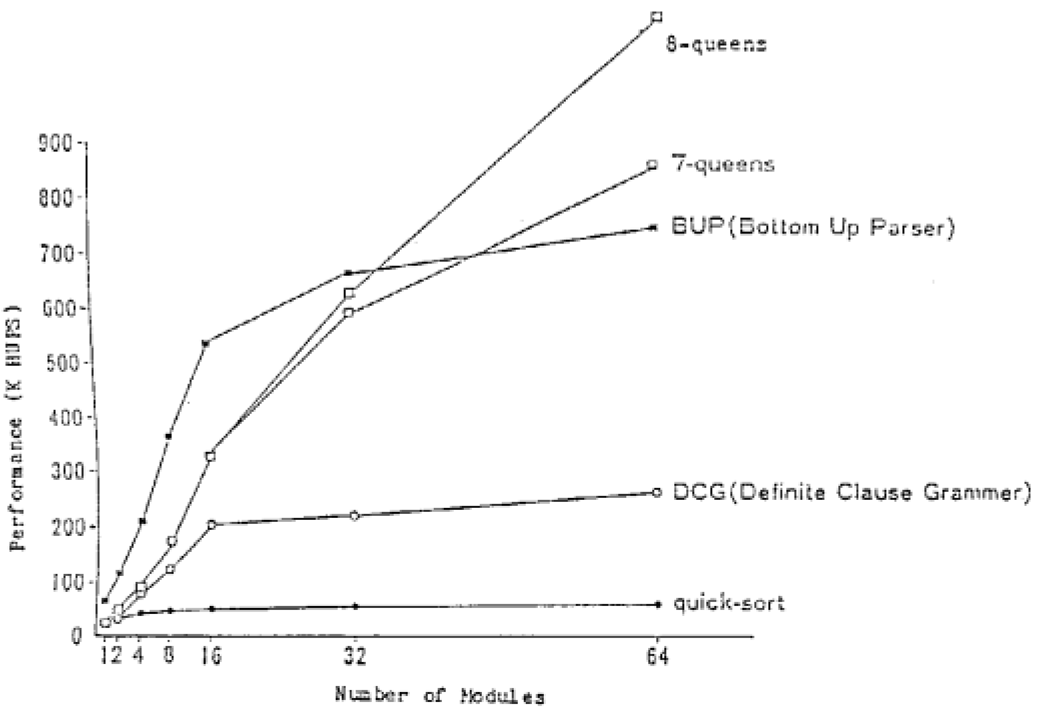


Fig. 5.1 Performance of OR-parallel Prolog Programs

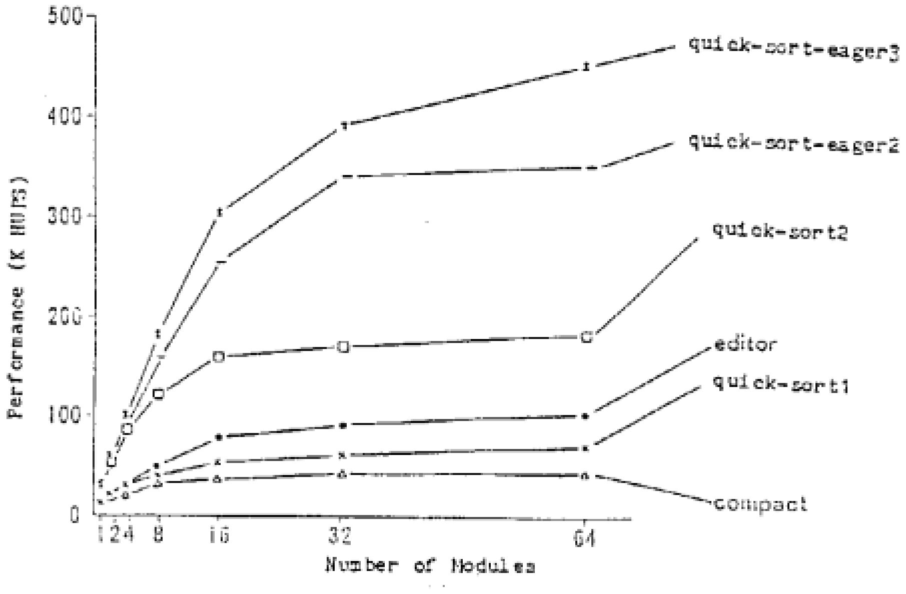


Fig. 5.2 Performance of Concurrent Prolog Programs

In the Fig. 5.1, the same dictionary and sample Japanese sentence for analysis are used in both the DCG and BUP programs. The sentence and the dictionary are very small, so the performance of these programs is optimal using about sixteen modules. The performance of BUP program is proportional to, and about twice that of the DCG program. The eight-queens program does not reach optimum in the simulated range and its performance was about one million HUPS at the maximum point.

In the Fig. 5.2, the performances of the compact or editor programs are not so high, because they do not have so much parallelism. Several quick-sort programs are tested. quick-sort1 sorts a list constructed from atomic constants. quick-sort2 sorts a list whose elements are constructed from lists with two constant elements and requires list comparison predicates to decide the sorting order. In order to exploit parallelism in the problem, quick-sort2 is updated so that list comparison is performed in the body rather than in the guard (quick-sort-eager2). quick-sort-eager3, in which the length of the list elements is extended to three, was also simulated.

Comparison results for the six-queens programs written in the two languages are given in terms of performance (Fig. 5.3(a)) and execution times (Fig. 5.3(b)). Two versions of Concurrent Prolog programs are given: one is a version in which stream merging predicates are used to control the solutions (CP-merge version); in the other version the solution set is given as a difference-list (CP-d-list version). The performance of these programs is nearly same, but the Concurrent Prolog versions need about three times as much computing time as OR-parallel Prolog version. The Concurrent Prolog programs here do not implement the stream handling predicates as built-in predicates described in Section 3. It seems clear that the Concurrent Prolog versions can be executed faster if built-in stream handling predicates are used.

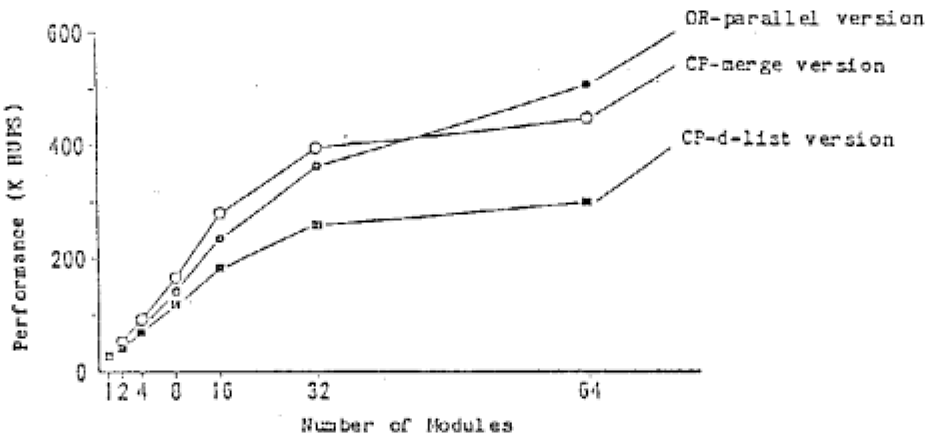
6. CONCLUSION

Execution models on the dataflow-based parallel inference machine for OR-parallel and AND-parallel Prolog, basic languages in KL1, were presented and primitive operators for supporting these two languages were described. It was shown that two types of logic programming languages with different aims can be supported on this machine.

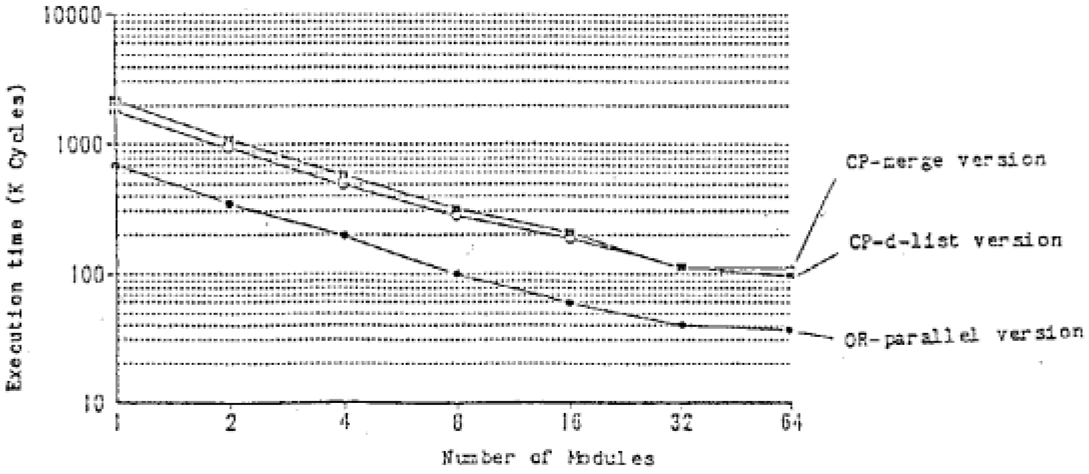
There are two basic functions embedded in these languages: one is unification and the other is nondeterminate control. Several primitives for performing these functions are introduced and programs written in these languages are compiled into data flow graphs corresponding to machine language codes. Thus, parallelism in the programs can be exploited naturally.

The machine architecture constructed from processing elements and structure memories was presented. The processing elements interprets the procedures represented by the data flow graphs in parallel. Structured data is distributed to structure memories and shared among these procedures. Thus, lazy copying of the structured data can be easily achieved.

The simulation results of OR-parallel and Concurrent Prolog programs indicate that performance can be significantly improved by exploiting parallelism. Detailed designs for the experimental machine have been developed and the machine is currently being debugged. Future efforts will involve the evaluation of the machine to serve as the basis for a highly-parallel inference machine.



(a) Performance



(b) Execution Time

Fig. 5.3 Comparison between OR-parallel and Concurrent Prolog

<Acknowledgement>

The authors extend their thanks to Director Kazuhiro Fuchi at ICOT, who afforded them the opportunity to pursue this research. Also much appreciated are the valuable advice and comments of Dr. Kunio Murakami, formerly Chief of the First Research Laboratory in ICOT and currently at NTT, Dr. Hidenori Ito, current Chief of the first Research Laboratory, and Mr. Rikio Onai, Senior Researcher, Mr. Toshihiko Miyazaki, who developed the compiler, and other ICOT research members.

<References>

[1] Ackerman,W.B., " A Structured Processing Facility for Data Flow Computers," Proceeding of International Conference on Parallel Processing, 1978.

[2] Amamiya,M., R.Hasegawa, O.Nakamura, and H.Mikami, "A List-processing oriented Data Flow Architecture," National Computer Conference 1982, pp. 143-151, June, 1982.

[3] Arvind, K.P.Gostelow, and W.E.Plouffe, "An Asynchronous Programming Language and Computing Machine," TR-114a, Dept. of ICS, University of California, Irvine, Dec., 1978.

[4] Arvind and R.A.Innucci, "A Critique of Multiprocessing von Neumann Style," Proceedings of 10th Internatiral Symposium on Computer Architecture, June, 1983.

[5] Clark,K. and S.Gregory, "PARLOG: Parallel Programming in Prolog," Research Report DOC 84/4, Imperial College of Science and Technology, April, 1984.

[6] Clark,K. and S.Gregory, "Notes on the Implementation of PARLOG," Research Report DOC 84/16, Imperial College of Science and Technology, 1984.

[7] Dennis,J.B. and D.P.Misnus, "A Preliminery Architecture for A Basic Data Flow Processor," Proc. of 2nd Symp. on Computer Architecture, Jan., 1975.

[8] Dijkstra,E.M., "A Discipline of Programming," Prentice-Hall, 1976.

[9] Gurd,J.R. and I.Watson, "Data Driven System for High Speed Parallel Computing," Computer Design, July, 1980.

[10] Kishi,M., E.Kuno, K.Rokusawa, and N.Ito., "Architecture of Experimental System of Dataflow-based Parallel Inference Machine," Proc. of 30th National Conference of Information Processing Society in Japan, 1985 (in Japanese).

[11] Kuno,E. and N.Ito, "Simulation Results of OR/AND Parallelism in Dataflow-based Parallel Inference Machine," Proc. of 30th National Conference of Information Processing Society in Japan, 1985 (in Japanese).

[12] Ito,N. and K.Masuda, "Parallel Inference Machine Based on the Data Flow Model," International Workshop on High Level Computer Architecture 84, Los Angeles, California, May, 1984.

[13] Ito,N., H.Shimizu, M.Kishi, E.Kuno, and K.Rokusawa, "Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog," New Generation Computing, Vol. 3, No. 1, 1985.

[14] Miyazaki,T., "GHC Implementation in multi-SIM," Internal Memo in ICOT, 1985 (in Japanese).

[15] Onai,R. and M.Asou, "Control Mechanisms of Guard and Read-only Annotation in Parallel Inference Machine," Proc. of 27th National Conference of Information Processing Society in Japan, 1983 (in Japanese).

[16] Shapiro,E.Y., "A Subset of Concurrent Prolog and its Interpreter," TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Jan., 1983.

[17] Ueda,K., "Guarded Horn Clauses," TR-103, Institute for New Generation Computer Technology, Tokyo, Japan, 1985.