

TR-487

Parallel Logic Programming  
on the Multi-PSI

by  
N. Ichiyoshi

July, 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**

# Parallel Logic Programming on the Multi-PSI

Nobuyuki Ichiyoshi

Institute for New Generation Computer Technology

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

## Abstract

The completion of the parallel KL1 implementation on the Multi-PSI, a pilot parallel inference machine, marks the beginning of experimental parallel programming research at ICOT. This paper reports on the experience we have had with our first experimental parallel programs running on the Multi-PSI, and discusses the preliminary results of their performance analyses. It also emphasizes the need to fill the gap between concurrent logic programming and the parallel inference machine, i.e. mapping from abstract/logical parallelism onto physical parallelism. We have found KL1 to be an excellent tool for experiments in this field because of the productivity and flexibility it provides. The completion of its parallel implementation on the Multi-PSI gives an opportunity for testing various ideas.

## 1 Introduction

The fifth generation project is an attempt to build a large-scale knowledge information processing system on a highly parallel computer. Its working hypothesis is that logic programming fills the gap between the high level knowledge information processing and the low level parallel processing. The concurrent logic language<sup>1</sup> GHC[27] is a major result of the research at ICOT. It allows most concepts in concurrent programming to be naturally expressed in the framework of logic programming.

---

<sup>1</sup>Concurrent logic language, committed choice language, and stream AND-parallel language all refer to the same concept, but with emphasis placed on different aspects. The term concurrent logic language implies the expressive power for concurrently running communicating processes. The term committed choice language emphasizes don't care nondeterminism and that the language is sound but incomplete as a logic programming language. The term stream AND-parallel language says what kind of parallelism is there (although, full GHC has OR-parallelism as well).

The utility of GHC for writing parallel algorithms has been sufficiently proven. Stream communication realizes process communication, the short circuit is an easy distributed termination detection technique. The lack of completeness property of a committed choice logic language is compensated for by the two all-solution search techniques: the layered stream method[20] and the technique for transforming a Prolog program into a GHC program [28, 29].

The claim that a concurrent logic language is suited for writing an open system – a system interacting with the outside world – has been proven by operating systems written in such languages[8, 22]. At ICOT, the parallel inference machine operating system PIMOS[5] was developed in a relatively short time on the PIMOS Development Support System (PDSS), a sequential implementation in a Unix environment, and was transported to the Multi-PSI with ease. KL1[5], the language in which the PIMOS is written, is based on Flat GHC. It is augmented by metaprogramming capabilities, but they do not destroy the clean semantics of the base language as the cut and the predicate *var* do in Prolog [16].

The remaining task in the fifth generation project should be to write knowledge information tasks in GHC, and to build a powerful parallel implementation of the language. The first part is as easy (or rather difficult!) with GHC as with any symbolic processing language. The simplicity and clear semantics of GHC helps programming and debugging and makes program transformation feasible[9, 31]. The second part is partially realized by the parallel KL1 implementation on the Multi-PSI/V2, and is expected to be more fully realized when the large-scale Parallel Inference Machine (PIM) is completed.

However, we found out it was not as simple as that, as we wrote programs to run on the Multi-PSI — it became clear that some programs ran efficiently on the machine, and others did not<sup>2</sup>. We do not always get an optimal speedup for a given number of processors. In fact, near-linear speedup was rarely possible.

In a way, it was expected that a concurrent logic language would serve as an interface between logic programming and the parallel machine, just as a machine language did for a conventional machine, or Lisp did for a Lisp machine. But the fact was that parallel software and the parallel machine do not meet at the concurrent language plane. Rather, there is a layer between the language and the machine, the layer of program mapping. A program written in a concurrent logic language defines data dependency and possible parallelism, but says nothing about how the logical concurrency is embodied as physical parallelism, which determines the program performance.

This gap between concurrent logic programs and the parallel machines must be

---

<sup>2</sup>On the shared memory multiprocessor, Evan Tick[26] did measurements of various types of benchmark programs in Flat GHC and OR-parallel Prolog.

filled to achieve real speedup. In our case, we are interested in scalable<sup>3</sup> speedups on scalable<sup>3</sup> architectures like those of the Multi-PSI and the PIM.

We take an experimental approach in the parallel logic programming research. That is, given a problem, we find a parallel algorithm, write it down in KL1, decides on some mapping strategy. Then we run the program on the Multi-PSI, measure the performance, and analyze results, and try new algorithms or new mapping strategies.

## 2 KL1

The concurrent logic language KL1[5] is based on Flat GHC, and has metaprogramming capabilities and pragmas. The metaprogramming capability of KL1 is realized by the *shoen* (pronounced 'sho-en') facility. While goals executed tail-recursively (*processes*) define small-grain threads of control, a *shoen* defines a larger-grain computational unit. A *shoen* has an in-coming *control stream* and an out-going *report stream*. The computation under a *shoen* can be stopped, re-started, and aborted by control stream messages, and the termination of computation, exceptional events, etc. are reported on the report stream.

The user can control the scheduling and load distribution by attaching pragmas to body goals. Goals are given execution priorities. By default, a child goal inherits the priority of the parent goal and is executed on the same processor as the parent goal, but they can be changed by a priority pragma (*Goal@priority(Prio)*) and a load distribution pragma (*Goal@processor(Proc)*). Pragmas do not change the semantics of the program. The separation of program semantics and program mapping is one of the merits of using KL1 in parallel programming research.

As compared to procedural parallel programming, in which the programmer has to be very careful about synchronization and data consistency, it is far easier to program in KL1. This increase in productivity is paid for by extra overheads on the part of the implementation. Perhaps, KL1 in parallel programming is like Lisp in AI programming: at early stages of research and development, productivity and modifiability are more important than the raw speed of the programs.

## 3 The Multi-PSI

The Multi-PSI is a multiprocessor running the concurrent logic language KL1[25]. The main purpose of its development was to provide a network-connected, scalable

---

<sup>3</sup>Something being *scalable* means that it can be parameterized by the size index  $N$ , and some important properties remain true as  $N$  becomes very large. For example, the shared-bus architecture is not scalable (under the current technology), because as the number of processors increases, memory access latency becomes far from constant short time due to bus contention.

machine for testing and evaluating parallel KL1 implementation techniques and for developing parallel software. The first version of the Multi-PSI was developed during 1986 and 1987. It consisted of six personal sequential inference machines (PSIs) connected by a  $2 \times 3$  mesh network. The KL1 implementation[13] was written in ESP, a Prolog-like system description language. It was a useful tool for testing the implementation and for running a few KL1 programs, but it was slow (roughly 1K append LIPS per processor). Measurements of this early implementation are given in [24]. As the next step, the Multi-PSI version 2 was developed and became operational in the late 1988. The processing element is the CPU of the PSI-II, the faster and smaller version of the PSI machine. It connects up to 64 processors by an  $8 \times 8$  mesh network. The parallel implementation was much improved [18], and was written in the microcode. It attains 130K append LIPS per processor. The raw speed of KL1 can still increase by global analysis by the compiler, sheer hacking, etc.

The Multiple Reference Bit (MRB) scheme[4] is adopted in the parallel KL1 implementation on the Multi-PSI. For a vector known by the MRB information to be single-referenced, a destructive update of an element may be done without violating the logical semantics. This enables the cost of vector element update to be kept to a low constant independent of the vector size. This is very important, because we do not want the computational complexity of a concurrent logic program to be of worse asymptotic order than a program written in a procedural language.

For performance measurements, the operating system PIMOS has a timer device, and the number of reductions made under a shoen can be reported. The implementation does not support load distribution profiling, but a special program called the *performance meter* was written in KL1 to display a rough image of processor utilization rate real-time for all processors. It consists of a master process in one processor and slave processes with the lowest system priority in all the processors (one slave process per one processor). The number of reductions made by a slave process is proportional to the idle time in the processor. The slave processes are asked by the master process to give the numbers of reductions every two seconds, and the processor utilization rates are displayed in the performance meter window using a color scale<sup>4</sup>.

## 4 Experimental Programs on the Multi-PSI

We have written KL1 programs to solve four problems. The programs have various types of algorithms, and their run time characteristics are also very different. They are as follows:

---

<sup>4</sup>The rather long interval of two seconds is determined by the display throughput of the window

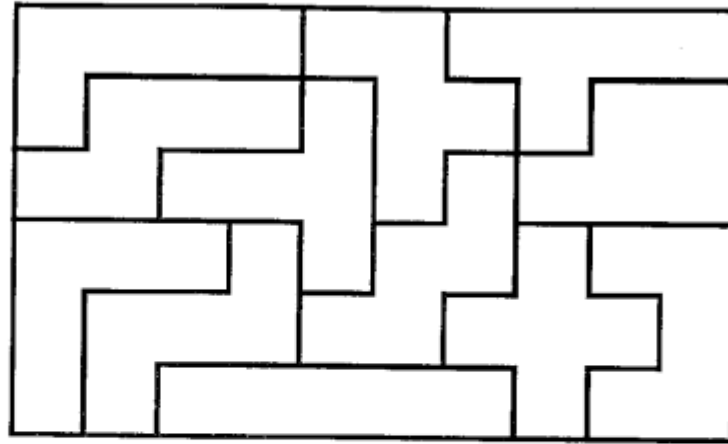


Figure 1: Pentomino

### Packing Piece Puzzle (Pentomino)

A rectangular box and a collection of pieces with various shapes are given (Fig. 1). 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 (Fig. 2). The program performs a distributed graph algorithm.

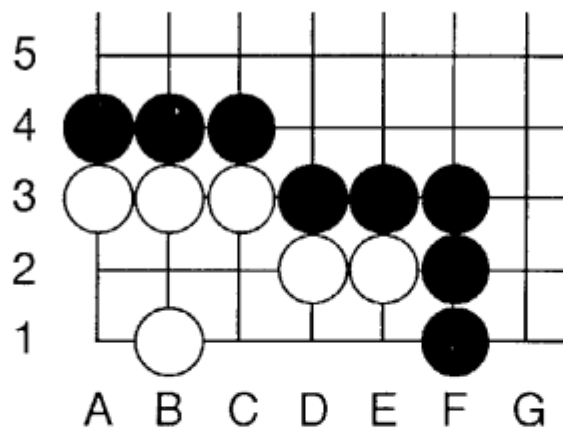
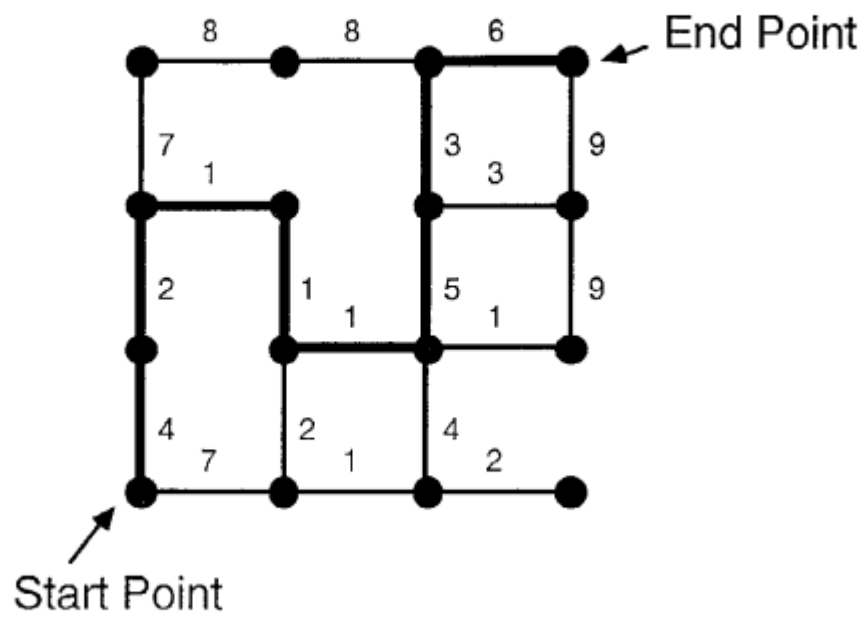
### Natural Language Parser

The problem is to construct all possible parse trees for an English sentence. The program called PAX 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. 3). 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<sup>5</sup> reached. The program does a parallel alpha-beta search.

<sup>5</sup>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.



We used those four programs for the Multi-PSI demonstration at FGCS'88. Since that time, some of the programs have been re-written and the performance improved 5 to more than 20 times.

#### 4.1 Packing Piece Puzzle (Pentomino)

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 alternative branches.

The program does a top-down exhaustive search of this OR-tree. A search process forks at an OR-node to spawn child processes corresponding to the OR-branches. The solutions are collected through a merge tree.

We mapped this logical process structure onto the Multi-PSI in the following way. The program starts on a master processor (say  $PE_0$ ). It does the searching up to a certain predefined level, and after that level is reached the child processes are evenly distributed to all processors (including itself), which do the rest of the searching. If (1) the top part of the search tree taken care of by the master processor is only a small portion of the entire search tree, (2) there are sufficient number of tasks to be distributed over the processors, and (3) the average size of the task is significantly larger than the distribution overhead (including communication, solution collecting), then we can expect a speedup close to the number of the processor.

In our case with the  $8 \times 5$  puzzle, the predefined search level was 2 ( $PE_0$  finds all possible placements of first two pieces to cover the top left corner), resulting in about 170 tasks to be distributed. When the subtrees were distributed to the processors as they are created, the performance meter showed some processors became idle early and some late. The adoption of on-demand load balancing made load (not the number of subtrees) more evenly distributed, and resulted in shorter computational times. On-demand load balancing was realized by placing a lowest priority idling goal in each processor. When a processor finishes a given task (with higher priority) or goes into suspension, the idling goal is scheduled, and it asks for more work from the master processor. We list the performance figures in Table 1. The plateau of speedup at about 32 processors in the on-demand load distribution performance seems to come from the bottleneck at the master process of providing work. This may be alleviated by creating sub-master processes in a few processors.



Table 1: Performance Figures for Pentomino (in seconds)

Number of processors	1	4	16	32	64
Semi-Static Load Distribution (Speedup)	270 (1.0)	85 (3.2)	24 (11)	18 (15)	9.4 (28)
On-Demand Load Distribution (Speedup)	270 (1.0)	– –	19 (14)	11 (25)	13 (21)

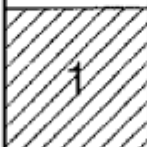
## 4.2 Shortest Path Problem

The first version of the program used a completely distributed algorithm. The nodes in the graph are represented by KL1 processes. Each of the node process retains  $P$ , a shortest known path from the start node and  $C$ , the cost of that path, and exchange shortest path information with its neighboring nodes. When a node receives a path information which is better than the one it retains, i.e. a path information  $cp(Cost, Path)$  such that  $Cost < C$ , it replaces  $C$  and  $P$  by  $Cost$  and  $Path$ . The node then sends to each of its neighboring nodes a path information  $cp(Cost + CE, [Self|Path])$ , where  $CE$  is the cost of the edge between the sending node and the receiving node, and  $Self$  is the identifier of the sending node. When a node receives a path information which makes no improvement, it simply ignores the information.

In the initial state, all nodes suppose that the shortest paths are unknown and the costs are infinity. The computation is initiated by throwing in a shortest path information  $cp(0, [])$  to the start node, meaning the empty path is a shortest path from itself and the cost is zero. The computation terminates when there remains no unprocessed path information. Termination is guaranteed for a graph whose edge costs are nonnegative. The original program used the short circuit technique to detect termination (path informations had circuit switches).

The original version of the program[23] was first compiled by a Flat GHC compiler [30] to run on DEC-20. It was transported to the Multi-PSI after the KL1 implementation became available.

For the performance testing, we generated a  $100 \times 100$  square graph with randomly determined edge costs ( $1 \sim 99$ ). To reduce inter-processor communication, we did not take the nodes as units of load distribution, but clusters of nodes (called *blocks*), so that inter-processor communication arises only across the block boundaries (Fig. 4). Let us call this mapping *2s* (2 dimensional simple). The problem with this mapping was that the shortest path information tends to spread from the start node in a wave front. This was observed in the performance meter window. Since at any point of computation only the processors on the wave front were busy,

13	14	15	16
9	10	11	12
5	6	7	8
 1	2	3	4

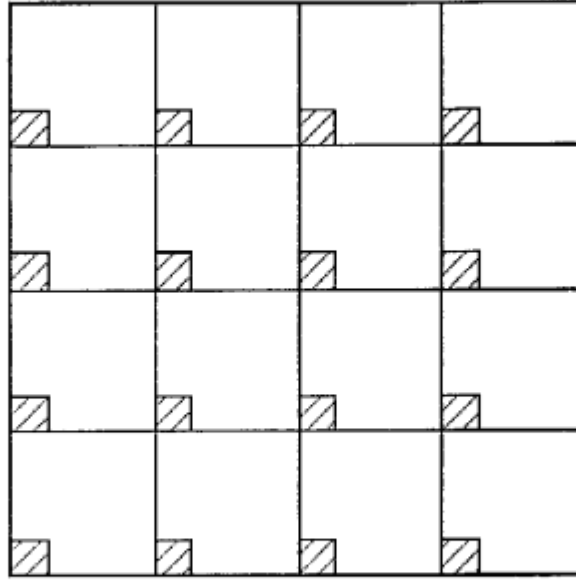
(The hatched block is mapped onto  $PE_1$ .)

Figure 4: Two Dimensional Simple

processor utilization ratio was low. Thus, we tried another mapping, where the graph was divided into smaller blocks and a processor took care of multiple blocks lying far apart (Fig. 5). Let us call this mapping  $2m$  (2 dimensional multiple). The increased processor utilization more than paid for the increased inter-processor communication overhead. The peak performance was attained by the  $4 \times 4$  multiple mapping for both 16 and 64 processors.

The completely distributed algorithm had a problem in computational complexity. It can be very inefficient if path information with large cost spreads first and is updated by lower cost path information, which is again updated by still lower cost path information. Actually, the worst case complexity is exponential in the number of nodes!

The program was completely re-written so that path information with lower cost may be given higher priority. This was done in the following way. A path information is represented not by a stream message but by a process. A path process with lower cost is given a higher execution priority than a path process with higher cost. When a path process visits a node process, it sends the path information to the latter, which checks it against the shortest path information it retains, and replies either fork (the new path information is better) or kill (the new path information is not better, and thus is ignored). On receiving the reply, the path process either forks into path processes to visit the neighboring nodes or it terminates. The node processes and path information processes belong to different *shoens*, and the termination detection



(The hatched blocks are mapped onto  $PE_1$ .)

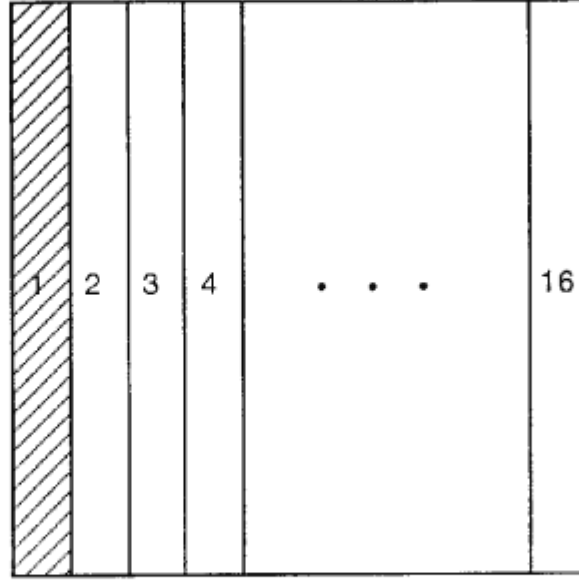
Figure 5: Two Dimensional Multiple

Table 2: Performance Figures for Shortest Path Problem (in seconds)  
(Two Dimensional Mapping)

Number of processors	1	4	16	64
Two Dimensional Simple (Speedup)	11.1 (1.0)	6.5 (1.7)	3.5 (3.1)	2.0 (5.6)
Two Dimensional Multiple ( $4 \times 4$ ) (Speedup)	– –	4.5 (2.5)	1.9 (6.6)	0.9 (12.0)

is delegated to the builtin termination detection mechanism of the shoen[21]. In one processor the program runs exactly like Dijkstra’s sequential shortest path algorithm. The program has the same computational complexity when run on more than one processor. The performance improved dramatically. The summary of performance figures are given in Table 2.

We can estimate the speedup for the direct mapping under the following simplifying assumptions: (1) the start node is located at the bottom left corner of the square graph, (2) the number of nodes in one mapping block is large, (3) a shortest path does not deviate from the straight line to a great extent (this is the case when there is no non-local pattern in edge cost distribution), and (4) the communication cost is negligible. Under these assumptions, the shortest path information advance in a wave front at some uniform speed. Let there be  $p = q^2$  processors. The graph is



(The hatched block is mapped onto  $PE_1$ .)

Figure 6: One Dimensional Simple

divided into  $q^2$  blocks. Let  $T$  be the time it takes one processor to cover one block. Since the time it takes one processor to compute the shortest paths is proportional to the number of nodes, it will take a processor  $p \times T$  time to solve the entire problem. When the program is mapped on the  $p$  processors, it will take  $q \times T$  time to solve the entire problem (there are  $q$  blocks on the diagonal, and it takes  $T$  time for the wave front to cover one block). Thus, the speedup is

$$\frac{p \times T}{q \times T} = \frac{q^2}{q} = q = \sqrt{p}.$$

The measured performance figures actually show roughly two-fold speedup for 4-fold increase in the number of processors.

It is more difficult to estimate the speedup for multiple mappings. In a real computation, we could guess that the wave front is not a line but a band with some non zero width. If so, the smaller the block size the more blocks the wave front covers, thus making more processors busy.

An interesting alternative for making the speedup ratio higher is to divide the graph into  $p$  thin rectangles (Fig. 6). We call this mapping (*1s*) (1 dimensional simple). Under the same simplifying assumptions, the speedup is expected to be

$$\frac{p^2}{2p-1} \sim \frac{p}{2} \text{ (when } p \gg 1).$$

However, the problem with this mapping is higher communication overhead. The communication overhead is expected to be proportional to the ratio between

Table 3: Performance Figures for Shortest Path Problem (in seconds)  
(One Dimensional Mapping)

Number of processors	1	4	16	64
One dimensional Simple (Speedup)	11.1 (1.0)	5.6 (2.0)	2.7 (5.1)	1.2 (9.2)

the length of the block boundary and the area of the block. For mapping (2s), the communication overhead  $co$  is estimated as

$$co = \frac{4(L/q)}{(L/q)^2} \cdot C = \frac{4qC}{L} = 4qC',$$

where  $L$  is the length of the side of the square, and  $C$  is a constant dependent of the program and the implementation,  $C' = C/L$ . This is proportional the square root of the number of processors. For mapping (1s), the communication overhead  $co$  is estimated as

$$co = \frac{2LC}{L^2/p} = \frac{2pC}{L} = 2pC',$$

which is directly proportional to the number of processors. This is an interesting tradeoff between processor utilization and communication overhead. We give the performance figures for mapping (1s) in Table 3. The communication constant  $C'$  can be calculated from the real speedup: it is estimated to be 0.02.

### 4.3 Natural Language Parser

The program performs PAX algorithm[17], a parallel parsing scheme for natural language analysis. It is a left-corner parser[1], in which phrases are constructed from bottom to top and from left to right. There is a very close one-to-one correspondence between the PAX algorithm and the bottom-up chart parsing[15]. In the latter, parse trees are constructed bottom-up starting from the words in the sentence, and adjacent partial parse trees of sentence fragments are checked against each other and a bigger parse tree is constructed according to the grammar rules. The partial parse results are maintained in a data structure called the chart, so that duplicate parsing is avoided. PAX algorithm can be thought of as an implementation of chart parsing in a concurrent logic language. Partial parse results are represented by processes, and the neighbor relation by streams.

Since the partial parse trees interact with each other, the program is communication intensive as was the shortest path program. But unlike the latter, PAX has more random communication patterns, because as the bigger parse trees are constructed, the distance (in words) between two interacting parse trees becomes larger.

Table 4: Performance Figures for PAX (in seconds)

Number of processors	1	16
Mapping d(1) (Speedup)	45 (1.0)	100 (0.45)
Mapping p(1) (Speedup)	–	19 (2.4)
Mapping p(7) (Speedup)	– –	15 (3.0)

We tried two different mapping strategies.

**d( $w$ )**  $W$  consecutive words are mapped on one processor. (When the sentence length divided by  $w$  is larger than the number of processors, word blocks are mapped onto the processors in a round robin.) All child processes remain in the same processor as the parent process.

**p( $w$ )** Same as above, except each child process is thrown to the processor where the input stream is generated.

In mapping d( $w$ ), since the parsing is done from left to right, the process  $P$  representing a parse tree for a sentence fragment will be created in the processor in charge of the rightmost word in that fragment. Processes  $Q_i$  representing the parse trees for sentence fragments to the right of and adjacent to that fragment will be located in the processors at the end of those fragments. Processes  $Q_i$  reads the stream generated by process  $P$ , resulting in one to many inter-processor communication.

In mapping p( $w$ ),  $Q_i$  move to the processor where  $P$  has been created. Though process  $P$  is likely to have moved to another processor, there is only one to one inter-processor communication<sup>6</sup>. The clustering of processes in this mapping strategy, however, could cause computational load to concentrate on a small number of processors.

The measurement results given in Table 4 show that (1) parallel execution resulted in longer computational time when mapping strategy d(1) was used, and that (2) only 3-fold speedup was attainable by 16 processors using mapping strategy p(1) and p(7). The reasons for this poor speedup (or speeddown!) are uneven load distribution and high rate of inter-processor communication overhead.

The parameter  $w$  is intended to control the rate of inter-processor communication in the following way. At the early stage of the parsing when the average distance of

<sup>6</sup>The stream communication is still one process to many processes. But since the KL1 implementation has a hashing mechanism to prevent duplicate data copying between two processors [14], the number of inter-processor messages is the same as when the number of  $Q_i$  is one.

process communication is small compared to  $w$ , most stream communication is likely to be between two processes in the same processor. It is rather surprising then that, in this communication intensive program, making  $w$  bigger does not make parsing time significantly shorter ( $p(7)$  vs.  $p(1)$ ). The following observation, though not confirmed, may explain this unexpected result. If the sentence is very redundant, a great deal of stream communication is expected in the later part of parsing, when inter-processor communication cannot be suppressed by even a large value of  $w$ .

The layered stream method[20] for all-solution search, closely related to the PAX algorithm, also incurs intensive stream communication. Achieving good speedup in this kind of programs is a challenge worth tackling.

#### 4.4 Tsumego Solver

The alpha-beta search[32] is a well-known technique for pruning search space in two-player game programs. When the best move is searched first, the alpha-beta search can theoretically look ahead the moves twice the depth as exhaustive mini-max search in a given time. Equivalently, the effective branching factor for the alpha-beta search is the square root of that for the exhaustive mini-max search. 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.

If the game tree search were done in purely parallel breadth first manner, there would be no pruning of search space. No matter how many ( $N$ ) processors there are, there is a number  $D$  ( $D \sim \log N$ ) such that a sequential alpha-beta search is faster than a parallel breadth first exhaustive search. A parallel search that has a pruning effect comparable to the sequential alpha-beta search is definitely needed. In the Pentomino and Shortest Path programs, uneven load distribution and communication overhead stood in the way of speedup. But in the Tsumego program, parallelization itself was the problem.

In the parallel alpha-beta search in the Tsumego program, we tried two kinds of scheduling. In both of them, the game tree is expanded in one master processor to a certain fixed depth and the subtrees rooted at that level are distributed to the processors, just like the Pentomino program. The moves of the second player at a given board situation are given the same priorities. In scheduling (a), the moves of the first player are sequentially searched. In scheduling (b), the searches of the first player's moves are given priorities so that if some branch is to the left of another branch in the search tree, the former always has a higher priority. A search subtrees with a higher priority comes before a subtree with a lower priority in the waiting task queue for load distribution.

The first scheduling is closer to the sequential alpha-beta search than the second one, but 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 scheduling (b) will have good speedup for a problem instance in which the pruning effect of sequential alpha-beta is small. We list some performance figures for several problem instances and different depths for subtree distribution in Table 5.

The figures support our expectation, but it is rather annoying that the computational time is sensitive to the depth of distribution as well as problems and scheduling strategies. We have yet to analyze these data, and also to try more sophisticated parallel alpha-beta search algorithms such as those in [7, 6].

## 5 Discussion

The basic formula for speedup  $S$  is

$$S = \frac{W}{\frac{W'+C}{UN}} = NU \cdot \frac{W}{W'+C} = NF,$$

where  $W$  is the amount of work done by a single processor,  $N$  the number of processors,  $U$  the average processor utilization,  $W'$  the total amount of work done by a multiprocessor,  $C$  the overheads peculiar to parallel processing, i.e. overheads of scheduling, synchronization and communication,  $F$  (the effective processor utilization)  $= UW/(W' + C)$ . This formula says, to achieve good speedup, for a given multiprocessor, we have to pay attention to the following items.

- (1) To increase processor utilization (to make  $U$  close to 1)

First, it is necessary to extract enough parallelism from the problem to match the number of processors. We are confident that large-scale problems have enough parallelism to match a highly parallel multiprocessor. For smaller problems, we need not use such a machine.

Second, it is necessary to distribute the extracted parallelism over the processors to keep them busy. This is the problem of load balancing.

For a program with uniform parallelism during computation, the load distribution may be decided statically. It was not the case with the shortest path program, as shown in the low processor utilization of the mapping 2s.

If there are plenty of unrelated tasks to do, as in the Pentomino program, fairly good load balancing can be realized by distributing roughly the same number of such tasks. But even then, dynamic balancing paid off because the task sizes vary.

The current load distribution pragma of KL1 may be too specific or too low-level. In the future, we would like to offer the programmer more a abstract view



Table 5: Performance Figures for Tsumego

Problem 1†		
Sequential alpha-beta	1,696 moves 15 sec	
Depth	Scheduling (a)	Scheduling (b)
1	1,696 moves 14 sec	13,481 moves 15 sec
2	1,696 moves 5.0 sec	18,262 moves 13 sec
3	1,696 moves 5.1 sec	7,901 moves 6.0 sec
4	1,696 moves 2.9 sec	11,656 moves 8.3 sec
5	1,696 moves 3.0 sec	7,300 moves 6.6
Max Speedup	5.2	2.5

Problem 2‡		
Sequential alpha-beta	21,257 moves 210 sec	
Depth	Scheduling (a)	Scheduling (b)
1	21,257 moves 203 sec	20,715 moves 25 sec
2	24,281 moves 114 sec	26,077 moves 24 sec
3	26,269 moves 139 sec	48,979 moves 31 sec
4	29,269 moves 70 sec	68,425 moves 45 sec
5	30,349 moves 79 sec	67,323 moves 49 sec
Max Speedup	3.0	8.8

† The best move is the leftmost branch.

‡ The best move is a far right branch.

of the machine. The Processing Power Plane (PPP)[3] is one candidate. It is an imaginary plane on which the programmer distributes the computational load. It is mapped onto the physical processors, but the machine monitors the computational load and moves the processor boundaries dynamically to make computational load evenly distributed over the processors. We may want to introduce some *elasticity* to the PPP to resist too much distortion for load balance. If the programmer knows that certain processes have close connections and communicate very frequently within themselves, he/she may cluster these processes and map to points in the PPP that are very close to each other. If there are many such clusters, the load balancing mechanism tends to balance the load, but, by the elasticity, not too much so to tear a single cluster into pieces to be mapped onto different processors.

- (2) To avoid doing work whose result may not be used afterwards (to keep  $W'/W$  close to 1)

The work whose result may not be used afterwards is often called *speculative* work as compared with *mandatory* work, work that must be done[11]. In the alpha-beta search, searching non-leftmost moves is a speculative work. The second version of shortest path program that used the priority mechanism did much less speculative work than the first one.

An interesting phenomenon is the superlinear speedup, that is, speedup greater than the number of processors ( $S > N$ ). The condition  $W' < W$  is necessary for this to happen. In the case of the alpha-beta search, if the best move  $M$  is a far right branch of the root, parallel search may find the good alpha value of  $M$  early and use it to prune (some subtrees under) the other branches. What happens here is that, for a certain problem instance, the speculative work done by the sequential algorithm can be larger than the speculative work done by the parallel algorithm. If it were known in advance that the best move lies in the right hand side, a (good) sequential algorithm should have searched that part first. This means that a superlinear speedup is always an unexpected outcome, and in the average case analysis it should not be possible.

On the other hand, another source of superlinear speedup could also come from data locality when the data size is huge. Consider the following extreme example. Suppose one processor can have local memory of 1K words and a disk (for simplicity, no I/O buffering is assumed), main memory access takes  $m$  microseconds, and disk access takes  $M$  microseconds ( $M \gg m$ ). The problem is summing 2K numbers. I would take one processor  $T = Km + KM$  microseconds to add up 2K numbers. If two of these processor-memory pair were connected by a channel, and the sending one word information takes  $C$

microseconds, it would take  $T' = Km + C$  microseconds to determine the sum. If  $KM > Km + 2C$ ,  $T' < T/2$ , speedup greater than the number of processors. (The source of the extra speedup was data compaction). This argument may be a little far fetched, but as we deal with bigger and bigger data set, data access locality will be of greater concern. In this sense, a multiprocessor is a step towards data parallelism<sup>7</sup>.

- (3) To keep low the overheads peculiar to parallel processing (to keep  $C$  small)

There are overheads inherent in parallel processing. Process switching, scheduling, synchronization, and communication are all such overheads. Keeping them at a low level should be of concern at all levels in a multiprocessor system, from algorithm design to user controllable scheduling, to load balancing, to compiler and implementation techniques and to architecture and hardware technologies.

At programming level, for example, process fusion is a program transformation technique that removes certain kind of synchronization overhead.

At parallel logic language implementation level, tail recursion optimization makes process switching less frequent. LIFO scheduling of executable goals by goal stack in our KL1 implementation and most other implementations helps keep suspension rate lower. Reducing inter-processor communication was a major design issue in the KL1 implementation.

Unfortunately, load balancing and communication localization tend to conflict with each other. If there were no penalty by communication, evenly distributing KL1 goals as they are spawned would result in a good load balance and near linear speedup. This is not the case, however, and seems to be one of the essential aspects in highly parallel computing.

The above discussion was how to attain highest speedup, given a fixed number of processors, that is, to make effective processor utilization close to 1. For a scalable architecture like the Multi-PSI, scalability of a program is also an important issue. A program in which 50% of processor time is consumed by communication overhead could still be useful if the overhead is independent of the number of processors and an absolute speedup is of the ultimate goal<sup>8</sup>.

We concentrate our attention on scalable MIMD multiprocessors. There has been much theoretical work on time complexity of parallel algorithms on the P-RAM (parallel random-access machine) [10]. Typically, an *efficient parallel algorithm* employs

<sup>7</sup>Data access locality is at cache level for a shared memory multiprocessor with snoopy cache, and at local memory level for a network-connected multiprocessor.

<sup>8</sup>For this kind of argument, *heavily loaded limit*, asymptotic speedup for a large number of processors as problem size increases [2], is a good conceptual tool.

a polynomial (in problem size) number of processors to solve the problem of size  $n$  in a polylogarithmic time, i.e.  $O(\log^k n)$  (such a problem is said to belong to class  $NC$ ). There are characterizations of  $NC$  in logic programs [19]. But current hardware technology does not allow a great number of processors to concurrently read and/or concurrently write the same memory location, as assumed in a P-RAM. Realization of such machines being unthinkable for now, parallel programming practice has to assume architectures in which memory contention exists and memory access latency becomes larger as access distance becomes larger. For such realistic scalable multiprocessors, research should naturally be focused on scalable parallel algorithms that achieve maximal speedups for a given number of processors. More precisely, for  $N$  number of processors, we look for a parallel algorithm that solves a problem with sequential time complexity  $T_s \geq O(N)$  in time  $T_p$  such that speedup  $S = T_s/T_p$  is close to  $N$ , or for hard problems,  $\sqrt{N}$ , etc.

## 6 Conclusion

In summary, the paper reported on the experimental parallel logic programs on the Multi-PSI, and analyzed the speedups obtained. We have come to realize the gap between KL1 programs and the parallel inference machine, i.e. the mapping of the abstract/logical parallelism in the parallel programs onto the physical parallelism realized on the parallel machines, and have started to address it. The mapping problem should be of importance not only for programmers of concurrent logic languages but for everyone who is engaged in multiprocessor programming for performance. There is as yet no universal method of good mapping — we have to devise one mapping strategy for one problem, and another one for another problem —, but, in a sense, that means the richness of the mapping problem.

KL1 and the Multi PSI turn out to be good tools for parallel program research. The pragma facility in KL1 makes experimenting scheduling and load distribution strategies without affecting program semantics. The Multi-PSI has a scalable architecture and is powerful enough to run large-scale programs. The issues that come out in scalable programs do arise in programs runnable on the Multi-PSI.

One important feature that is lacking in the current Multi-PSI environment is performance measurement tools. Currently, the execution time and the performance meter pattern are the only clues for program performance that are readily available to programmers. Counting the numbers of various inter processor messages is a painstaking task, possible only with the help of an implementor. We definitely need better performance profiling tools — tools that show process history, processor utilization graph, message handling rate, etc. <sup>9</sup>

---

<sup>9</sup>The MuT system has a few such tools [12].

## 7 Acknowledgments

I would like to thank my colleagues in the parallel KL1 implementation on the Multi-PSI and parallel programming research. Without the efforts of the following people, this work was not possible: Reiko Sato, who coded the performance meter, Masakazu Furuichi of Mitsubishi Electric, who wrote the Pentomino program, Kumiko Wada of Oki Electric Industry, who wrote the shortest path program, Iiroyuki Sato, a former member of ICOT 4th Laboratory, now back in Mitsubishi Electric, who wrote the PAX program, and Kasumi Susaki who helped him, Hiroaki Oki of Future Technology Laboratories and Shin'ichi Sei who wrote the Tsumego program. Satoshi Onishi helped me by MacDrawing the figures and Akira Imai taught me how to include postscript figures in the  $\text{\LaTeX}$  text.

Kazuo Taki was the first at ICOT to point out the *missing link* between concurrent logic programs and the parallel inference machines, and started to organize a group of people to embark on that research area. Takashi Chikayama is the main designer of KL1, and has predicted the necessity of fine-grain priority mechanism. He wrote the original version of the second algorithm of the shortest path problem.

Finally, I am indebted to Shun'ichi Uchida, the chief of ICOT Fourth Laboratory for leading the Multi-PSI project, and Kazuhiro Fuchi, the director of ICOT, for giving me the opportunity in participating in this exciting frontier of research and development.

## References

- [1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 1: Parsing. Prentice Hall, 1972.
- [2] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, 1989.
- [3] T. Chikayama. Load balancing in a very large scale multi-processor system. In *Proceedings of Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems*. SICS, 1986.
- [4] T. Chikayama and Y. Kimura. Multiple reference management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 276–293, 1987.
- [5] 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.

- [6] E. W. Felten and S. W. Otto. A highly parallel chess program. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 1001–1009, 1988.
- [7] J. P. Fishburn. *Analysis of Speedup in Distributed Algorithms*. Computer Science: Distributed Database Systems, No. 14. UMI Research Press, 1984.
- [8] I. Foster. *Parlog as a Systems Programming Language*. PhD thesis, Imperial College, London, March 1988.
- [9] K. Furukawa, A. Okumura, and M. Murakami. Unfolding rules for GHC programs. *New Generation Computing*, 6(2,3):143–157, 1988.
- [10] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
- [11] R. H. Halstead, Jr. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–502, December 1986.
- [12] R. H. Halstead, Jr., May 1989. Personal communication.
- [13] N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of Flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 257–275. The MIT Press, 1987.
- [14] N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. A new external reference management and distributed unification for KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 904–913, 1988.
- [15] M. Kay. Algorithm schemata and data structures in syntactic processing. Technical Report CSL-80-12, Xerox Palo Alto Research Center, October 1980.
- [16] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
- [17] 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.
- [18] 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*, 1989.

- [19] Y. Okabe and S. Yajima. Parallel computational complexity of logic programs and alternating Turing machines. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 356–363, 1988.
- [20] A. Okumura and Y. Matsumoto. Parallel programming with layered streams. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 224–231, 1987.
- [21] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. I Architecture*, pages 18–22, 1988.
- [22] E. Shapiro. Systems programming in Concurrent Prolog. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, 1984.
- [23] S. Takagi. A collection of KL1 programs, Part 1. ICOT Technical Memorandum TM-0311, ICOT, 1987.
- [24] K. Taki. Measurements and evaluation for the Multi-PSI/V1 system. In K. Fuchi and L. Kott, editors, *Programming of Future Generation Computers II*, pages 365–391. Elsevier Science Publishers B.V. (North-Holland), 1988.
- [25] K. Taki. The parallel software research and development tool: Multi-PSI system. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 411–426. Elsevier Science Publishers B.V. (North-Holland), 1988.
- [26] E. Tick. A performance comparison of AND- and OR-parallel logic programming architectures. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.
- [27] K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. ICOT Technical Report TR-208, ICOT, 1986.
- [28] K. Ueda. Making exhaustive search programs deterministic. *New Generation Computing*, 5(1):29–44, January 1987.
- [29] K. Ueda. Making exhaustive search programs deterministic, Part II. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 356–375, 1987.
- [30] K. Ueda and T. Chikayama. Concurrent Prolog compiler on top of Prolog. In *Proceedings of 1985 Symposium on Logic Programming*, pages 119–126. IEEE Computer Society, 1985.

- [31] K. Ueda and K. Furukawa. Transformation rules for GHC programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 582–591, 1988.
- [32] P. H. Winston. *Artificial Intelligence*. Addison-Wesley Publishing Company, second edition, 1984.