ICOT Technical Memorandum: TM-0822

TM-0822

Research Issues in Parallel Knowledge Information Processing

by
N. Ichiyoshi

November, 1989

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

Research Issues in Parallel Knowledge Information Processing

Nobuyuki Ichiyoshi

Institute for New Generation Computer Technology

1-4-28 Mita, Minato-ku, Tokyo 108, Japan

Abstract

The Japanese fifth generation computer systems (FGCS) project attempts to build a parallel inference machine (PIM) to provide the computational power needed in large-scale knowledge information processing. This paper describes the outline of the concurrent logic language KL1 and the PIM architecture. It also discusses some of the issues to be addressed in parallel knowledge information processing, including load distribution, communication overhead and speculative computation, and reports on the performance results of experimental programs on the pilot parallel inference machine Multi-PSI.

1 Introduction

The Japanese fifth generation computer systems project (FGCS) is a ten year project that started in 1982 [6]. It attempts to build a massively parallel computer to provide the computational power needed in large-scale knowledge information processing. Since the machine runs a logic programming language in parallel, it is called a parallel inference machine (PIM). Several PIMs which differ in architectural details are under development [3]. They are scheduled to come into existence in 1990–1991.

The kernel language of the FGCS is the concurrent logic language KL1[1]. KL1 is suited to describe concurrent systems in which a lot of processes cooperate to solve a large-scale problem.

We built a pilot parallel inference machine Multi-PSI [9] in late 1988. Its is a loosely-coupled MIMD multiprocessor with maximum of 64 processors with large memory. The purposes of the development of the Multi-PSI were (1) to test various new implementation techniques of concurrent logic languages, and (2) to develop a parallel operating system and rudimentary application programs by the time the PIM becomes operational.

A number of medium-scale parallel programs have been written in KL1, and the performance was measured on the Multi-PSI. Though still limited, the experience has taught us the importance of scheduling and load balance in programs on a loosely-coupled machine. The realization of logical parallelism in a program as physical parallelism on a multiprocessor is called *mapping*.

2 Concurrent Logic Language KL1

The parallel inference machine runs a concurrent logic language KL1 [1], which can be executed in parallel and is suited for describing interacting processes or objects. A KL1 program consists of a collection of *guarded Horn clauses* [12] of the form:

$$H \ :- \ G_1, \ \ldots, \ G_m \mid B_1, \ \ldots, \ B_n. \ (m, n \geq 1)$$

where H, G_i and B_i are atomic formulas. H is called the *head*, G_i the *guard goals*, B_i the *body goals*. The guard part consists of the head and the guard goal, the body part consists of the body goals, and they are separated by the *commitment operator* ($\mid$). The declarative reading of the above clause is simply:

$$\text{If } G_1, \ \ldots, \ G_m \text{ and } B_1, \ \ldots, \ B_n \text{ then } H.$$

A collection of guarded Horn clauses whose heads have the same predicate symbol P and the same arity N, define a procedure P with arity N (denoted P/N).

One example of a KL1 procedure is as follows:

```
filter([],F,Out) :- true | Out = [].
filter([X|Xs],F,Out) :- X mod F =:= 0 | filter(Xs,F,Out).
filter([X|Xs],F,Out) :- X mod F =\= 0 |
    Out = [X|Out1], filter(Xs,F,Out1).
```

Declaratively, `filter` is a ternary predicate whose first argument is a list of integers, the second an integer, the third a list of integers. filter(In, F, Out) is read as "In filtered by F is Out". `[]` denotes the empty list (called *nil*), `[`H`|`T`]` denotes the list whose head (first element) is H and the tail is T.

Operationally, the head corresponds to a procedure call, the guard part represents the test on the input arguments, and each of the body goals represents a procedure invocation. In particular, the equality symbol (`=`) in the guard represents a passive pattern matching, and that in the body represents assignment to the output argument. When the guard testings of one or more clauses have succeeded, one of those clauses is selected (called *commitment*), and its body goals are called. The body goals can be executed in parallel. If an input argument is not ready for testing (because the value has not been computed yet), the guard testing is suspended.

Thus, KL1 provides two basic mechanisms of concurrent programming: concurrent execution and synchronization.

Ideally, a logic programming language system is both sound and complete as a theorem proving system. SLD-resolution on which Prolog is based has such a property [7]. The commitment to a single clause makes the KL1 (or any *committed-choice language*) system a sound but incomplete theorem prover. This is not to be regretted because KL1 is designed not as a theorem prover in a small axiom system, but as a practical production language for describing concurrent systems while retaining some of the nice properties of logic programming languages (such as possibility of declarative reading, referential transparency of logical variables, ease of mechanical program transformation).

A KL1 goal can be interpreted as representing a process state, and a chain of goals recursively calling the same procedure can be interpreted as representing a process which interacts with the environment (input/output arguments).

For example, when `filter/3` is called with the first and second argument concrete values, and the third an unbound variable, it recursively calls itself until the end of the list (first argument) is reached, and incrementally returns the filtered output.

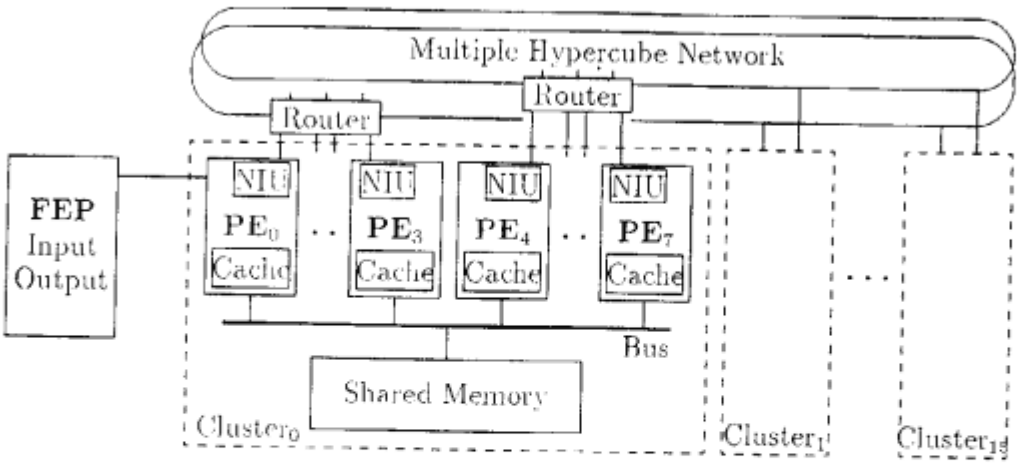
Added to the above basic mechanism are the execution control and mapping facilities. The descendant processes of one goal can be grouped as a unit of control, and can be stopped, re-started, and aborted, etc. The mapping facility includes priority and load-distribution specification. The programmer can annotate the program by attaching *pragmas* to body goals to specify under what priority (specified by *Goal*@`priority`(*Prio*)) and in which processor (specified by *Goal*@`processor`(*Proc*)) the goal should be executed.

The sequentiality specified by the raw semantics of Guarded Horn Clauses is that comes from parent-child relationship of processes and from data dependence. The priority mechanism allows the programmer to specify less strict sequentiality, such as mandatory work before speculative. To this end, the KL1 implementation on the Multi-PSI provides a fine priority system (there are 4,096 physical priority levels).

Currently, the programmer must tell the implementation which goals to execute on which processors.[1] This is because simple-minded automatic load balancing could result in closely communicating processes to be executed on different processors and cause much communication overhead. In contrast, processor allocation in PIM clusters will be done automatically by the implementation.

KL1 program defines large amount of fine-grain parallelism, and it is up to the implementation and the programmer to decide how to map this logical parallelism into physical parallelism realized on the parallel inference machine. The priority

[1]The processor numbers can be dynamically computed.



FEP: Front End Processor
NIU: Network Interface Unit
PE: Processing Element

Figure 1: Architecture of PIM/p

and processor pragmas represent the user-definable part of the mapping. Since the pragmas do not change the meaning of the clause, the program semantics and mapping are separated in KL1.

3 Architecture of Parallel Inference Machine

3.1 PIM Architecture

The rationale of designing the parallel inference machine architecture was performance and scalability. Scalability means the architecture should be parameterized by the number of processors n and n should scale up (in PIM, up to about 1,000). One of the PIM (PIM/p) [3] being developed adopted a two-level architecture: It is a loosely-coupled system at the higher level, and is a shared memory multiprocessor (called a *cluster* at the lower level (Fig. 1). Clusters at the lower level provides powerful processing units with 8 processors per each, and they are connected by a hypercube network to form a large-scale parallel computer.

3.2 Multi-PSI Architecture

The pilot parallel inference machine Multi-PSI [9], which has been operational since late 1988, is a mesh-connected multiprocessor. Each processing element is the same as the CPU of the PSI-II [10] personal sequential inference machine, plus the network

interface.

4 Parallel Knowledge Information Processing

Tightly-coupled medium scale computers, such as the Sequent Symmetry, have come into practical use in a wide variety of applications, and large-scale parallel network-connected computers have shown near-linear speedup in executing numerical computation programs (such as partial differential equation solvers) for large problem sizes. It is time large-scale parallel computers begin to show their effectiveness in large-scale knowledge information processing.

However it is not as easy to obtain near-linear speedup knowledge information processing as in most numerical computation problems for the following reasons:

- The sizes of the problems and subproblems are usually unpredictable, and problem partition and processor allocation are more difficult.
- Data access patterns are irregular (low locality), which cause longer latency, synchronization overhead and require more communication bandwidth.
- Often, a main problem is search in some problem space, and just going parallel causes a lot of redundant computation (in other words, the amount of *speculative computation* may increase in parallel computation).

Care should be taken to reduce those overheads in order to make most of the computational power of the parallel machines. These are newly added dimensions to sequential symbolic processing. Closer look at problem decomposition, communication patterns and speculative computation would be needed, but the efforts would be paid off by a deeper understanding of the problem at hand, and of course, the ability to solve problems of much larger-scale than would be possible on sequential machines.

There has not been any good-for-all theory of parallel algorithms and mapping. We have to attack each type of problems separately. But then, this was the case with designing good sequential algorithms, too.

5 Experimental Parallel Programs

We have written KL1 programs for a few problems, and taken performance measurements [4]. The programs show different run-time characteristics and different degree of speedup. Here are the programs:

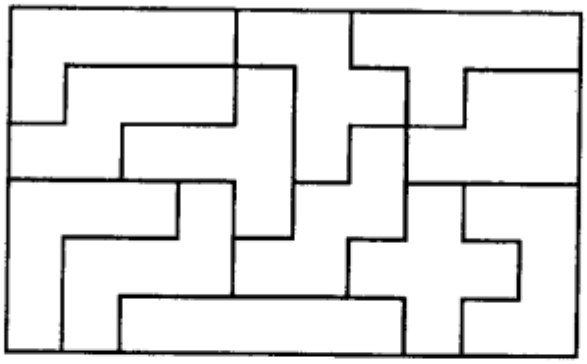


Figure 2: Pentomino

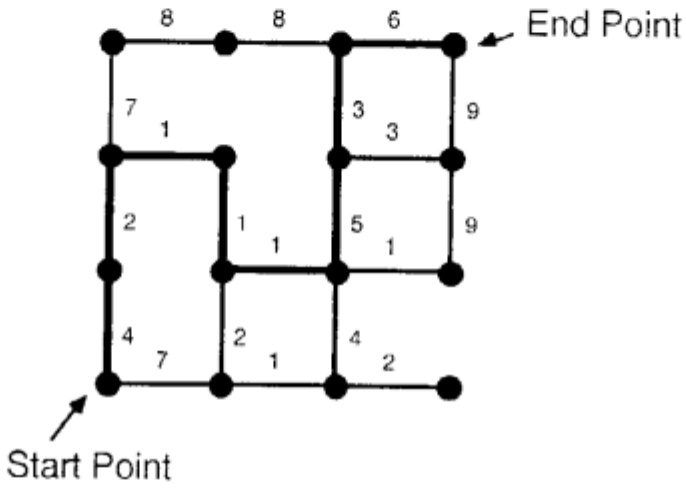


Figure 3: Shortest Path Problem

Packing Piece Puzzle (Pentomino)

A rectangular box and a collection of pieces with various shapes are given (Fig. 2). The goal is to find all possible ways to pack the pieces into the box. The puzzle is also known as the Pentomino puzzle, when the pieces are all made up of 5 squares. The program does a top-down OR-parallel all solution search.

Shortest Path Problem

Given a graph, where each edge has an associated nonnegative cost, and a start node in the graph, the problem is to find a shortest path to every node in the graph from the start node (single-source shortest path problem). (Fig. 3). The program performs a distributed graph algorithm. We used a 200×200 grid graph with randomly generated edge costs.

Natural Language Parser

The problem is to construct all possible parse trees for an English sentence.

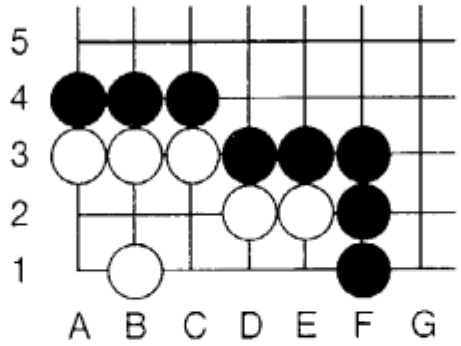


Figure 4: Tsumego

The program called PAX [8] does a bottom-up chart parsing. It is a communication intensive program.

Tsumego Solver

A Tsumego problem is to the game of Go what the checkmate problem is to the game of chess. The black stones surrounding the white stones try to capture the latter by suffocating them, while the white tries to survive (Fig. 4). The problem is to find out the result assuming the black and the white do their best. The result is one of (1) the white doomed, (2) the white surviving, or (3) the Ko situation[2] reached. The program does a parallel alpha-beta search.

An experimental formula of parallel speedup is as follows:

$$S_p = \frac{T_1}{T_p} = \frac{W_1}{W_p(1 + C)/Up} = \frac{W_1}{W_p(1 + C)}Up,$$

where p is the number of processors, S_p the speedup, T_1 the execution time by one processor, T_p the execution time by p processors, W_1 and W_p the amount of computation done by one processor and p processors, respectively, C the overhead incurred by parallel execution (including communication overhead), and U processor utilization rate. This formula says, to obtain good speedup (close to p), (1) processors must be kept busy (U high), (2) redundant computation must be kept small (W_p close to W_1), and (3) parallel overhead must be kept small (C small).

In Pentomino, there is no speculative computation ($W_p = W_1$), and alternative search branches do not communicate with each other ($C \approx 0$). By dynamically balancing the load of the processors ($U \approx 1$), near linear speedup was obtained.

[2]The Ko is a special rule in Go to avoid infinite repetition of two alternating states. In the case of Tsumego, the Ko situation means that the survival is decided by tradeoff with gains or losses in other part of the board.

The shortest path program has a lot of inter-*process* communication, but the communication is between neighboring vertices. A mapping that respects the locality of the original grid graph can keep the amount of inter-*processor* communication low. Reasonable speedup was obtained.

The PAX program was the most difficult to speed up — we could get only threefold speedup so far. The main reason is that the amount of communication is large and the communication is not restricted to between neighboring processes, which makes it difficult to localize it within processors. This also makes dynamic load balancing extremely difficult (the processor utilization is about 20% in 16 processor execution). Finding a good mapping method for this type of problem (kind of dynamic programming) is a challenge to us.

The Tsumego program did parallel alpha-beta search up to the leaf nodes of the game tree. The sequential alpha-beta pruning can reduce the effective branching factor of the game tree by half in the best case [5]. Simply searching different alternative moves in parallel would lose much of the pruning effect. In other words, the parallel version might do a lot of redundant speculative computation. Parallel alpha-beta algorithms have been extensively studied. For example, Fishburn [2] has shown that simple tree splitting allocation has the theoretical speedup of $p^{0.5}$, where p is the number of processors, when the best moves are searched first. Mandatory work first scheduling has the speedup $p^{0.8}$ for a best-first lookahead tree of fanout degree 38 (typical in chess), though it does not do deep cutoffs. Our program takes tree-splitting allocation scheme to distribute work among processors, and the search branches are given execution priorities so as to do less speculative computation before more speculative computation. We have yet to analyze the speedup, though the performance results have been similar to Fishburn's first scheme.

Here is the summary of characteristics and speedups for the above programs (Table 1).

Table 1: Characteristics and Speedups of Experimental Programs

Program	Amount of Communication/Pattern	Amount of speculative computation	Speedup	
			16 PEs	64 PEs
Pentomino	None	None	15	50
Shortest Path	Large/Local	Medium	8	20
PAX	Large/Global	None	3	-
Tsumego	Little	Large	3~ 10	-

6 Conclusions and Future Work

The Japanese fifth generation computer systems project attempts to build a parallel inference machine (PIM) to provide the computational power needed in large-scale knowledge information processing. The completion of the pilot parallel inference machine Multi-PSI in late 1989 marks the beginning of parallel logic programming practice at ICOT. We pointed out the main factors in speeding up parallel knowledge processing programs.

KL1 is a comfortable language in which to write parallel symbolic processing programs. Especially, the separation of program semantics and mapping makes it very easy to test various alternative mapping strategies.

The Multi-PSI is powerful enough to run medium to large scale programs reasonably fast, and importantly, the issues that come out in programs on large-scale loosely-coupled parallel computers do show themselves in programs runnable on the Multi-PSI. The parallel inference machine operating system (PIMOS) has been developed [1] on the Multi-PSI, and is being expanded with programming and debugging utilities.

We plan to write more programs for problems of different types than the ones reported here, and study scheduling and load distribution strategies.

In AI problems, finding optimal solutions is very often intractable. In such cases, the realistic approach would be to come up with programs to find *satisficing* [11] solutions. We would like to explore parallel satisficing programs. Such programs may not need strict synchronization, which is a good news for any parallel computers.

References

[1] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 230–251, 1988.

[2] J. P. Fishburn. *Analysis of Speedup in Distributed Algorithms*. Computer Science: Distributed Database Systems, No. 14. UMI Research Press, 1984.

[3] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the parallel inference machine (PIM) architecture. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 208–229, 1988.

[4] N. Ichiyoshi. Parallel logic programming on the Multi-PSI. ICOT Technical Report TM-487, ICOT, 1989.

[5] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293-326, Winter 1975.

[6] T. Kurozumi. Present status and plans for research and development. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 3–15, 1988.

[7] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[8] Y. Matsumoto. A parallel parsing system for natural language analysis. In *Proceedings of the Third International Conference on Logic Programming*, pages 396-409. Springer-Verlag, 1987. Lecture Notes on Computer Science 225.

[9] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 436–451, 1989.

[10] H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine: PSI-II. In *Proceedings of 1987 International Symposium on Logic Programming*, pages 104–113, 1987.

[11] H. A. Simon. *The Sciences of the Artificial*. MIT Press, 1969.

[12] K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. ICOT Technical Report TR-208, ICOT, 1986.