A Parallel Inference Machine

Hidehiko Tanaka, University of Tokyo

Logic programming should be adaptable to parallel processing. The architectural design of the parallel inference machine called PIE employs it.

Striking progress in computer technology has given us single-chip computers whose processing power far exceeds that of the first-generation computers. We also have various high-level languages, operating systems, and database systems. As a result, we can now write programs for almost any kind of application, provided that we can explicitly describe their algorithms. This means that computers can replace people in many areas because of their high-speed processing and large memory capability. However, there remain many application fields with hard-to-solve problems. One such is the knowledge-information processing field, where fifth-generation computer systems are expected to play an important role.

A machine to cope with knowledge-information processing should support extensive storage of data and high-speed inference using the data. Up to now, inference processing has involved implementing functional languages and logic-programming languages, such as Lisp and Prolog, on conventional sequential computers. However, the need for processing power of new applications in knowledge-information processing may exceed the capabilities of conventional computers.

The architecture of the parallel inference machine makes it a possible candidate for coping with such processing requirements. Computer architectures proposed for parallel inference machines[1] include the data-flow machine[2,3] and the high-level language machine[4,5,6] for parallel inference. The machine architecture I discuss in this article is for a high-level language machine for parallel logic programming.

Inference processing

Prolog is a typical logic programming language based on first-order predicate logic. Each statement, called a *clause*, takes the form

PO if P1 and P2 and --- Pn.

where P*i* is a predicate with some arguments and $n \geq 0$. (For ease of understanding, I use a simple nonstandard notation.) For example, a program that represents the family relations of parent and grandparent is expressed as

grandparent(X,Y) if parent(X,Z) AND parent(Z,Y). (1)
parent(john, jack). (2)
parent(john, mary). (3)
parent(ann, mary). (4)
parent(mary, paul). (5)
parent(mary, peter). (6)
parent(bill, peter). (7)

where words beginning with capital letters represent variables and lowercase words atoms.

When the question "Who is the grandparent of peter?" (grandparent (X,peter)?) arises, inference operations execute as follows: Using Rule (1), the original question is converted to a new question of the form

parent(X,Z) AND parent (Z, peter)? (8)

where X and Z are variables to be determined.

0018-9162/86/0500-0048$01.00 © 1986 IEEE

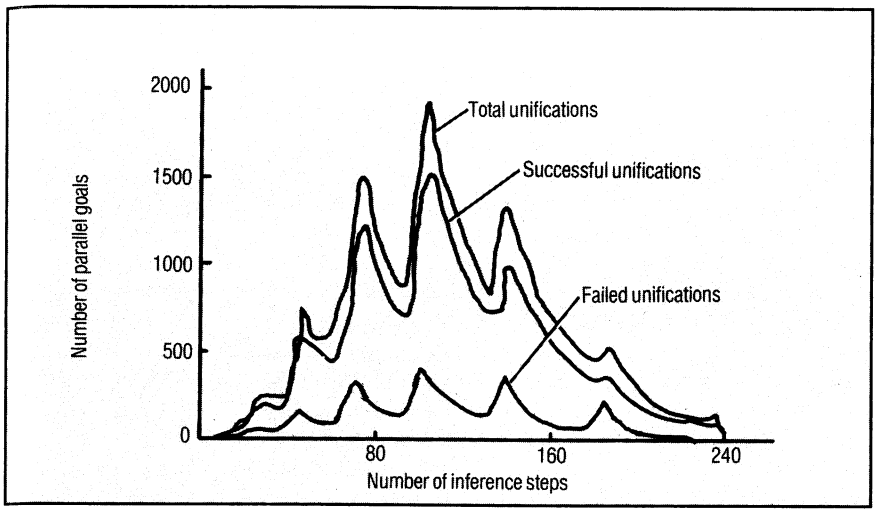

Figure 1. Number of parallel goals (eight queens).

To solve this new question, we have two alternatives: to solve the first part (parent (X,Z)) first, or to solve the second part (parent(Z, peter)) first. Because this decision affects the efficiency of search operations, we need some algorithms to select the best way to proceed. In this example, it is more efficient to evaluate the second part before the first. Accordingly, conditions that fulfill the second part of Rule (8) are searched for in the program. Two values are found for variable Z from Rules (6) and (7)—Z = mary and Z = bill. For each value, the first condition of Rule (8) is tested next and two solutions—(X = john, Z = mary) and (X = ann, Z = mary)—are found.

When we call the question at each step a *goal*, the operation steps of Prolog can be considered goal-reduction steps. In conventional sequential Prolog, the search and test operations (called *unifications*) are executed one by one, but parallel search and test operations can be implemented through parallel machine architecture to obtain a high-speed machine.

Parallelism. A few sources of parallelism can be distinguished for parallel execution of Prolog as follows:

(1) AND parallelism;
(2) OR parallelism; and
(3) argument parallelism.

When the goal is expressed as "Q1 AND Q2—-Qm?", AND parallelism can be used to search for conditions for all literals ($Qi : i = 1,2,—,m$) in parallel. Generally speaking, because arguments in a goal may be related to each other, a consistency check may be needed following the parallel search of AND conditions to ensure that all references to the same variable have the same value. OR parallelism can be used to search and test all conditions for each literal Qi in parallel. The conditions found by OR parallelism are independent of each other. Therefore, the operations can be executed independently. *Argument parallelism* refers to the process of unifying several arguments of a literal in parallel. As arguments may be related to each other, a consistency check is needed in general.

Figure 1 shows an example of measurement of OR parallelism for an eight-queens chess problem. The program finds all placement patterns for eight queens on an eight-by-eight chessboard. The horizontal axis is the number of inference steps and the vertical axis, the number of parallel executions at the step. Due to the multiplication of OR parallelism, the maximum number of parallel goals amounts to about two thousand.

Other operations. For practical programming language considerations, the Prolog functions stated above are not enough. Candidates for supplementary functions include the

(1) NOT operation,
(2) SET operation, and
(3) GUARD operation.

Although pure Prolog statements assert only that a clause is true, in some cases of rules and facts it can be more efficient to assert that the clause is false. NOT expresses such rules and facts. The SET operation gathers values that fulfil some condition in a set. The GUARD operation selects a goal nondeterministically from among several OR goals. Goals not selected are discarded.

Architectural points

When designing a parallel machine architecture that can exploit large-scale parallelism, we should consider several architectural points.

Parallelism to be supported. As stated before, at least three kinds of parallelism can speed up processing. Although we would like to use all three kinds, AND and argument parallelism are difficult to implement. However, we can achieve AND parallelism through pipeline processing of two literals connected by the AND relation, as shown in Figure 2. In the figure, the goal is "Q1 AND Q2 ?" When the unify processing of literal Q1 finishes, the results are fed to literal Q2. When the number of results of Q1 is greater than 1, generation of the second result of Q1 can occur in parallel with the unification of Q2 using the first result. Although conventional AND parallel operation needs a consistency check afterward, this pipeline operation does not and thus is efficient to implement.

Goal-search algorithm. Because inference processing corresponds to a search operation on a tree such as the one in Fig-

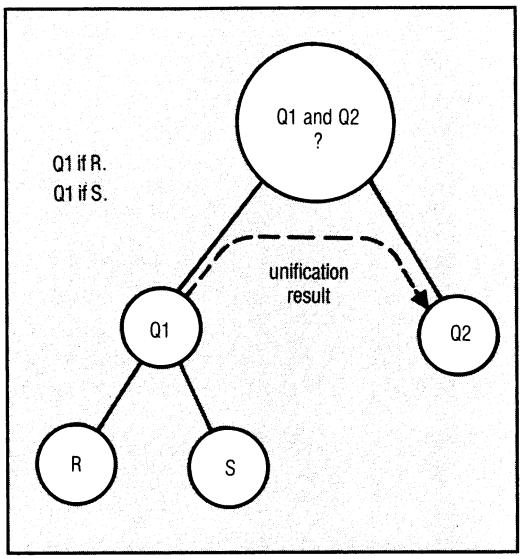

Figure 2. Pipeline processing of an AND relation.

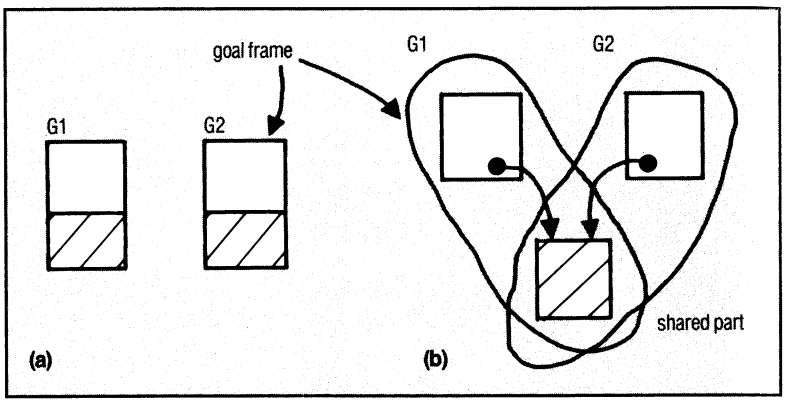

Figure 3. Typical representation scheme of goal frames showing (a) perfect-copy scheme and (b) data-sharing scheme.

ure 2, the search strategy has an important effect on processing efficiency. Four issues are important regarding this algorithm. One issue is the tree-traverse strategy, such as depth-first and breadth-first. In sequential processing, a tree traverse is done node by node, so that failure at one node causes backtracking to an alternative path. In parallel processing, many kinds of search strategies exist, mixing depth-first and breadth-first strategies. One such we call *conclusion precedence.* In this strategy, higher priority is given to a goal frame theoretically nearer to the conclusion (failure or success), using as the parameter the depth number of the node where the literal is introduced to the goal frame. Although in our tentative simulations this strategy shows[4] comparatively good performance both in finding the first solution and in getting all solutions for the simulation programs, we need to study this aspect more thoroughly.

The second issue, the literal selection strategy, should be solved first in each goal frame. For this selection, we can use such parameters as the number of candidate clauses and the argument binding status of each literal.[4]

The third issue is the utilization of the programmer's choice of search strategy, such as read-only annotation. Read-only annotation restricts the location of variable binding, thus affecting the search sequence.

The fourth issue is the elimination of recomputation that may occur many places in a tree. For this purpose we need a mechanism that memorizes past experience.

Processing model. A parallel-inference machine has a large storage area that holds many goals. I call the data structure of a goal a *goal frame.* Two typical schemes of goal frame data structures are shown in Figure 3. One, the perfect-copy scheme, allocates a separate memory area to each goal frame. The other, the data-sharing scheme, shares common parts of goal frames, identifying them by means of pointers. Although copying overhead is induced, the perfect-copy scheme lends itself well to load-leveling mechanisms. On the other hand, the data-sharing scheme is efficient in memory and time consumption, although somewhat complicated to implement. In reality, a combination of both schemes may be best. Some part of a goal should be shared among many goals; the rest of it should be stored independently. The key point is to find the proper combination.

Machine organization. For each unification of a goal, a program composed of many rules and facts is searched to find a candidate clause that can be used to unify a literal of the goal. Because a parallel inference machine has many processors, many simultaneous accesses to program memory arise. To alleviate access contention, the machine should be equipped with many separate memories (called *definition memories*) to store the same program. When program size grows, all programs cannot be stored in the high-speed definition memory. Accordingly, in the future we should incorporate secondary storage to facilitate the cache concept, where only the necessary part of a program is copied to the definition memory.

Connection networks of element modules are also an important issue. To realize a parallel machine of more than 100 processor elements, we should consider clustering processor elements to take advantage of processing locality. By dividing one cycle of operation into several stages, we can take pipeline control into account to achieve sufficiently high speed. One example of pipeline stages is shown in Figure 4.

In general, there are several candidate clauses for one literal of a goal. In Figure 4, the final three stages of unification for each candidate clause are independent of each other and can be processed in parallel.

Machine control. A characteristic of parallel-inference processing is dynamic load generation and consumption. For any kind of parallel machine with this load characteristic, a dynamic load-control scheme is essential for utilizing machine resources effectively. Figure 5 shows a dis-

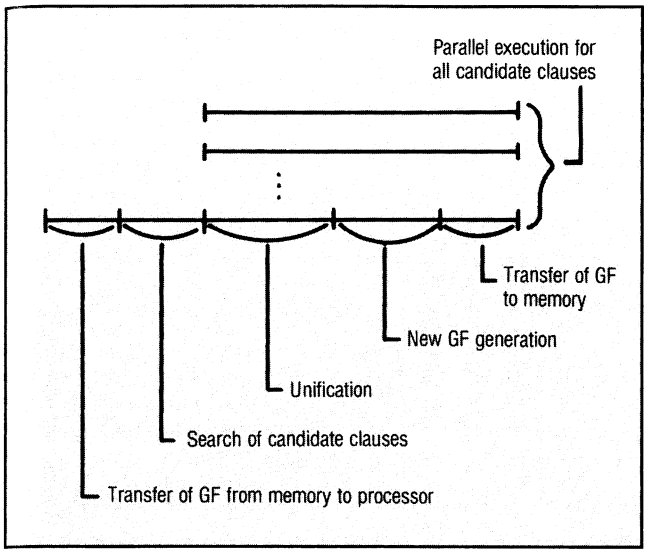

Figure 4. Pipeline stages of inference operation.

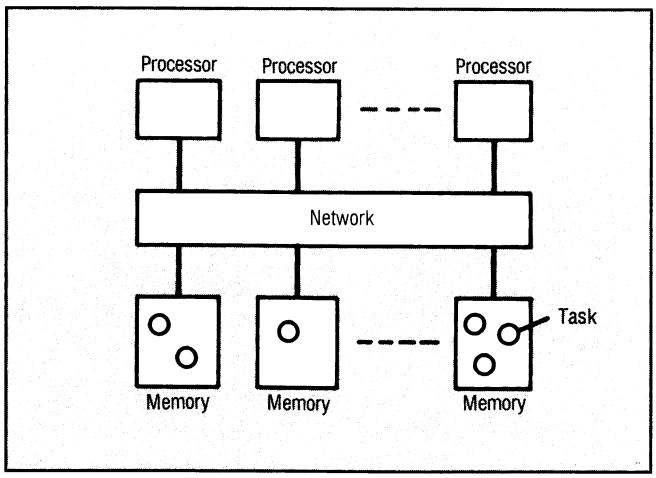

Figure 5. Distributed model of load control.

tributed model of load control. Tasks are stored in memory modules. Each processor fetches a task from a memory module, processes it, and generates a few tasks as the results. Load control can be done in two ways. One method involves fetching a task from a memory module with many tasks. The other method involves dispatching newly generated tasks to lightly loaded memory modules.

Only one of the two methods should be needed for most situations. One memory module can be local to a processor. Accordingly, the following organization offers a good solution for load control:

(1) Tasks are always fetched from the local memory module, and

(2) Newly generated tasks are dispatched depending on the load of each memory module.

Because this scheme assumes monitoring of load balance among modules, we need to incorporate a measurement mechanism. However, the information from a search tree can be used for load control instead of dynamic monitoring.

The other feature of machine control relates to the implementation of supplementary functions such as NOT, SET, and GUARD operations. Although OR-related goals are independent of each other, goals become interrelated when these functions are introduced. For example, Figure 6 shows the implementation of NOT operations. When "NOT Q ?" is executed as a goal and there exist two definition clauses for Q, the unification generates two AND nodes (NOT R AND NOT S). This contrasts with the generation of two OR nodes when the goal is simply "Q ?"

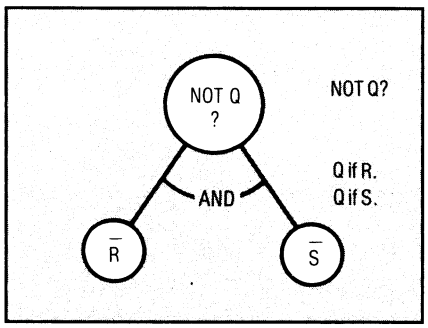

Figure 6. The implementation of NOT operation.

The same situation occurs when the other supplementary functions are used. We need some global control mechanism to implement these functions and to interrelate the processing of goals.

Proposed architecture–PIE

Based upon the consideration stated above, we[4,7] have proposed a parallel inference machine called PIE, for Parallel Inference Engine. Figure 7 shows the global organization of PIE. Because the PIE architecture continues to evolve, this figure shows the second version of the architecture.

This machine has a two-level organization. The Level-1 system is a cluster of inference units, or IUs. The second level is a subset of the Level-1 systems. PIE assumes 64 Level-1 systems. It also assumes the number of inference units in a Level-1 system to be 16, so the total number of inference units is 1024. The system manager, or SMR, of PIE manages global activity through intercluster control of Level-1 systems.

Figure 8 shows the detailed organization of a Level-1 system, including seven modules and three networks:

- memory module, or MM;
- unify processor, or UP;
- definition memory, or DM;
- lazy-fetch buffer, or LFB;
- activity controller, or AC;

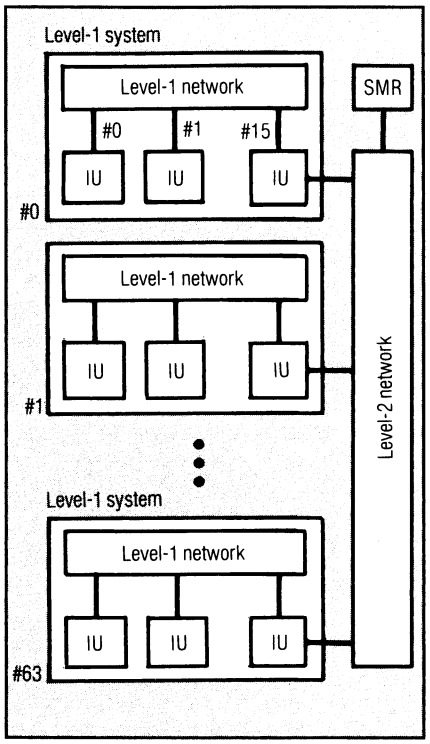

Figure 7. Global organization of PIE.

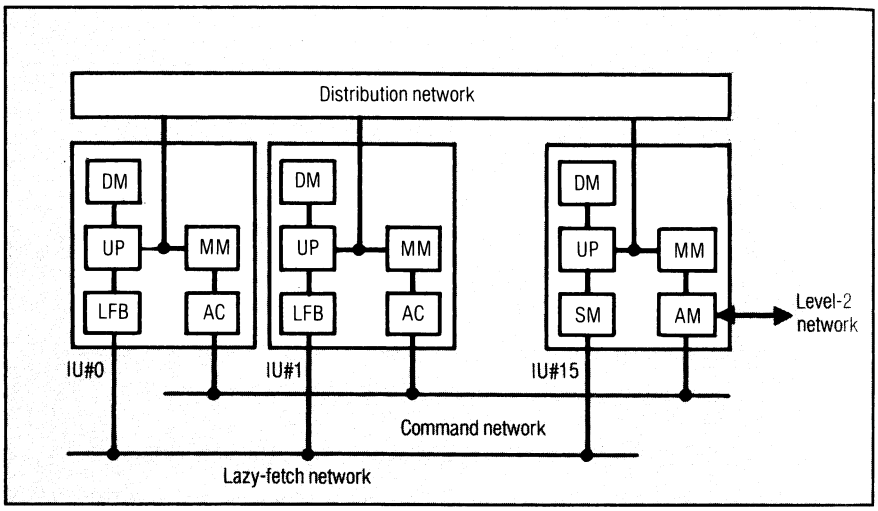

Figure 8. The organization of a Level-1 system.

- structure memory, or SM;
- activity manager, or AM;
- distribution network, or DN;
- lazy-fetch network, or LFN; and
- command network, or CN.

IU #15 is special in that it acts as an interface module to a Level-2 network. It also includes an SM that stores structure data. This SM is shared by all IUs in the Level-1 system. The AM is a centralized controller of the Level-1 system.

Memory module and unify processor. The MM stores goal frames. All MMs in a Level-1 system make up a goal pool. The UP fetches a goal frame from its local MM, unifies the frame, and stores the resulting new goal frames in the goal pool. For this store operation, the memory module in which the new goal frame is stored is determined by a load-dispatching algorithm. That is, if the number of local goal frames stored is greater than a threshold, the goal frame is output to DN and transmitted to an external MM. The destination MM is selected randomly so that the number of goal frames is less than a threshold.

This strategy is called *empty self*. Alternative strategies are *first self* and *random*. In first-self strategy, the most recently generated goal frame is always stored in the local MM and the rest are transmitted randomly to other, external MMs. Random strategy distributes all new goal frames randomly to all MMs.

Figure 9 shows a simulation result of the load-leveling strategies. The vertical axis designates the number of working UPs; the number of physical UPs is assumed to be 64. As can be seen from this figure, empty-self strategy is best in the sense that the load is dispatched faster to idle UPs and that processing ends first.

Figure 10 shows a simulation result of the relative execution speed of PIE. The horizontal axis designates the number of UPs. The vertical axis designates the relative speed compared to a single UP machine. The application programs are the eight-queens and six-queens chess programs, and a cryptoarithmetic program called LL2P. In this example, the parallel speed gain for the eight-queens program is about 170 when the number of UPs is 256.

In these sample programs, the processing time of one inference step is about 300 micro steps for the eight-queens program and 900 micro steps for the cryptoarithmetic program, assuming microprogram-controlled hardware. Accordingly, the processing time is about 60 and 180μs, respectively, with a clock period of 200ns, so that the granularity of this machine is rather coarse.

When a new goal frame is generated, the frame size grows linearly with respect to the unification steps if the conventional unification algorithm is used. However, Moto-Oka et al.[4] have shown by simulation that the size can be kept roughly constant if a size-reduction process is applied to new goal frames. This process eliminates unneeded intermediate variables. Although a rather costly operation, it improves the effective parallelism of the copy scheme.

Definition memory and structure memory. DM stores all definition clauses of the program. All DMs have the same copy of the program.

DM searches candidate definition clauses that may match the head predicate presented by UP. Clauses found are fed to the UP one by one. Following this, the UP performs unification one by one.

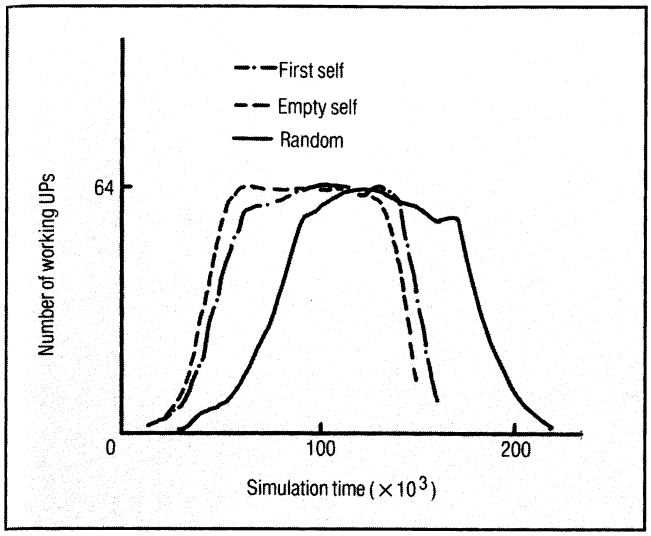

Figure 9. Comparison of load-leveling strategies.

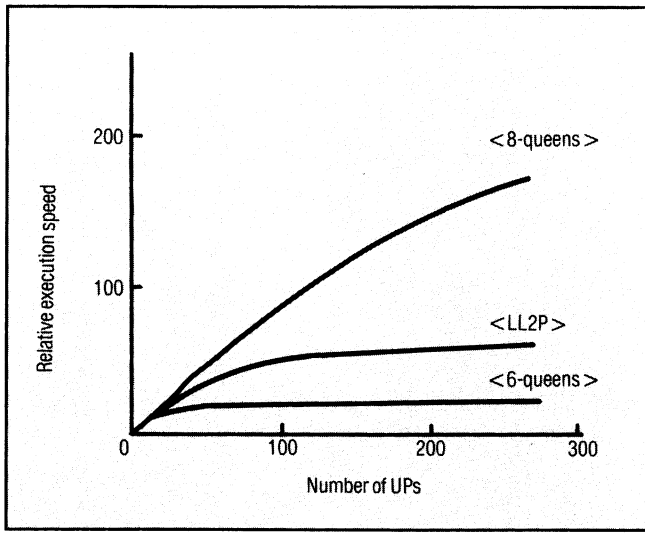

Figure 10. Relative execution speed of PIE.

PIE adopts a data-sharing scheme for goal frames. The shared part is stored in SM and the nonshared part in MM. When a goal-fetch operation is done by a UP, the nonshared part is transferred from MM to the UP, whereas the shared part is not transferred. Access to the shared part occurs on a demand basis. That is, if access becomes necessary within unification, it is done. This access is called *lazy fetch*. The data accessed by one lazy-fetch operation is not the whole volume of structure data, but a few words of fixed length. LFB behaves as a cache to decrease SM access.

In PIE, not all of the shared data is stored in SM. For simple implementation, data in SM are restricted to immutable data such as structures made of ground instances only. These data are not changed once created. When it becomes clear that the data will not be accessed any more in future, the data are deleted and the memory area is garbage-collected into a free-cell list. Due to the immutability of content, the garbage-collection algorithm becomes very simple.

The number of IUs in a Level-1 system is limited by the traffic of SM access. From preliminary simulation results,[4] it is estimated to be less than 20. The effect of providing SM is to reduce the goal frame size into less than a half, although its precise value depends on each application.

Activity controller and activity manager. As stated before, goal frames should be interrelated so as to support some supplementary functions. For this purpose, all goal frames are managed by a tree structure stored in ACs, as shown in Figure 11. This tree is called an *inference tree*. Each AC maintains the information of nodes for which goal frames are stored in the local MM. Nodes are also connected to remote nodes in the other modules, using pointers. When a new goal frame is generated, a new node area is reserved and tree connections are created. When some goal succeeds or fails, the corresponding node transmits the result to its parent node and deletes itself.

At all times the AC monitors the local load in terms of the number of goal frames and periodically transmits the summing-

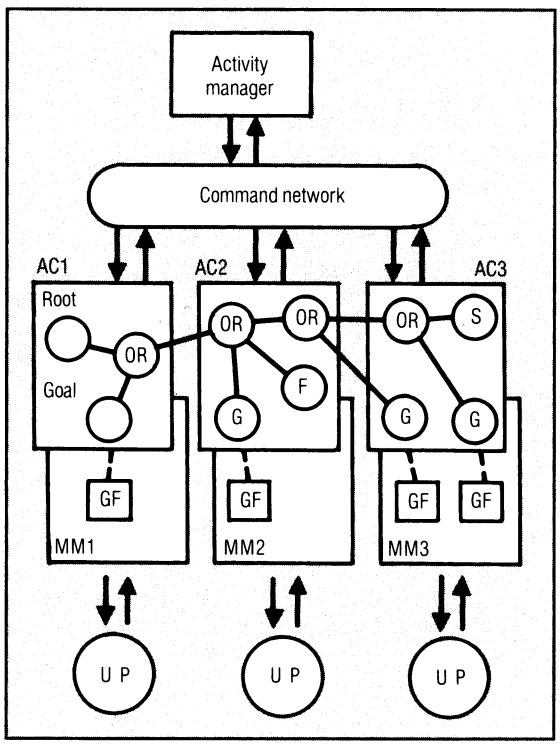

Figure 11. Management of goal frames.

up report to AM. After receiving load information from all ACs, the AM fixes a threshold value for load control and broadcasts this value to all ACs. The AC sets up the local IU using the threshold value for the empty-self algorithm.

AM serves as an interface module to a Level-2 network as well. In the Level-2 network, all AMs in PIE behave for SMR just like ACs in a Level-1 system. The SMR controls the load balance among Level-1 systems by fixing another theshold value for all AMs. When the total load of a Level-1 system exceeds the threshold value, the AM dispatches part of the load to another Level-1 system. At this time, the AM concatenates the shared part of a goal frame to the nonshared part and transmits the resulting goal frame in complete form to the other Level-1 system. Accordingly, the structure data stored in SM is not shared beyond a Level-1 system.

Concluding points

One of the key points in achieving a highly parallel machine is the selection of granularity of processing tasks. This granularity should be determined based on the balance between processing time of each task and other overhead time, such as data transfer delay and control delay. In the case of inference processing, this granularity can be set comparatively large, as a unification operation usually takes hundreds of conventional machine code cycles.

The second important point is the balance between copying and sharing of data. Although copying makes machine organization simple, sharing makes operation and memory utilization efficient.

The third important point is control overhead. For a highly parallel machine, the central facility should be invoked as little as possible to avoid becoming a bottleneck in the system. Global control tends to become such a bottleneck. We need a smart mechanism to select between distributed and centralized control and thus give us satisfactory results.

I have presented here a parallel machine architecture, PIE, which I expect to be the base for highly parallel inference machines. On a small sample of problems, experimenters have obtained speedups ranging from 20 to 170 times the single-processor system speed for a system with 256 processors. The experiments show that the architecture has the potential to attain very high performance from parallel operation, but that speedup can be fairly low as well. Additional research will determine what speedup we can expect for particular classes of problems. We also need to develop techniques for achieving very high speedup on those classes of problems that currently exhibit poor speedup. Further study should focus on more precise evaluation, analysis,[8] and building an experimental system.

Finally, we should incorporate several improvements in the machine organization, such as pipeline control, associative search of clauses, garbage collection hardware, and support of secondary memory. □

References

1. T. Moto-Oka, "Challenge for Knowledge Information Processing Systems," *Proc. Int'l Conf. Fifth Generation Computer Systems,* JIPDEC, Oct. 1981.
2. R. Hasegawa and M. Amamiya, "Parallel Execution of Logic Programs Based on Dataflow Concepts," *2nd Int'l Conf. Fifth Generation Computer Systems,* Nov. 1984, pp. 507-516.
3. T. Ito et al., "Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog," *New Generation Computing*, Vol. 3, No. 1, OHMSHA, Ltd. and Springer Verlag, 1985, pp. 15-41.
4. T. Moto-Oka et al., "The Architecture of a Parallel Inference Engine—PIE," *Proc. 2nd Int'l Conf. Fifth Generation Computer Systems*, Nov. 1984, pp. 479-488.
5. J. S. Conery and D. F. Kiblar, "Parallel Interpretation of Logic Programs," *Proc. Conf. Functional Programming Languages and Computer Architecture,* 1981, pp. 163-170.
6. R. Onai et al., "Architecture of a Reduction-Based Parallel Inference Machine: PIM-R," *New Generation Computing,* Vol. 3, No. 2, OHMSHA, Ltd. and Springer Verlag, 1985.
7. A. Goto, H. Tanaka, and T. Moto-Oka, "Highly Parallel Inference Engine PIE—Goal Rewriting Model and Machine Architecture," *New Generation Computing*, Vol. 2, No. 1, OHMSHA, Ltd. and Springer Verlag, 1984, pp. 37-58.
8. H. Tanaka, "The Fifth Generation Computer System and its Computer Architecture," *6th Int'l Conf. Computing Methods in Applied Sciences and Engineering,* Dec. 1983.

Hidehiko Tanaka is an associate professor in the Department of Electrical Engineering, University of Tokyo. From 1978 to 1979, he was a visiting professor at the City College of New York. He received the Yonezawa Award from IECE of Japan in 1970. He is also an associate editor of *New-Generation Computing*.

His current research interests include parallel computer architectures such as parallel inference machines and knowledge base machines, distributed systems organization, and object-oriented systems. He designed the Network-Oriented OS, the parallel inference engine PIE, and the cooperative distributed Service Base System.

Tanaka received the BE in electronics engineering and the Doctorate of Engineering in electrical engineering from the University of Tokyo in 1965 and 1970, respectively.

Readers may write to Tanaka at the Dept. of Electrical Engineering, The Faculty of Engineering, The University of Tokyo, 7-3-1 Hongo, Bunkyo-Ku, Tokyo 113, Japan.