TR-0839

Programming Environment of PIMOS

by
S. Ishida & T. Chikayama

April, 1993

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~ 5

Institute for New Generation Computer Technology

Programming Environment of PIMOS

Shigeru Ishida Takashi Chikayama
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
{ishida, chikayama}@icot.or.jp

Abstract

In the Japanese fifth generation computer systems project, the parallel inference machines, PIMs, have been developed to provide the computational power required for high performance knowlege information systems. They are designed to run a concurrent logic programming language, KL1, which is based on a flat version of GHC. The operating system for PIMs, PIMOS, is designed to control highly parallel programs efficiently and to provide a comfortable software development environment for the KL1 language. This paper describes the programming environment of PIMOS, focusing on its debugging facilities.

1 Introduction

In the Japanese fifth generation computer systems project, the parallel inference machines, PIMs[1][2], have been developed to provide the computational power required for high performance knowlege information systems. PIMOS[3][4] is the operating system which is designed to control highly parallel KL1 programs efficiently to draw maximum potencial from PIMs.

This paper introduces several facilities for debugging known to be effective through our experiences of developing KL1 programs. Section 2 describes the outline of the KL1 language and debugging of KL1 programs. Section 3 introduces the debugging facilities. Section 4 introduces the performace analysis facilities. Section 5 gives conclusion and future plans.

2 KL1

2.1 Outline of KL1

KL1[5] is a concurrent logic programming language based on a flat version of GHC[6].

2.1.1 Execution Model of KL1

A KL1 program consists of clauses of the following form:

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

where H, G_i, and B_j are respectively called a clause head, a guard goal, and a body goal. The '|' operator is called the commit operator. The part of a clause before '|' is called a guard, and the part after '|' is called a body. A KL1 clause is executed as follows:

- In contrast to Prolog, all clauses for the same predicate can be tried in parallel.

```
producer(Stream) :- true |
    Stream = [msg|NextStream],     % Sender
    producer(NextStream).
consumer([Msg|Stream]) :- true |   % Receiver
    consumer(Stream).

?- producer(S), consumer(S).       % Invorking Goals
```

Figure 1: Example of Stream Communication

- The head and the guard of a clause specify the conditions for the clause to be selected. The guard goals are tested sequentially left to right. If a guard goal cannot succeed without instantiating the variables in the caller goal, then the guard goal suspends until they are instantiated.
- The commit operator selects one of the clauses whose guard has succeeded.
- The body goals of the selected clause are then executed in parallel.

Using this mechanism, communication between processes and their synchronization can be effected through shared variables, as described in the next section.

2.1.2 Stream Communication

The stream communication is a basic programming method of KL1 programming. In KL1 programs, communication is realized by using shared variables, such as instantiating a shared variable to a data structure consisting of a message and a new shared variable. This new shared variable can be used in the succeeding communication, realizing continuous communication between processes. This communication style is called "Stream Communication". Figure 1 shows an example of stream communication. Normally, execution of KL1 programs progresses exchanging messages among processes through streams.

2.2 Debugging of KL1 Programs

2.2.1 Frequently Found Bugs in KL1 Programs

Bugs frequently found in KL1 programs can be categorized as follows.

Reduction Failure: When a goal invocation does not match the head of any clauses, or the guard conditions of all the clauses are found to fail, its reduction fails.

Unification Failure: When two variables with inconsistent values are unified in the body, the unification fails.

Unintended Commitment to a Clause: By mistakes in clause selection conditions, a clause may be selected unintentionally. This often is the origin of other problem in this list.

Infinite Loop: It is hard to find a process that fell into an infinite loop without making any output. Such a process may consume all the computational resource, such as memory. Tracing the whole execution may do for a small program, but for a large program, tracing everything is unrealistic.

Deadlock: Deadlock is a state in which two or more processes are awaiting for completion of operation one another. We here use this word in a broader sense for the state of a process awaiting for some event that will never happen. It is difficult to find such a process as it makes no actions to be observed. In addition, when one process suspends its execution this way, many other processes depending on its output may also suspend forever. This dependency chain may include a large number of processes. Thus, reporting all such processes may convey too much information, making it harder to find the origin of the problem.

2.2.2 Improving the Performance of Programs

There are several approaches to improve program performance. The following are most frequently useful ones.

Finding Frequently Called Predicates: By finding predicates that are most frequently called, performance improvement efforts can be more sharply focused.

Priority Control: The order of execution of processes can be arbitrarily chosen for obtaining the correct result thanks to the data-flow synchronization of KL1. However, some scheduling may result in much better performance than others. The priority control mechanism of KL1 allows the programs to roughly specify the execution order for better performance.

Load Distribution: For efficient execution on multiprocessor systems, distribution of computational load is important. On the other hand, too much distribution may require too much communication, resulting in lower performance. Thus, the trade-off between load balancing and communication locality is a key to obtain higher performance, especially on large scale loosely coupled systems. KL1 provides a simple load distribution mechanism that specifies where to execute goals. Finding a good load distribution algorithm using this simple mechanism is one of the most important research topics.

3 Debugging Facilities

3.1 Listener

The Listener is a top level interface of the KL1 program execution, similar to the top level of interactive Prolog systems. The Listener has the following functions.

Goal Execution: The user can interactively execute goals in the Listener top level.

Tracing: Users can see the detailed reduction process of specified goals. Besides the normal single stepping, spy function allows specifying only some predicates to be traced. Execution order can be altered by suspending the execution of certain goals temporarily. Multiwindow trace function displays trace information in multiple windows deviding the traced goals into several groups.

Inspecting and Monitoring Functions: The inspector and the variable monitor can be used to inspect data structures, possibly during their creation. Their details will be described in the following sections.

Execution Profiler: The Listener can collect profiling information. The measured data can be analized and graphically displayed by a performance analysis tool, ParaGraph(see below).

Detection of Deadlock: The Listener reports the deadlock(suspensions that will never be resolved) detected during the garbage collection. The detected goal is the goal that is the "origin" of the causality relationship of deadlock goals[8].

3.2 Inspector

The inspector is a tool to analyze and display data structures interactively. (Figure 2)

```
@0@4096 29    layered: filter([ & , & |E1],1,1,W)
41    * (1)     filter([ & , & |E1],1,1,W)? inspect 1  % inspecting subgoal 1
filter([3*'[' & ']',4* V1 |T1],1,1,W)> me              % list elements
0 : filter
1 : [3*'[' & ']',4* V1 |T1]
2 : 1
3 : 1
4 : W
filter([3*'[' & ']',4* V1 |T1],1,1,W)> 1               % go down elements 1
0 : 3*[ & |Q1]
1 : 4* V1
tail : T1
```

Figure 2: Example of Inspector

3.3 Variable Monitor

For inspecting the future value of a currently uninstantiated variable, a data-driven monitoring process can be initiated from the inspector or the tracer. The value of the monitored variable is reported on its instantiation. It is also possible to incrementally monitor the instantiation of a list structure, which is useful for tracing communication streams. (Figure 3)

```
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)? m G    % monitoring the variable
@0@4096   1   layered: area(ok,begin,B)
   7     * (1)  colors(G)
   8     * (2)  area1(ok,begin,G,B)?
aq:1 G == [red,yellow,blue,white] ?   % reporting the instantiation of the variable
```

Figure 3: Example of Variable Monitor

4 Performance Analysis Facilities

4.1 Runtime Monitor

The Runtime Monitor shows the rate of processor utilization during program execution. The utilization rate is displayed with color patterns or a monochrome density chart periodically. Figure 4 shows an example display of the Runtime Monitor. The y axis is the processor numbers, the x axis is the cycle numbers which corresponds to the time axis.

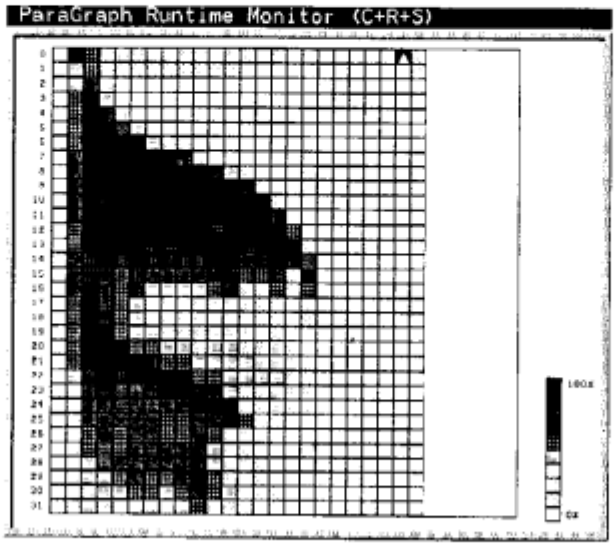

Figure 4: Example Display of Runtime Monitor

A triangle appears when garbage collection took place. Other information can also be displayed such as what kind of low level messages are transmitted and received on which processor and when. The overhead of running the Runtime Monitor is less than 5% .

4.2 ParaGraph

ParaGraph[7] is a tool for more detailed execution profile of KL1 programs. The profiling data has three axes, "What"(the predicate), "When"(time period) and "Where"(processor number). The available imformation is the number of reductions and suspensions.

In sequential program execution, "Where" is fixed and "When" axis is not important, because the execution order is determined. In parallel program execution, all these three dimensions are essential.

Detecting infinite loops

ParaGraph is not originally intended to be a tool for program debugging. However it is also useful in detecting infinite loops.

Figure 5 shows an example of detecting an infinite loop using ParaGraph. The predicate that fell into an infinite loop can be detected at a first glance by the transition of reduction numbers.

5 Conclusion

This paper described the programming environment of PIMOS. The system, although in an experimental stage, has been used since 1988 for development of many application software and PIMOS itself.

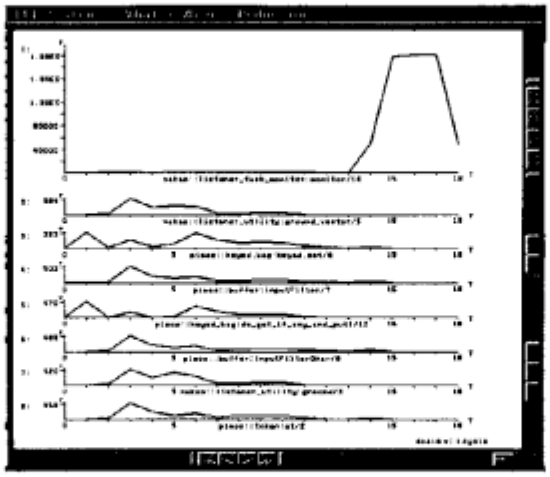

Figure 5: Example of detecting an infinite loop by using ParaGraph

To further improve the environment, development of the following features is planned.

Static Analysis Tools: Currently only a quite simple static program analysis tool is provided. More sophisticated tools, such as one analyzing input-output relations among clauses are planned[9].

Higher-Level Language: A higher-level language upon KL1, AYA[10], is under development. AYA supports object-oriented programming function as its language feature. This makes the description much more concise preventing bugs such as misspelling of variables or mistake the order of arguments. Programs written in AYA are compiled into KL1 and executed.

Portable Implementation of KL1: A language processor that compiles KL1 programs into C is planned for portability of KL1 programs[11].

References

[1] A. Goto, et al. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208–229, ICOT, 1988.

[2] K. Taki. Parallel Inference Machine PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pp.50–72, ICOT, 1992.

[3] T. Chikayama, et al. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230–251, ICOT, 1988.

[4] T. Chikayama. Operating System PIMOS and Kernel Language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pp.73–88, ICOT, 1992.

[5] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine, In *Computer Journal*, Dec. 1990.

[6] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[7] S. Aikawa, et al. ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pp.286–293, ICOT, 1992.

[8] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *Proceedings of 7th International Conference on Logic Programming*, 1990.

[9] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *Proceedings of 7th International Conference on Logic Programming*, 1990.

[10] K. Susaki and T. Chikayama. A Process-Oriented Language AYA upon KL1. Technical Report TR-652, ICOT, 1991. (In Japanese)

[11] T. Chikayama. A Portable and Reasonably Efficient Implementation of KL1. Technical Report TR-747, ICOT, 1992.