ICOT Technical Memorandum: TM-0827

TM-0827

Guide to the Japanese Demonstrations at Joint Japanese/American Workshop on Future Trends in Logic Programming

by
K. Rokusawa

November, 1989

© 1989, ICOT

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

Guide to the Japanese Demonstrations

Joint Japanese/American Workshop

on

Future Trends

in

Logic Programming

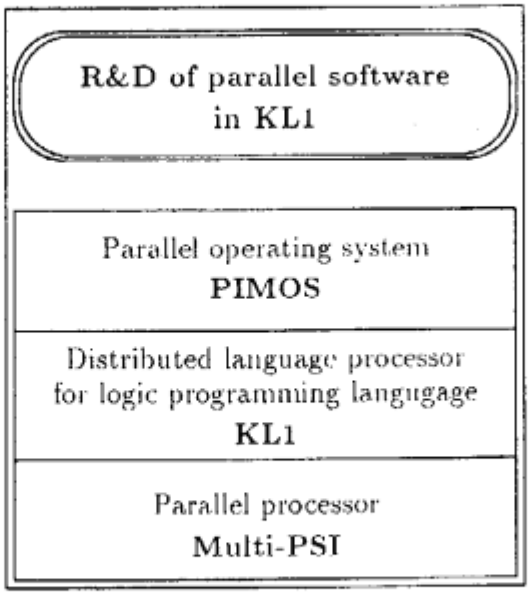
October 11–13, 1989

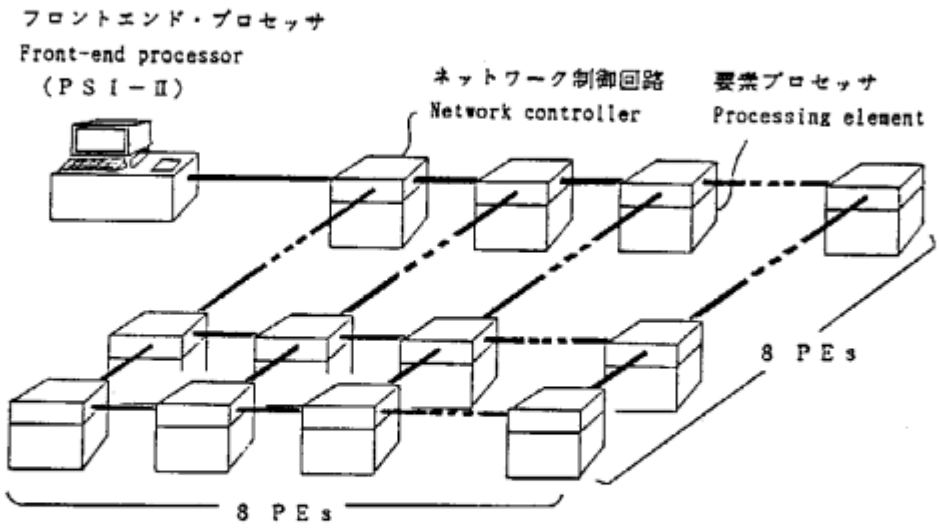
Institute for New Generation
Computer Technology

4-28, Mita 1-chome, Minato-ku, Tokyo 108 Japan
phone: +81-3-456-2511 fax: +81-3-456-1618

A List of Demonstrations

1. Parallel Software Development System *Multi-PSI and PIMOS*
   - Hardware for Parallel Software Development Multi-PSI
   - Distributed KL1 Implementation
   - Parallel Inference Machine Operating System PIMOS
   - Parallel Application Programs
     (1) Natural Language Parser PAX
     (2) Tsumego Solver
     (3) Packing Piece Puzzle
     (4) Shortest Path Problem Solver
2. Knowledge Application Oriented Advanced DBMS *Kappa*
3. Computer Aided Proof *CAP-LA*
4. Constraint Logic Programming Experimental System *CAL*
5. A General-Purpose Reasoning Assistant System *EUODHILOS*
6. Visualization and Transformation Apprentice for Concurrent Logic Programming *VISTA*
7. PIMOS Development Support System *PDSS*
8. Experimental A'UM System *XAS*

Title	**Parallel Software Development System**
Purpose	Research and development of parallel operating systems, parallel algorithm design and load distribution methods on an implementation of a concurrent logic programming language on a parallel processor
Outline & Features	• The Multi-PSI connecting up to 64 CPUs of the sequential inference machine (PSI) • A high performance distributed language implementation for a concurrent logic programming language (KL1) • A parallel operating system (PIMOS) for research and development of parallel software • Used to extend and optimize the KL1 language processor and the PIMOS for parallel inference machines (PIMs)
System Configuration	**Parallel software development system** 

Title	**Hardware for Parallel Software Development – Multi-PSI**
Purpose	The Multi-PSI hardware offers high processing power and useful functions for (1) research and development of parallel software and distributed processing mechanisms, and (2) designing PIM architecture.
Outline & Features	The CPU and memory of the compact version of the sequential inference machine, PSI, are used as the processing elements (PEs) of the Multi-PSI. PEs are connected to each other to form a two-dimensional mesh network by a specially designed message switching and automatic routing network controller. The system can be configured with up to 64 PEs in units of eight PEs. The compact version of the PSI, the PSI-II, is used as the front-end processor (FEP). Up to four front-end processors can be connected to the network to perform I/O functions for the Multi-PSI system. Tag architecture : 8-bit tag + 32-bit data PE control : Horizontal microprogram control Cycle time : 200 nsec (whole system synchronized) Main memory : 80 MB (16 MW)/PE max. Network channel : 5 MB/s Devices : 8K, 20K-gate CMOS gate array LSI, and others
System Configuration	

Processing element (PE)

The wide (53-bit) micro instruction architecture of PEs allows flexible experiments of KL1 language implementation. The tagged architecture enables efficient execution of logic programming languages. It is especially suitable for high level abstract machine instructions because tag manipulation and multi-way branching on tag values can be performed in parallel with an ALU operation.

Network controller

A PE communicates with four adjacent nodes through bidirectional channels, each nine bits wide. Messages received are stored in the `Read buffer` for the PE in the node, or forwarded to another node through the channel chosen by looking up the `Path table` with the destination address of the message.

A buffer of 48 bytes for each output channel is used for busy waiting synchronization with the adjacent node.

When a complete message is stored in the `Read buffer`, the PE of the node is notified of it (by a `NWINT` signal), and the PE will process the message as early as convenient for its internal processing.

Message sending from the PE is initiated automatically when a complete message is stored in the `Write buffer` of the network controller.

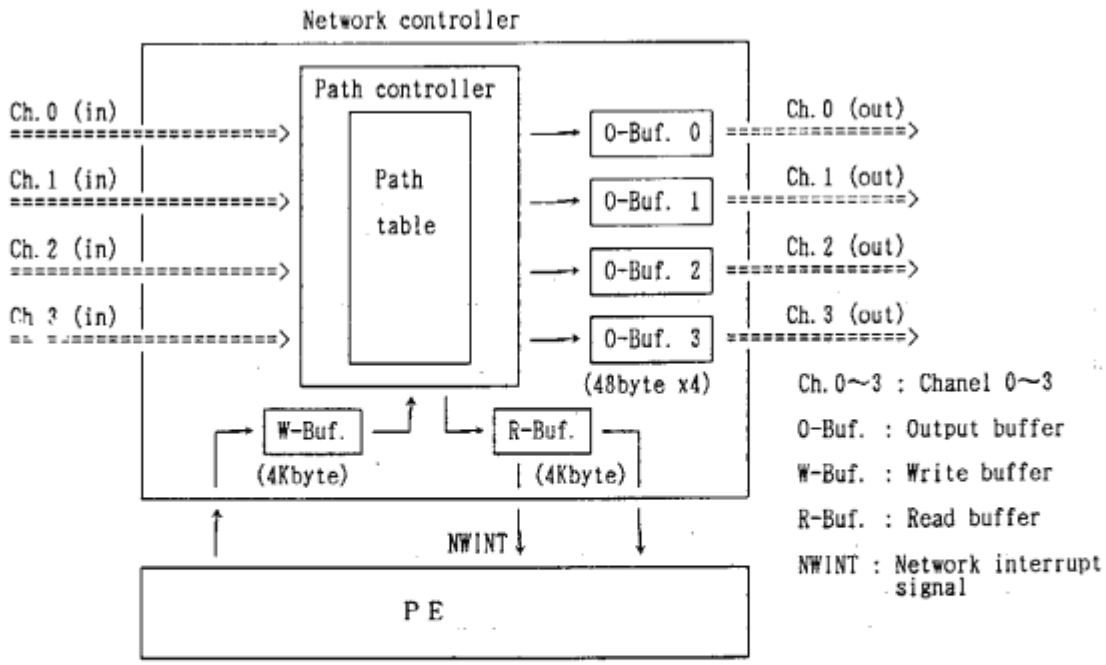


Figure. Network controller and Processing element (PE)

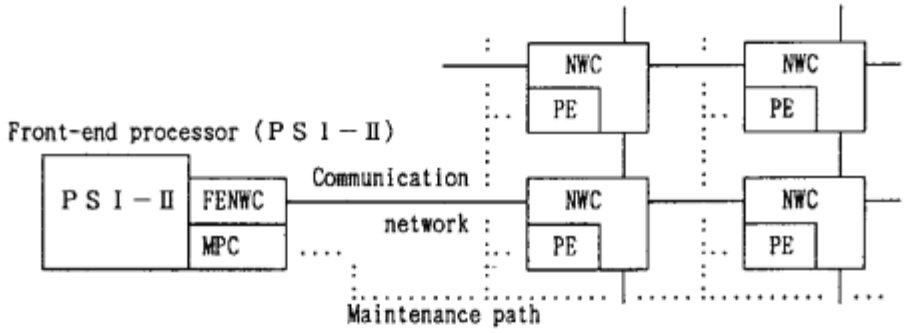
Hardware implementation

A PE is implemented on three printed circuit boards (PCBs) with nine 8K-gate CMOS gate array LSIs. The network controller is implemented on one PCB using two 20K-gate CMOS gate arrays.

One PE can have up to 80M bytes of main memory on four PCBs, each of which contains 20M bytes, using 1M bit dynamic RAMs.

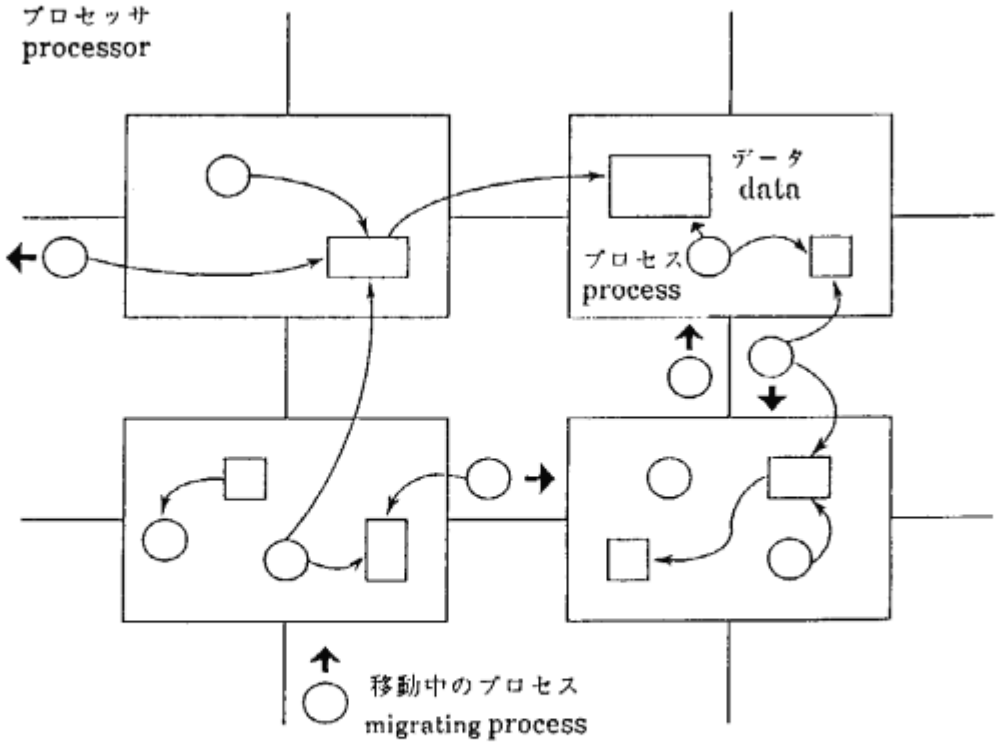
One cabinet of the Multi-PSI contains eight nodes, and one system can have up to 64 nodes by connecting eight such cabinets. The entire system is synchronized by a single clock distributed to all cabinets.

Up to four front-end processors (FEPs) can be connected to the network for I/O. One of them, the master FEP, also performs console processor (CSP) functions, such as system start-up and diagnosis, through a specially devised maintenance path.



PE : Processing element
NWC : Network controller
FENWC : Front-end network controller
MPC : Maintenance path controller

Figure. Multi-PSI and its front-end processor

Title	# Distributed KL1 Implementation
Purpose	The distributed KL1 implementation manages KL1 processes and data distributed over the network-connected processors of the Multi-PSI and executes KL1 programs efficiently.
Outline & Features	KL1 programs are compiled by an optimizing compiler into abstract machine instructions, which are executed by the microcode (150 KLIPS (Kilo Logical Inferences Per Second) per processor). The distributed KL1 implementation is designed to reduce the amount of inter-processor communication by utilizing the single-assignment property of KL1 and by various other techniques. Innovative intra- and inter-processor garbage collection schemes are implemented.
System Configuration	

1 KL1

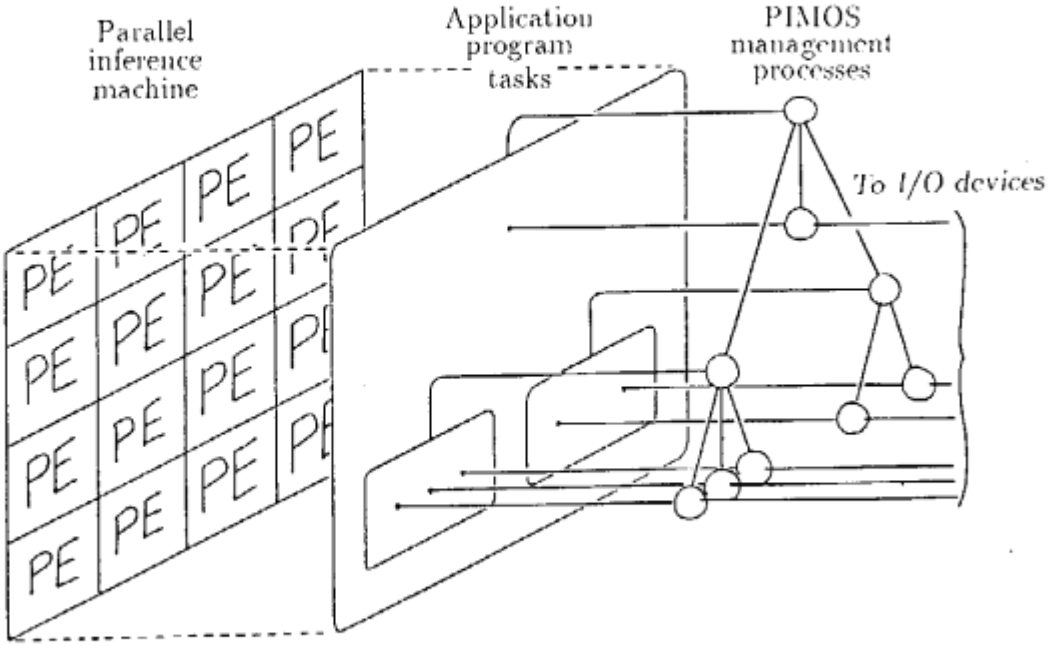
KL1 is a stream AND-parallel logic programming language. In stream AND-parallelism, concurrently executed processes that share data constitute the process structure in the form of cause-consequence chains. Unlike OR-parallelism and independent AND-parallelism, in which concurrently running processes do not communicate with each other, programs originally written for sequential machines cannot be readily executed in a stream AND-parallel manner, but complex cooperative problem solving can be best modeled in stream AND-parallelism.

2 Multi-PSI Architecture

There are two categories in parallel computer architectures: shared and non-shared memory architectures. In the shared memory architecture, processors communicate with each other by writing and reading shared memory; in the non-shared memory architecture, they communicate by sending and receiving messages over the communication channels. The shared memory architecture has the advantage of relatively low communication overhead, but the maximum number of processors is severely limited because of the memory access bottleneck. The advantage of non-shared memory architecture is its scalability. It was chosen for the Multi-PSI, since the machine is to serve as an experimental machine for the Parallel Inference Machine (PIM) project which aims at building a parallel machine with up to 1,000 processors. Programs on a non-shared memory machine, however, need to be designed with the problem of high communication overhead in mind.

3 Distributed KL1 Implementation

The task of KL1 implementation is to execute KL1 programs efficiently on the Multi-PSI. Algorithms have been developed to keep the amount of inter-processor communication as low as possible. Since the rate of memory consumption in KL1 programs is high, new techniques are employed in garbage collection (GC), reclamation of memory area that is no longer used. One is the Multiple Reference Bit (MRB) technique that uses one-bit pointer tags to recognize reclaimable data in a local processor, and another is the Weighted Export Counting (WEC) technique suited for inter-processor incremental GC. The implementation provides metaprogramming capabilities to support the operating system. They include the "shoen" facility — core of resource and task management, priority execution, and user-programmable load distribution mechanism.

Title	**Parallel Inference Machine Operating System: PIMOS**
Purpose	The PIMOS aims at providing operating system facilities through which application programs can easily and fully utilize the processing power of the parallel inference machines.
Outline & Features	**Described in KL1:** The PIMOS is completely described in the concurrent logic programming language KL1. Making use of the *meta*-programming features of the KL1 language, the design of the PIMOS is independent of various hardware parameters, such as the number of available processors in the system. **Single OS on multiple processors:** The PIMOS is not an aggregate of independent operating systems on each processor, but one single integrated operating system. However, not only the application programs but also various parts of the PIMOS are executed on multiple processors in parallel. Computation and communication bottlenecks due to information centralization are avoided by distributing management information close to the application program tasks. **Providing basic system functions:** The PIMOS provides basic functions required in operating systems, such as management of execution, resource allocation and input/output devices. Additional services are planned.
System Configuration	All the demonstration programs shown on the Multi-PSI systems are operated under the supervision of the PIMOS. 

The functions of the PIMOS are demonstrated by operating the application programs in their demonstrations.

Stream communication and filters

All communication between the PIMOS and application programs is made through *streams*, which is one of the most advantageous features of the AND-parallel logic programming languages. *Filters* inserted to such streams handle abnormal termination of application programs and protect the operating system from failures in application programs.

Distributed processing by the resource tree

Each application program task or input/output device used in such tasks has a PIMOS management process associated with it. The *processes* in this case are quite light-weight ones provided by the microcode of the KL1 language processor, which correspond to *objects* in object-oriented programming languages.

In the PIMOS, tasks can be created inside a task, which, as a whole, forms a tree structure. Thus, the corresponding PIMOS management processes also form a tree structure. This tree is called the *resource tree*. All the PIMOS management information is distributed to the management processes, which are the nodes of this resource tree.

The management process is allocated on the processor where the corresponding application program task is running (when the task uses multiple processors, the processor on which the request was made to create a new task or to open an I/O device). This distribution of management processes also distributes the management overhead to multiple processors. Also, by placing management information near the application program tasks, communication congestion to a single processor, as expected when all the information is centralized in a single table, is avoided and the total amount of inter-processor communication is minimized.

Shell

Execution of an application programs supervised by the PIMOS can be initiated by invoking it from the command interpreter (*shell*). Based on the functions equipped by the PIMOS, the shell provides the following features for the user.

- Starting, suspending, restarting and aborting the execution of jobs
- Controlling foreground and background jobs
- Defining the standard input/output of jobs
- Inter-task communication via *pipes*
- Controlling resource allocation to jobs
- Handling exceptions in application programs

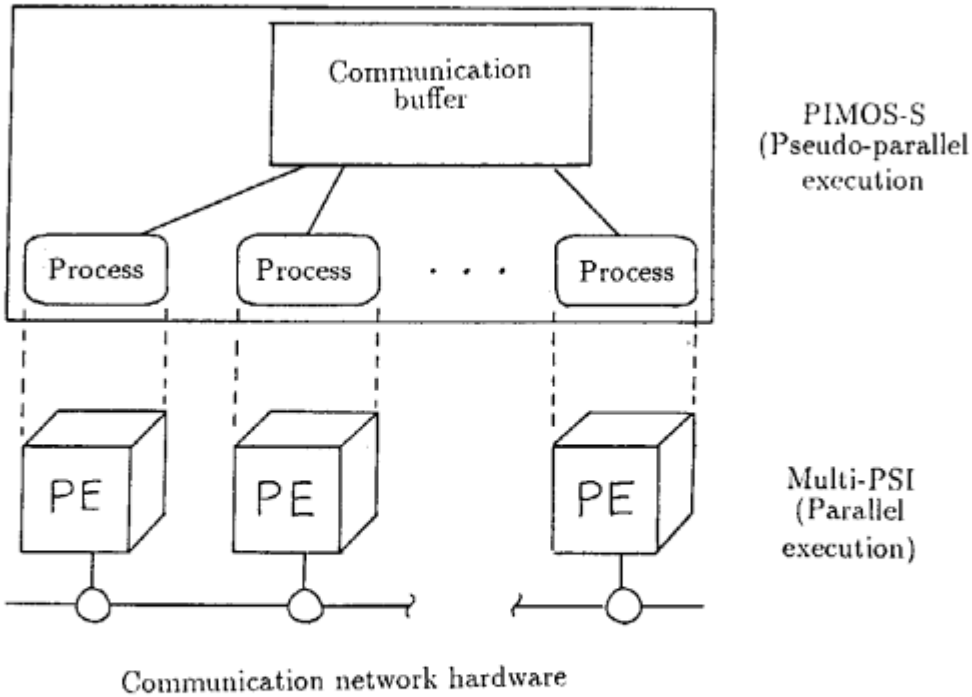
Utility programs invoked from the shell provide monitoring of the status of task execution and allocation of resources such as input/output devices.

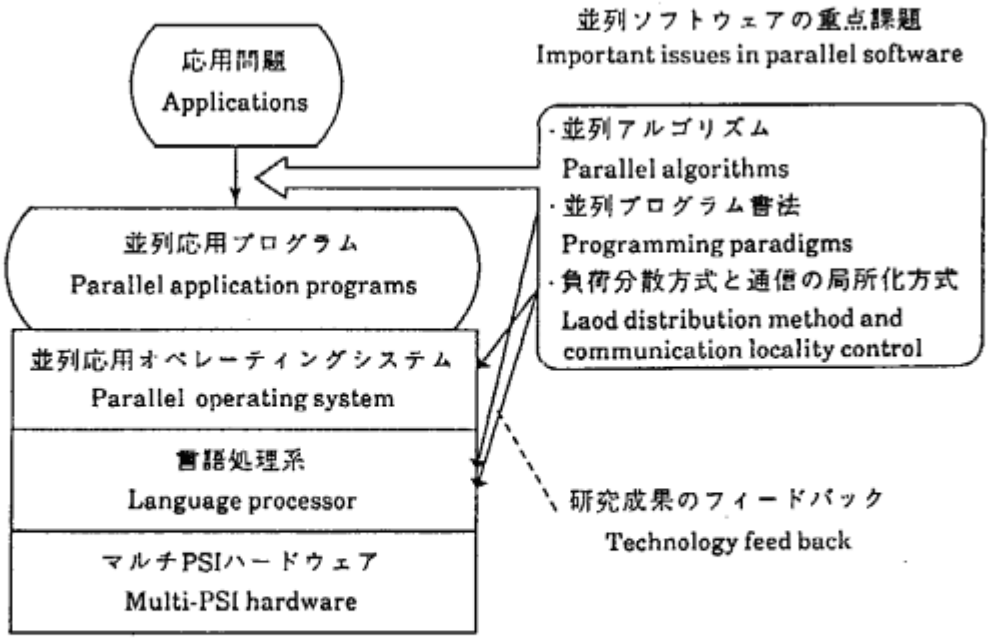
Input and output devices

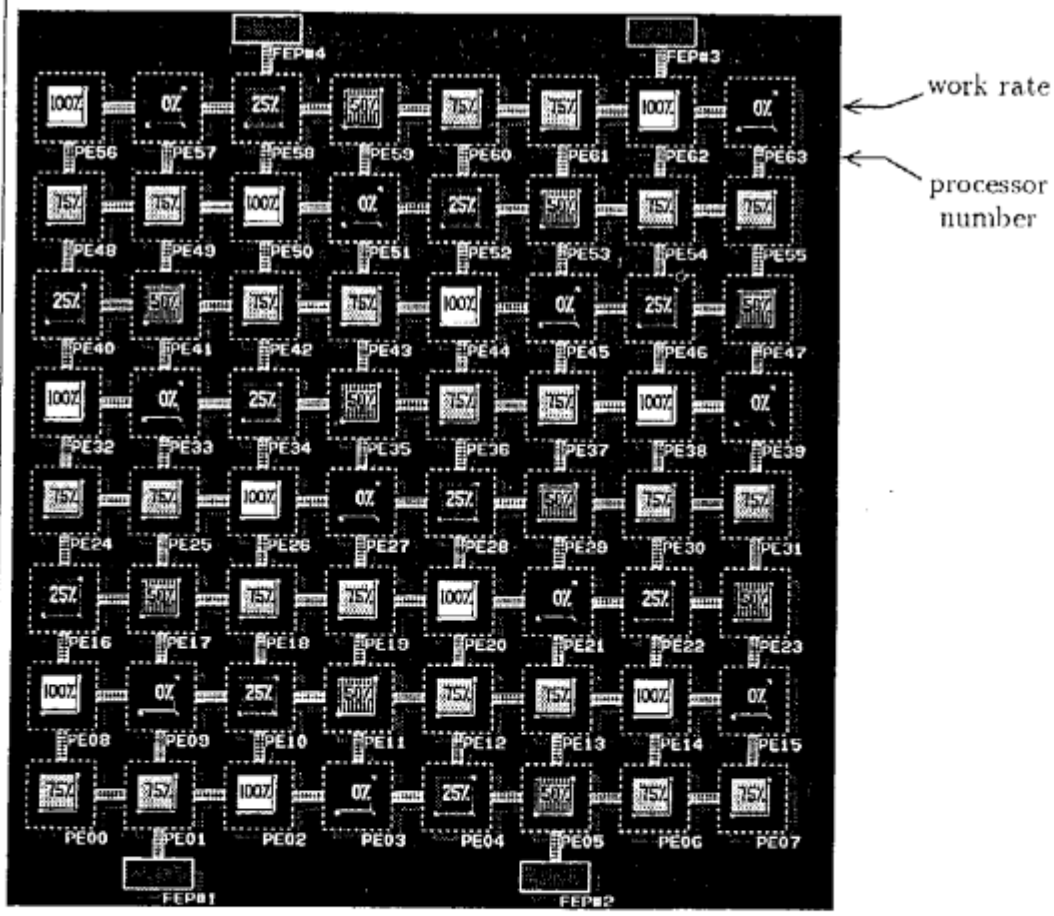
The PIMOS currently provides functions to access high-level I/O features of the operating system (SIMPOS) on the front-end processor (PSI-II), such as files and display windows, from KL1 programs. Various display facilities used in demonstration programs are operating on the front-end processor, controlled by KL1 programs on the Multi-PSI through the standard interface provided by the PIMOS.

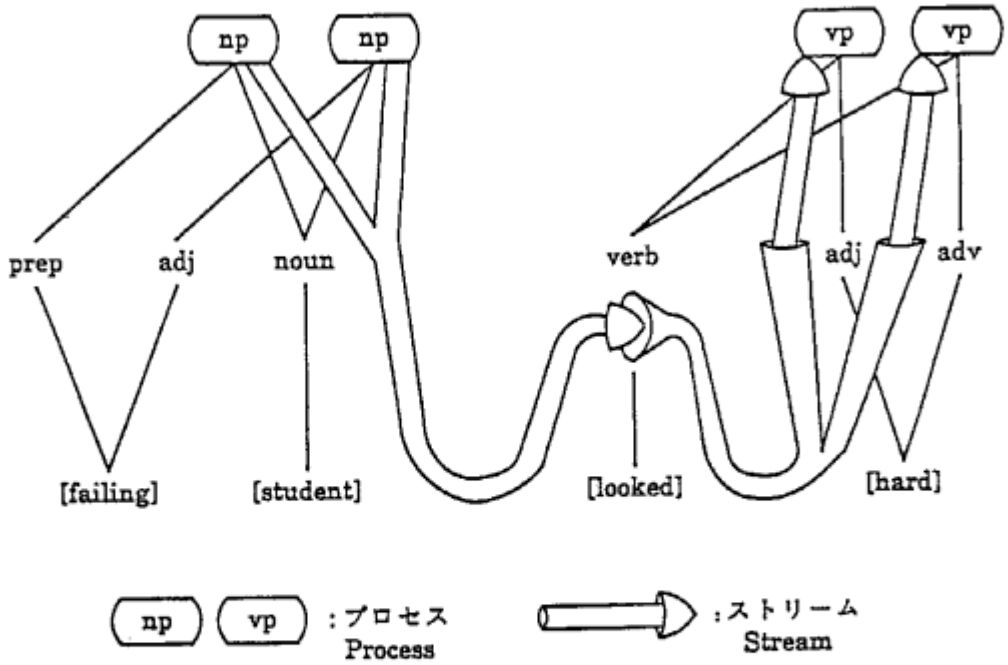
Further details of the design of the PIMOS are given in the following paper, presented at the session ICOT-SS1 (3:30 p.m., Wednesday, Nov. 30th).

"Overview of the Parallel Inference Machine Operating System"

Title	**Parallel Software Development Environment on PSI-II: PIMOS-S**
Purpose	The PIMOS-S provides a parallel software development environment qualitatively equivalent to the Multi-PSI system on much less costly PSI-II workstations.
Outline & Features	**Compatibility:** The language and operating system features provided are fully compatible with the Multi-PSI system. Parallel software developed on PSI-II workstations using PIMOS-S can be executed on the Multi-PSI without any change. **Pseudo-parallelism:** Processing elements of the Multi-PSI running in parallel are simulated by pseudo-parallel processes. **High performance:** The same microcode as used in the Multi-PSI system is utilized. Thus, the provided performance is equivalent to its one processing element. **Decent debugging environment:** A debugging environment qualitatively equivalent to the Multi-PSI system is provided. It is even better in some aspects; e.g., non-determinacy due to parallel execution can be eliminated by applying pseudo-random scheduling.
System Configuration	

Title	Parallel Software Research
Purpose	To develop novel software technology essential for making large scale network-connected parallel machines work efficiently for various application fields.
Outline & Features	Almost every software technology in programming and execution of application programs must be reconstructed for parallel processing to achieve a satisfactory execution efficiency of large scale parallel machines. Especially those listed below are essential. • To develop algorithms with much parallelism without increasing the amount of required computation • To accumulate programming styles or paradigms which give guidelines to parallel programming for various types of large scale problems • To develop load distribution methods for balanced work load and high communication locality The research has just started by implementing parallel programs for several types of applications (e.g. the programs demonstrated). The results of the research will be fed back to improve the functions of the operating system and the language processor.
System Configuration	

Title	**Processor Work Rate Measurement Program for The Multi-PSI** **— Performance Meter —**
Purpose	To evaluate load distribution of user programs through real time visual display of processor work loads.
Outline & Features	Work rate of each processor of the multi-PSI is displayed graphically at real time. Display interval is two seconds (can be changed). The measurement program is written in KL1.
System Configuration	The program is constructed of measuring processes allocated to all processors and a management process which gathers the measurement results. The results are packed and sent to a display device on the front-end processor via the operating system, PIMOS.  Display example of the Performance meter

Title	Parallel Application Program (1): Natural Language Parser
Purpose	The natural language processing system originally used in DUALS is implemented in parallel using *layered stream method*. A load distribution method is studied and evaluated for it in order to realize a very fast natural language parser.
Outline & Features	**Outline** The PAX analyzes natural language sentences, makes parse trees, and displays them. Parsing is performed by bottom-up, parsing messages between processes which correspond to each node of parse tree. **Parsing program** The parsing program is generated from definite clause grammar by a translator, which adds load distribution code automatically. **Problem characteristics** All solutions are searched in the parsing algorithm. However, the result is not always unique because of the ambiguity of grammar, particularly in natural language. **Programming paradigm** The *layered stream method*, a broadly applicable paradigm for all solution search problems, is used. **Load distribution** A load allocation which minimizes inter-PE communication is examined.
System Configuration	 解析木と階層化されたストリーム通信 Parse tree and layered stream communication

Configuration

The PAX is a natural language processing system. It analyzes natural language sentences, makes parse trees, and displays them in the window. This system is divided into three parts, the input part of sentences, the analysis part (parser), and the output part of results (parse trees).

The parser is the main part of this system and only this part runs on the Multi-PSI in parallel. The parsing program is generated from definite clause grammar by a translator.

Both the input program and the output program are written in ESP and run on the front end processor PSI-II machine.

Problem characteristics

For parsing, a bottom-up breadth-first algorithm called left-corner parsing is used. Generally, all solutions are searched in the parsing algorithm. However, the result is not always unique because of the ambiguity of grammar, particularly in natural language.

In general, we use the following way to write all solution search programs in languages that do not support the backtracking mechanism, such as KL1. First, we gather all solutions as a set, then the set is sieved gradually to select the adequate one. In the PAX, the *layered stream method*, a broadly applicable paradigm for all solution search problems, is used.

Algorithm

The PAX analyzes the input sentence by using the bottom-up parsing method with generating processes which correspond to each node of the parse tree, based on each word. The analysis progresses by messages passing between two adjacent processes through the stream. Each message contains partial results (partial parse trees).

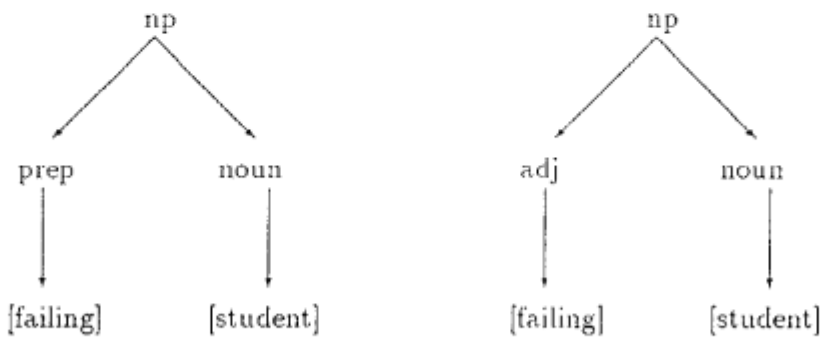
We explain it by using an example shown in the figure of first page. This example analyzes the sentence "failing student looked hard" with the following grammar.

```
s    --> np, vp.          adj  --> [failing].
np   --> adj, noun.       adj  --> [hard].
np   --> prep, noun.      prep --> [failing].
vp   --> verb, adj.       adv  --> [hard].
vp   --> verb, adv.       verb --> [looked].
                          noun --> [student].
```

D e t a i l s (2/3)

First, the input words generate processes with streams which connect two adjacent words.

This picture is a snapshot of parsing. The noun phrase (*np*) and verb phrase (*vp*) processes have already been made, and each *np* process (there are two) sends partial parse trees that it made to *vp* processes through the stream. The partial parse trees are as follows:



There are two *np* and *vp* processes, because there are two canditates for *failing* : adjective (*adj*) and preposition (*prep*). *hard* also has two candidates, adjective (*adj*) and adverb (*adv*).

Communication between an *np* process and a *vp* process uses the layered stream that was connected when processes were generated from the words.

1. Each *np* process sends its partial parse trees through the stream between *student* and *looked*.

2. Both the *adj* process and *adv* process are adjacent to the *verb* process and are connected with the stream between *looked* and *hard*. When they communicate with each other, the *verb* process puts the stream connecting *student* and *looked* into the message.

3. On receiving messages from the *verb* process, the *adj* process and *adv* process generate the *vp* process above them. The stream between *student* and *looked* are got from the message and given to the *vp* processes.

4. Each *vp* process receives the stream between *student* and *looked*, and can receive the message from the *np* process through it.

In this way, the *np* process communicates with the *vp* process. Each *vp* process receives two partial trees from the *np* processes, and constructs its trees and generates the last node *sentence*. (This completes parsing.) It means that four parse trees are made.

Load Balancing

Two kinds of parallel execution are applicable to the PAX:

1. Making adjacent nodes can be done in parallel. In the above example, the processes to make an *np* process and a *vp* process can be executed in parallel.
2. When there are two or more node candidates, each process corresponding to node can be executed in parallel. In the above example, there are two candidates for *failing* (*prep* and *adj*), and they can be executed in parallel.

If all the processes mentioned above are distributed to different processors, too much inter-processor communication occurs.Because the messages from some candidates are merged into one stream, which is sent some candidates again. That is N to *one* to M communication occurs.

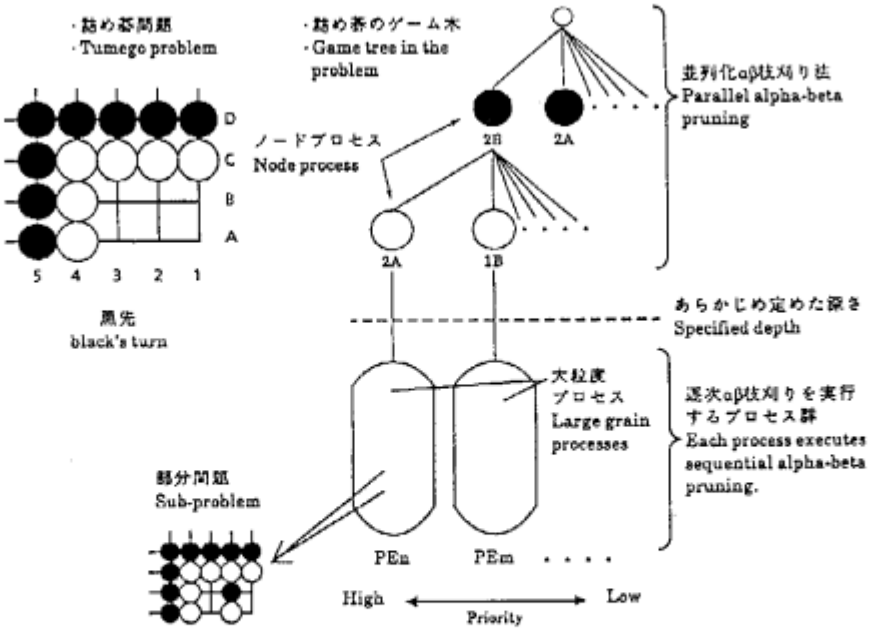
In order to minimize inter-processor communication, we use following load balancing method. Processes that receive a message from the same stream are executed on the same processor.

First, processes that correspond to each word are allocated to different processors. A process corresponding to a word puts its processor number into the head of a stream which connects the process and a process of next word. The process of next word reads this stream and moves to the processor that is designated by the processor number. In this way, inter-processor communication decreases to N to *one*.

Demonstration

PAX analyzes sentences according to the grammar listed in "Oxford Advanced Learner's Dictionary of Current English" which consists of about 420 grammar rules and about 200 words of vocabulary. In the demonstration, PAX analyzes sentences explaining PAX itself and show the change of execution time changing the following parameters.

- The number of processors used.
- The load distribution method; The following two methods are tried.
  - The method which minimizes inter-processor communication by moving the processes to the processors where the accessed data reside.
  - The method where the process never moves.
- The number of consecutive words allocated to one processor.

Title	**Parallel Application Program (2): Tsumego Solver**
Purpose	A parallel algorithm, programming paradigm and load distribution method for the game tree search problem are studied and evaluated implementing a Tsumego solver.
Outline & Features	The solution (life, death or tie) is calculated for a given Tsumego problem and the first move is made. **Characteristics of the problem :** Game tree search (to leaf nodes). **Algorithm :** The alpha-beta pruning method is modified for parallel execution. The search tree is expanded in parallel, and processes corresponding to each node exchange messages with each other. Priority is attached to each process to realize an efficient pruning for the search tree. **Load distribution :** Large grain processes are allocated to idle processors dynamically.
System Configuration	

1 Overview

A parallel algorithm, scheduling and load distribution method for the game tree search problem are studied. The alpha-beta pruning method is modified for parallel execution. Priority is attached to all the node processes to realize an efficient pruning for the search tree. Processes are allocated dynamically in order to balance work loads of all processors.

2 Tsumego problem and solver

Tsumego problem is to determine life, death or tie of the surrounded stones given the Go board state (placement of white and black stones) and whose turn is next. The program shows the first move according to the result.

The program is essentially a game tree search. In the search, the alpha-beta pruning method is used with some modification for parallel execution.

Exhaustive search down to leaf nodes is made in this solver.

3 Alpha-beta pruning for parallel execution

In the conventional alpha-beta search, the game tree search is done in depth first manner. Since the result of a subtree search is used for pruning searches of other part of the game tree, the alpha-beta search has a sequential bottleneck. Therefore, the alpha-beta pruning method should be modified for parallel execution.

If the game tree search were done in purely parallel breadth first manner, there would be no pruning of search space. A priority control is used to realize an efficient pruning for the search tree.

4 Algorithm

The game tree is expanded in one master processor to a certain depth using the parallel alpha-beta pruning method. Blow the depth, the conventional sequential alpha-beta search is executed. Each sequential alpha-beta search is performed by a large grain process, which is the unit of distributing computational load to processors. The parallel alpha-beta search is as follows.

1. Make a move for the given Go board situation. Pick up stones to be captured if any; Trace of adjacent same colored stones and if they are completely surrounded by enemy stones, they are captured.

2. Make judgment whether the game tree is expanded to a certain depth. When the game tree is expanded to a certain depth, sequential alpha-beta search is executed.

3. When not expanded, the search is continued spawning children node processes.

Each node process tests conditions for branch pruning or termination, exchanging information through streams. The information for renewal of alpha and beta values flows downwards, values of terminating nodes upwards, and termination commands caused by pruning downwards.

5 Scheduling using priority

In the parallel alpha-beta search, following three kinds of scheduling are tried.

(a) The moves of the first player are sequentially searched.

(b) The searches of the first player's moves are given priorities so that left-hand tree searches always prior than right-hand ones. The moves of the second player are given the same priorities.

(c) Each search of the first player's moves is given individual priority according the individual game tree. The moves of the second player are given the same priorities.

Scheduling (a) is closer to the sequential alpha-beta search than (b) and (c), and has less parallelism.

It is expected that for a problem instance where the pruning effect in the sequential alpha-beta search is large, scheduling (a) will do well, while (b) and (c) will have good speedup for a problem instance in which the pruning effect of sequential search is small.

6 Load distribution

On-demand dynamic load distribution is used in order to balance work loads of all processors.

When a processor becomes idle, it sends a message requesting a new process to the master processor in which the game tree is expanded in parallel. On receiving the message, the master processor distributes a process as the response to the message.

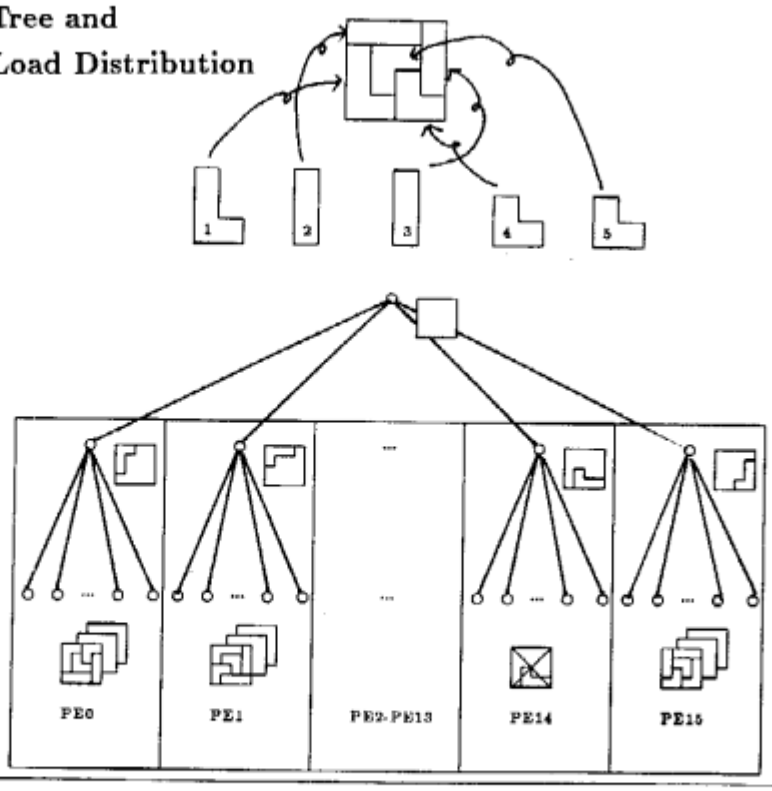
7 Outline of the demonstration

1. Sequential alpha-beta search is tried to the problem where the best move is searched first.

2. Parallel alpha-beta search using 16 PEs is tried to the same problem above. Both scheduling (a) and (b) are used.

3. Parallel alpha-beta search using 16 PEs is tried to the problem where the best move is searched after many moves.

4. Using scheduling (c), parallel alpha-beta search is tried to the same problem above.

Title	**Parallel Application Program (3): Packing Piece Puzzle**
Purpose	Dynamic load balancing scheme for OR-parallel search programs is studied. Multi-level load balancing scheme is proposed, and evaluated by implementing all-solution exhaustive search Pentomino program.
Outline & Features	Packing Piece Puzzle is a puzzle, a rectangular of a collection of pieces with various shapes. The problem is to find all possible ways to pack the pieces into the box. This puzzle is known as the Pentomino puzzle, when the pieces are all made up of 5 squares. This is a typical OR-parallel search program, and multi-level dynamic load balancing scheme is applied. **Program structure:** An OR-parallel exhaustive search by forking tasks at each alternative choice, forming a tree structure. **Load distribution:** Tasks are distributed to idle processing elements (PEs), in order to balance work loads. When a PE becomes idle, it send a message to the master PE, requesting a new task. To overcome the distribution bottlenecck, multi-level load balancing is introduced.
System Configuration	**Search Tree and Load Distribution** 

1 Overview

In the demonstration, packing piece puzzle of 10 pieces(Fig.1) is solved with different number of processing elements (PEs), and speedup by parallel execution and effectiveness of load balancing are shown.

The demonstration is carried out as follows.

- Program is executed on 1 processor, and execution time is displayed.
- Program is executed on 16 processors, and execution time is displayed.
- Load balancing can be observed by the performance meter.
- Near-linear speedup is obtained.

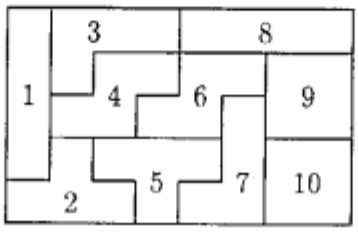


Figure 1: Packing Piece Puzzle

2 Description of the program

To solve this puzzle, the program starts with the empty box, and finds all possible placements of a piece to cover the square at the top left corner, then, for each of those placement, finds all possible placements of a piece (out of the remaining pieces) to cover the uncovered square which is the topmost leftmost, and so on until the box is completely filled. Each partly filled box defines an OR-node, where the possible placements of a piece to cover the uncovered topmost leftmost square define child nodes.

The program does a top-down exhaustive search of this OR-tree. Here, deepening the tree depth corresponds to pack one piece. Number of OR-nodes increases as the search level deepens, but since some OR-nodes are pruned when there are no more possible placements, number of OR-nodes decreases below a certain tree depth.

3 Load balancing scheme

Load balancing is done on master PE by partitioning a program into mutually independent subtasks (Subtask Generation), and by distributing subtasks to idle PEs so as to balance work loads (Subtask Allocation). To detect idle PEs, on-demand distribution method is utilized. When a PE becomes idle, it sends a message to the master PE, requesting a new subtask. Subtask generation is done until the search reaches the certain depth in the tree.

However, as the number of processors increases, the rate of subtask execution eventually becomes larger than the rate of subtask supply. In other words, subtask generation becomes a bottleneck.

To overcome this bottleneck, we have introduced multi-level load balancing scheme. Each subtask generator is in charge of a certain fixed number of processors, which form processor groups (PG). N processors are grouped into M processor groups, therefore, each PG is composed with $\frac{N}{M}$ PEs and a certain PE in a PG is called group master PE.

In Fig.2, two-level load balancing scheme is shown. At the first level distribution, super-subtasks are distributed to idle PEs to balance the loads of PGs. At the second level, subtasks are distributed to idle PEs to balance the loads of PEs which belong to a PG.

This scheme is scalable to any number of processors because of this multi-level structure.

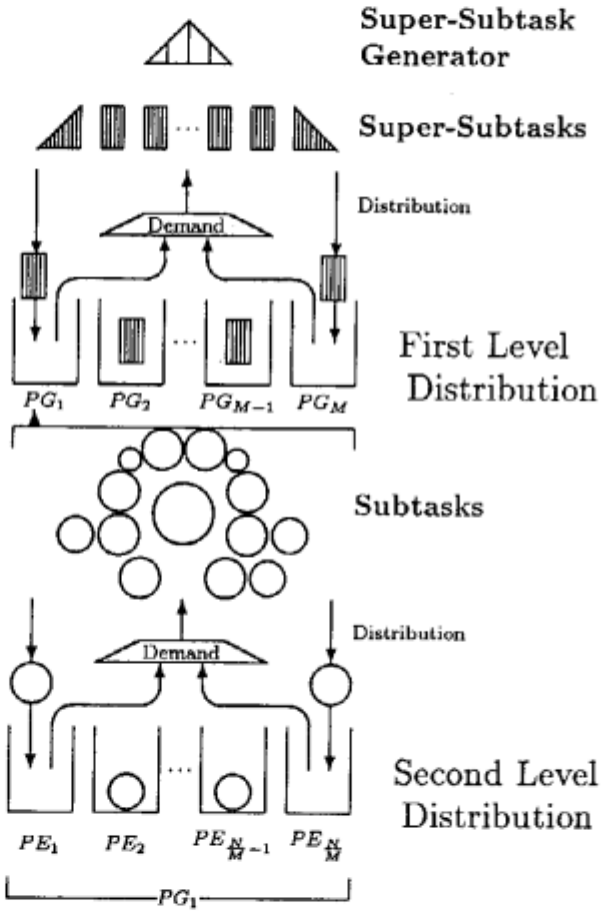


Figure 2: Structure of Multi-Level Load Balancing

4 Speedup Measurement

Execution times are measured for one-level load balancing and two-level load balancing. Speedup (S_N) is defined as the ratio of execution time on 1 PE (T_1) to N PEs (T_N), and calculated by $\frac{T_1}{T_N}$, and it is described in Figure 3.

Speedup of one-level load balancing is getting saturated because of the subtask generation bottleneck. However, it is improved by two-level load balancing, and near-linear speedups are obtained: 7.7 with 8 PEs, 15 with 16 PEs, 28.4 with 32 PEs, 50 with 64 PEs.

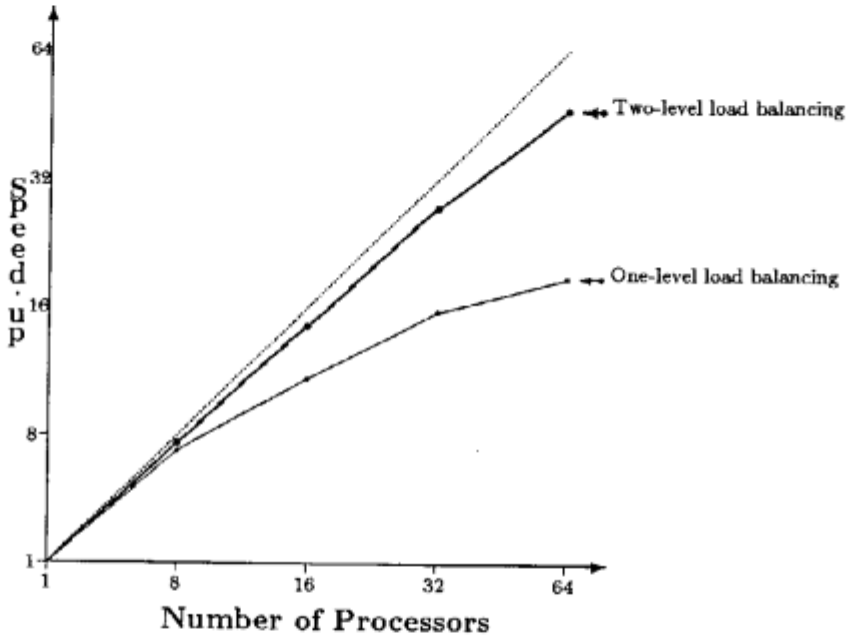
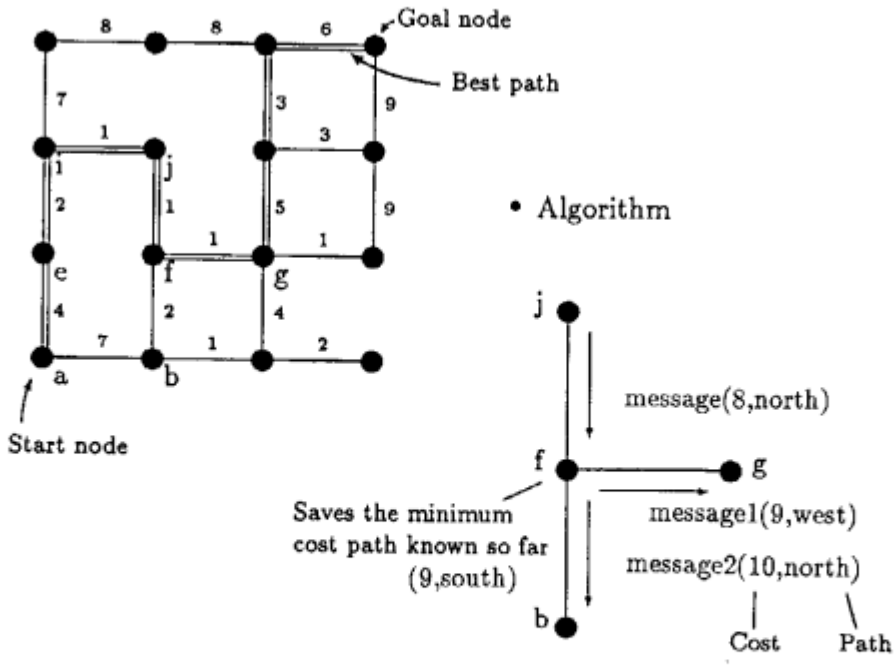


Figure 3: Speedups

5 Conclusion and Future Works

This scheme is efficient not only for OR-parallel search problems, but also applicable to some types of search problems such as alpha-beta pruning problems, which does not involve frequent inter-processor communication. Applying the multi-level load balancing scheme to such programs is our future works.

Title	**Parallel Application Program (4): Shortest Path Problem Solver**
Purpose	A parallel algorithm, programming paradigm and load distribution method for the best solution search problem are studied and evaluated by implementing a shortest path problem solver.
Outline & Features	The single-source shortest path problem is to search for the minimum cost paths between a given start node and all other nodes of a network in which each network branch has a non-negative cost. Large networks with tens of thousands of nodes are generated using random numbers as the test data. **Type of the problem :** Best solution search. **Algorithm :** Processes corresponding to each network node exchange messages with each other. Each message contains path and cost from the start node. Priority is attached to each message so that a message with lower cost is sent earlier than a message with higher cost. Each node remembers the shortest path notified by the messages arrived so far and its cost. **Programming technique :** A message is represented by a process so that a message has a priority. **Load distribution :** Making more processors work for the part of the network where communication is dense.
System Configuration	 Message1 will be sent earlier than message2.

Outline

The single-source shortest path problem is to find the minimum cost paths between a given start node and all other nodes of a network in which each network branch has a non-negative cost. In the demonstration, the network consists of about ten thousand nodes and is generated using random numbers.

In the demonstrated program, processes are generated for each network node and computation is performed by exchanging messages between them. The order of required computation with this algorithm is smaller than that with the algorithm in which processes are forked for each candidate path. Using priority control, efficient pruning for the search branches is done. As a result of that, the program works in the same order of computational complexity as well-known Dijkstra's algorithm.

Algorithm

A message contains the path from the start node to the receiver node and its cost. The computation is initiated by sending a message with an empty path and zero cost to the start node. All the nodes remember the minimum cost to reach the node notified by the messages received so far. Initially, the cost remembered by all the nodes is infinite(Figure 1).

When a message arrives at a node and the cost notified by the message is smaller than the minimum cost remembered in the node, the new cost is saved and messages with better paths and costs are sent further to the adjacent nodes (Figure 2). If the message has a larger cost value than the known minimum, it is simply discarded.

Since a message is represented by a process, sending message means a creation of a message process, while receiving message means an execution of a message process. Each message process has a priority decided by the cost. Thus, a message with a lower cost is received earlier than a message with a higher cost.

When all the messages on the network are discarded, each node has the shortest path from the start node and its cost.

Load Balancing

The heaviest part of the processing is communication, and the communication is initiated at the start node and propagates gradually to the whole network in waves.

The program tries to balance the load based on the following two ideas.

- Divide the network into sub-networks and distribute processes for sub-networks to distinct processors.
- Make more processors work for the part of the network where communication is dense.

Mapping Strategies

The following three mapping strategies are tried. In each mapping, $p = q^2$ processors are employed.

Two-Dimensional Simple Mapping

Divide the network into $q \times q$ sub-networks and map each sub-network onto the corresponding processor.

Two-Dimensional Multiple Mapping

Divide the network into k super-sub-networks, each of which is again divided into p sub-networks just as in the two-dimensional simple mapping. Each processor is responsible for k sub-networks, each one from each super-sub-network.

One-Dimensional Simple Mapping

Divide the network simply as p narrow rectangular strips and map them onto the processors.

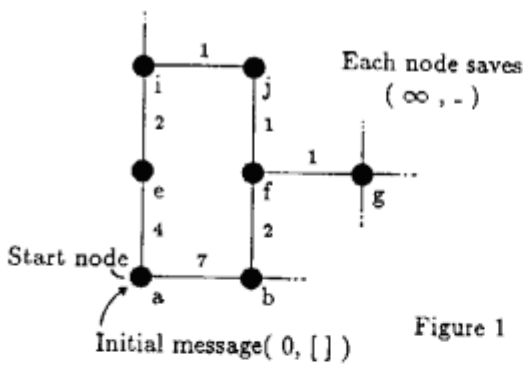


Figure 1

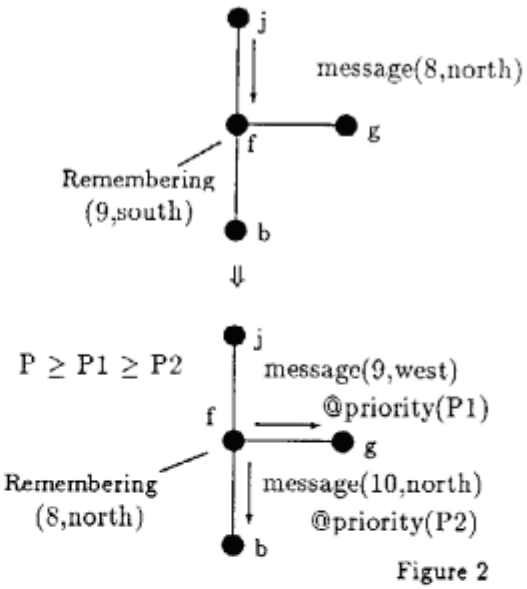
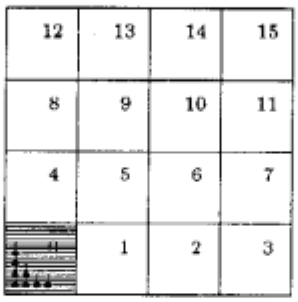
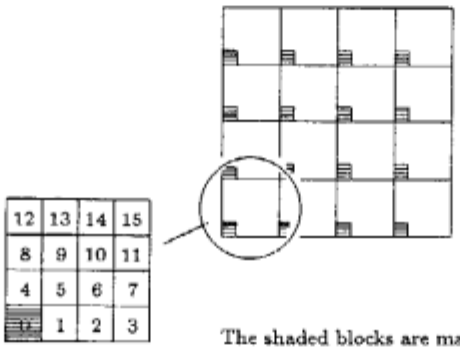


Figure 2



The shaded block is mapped onto processor 0.

Figure 3: The decomposition of a graph for the two-dimensional simple mapping

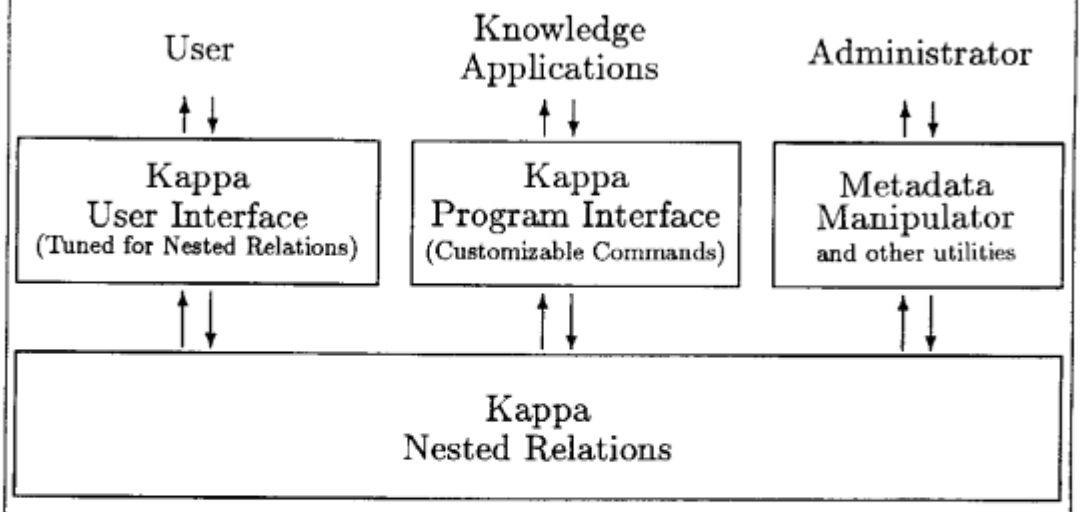


The shaded blocks are mapped onto processor 0.

Figure 4: The decomposition of a graph for the two-dimensional multiple mapping

The shaded block is mapped onto processor 0.

Figure 5: The decomposition of a graph for the one-dimensional simple mapping

Title	Knowledge Application Oriented Advanced DBMS : Kappa
Purpose	Kappa, a DBMS based on the nested relational model, is implemented on PSI-II in order to study management of very large and/or complex structured databases in the logic programming environment and to provide a platform for implementing various knowledge applications (including deductive DBs and semantic networks).
Outline & Features	Kappa is a DBMS with the following features: 1. Nested relational model is adopted. 2. Terms stored as one data type are retrieved by unification. 3. Large amounts of data, such as electronic dictionaries, mathematical knowledge and genetic information, are effectively accessed. 4. An User interface suitable for nested relations and program interface customizable for various applications are provided. 5. It's written in object-oriented logic programming language ESP, and will be rewritten in parallel language KL1.
System Configuration	

GenBank Sequence DB on Kappa

1 Guidance

1.1 Features of Nested Relational Model

The definition of the nested relational model (according to the standpoint of Kappa) is:

$$NR \subseteq E_1 \times \ldots \times E_n$$
$$E_i ::= D \mid 2^{NR}$$

while that of the relational model is:

$$R \subseteq D_1 \times \ldots \times D_n$$

where D_i is a domain, R is a relation and NR is a nested relation.

In Kappa (the nested relational model) :

1. We can represent tree-structure naturally.
   - It's suitable for complex structured data.
   - It's more user-friendly than the relational model.
2. We can utilize features of the relational model.
   - Extended relational algebra is available.
   - Entity-relationship concept are effective at the design and the management phases.
   - Deductive database system is implemented on Kappa.

1.2 Schema of GenBank/Kappa

Schema based on nested relational model for GenBank data is shown in Fig. 1 in detail.

gene : main table which has locus name, definition, accession, keywords, identifiers to the other tables, and so on.

reference : table which has its authors, title, journal where it has appeared, and so on.

feature : consists of a region of the sequence and its feature.

seqdata : sequence data represented in string form.

1.3 Stored Data

Data we stored for this demonstration is sequences of invertebrate, virus, bacteria and phage. The total amount is 7285 entries, 10 mega bases, and 22 mega characters in the original data, which is stored into 60 mega bytes (30 mega characters) including 10 indexes and about 16 mega bytes (8 mage characters) of *'textdata'* in Kappa.

GenBank sequence DB (89.6.15) has 26323 entries, 32 Mega bases. So it will cost about 240 mega bytes in Kappa.

2 Demonstration

2.1 Show Schema : Metadata Manipulator

We can see the schema of the database.

We show the schema of all four tables of GenBank/Kappa (gene, seqdata, reference and feature, Fig. 1) through metadata manipulator.

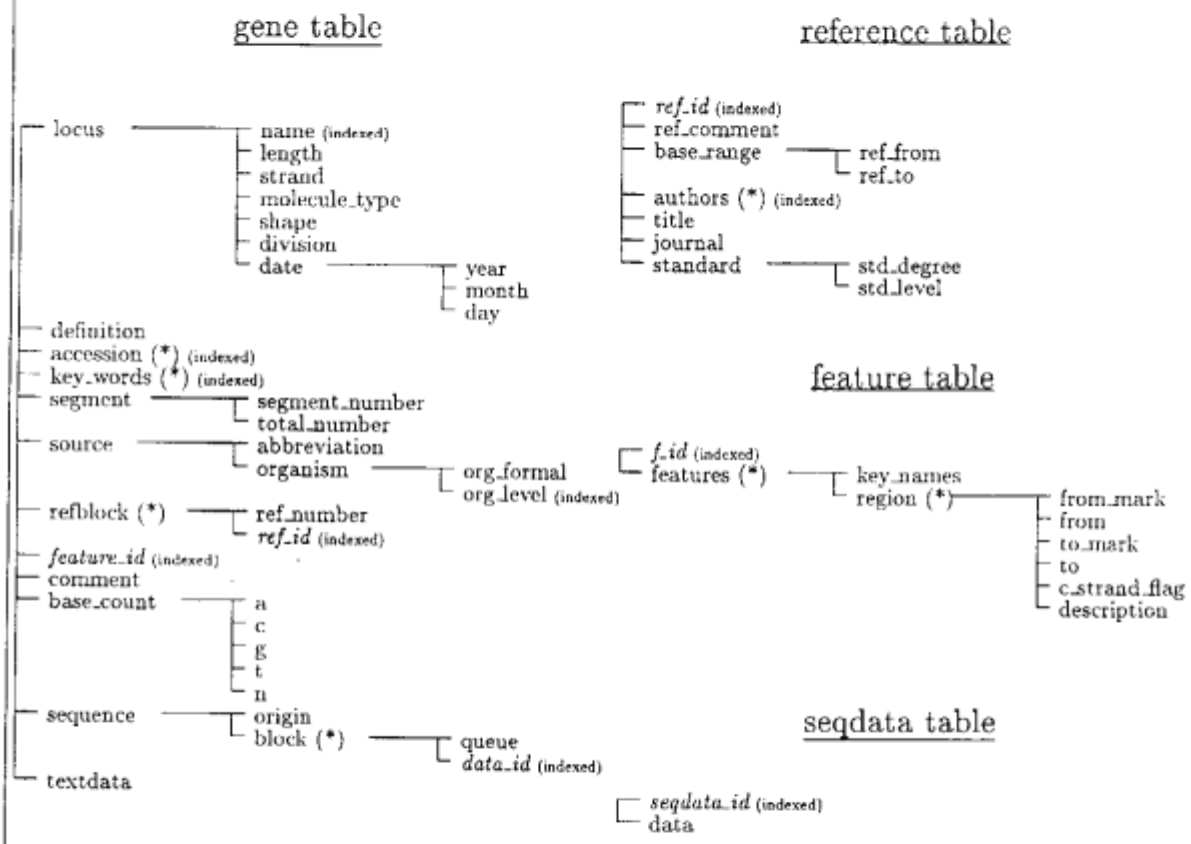


Fig. 1 *Schema of GenBank/Kappa* ('(*)' means repeating)

2.2 Display Tables : Kappa User Interface

We can see how the data is stored.

We show all four tables by projecting into smaller tables (Fig 2.1), and the contents of their attributes by clicking each cell. Kappa can have attributes with variable length directly in each record. Now in the attribute *textdata* we can see the excerpt of each flat gene data as it is in GenBank original (Fig 2.2).

Kappa can also have attributes with multi-values. So it is necessary to scroll values in the user interface.

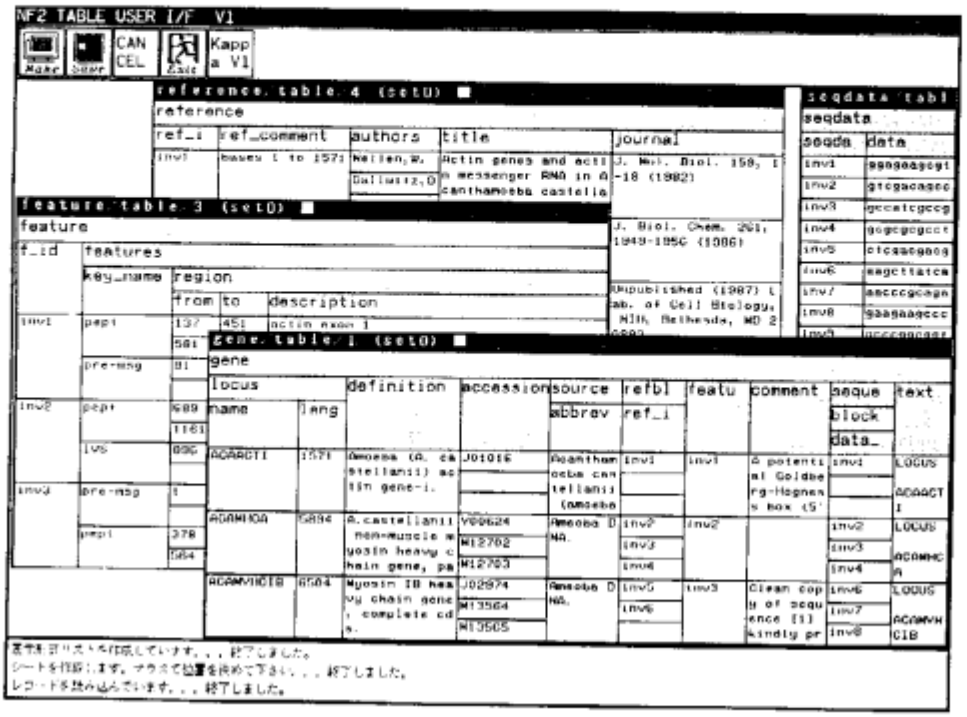


Fig. 2.1 *Kappa User Interface*

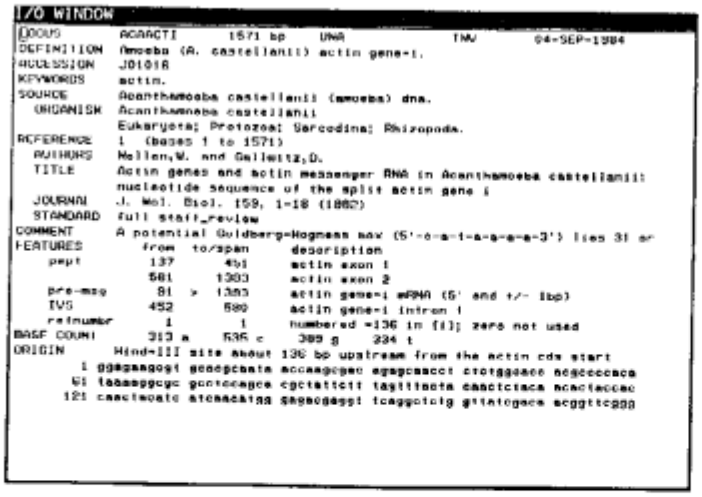


Fig. 2.2 *GenBank Original Textdata*

2.3 Retrieve Data

We can retrieve data by various conditions.

Example: make a table of genes whose reference Mr. Smith,C. writes.

$$\pi_{\text{ref_id}}\left(\sigma_{\text{authors='Smith,C.'}}(\text{reference})\right) \bowtie \text{gene}$$

Tables and attributes are shown in Fig. 3.

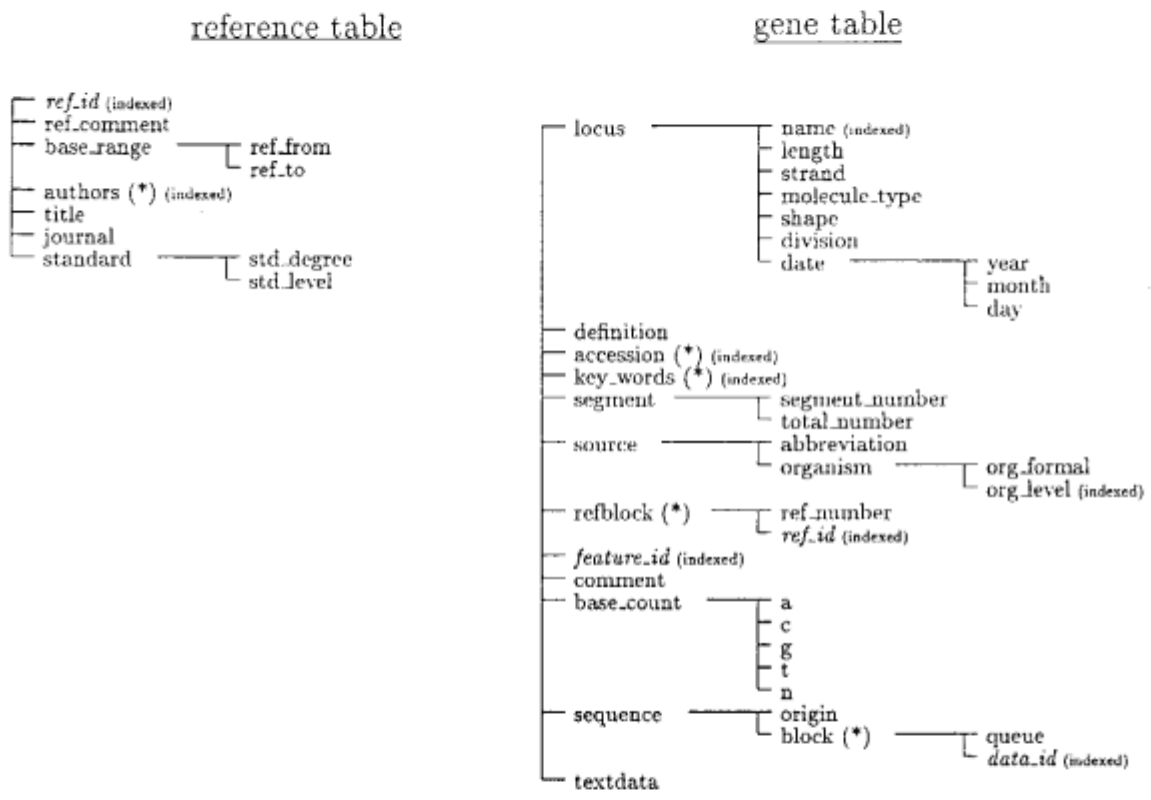


Fig. 3 *Tables and Attributes*('(*)' means repeating)

2.4 Translate into Protein & DP matching

We can translate DNA code into amino acid sequence and execute DP matching.

Flowchart is shown in Fig. 4.1.

Translation We select a DNA sequence in the feature table to translate into an amino acid sequence. The sequence is translated according to the table shown in Fig. 4.2.

DP matching We compare the *'translated'* sequence with the object sequence determined in advance. The sequences are compared according to the table selected by the user shown in Fig. 4.3 for example.

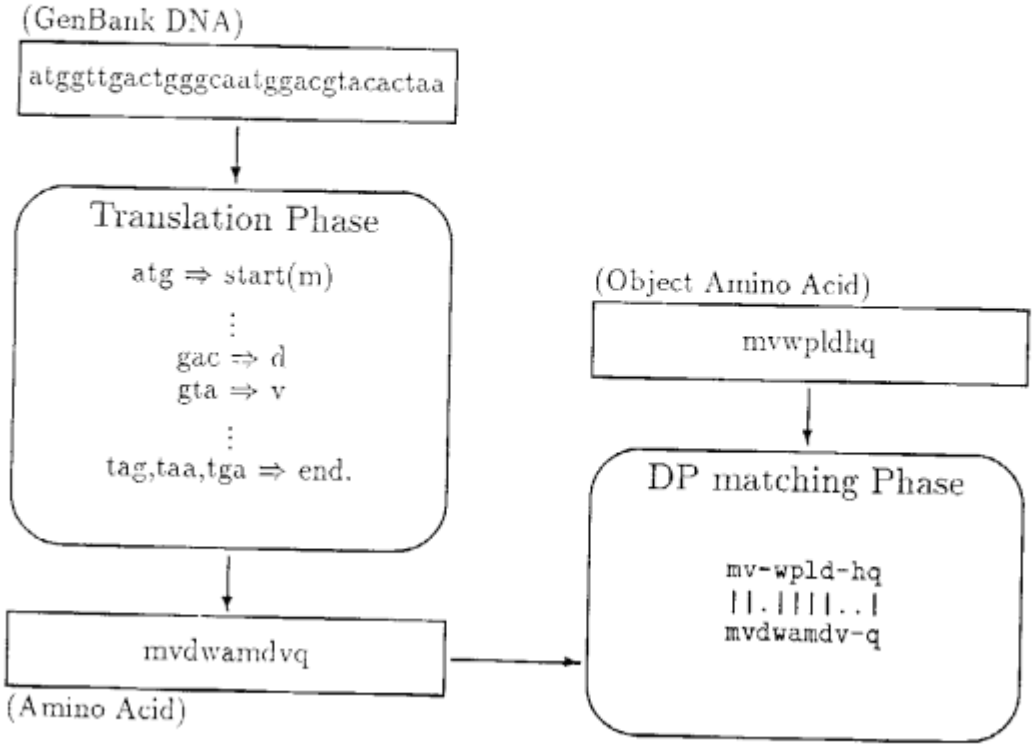


Fig. 4.1 *Translation and DP matching*

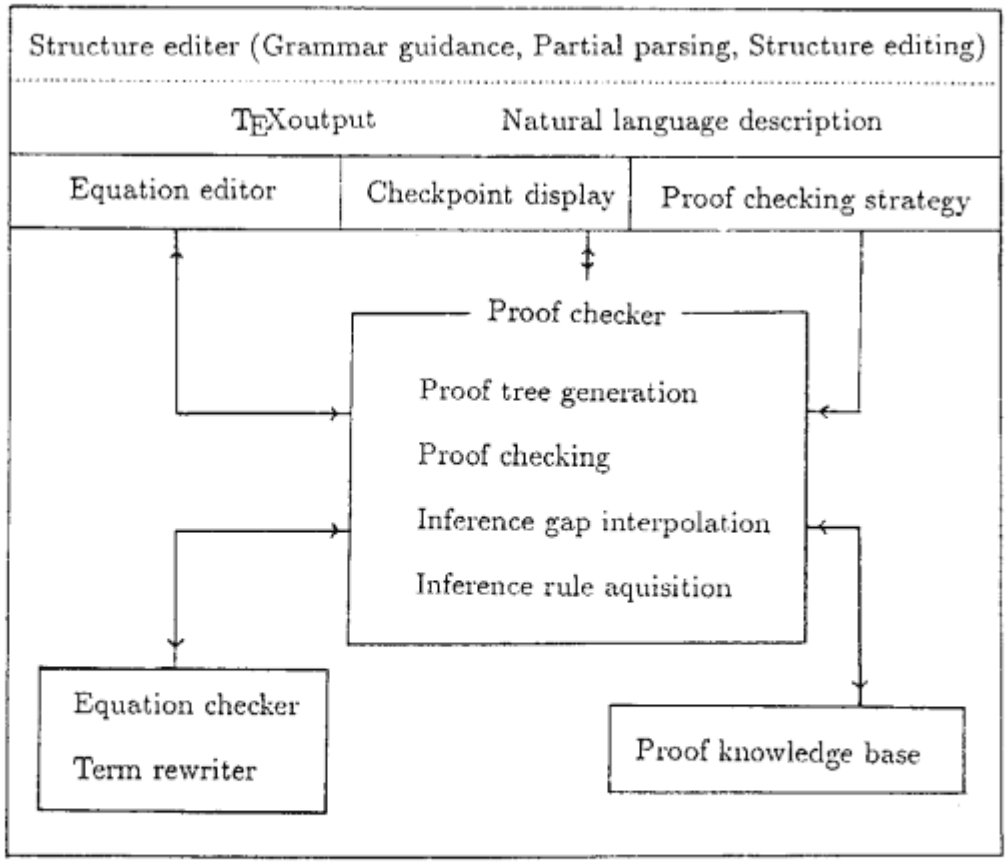
Fig. 4.2 *DNA-Amino Acid Table*

1st (5'end)	2nd T	2nd C	2nd A	2nd G	3rd (3'end)
T	f	s	y	c	T
					C
	l		*	*	A
			*	w	G
C	l	p	h	r	T
					C
			q		A
					G
A	i	t	n	s	T
					C
			k	r	A
	m				G
G	v	a	d	g	T
					C
			e		A
					G

(* terminate codon)

Fig. 4.3 *Amino Acid Difference Table*

Mutability	c	s	t	p	a	g	n	d	e
c: cysteine	6								
s: serine	4	6							
t: threonine	2	5	6						
p: proline	2	4	4	6					
a: alanine	2	5	5	5	6				
g: glycine	3	5	2	3	5	6			
n: asparagine	2	5	4	2	3	3	6		
d: aspartic acid	1	3	2	2	4	4	5	6	
e: glutamic acid	0	3	3	3	4	4	3	5	6
q: glutamine	1	3	3	3	3	2	3	4	4
h: histidine	2	3	2	3	2	1	4	3	2
r: arginine	2	3	3	3	2	3	2	2	2
k: lysine	0	3	4	2	3	2	4	3	4
m: mathionine	2	1	3	2	2	1	1	0	1
i: isoleucine	2	2	3	2	2	2	2	1	1
l: leucine	2	2	2	3	2	2	1	1	1
v: varine	2	2	3	3	5	4	2	3	4
f: phenylalanine	3	3	1	2	2	1	1	1	0
y: tyrosine	3	3	2	2	2	2	3	2	1
w: tryptophan	3	2	1	2	2	3	0	0	1

Title	Computer Aided Proof (CAP-LA)
Purpose	(1) Man-machine cooperation in mathematical problem solving (2) Assistance in proof checking and formula manipulation
Outline & Features	**outline** (1) Checking theorems and proofs in linear algebra (2) Assistance in writing therorems and proofs **features** (1) Checking proofs with inference gaps (2) Interactive checking and debugging (3) Proof-structure-oriented writing and editing
System Configuration	

D e t a i l s (1/3)

Proof written in PDL (Proof Description Language)

```
theorem
  det_trans:
  all A:squ_mat.
    det(A)=det(trans(A))
  proof
    let A:squ_mat be arbitrary;
    row(A)=col(trans(A));
    col(A)=row(trans(A));
    det(A)=sigma p:perm<col(A)>.
              (sgn(p)*pi i:1..col(A).
                        A[p(i), i])
           =sigma p:perm<col(A)>.
              (sgn(inv(p))*pi i:1..col(A).
                          A[(inv(p))(i), i])
            using sigma_pi
           =sigma p:perm<col(A)>.
              (sgn(p)*pi i:1..col(A).
                        A[(inv(p))(i), p(i)])
            using sigma_pi
           =sigma p:perm<col(A)>.
              (sgn(p)*pi i:1..col(A).
                        A[(inv(p))(p(i)), p(i)])
            using sigma_pi
           =sigma p:perm<col(A)>.
              (sgn(p)*pi i:1..col(A).
                        A[i, p(i)])
            using sigma_pi
           =sigma p:perm<col(A)>.
              (sgn(p)*pi i:1..col(A).
                        (trans(A))[p(i), i])
            using sigma_pi
           =det(trans(A))
  end_proof
end_theorem
```

TEX output

THEOREM1: *det_trans:*
For all A ∈square matrix,

$$\det(A) = \det({}^tA)$$

PROOF:
Now let A ∈square matrix be arbitrary.

$$\mathrm{row}(A) = \mathrm{col}({}^tA)$$

$$\mathrm{col}(A) = \mathrm{row}({}^tA)$$

$$\det(A) = \sum_{p \in S_{\mathrm{col}(A)}} \mathrm{sgn}(p) \prod_{i=1}^{\mathrm{col}(A)} A_{p(i),i}$$

$$= \sum_{p \in S_{\mathrm{col}(A)}} \mathrm{sgn}(p^{-1}) \prod_{i=1}^{\mathrm{col}(A)} A_{p^{-1}(i),i}$$

using sigma_pi

$$= \sum_{p \in S_{\mathrm{col}(A)}} \mathrm{sgn}(p) \prod_{i=1}^{\mathrm{col}(A)} A_{p^{-1}(i),p(i)}$$

using sigma_pi

$$= \sum_{p \in S_{\mathrm{col}(A)}} \mathrm{sgn}(p) \prod_{i=1}^{\mathrm{col}(A)} A_{p^{-1}(p(i)),p(i)}$$

using sigma_pi

D e t a i l s (2／3)

Equation Editor

(1) Initialization

```
CAP-LA_48   ===> Structure Mode...
theorem
  det_trans:
  all A:squ_mat.
    det(A)=det(trans(A))
  since
    let A:squ_mat be arbitrary;
    det(A)=det(trans(A))
  end_since
end_theorem
SEMACS(esp)[95,15] demo11 sys>user>
Read: icps:180::>sys>user>semacs>te
```

```
EQUALITY EDITOR ==> RIGHT HAND SIDE
det(A)

det(trans(A))
SEMACS(esp)[57,6] *48/2*   --Top-- *
```

(2) Formula manipulation

```
EQUALITY EDITOR ==> LEFT HAND SIDE
sigma p:perm(col(A)),
  ((sign(p)*pi i:1..col(A),
          (A[i, (p)(i)]))))

det(trans(A))
SEMACS(esp)[57,6] *48/1*   --Top-- *
this rule (y/n/a(bort))?y
replace(i_1): p
replace(i_2): i
```

```
RULE WINDOW
Rule Tag:def func det
Condition:(A:squ_mat)
```

(3) Final result

```
CAP-LA_48   ===> Structure Mode...
  all A:squ_mat.
    det(A)=det(trans(A))
  since
    let A:squ_mat be arbitrary;
    det(A)=sigma p:perm(col(A)),
              ((sign(p)*pi i:1..col(A),
                          (A[i, (p)(i)]))))
           *sigma p:perm(col(trans(A))),
              ((sign(p)*pi i:1..col(trans(A)),
                          (A[(p)(i), i]))))
           *sigma p:perm(col(trans(A))),
              ((sign(p)*pi i:1..col(trans(A)),
                          ((trans(A))[i, (p)(i)]))))
           *det(trans(A))
  end_since
SEMACS(esp)[95,15] demo11 sys>user>semacs>text>det_trans..1 --4%-- *
```

D e t a i l s (3/3)

Structure-oriented proof development

(This example illustrates "universal quantifier elimination")

Before elimination

```
CAP-LA_48   ===> Structure Mode...
theorem
  det_trans:
  all A:squ_mat.
    det(A)=det(trans(A))
end_theorem

                                  menu
                                    <rule_menu>
                                    univ_elim
                                    contradiction
```

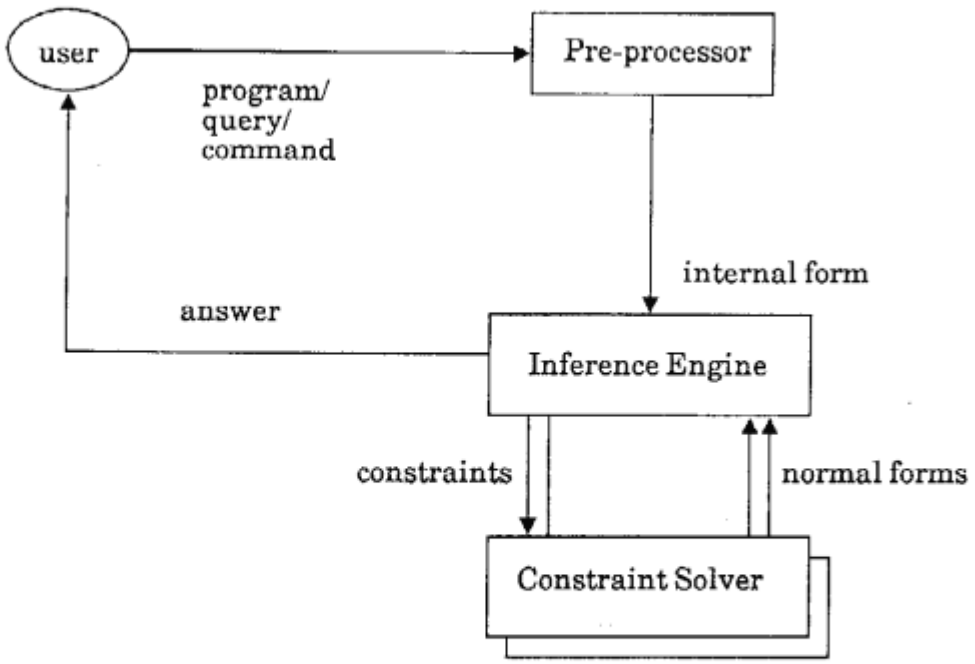
After elimination

```
CAP-LA_48   ===> Structure Mode...
theorem
  det_trans:
  all A:squ_mat.
    det(A)=det(trans(A))
  since
    let A:squ_mat be arbitrary;
    det(A)=det(trans(A))
  end_since
end_theorem
```

Proof checking (This example illustrates an error in indexing)

```
CAP-LA_55   ===> Structure Mode...
theorem
  trans_trans:
  all m, n:pos, A:matrix<m, n>.
    (trans(trans(A))=A)
  since
    let m, n:pos, A:matrix<m, n> be arbitrary;
    trans(trans(A))=A
                    since using theorem mat_EQ:
                      col(trans(trans(A)))=row(trans(A))
                                          =col(A);
                      row(trans(trans(A)))=col(trans(A))
                                          =row(A);
                      all i:1..col(trans(trans(A))), j:1..row(A).
                        (trans(trans(A)))[i, j]=(A)[i, j]
                      since
                        let i:1..col(trans(trans(A))), j:1..row(A) be arbitrary;
                        (trans(trans(A)))[i, j]=(trans(A))[i, j]
                                                =(A)[i, j]
                      end_since
SEMACS(esp)[96,19]  sys>user>semacs>text>TRTR.88.1 --29%-- *
Grammar name? >pd1
Top Category name(theory)? >
Buffer(TRTR): ^G
CAP CHECKER
AUNIFY MONITOR

 goal:(trans(trans(A)))[i,j] = (trans(A))[i,j]
 rule:(trans(_A))[_B,_C] = _A[_C,_B]
found difference of...
i and _B, j and _C, i and _C, j and _B

RULE POOL MONITOR

restore function : trans  ...  O.K.

EQUALITY TRS MONITOR

(6,2)Goal>(trans(trans(A)))[i,j] = (trans(A))[i,j] : (by equa
lity)

press any key
```

```
TRS RULES

<EQUALITY>
>>col(trans(A)) = n
>>row(trans(trans(A))) = n
>>col(trans(trans(A))) = m
>>row(trans(A)) = m
>>row(A) = n
>>col(A) = m
<INEQUALITY>
>>n >= 1
>>n >= j
>>j >= 1
>>m >= 1
>>m >= i
>>i >= 1
```

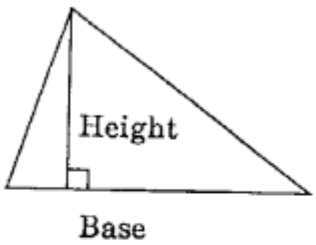
Title	**Constraint Logic Programming Experimental System CAL**
Purpose	1) Programs easier to write and read 2) Highly abstract programming 3) Research on efficient problem solving techniques
Outline & Features	1) Amalgamation of logic programming and constraint solving 2) Multiple Constraint Solvers 3) Solution of non-linear algebraic equations and Boolean equations 4) Natural extension of Prolog
System Configuration	

★ Examples of CAL : Heron's Formula(non-linear)

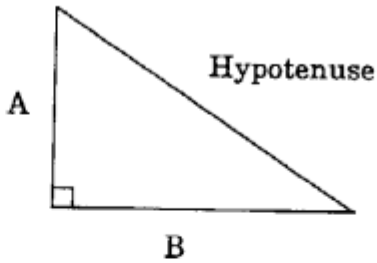
Program & Query&Answer

```
:- public triangle/4, triangle1/4.
surface(Height, Base, Area) :- Base * Height = 2*Area.
pythagoras(A, B, Hypotenuse) :- A^2 + B^2 = Hypotenuse^2.
triangle(A, B, C, S) :-
        C = CA + CB:alg,
        pythagoras(CA, H, A),
        pythagoras(CB, H, B),
        surface(H, C, S).
triangle1(A, B, C, S) :- precedence(S :> 0), triangle(A, B, C, S).

?- heron:triangle1(a, b, c, s).

s^2 =  -1/16*b^4+1/8*a^2*b^2-
       1/16*a^4+1/8*c^2*b^2+1/8*c^2*a^2-1/16*c^4
```

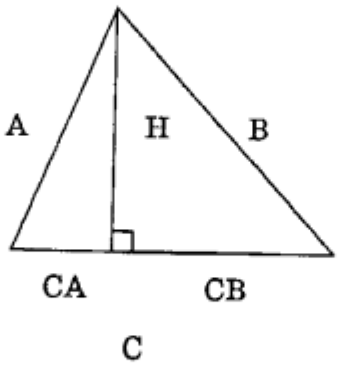
surface:



pythagoras:



triangle:



★ Examples of CAL : Cone Volume using Lagrange(dynamic constraints)

Program & Query&Answer

```
:- public lagrange/3.
lagrange(F, Constraints, Vars) :-
        construct_l(Constraints, L),
        partials(Vars, F, L).
partials([ ], _, _) :- !.
partials([Var|Vars], F, L) :-
        dif(F, Var) = dif(L, Var):alg,
        partials(Vars, F, L).
construct_l([ ], 0) :- !.
construct_l([C|Cs], C*Alpha + L) :-
        C = 0:alg,
        construct_l(Cs, L), !.

?- lagrange((1/2+r)^2*a+(1/2-r)^2*b,
            [a^2+(1/2+r)^2 = 1, b^2+(1/2-r)^2 = 1],
            [a, b, r]).

r^7=(29/12)*r^5+(-17/48)*r^3+(5/576)*r
```

Maximize $f(x_1, \ldots)$
s. t. $c_1(x_1, \ldots) = \ldots = 0$

Lagrange ⇒

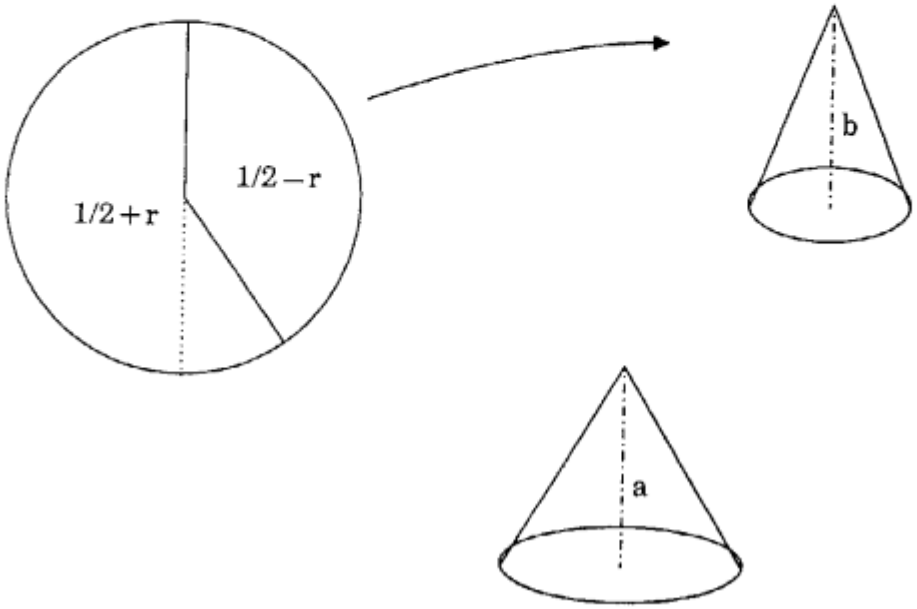
Solve
$\partial l/\partial x_1 = \partial f/\partial x_1, \ldots$
$c_1(x_1, \ldots) = \ldots = 0$
where $l(x_1, \ldots) = c_1(x_1, \ldots)\alpha_1 + \ldots$

Maximize $f(a, b) = a(1/2+r)^2 + b(1/2-r)^2$
s. t. $a^2+(1/2+r)^2-1=0$,
$b^2+(1/2-r)^2-1=0$

Lagrange

Solve
$a^2+(1/2+r)^2-1=b^2+(1/2-r)^2-1=0$
$\partial l/\partial a = \partial f/\partial a, \partial l/\partial b = \partial f/\partial b$
where $l = (a^2+(1/2+r)^2-1)\alpha_1 + (b^2+(1/2-r)^2-1)\alpha_2$

D e t a i l s ($^3/_5$)



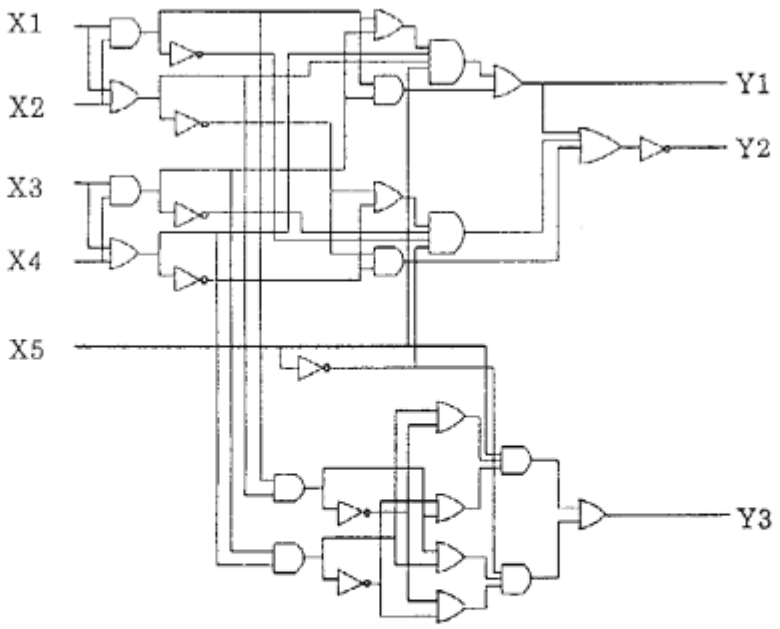
Query

```
?-lagrange((1/2+r)^2*a+(1/2-r)^2*b,
           [a^2+(1/2+r)^2 = 1, b^2+(1/2-r)^2 = 1],
           [a, b, r]).
```

★ Examples of Boolean CAL : Counter Circuit

Program&Query&Answer

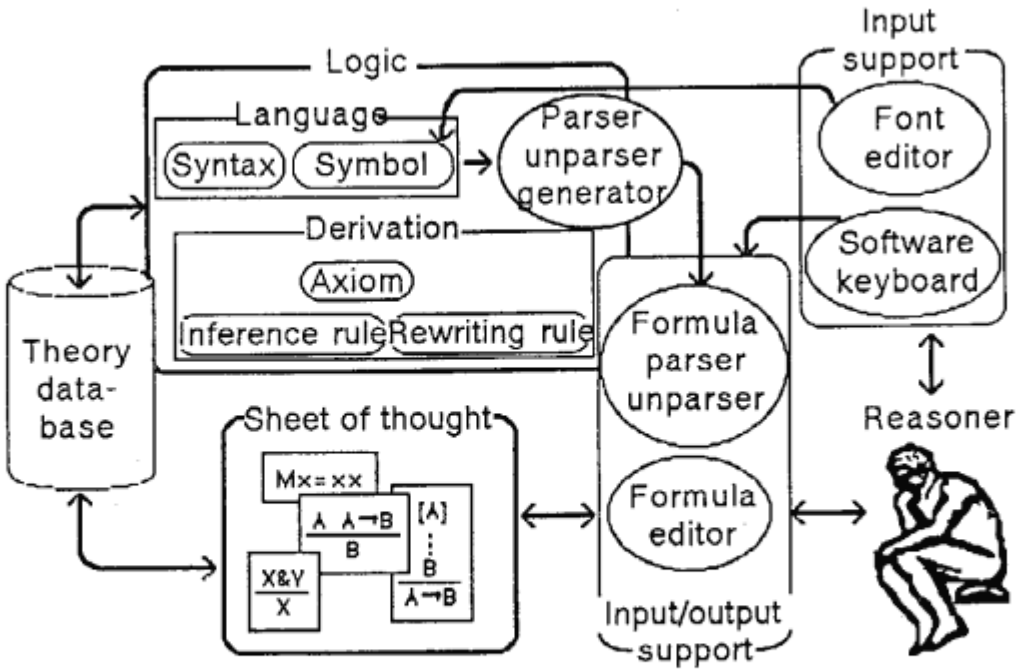
```
:- public circuit/8.
circuit(X1, X2, X3, X4, X5, Y1, Y2, Y3) :-
    I1 = X1&X2:bool,      I2 = X1 V X2:bool,      I3 = X3&X4:bool,
    I4 = X3 V X4:bool,    I5 = I1:bool,           I6 = I2:bool,
    I7 = I3:bool,         I8 = I4:bool,           I9 = I1 V I3:bool,
    I10 = I1&I3:bool,     I11 = I6 V I8:bool,     I12 = I6&I8:bool,
    I13 = X5:bool,        I14 = I5&I2:bool,       I15 = I7&I4:bool,
    I16 = I14:bool,       I17 = I15:bool,         I18 = I15 V I16:bool,
    I19 = I14 V I17:bool, I20 = I14 V I15:bool,   I21 = I16 V I17:bool,
    I22 = I9&I4&I2&X5:bool,   I23 = I11&I7&I5&I13:bool,
    I24 = X5&I18&I19:bool,    I25 = I13&I20&I21:bool,
    I26 = I22 V I10:bool,     I27 = I26 V I23 V I2:bool,
    Y1 = I26:bool,        Y2 = I27:bool.          Y3 = I24 V I25:bool.

?- count:circuit(x1,x2,x3,x4,x5,1,0,1).
?- count:circuit(1,x2,x3,x4,x5,y1,y2,y3), x2 = x3 : bool.x5 = 1.
```



★ Conclusions

- *Constraint Logic Programming*
  1) Constraint Logic Programming extends unification to constraint solving, allowing symbolic answers to queries
  2) Powerful semantically clean, language in which to generate and solve constraints.
  3) More powerful pruning of search space.

- *CAL*
  1) Solves linear and non-linear equalities over complex numbers using Gröbner Base method.
  2) Solves Boolean constraints using Boolean Gröbner Base method developped at ICOT.

Title	**A General-Purpose Reasoning Assistant System EUODHILOS**
Purpose	EUODHILOS (Every Universe Of Discourse Has Its LOgical Structure) is a general-purpose reasoning assistant system that allows users to interactively define the syntax and inference rules of a formal system and to construct proofs in the defined system. *(It can be specified as ∀Universe∃Logic.EUODHILOS(Universe, Logic).)*
Outline & Features	(1) Formal system description language (a) Language system (symbols, terms, formulas, etc.) ♦ definite clause grammar formalism ♦ automatic generation of a bottom-up parser, an unparser and internal structures of expressions (b) Derivation system (axioms, inference rules, etc.) ♦ inference rules and derived rules: natural deduction style ♦ rewriting rules: definition by a pair of forms before and after rewriting (2) Proof construction facilities ♦ Sheets of thought: a field of thought where we are allowed to compose a proof from its fragments, to separate a proof, or to reason using lemmas, etc. ♦ Proving methodology based on several sheets of thought: forward reasoning, backward reasoning, reasoning in a mixture of them, schematic proof, etc. ♦ Tree-form proof with justifications indicated in the right margin (3) Visual human-computer interface for reasoning: formula editor, software keyboard, stationery for reasoning
System Configuration	

A Scenario of the demonstration

- a case of the intuitionistic type theory -

(1) Specifying a logic

First we define the language system to be used by the definite clause grammar formalism, for example, the judgement in the intuitionistic type theory is defined as follows :

judgement --> term, epsilon, type ;
epsilon --> "∈" ;

Then, we define a rule of inference, a derived rule and a rewriting rule in a natural deduction style presentation, for example, the λ-introduction rule and the negation are defined respectively as follows :

$$\frac{\begin{array}{c}[x \in A] \\ \vdots \\ F[x] \in B\end{array}}{\lambda x.F[x] \in A \supset B}\ (\lambda\text{-I}) \qquad \frac{A \supset \bot}{\sim A}\ (\text{def})$$

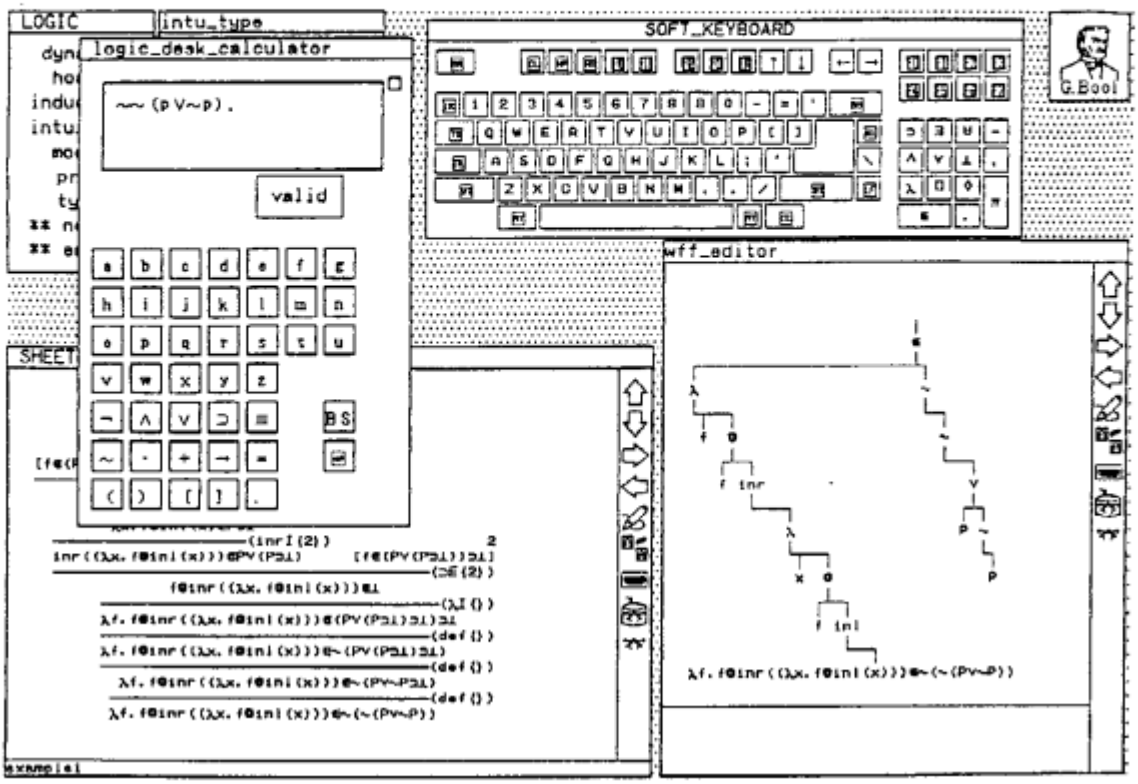
(2) Constructing a proof in the defined logic

The proof is interactively constructed in a tree-form on sheets of thought through reasoning forward or backward and/or connecting or separating several proof fragments. A proof example in the intuitionistic type theory proceeds as follows :

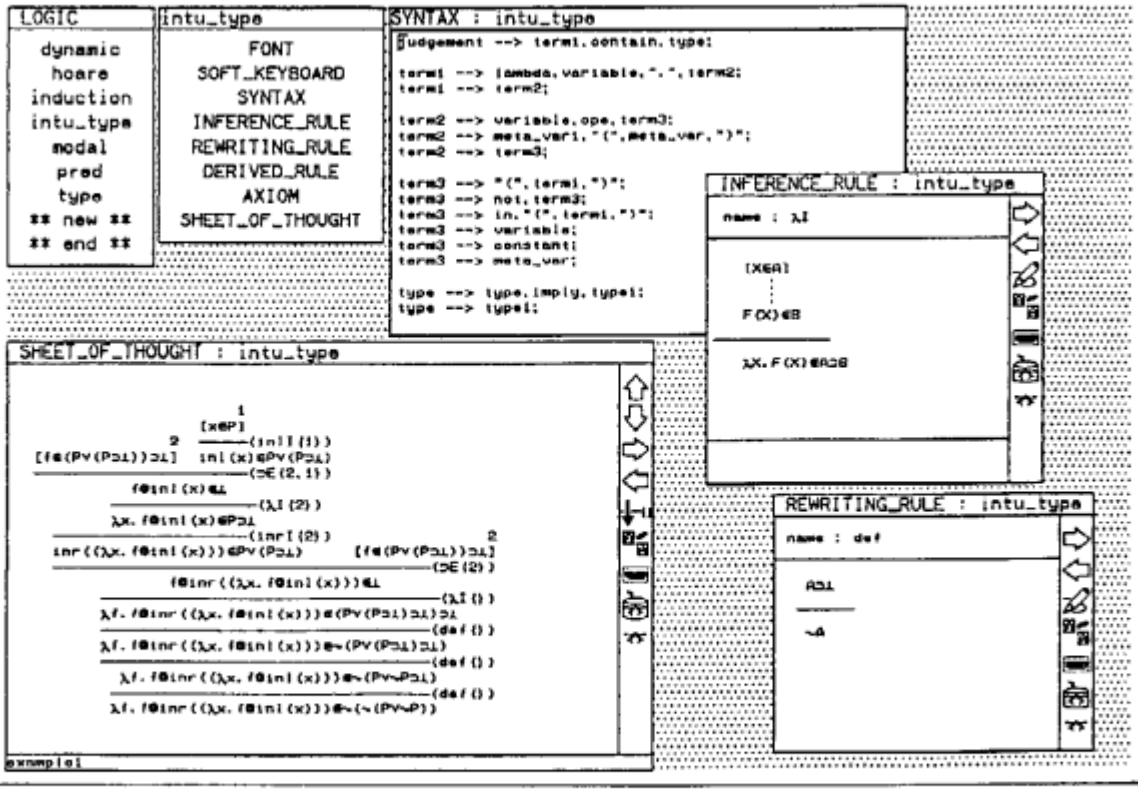
$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[f \in (P \vee (P \supset \bot)) \supset \bot]^2 \qquad \dfrac{[x \in P]^1}{\mathrm{inl}(x) \in P \vee (P \supset \bot)}\ (\text{inl-I } \{1\})}{f \bullet \mathrm{inl}(x) \in \bot}\ (\supset\text{-E } \{2,1\})}{\lambda x.\ f \bullet \mathrm{inl}(x) \in P \supset \bot}\ (\lambda\text{-I } \{2\})}{\mathrm{inr}(\lambda x.\ f \bullet \mathrm{inl}(x)) \in P \vee (P \supset \bot)}\ (\text{inr-I } \{2\}) \qquad [f \in (P \vee (P \supset \bot)) \supset \bot]^2}{f \bullet \mathrm{inr}(\lambda x.\ f \bullet \mathrm{inl}(x)) \in \bot}\ (\supset\text{-E } \{2\})}{\lambda f.\ f \bullet \mathrm{inr}(\lambda x.\ f \bullet \mathrm{inl}(x)) \in (P \vee (P \supset \bot) \supset \bot) \supset \bot}\ (\lambda\text{-I } \{\})}{\lambda f.\ f \bullet \mathrm{inr}(\lambda x.\ f \bullet \mathrm{inl}(x)) \in \sim\sim(P \vee \sim P)}\ (\text{def } \{\})$$

(The actual screen layout is shown in the next page)

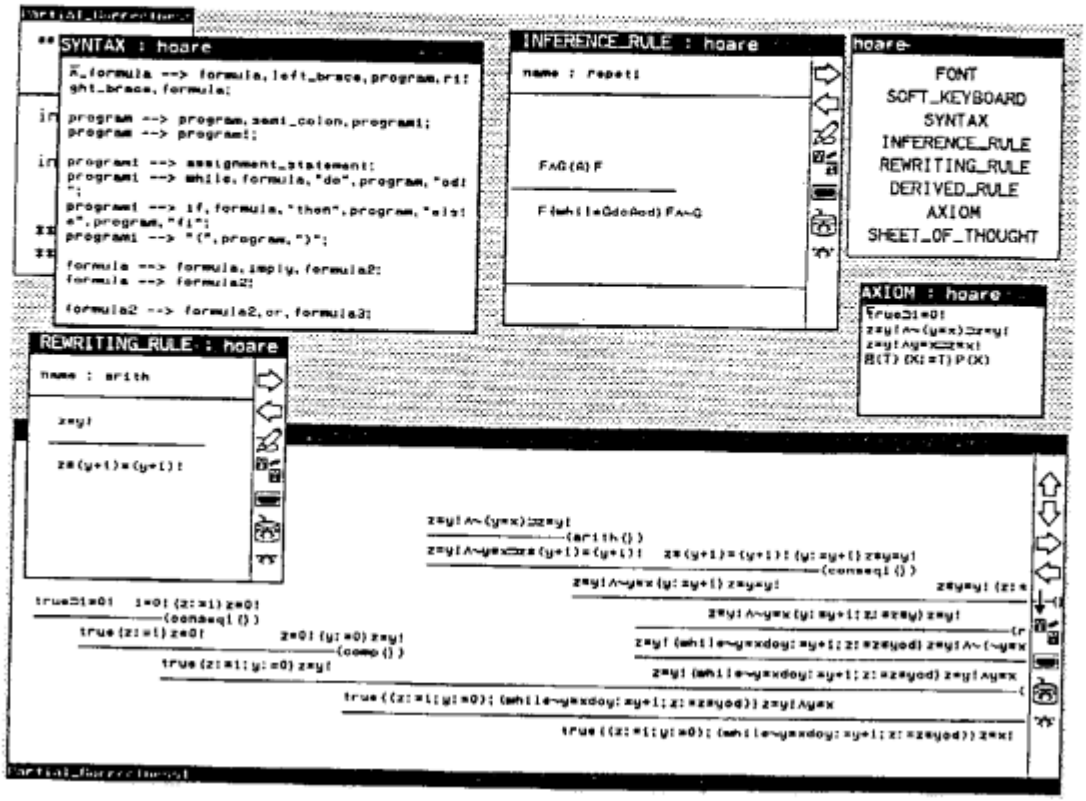
Reasoning-Oriented Human-Computer Interface



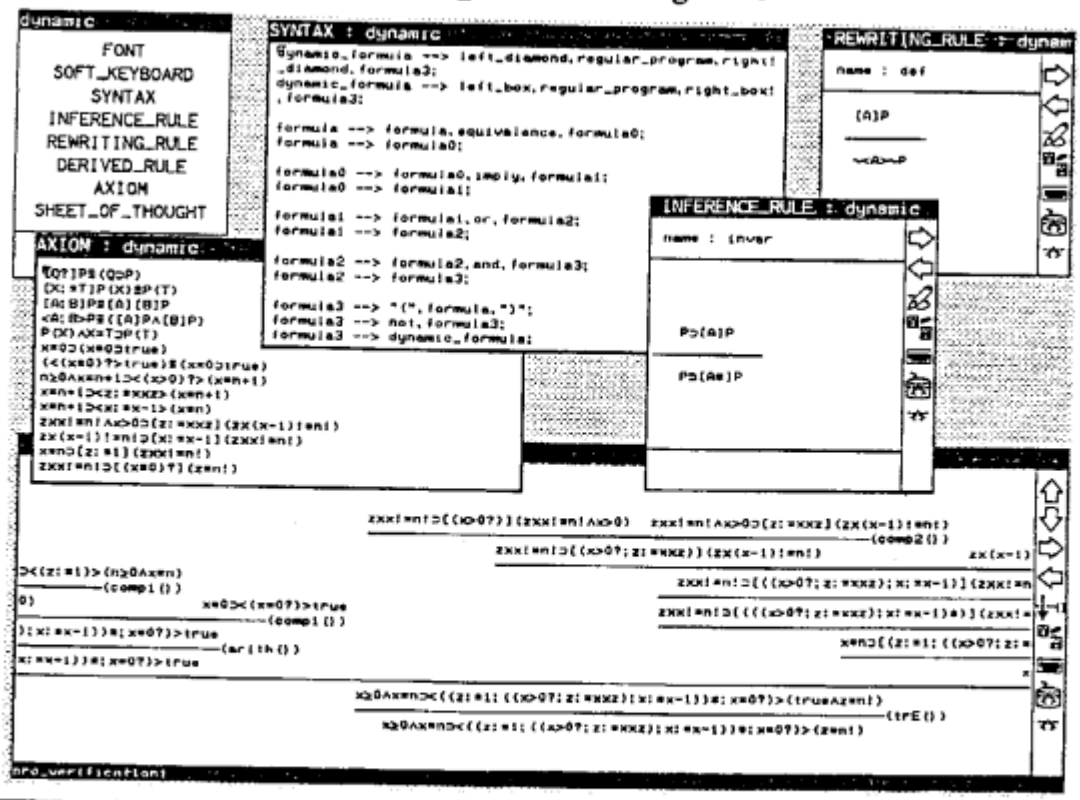
Intuitionistic Type Theory and A Constructive Proof

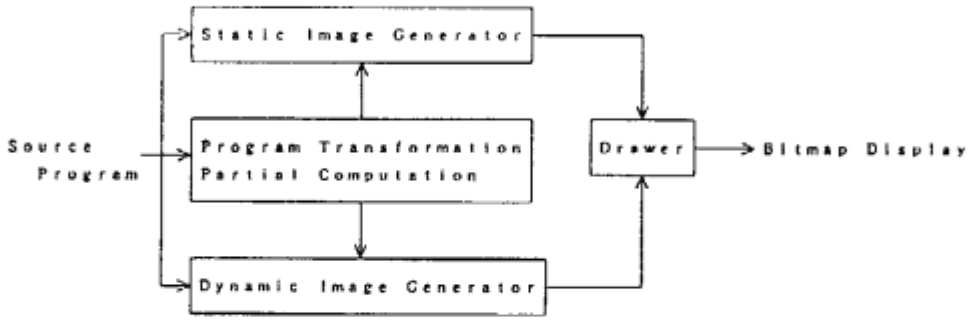


Hoare Logic and Program Verification



Dynamic Logic and Reasoning About Programs



Title	V I S T A : Visualization and Transformation Apprentice for Concurrent Logic Programming
Purpose	The system helps programmers develop concurrent logic programs easily by providing visualization of program structures which may be useful to understand programs.
Outline & Features	◇ VISTA has two visualization modes - Static mode : a structure of the program is visualized by unfolding a given top level goal. - Dynamic mode : an execution of the program is visualized by replacing a parent process with child processes and by redrawing the stream lines between processes. ◇ VISTA can be used to examine concurrent programming techniques - Layered-stream program - Knuth-Bendix completion program ◇ VISTA can be used to compare different versions of programs - Two versions of unification program - Partially evaluated program and its original
System Configuration	

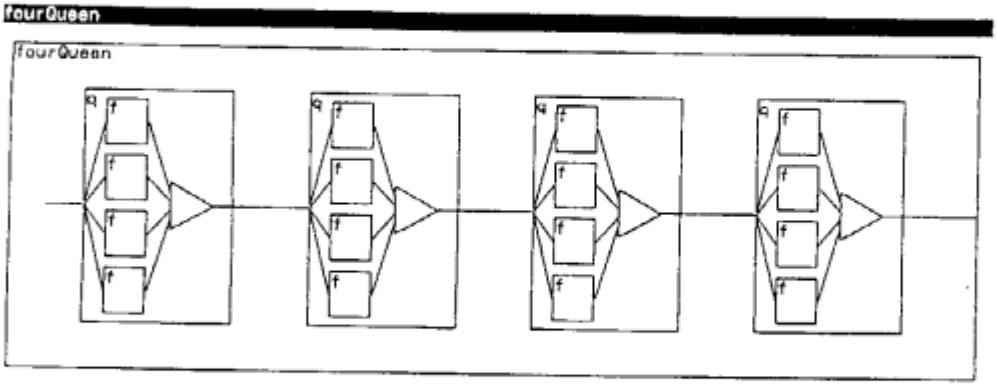
D e t a i l s (1/4)

◇ Layered-stream program

Layered stream is a type of data structure, which is designed for efficient search programming in GHC. Through a layered stream, information is propagated to consumer processes as soon as possible, even if the information is incomplete. Thus it provides very high parallelism. In our demonstration, a 4-Queens program using a Layered Stream is visualized.

Source program:

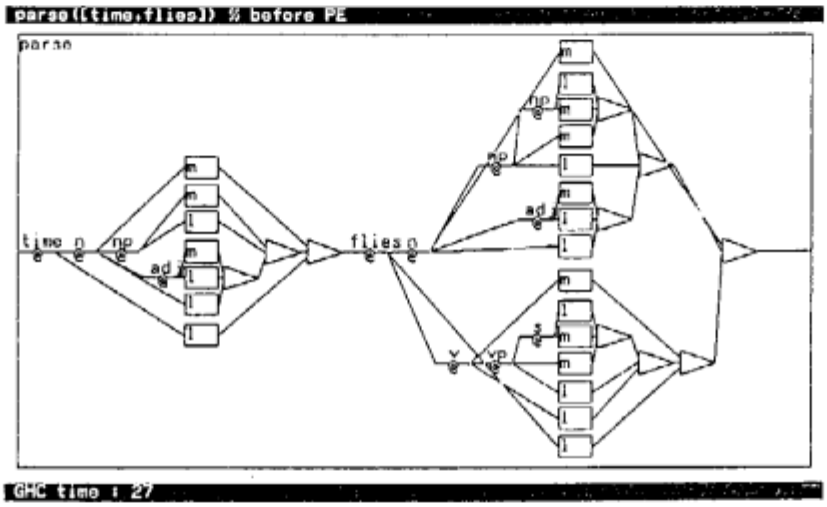
```
fourQueens(Q4) :- true |
        q(begin,Q1), q(Q1,Q2), q(Q2,Q3), q(Q3,Q4).

q(In,Out) :- true |
        filter(In,1,1,Out1), filter(In,2,1,Out2),
        filter(In,3,1,Out3), filter(In,4,1,Out4),
        Out = [1*Out1,2*Out2,3*Out3,4*Out4].

filter(begin,_,_,Out) :- true | Out = begin.
filter([],_,_,Out) :- true    | Out = [].
filter([I*_|Ins],I,D,Out) :- true       | filter(Ins,I,D,Out).
filter([J*_|Ins],I,D,Out) :- D =:= I-J | filter(Ins,I,D,Out).
filter([J*_|Ins],I,D,Out) :- D =:= J-I | filter(Ins,I,D,Out).
filter([J*In1|Ins],I,D,Out) :- J \= I, D =\= I-J, D =\= J-I |
        D1 := D+1, filter(In1,I,D1,Out1),
        filter(Ins,I,D,Outs),
        Out = [J*Out1|Outs].
```

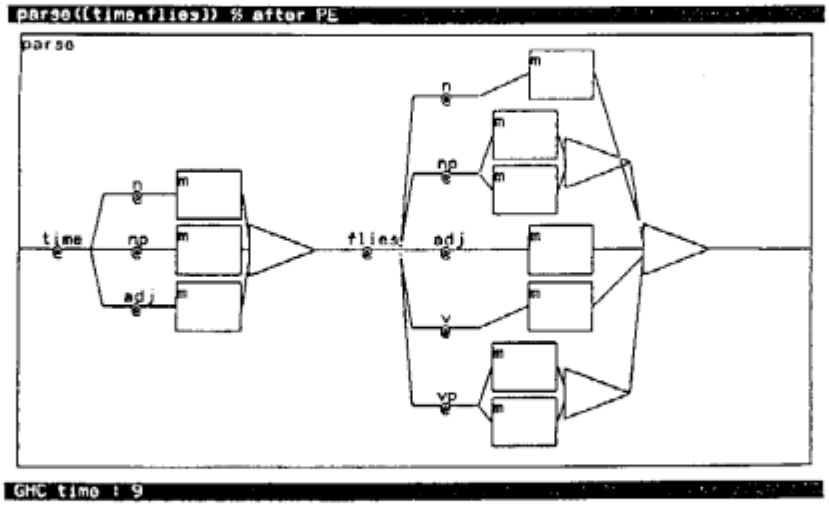


D e t a i l s (2/4)

◇ Parsing program

A parsing program called Meta-PAX is presented. Meta-PAX, given a set of grammar rules and a sentence, checks whether the sentence is acceptable. A sentence varies each time Meta-PAX is used whereas a set of grammar rules is fixed for some time while Meta-PAX is used, for parsing many sentences. Using partial evaluation, a version of the Meta-PAX program specialized for the given set of grammar rules is obtained which is more efficient than the original program.

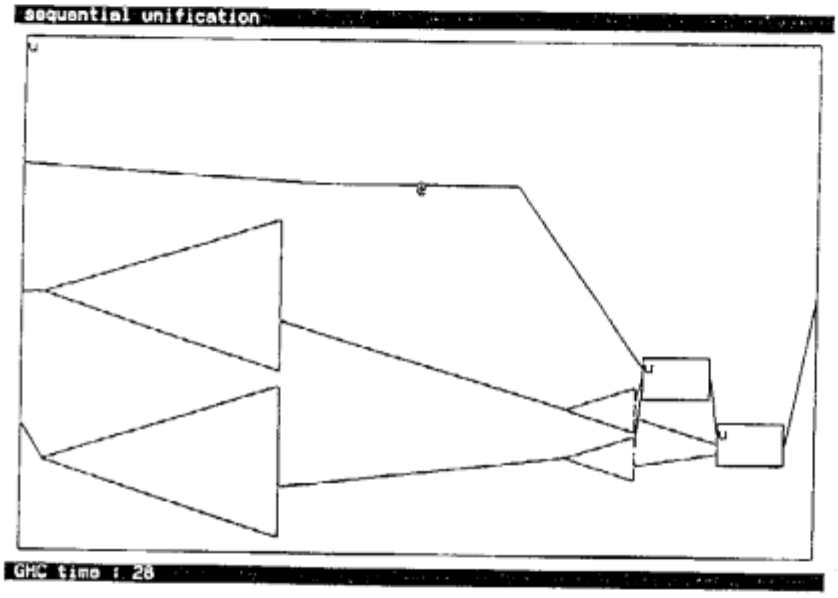


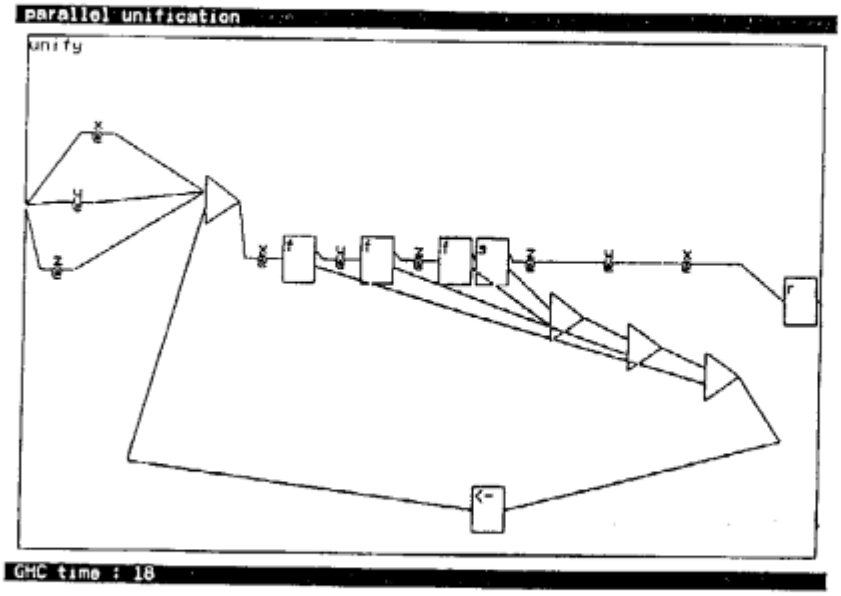


D e t a i l s (3/4)

◇ Unification program

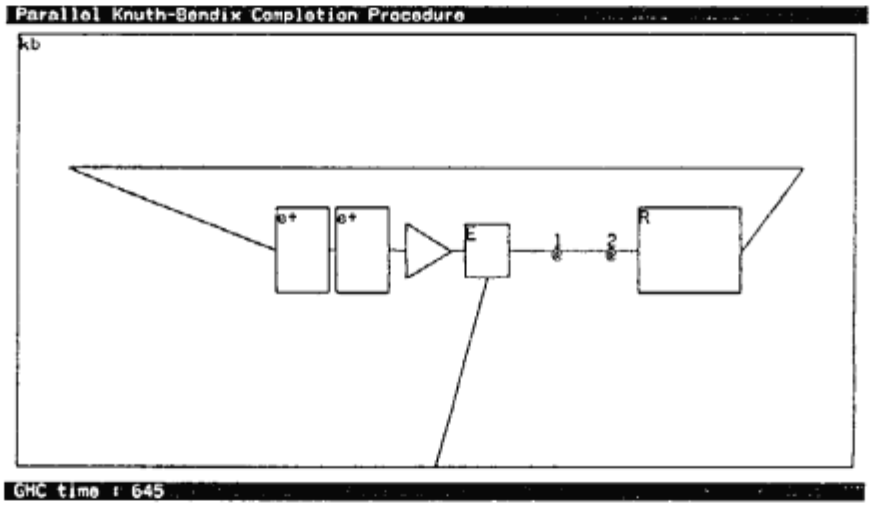
Two programs for unifying two terms are presented. The difference between the programs can be easily seen by visualizing them. The first one tries to unify two compound terms one branch at a time, updating the intermediate result of the unifying substitution. The second one tries to generate a unifying substitution at each branch of the given compound terms in parallel and tries to check consistency between separately generated substitutions.

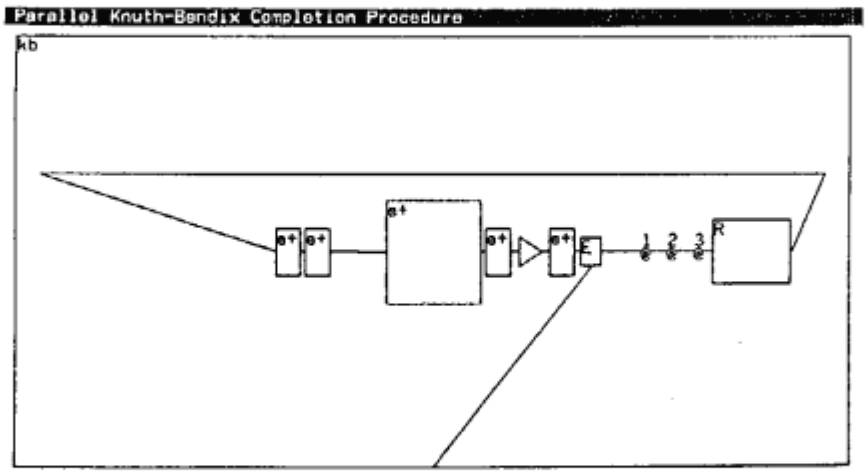


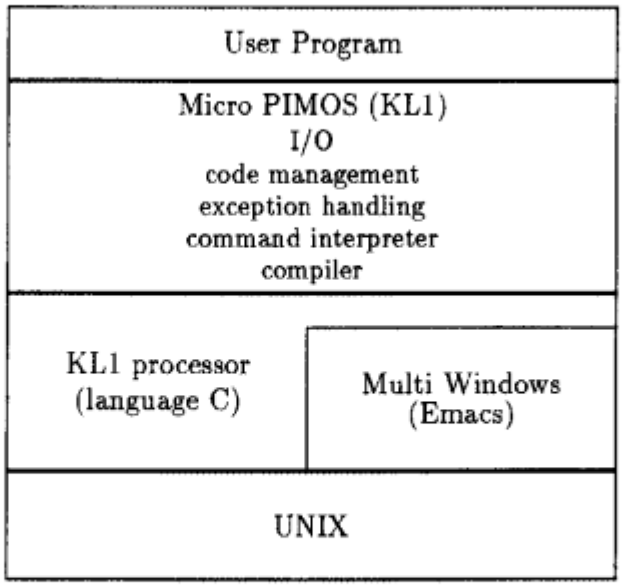


◇ Knuth-Bendix completion program

The Knuth-Bendix completion program is a kind of a compiler that, given a set of equations, produces a set of term rewriting rules which has a nice property called canonical. There are two major processes running in parallel in the program: one for converting an equation to a rewrite rule, the other for generating a new equation (called a critical pair) from a pair of rewrite rules. Critical pair generation can be done in parallel for every pair of rewriting rules, whereas conversion of an equation has to be serialized so that the simpler rewriting rule is generated first. The serialization of the conversion process is realized by using a unique programming technique that makes the most of stream parallelism.





Title	**PIMOS Development Support System: PDSS**
Purpose	PDSS is a KL1 system developed for supporting the development of PIMOS. It runs on Unix machines.
Outline & Features	**Compatibility:** PDSS is highly compatible with the KL1 system on Multi-PSI. Most of KL1 programs tested on PDSS can be installed directly onto Multi-PSI. Main differences between PDSS and Multi-PSI are as follows: • Some function, such as atom management, is processed by the PDSS compiler, while it is done by the PIMOS software in Multi-PSI. • Code management is also done by the PDSS compiler, while it is done by the KL1 system in Multi-PSI. • The I/O protocol of PDSS is different from that of PIMOS. • The processor-allocation specification is not available, since PDSS ia a single processor system. **Portability:** PDSS runs on Unix machines with GNU emacs or Nemacs. (PDSS 2.51 is runnable on Unix 4.2 bsd with GNU emacs 18.53.11 or Nemacs 3.0). Unix users can use PDSS easily on their own Unix machine. **Friendly Interface:** PDSS provides a friendly user-interface by making much use of emacs library function. The user can invoke PDSS easily from an emacs session. **Flexibility as a Test-bed:** The primary aim of PDSS development is to make sure of the design of PIMOS. PDSS is designed to be able to modify its own design as PIMOS development proceeds.
System Configuration	

Details

KL1 Language Specification: A KL1 dialect runs on PDSS, which is almost compatible with the KL1 dialect on Multi-PSI and is distinguished from GHC at the following points:

- *Sequentiality execution of guard:* Unification of head parameters and execution of guard goals are executed sequentially from left to right.
- *Guard restrictions:* Only built-in predicates can be specified in the guard.
- *Equality of variables:* Equality of unbound variables is not checked in the guard.
- *Modularity:* Clustering clauses in modules allows modular compilation and debugging.
- *Sho-en:* The Sho-en feature is introduced as a functional unit to control the execution priority and resource allocation.
- *Exception handling:* Exception handling for KL1 programs is described in KL1 itself, using the Sho-en feature and second-order predicates.
- *Failure handling:* All failures are regarded as exceptions within KL1.

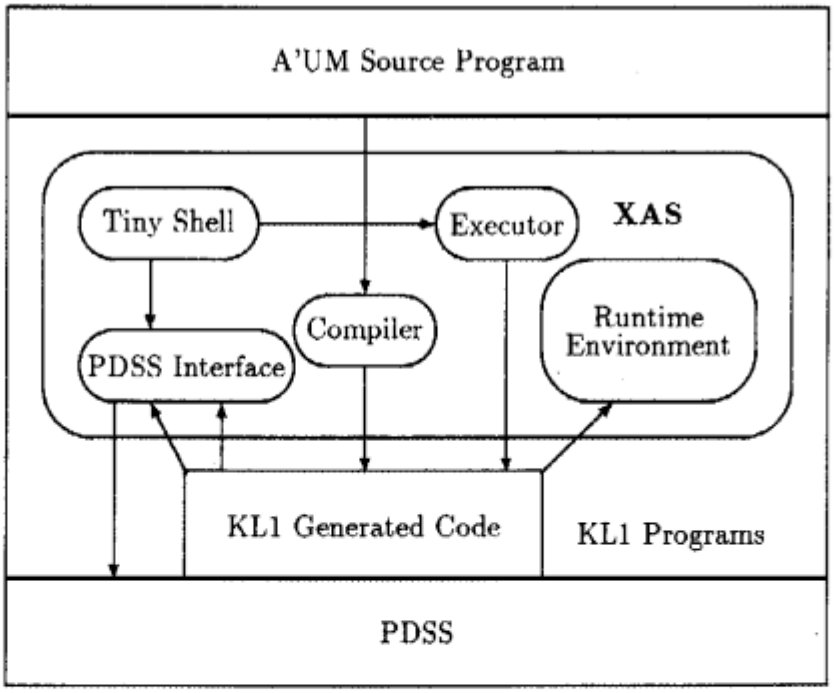
Micro PIMOS: Micro PIMOS is a very simple system which provides various services for KL1 users on PDSS. It is basically designed for single user, single task operations. Micro PIMOS supports the following services:

- Command interpreter
- I/O functions (windows, files, etc.)
- Code management
- Display of exception information

Tracer: Tracing is done for goal invocations. The user can trace goals, set up spy points to leap and monitor variables. When a goal is in either of the following states, called *trace points*, it is traced:

- Goal invocation
- Suspension due to an uninstantiated argument
- Resumption by instantiation of the variable
- Goal failure
- Swap out (caused by interruption or scheduling of a higher priority goal)

Deadlock Detection: Two kinds of deadlock detection mechanisms are supported. One is activated during the global GC, and the other works during execution.

Title	**Experimental A'UM System: XAS**
Purpose	XAS is an experimental A'UM system developed for executing A'UM programs on top of PDSS (PIMOS Development Support System).
Outline & Features	**Stream-based Object-oriented:** A'UM is a concurrent object-oriented programming language whose computation model is defined by stream operations. A'UM aims at high parallelism and high expressivity and makes it easy to write large scale parallel programs. **Compiled to KL1:** In XAS, A'UM programs are compiled into KL1 programs, so they can be executed on PDSS. **Described in KL1:** The entire system of XAS is written in KL1 except for a part written in A'UM itself, so it can be invoked from PDSS. **Portability:** XAS is a system built on top of PDSS which runs on Unix machines. (XAS 1.0 is runnable on PDSS 2.51.) Unix users can use XAS easily as well as PDSS on their own Unix machines. **Flexibility as a Test-bed:** The primary aim of XAS development is to make sure of the design and implementation of an A'UM abstract machine. XAS is designed to be able to modify its own design as the design and implementation of the A'UM abstract machine proceeds.
System Configuration	

Details

1. A'UM

A'UM is characterized by its stream-based computation and object-oriented abstraction. The A'UM computation world consists of objects which communicate by message passing. There is no distinction between primitive objects, such as integers, and abstract objects, such as stack objects. The notion of an A'UM object is very similar to that of the Actor. An object encapsulates a set of internal states and provides a set of protocols to the outside; an object is activated by an event, and then according to the event, it takes several actions in parallel. Multiple class inheritance is supported.
The major difference is that A'UM objects communicate with one another via streams, while actors do so by directly specifying their mail addresses. The ordering (or sequentiality) and non-ordering (or parallelism) of events are all explained in stream operations, that are sending a message, closing a message, merging streams, appending streams, receiving a message and detecting a stream being closed.

2. XAS

XAS consists of the following facilities:

Tiny Shell: XAS has its own tiny shell in which the following functions are available:

- to change the current directory,
- to refer to a directory,
- to compile, load and execute A'UM programs, and
- to compile and load KL1 programs.

Compiler: The A'UM compiler generates from an A'UM program a KL1 (KL1-C) program which is translated to a KL1 abstract code (KL1-B) by the KL1 compiler embedded in PDSS.

Executor: The program executor is an interface between PDSS and XAS. It creates the first A'UM object to initiate an A'UM program.

Runtime Environment: XAS provides a set of primitive and built-in classes as a runtime environment. Those included are integer, atom, boolean, string, list, vector as well as of I/O classes, such as windows and files. The A'UM user can access Emacs windows and Unix files, both as line streams.