TR-0844

ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems

by
S. Aikawa (Fujitsu), M. Kamiko (Fujitsu)
& T. Chikayama

May, 1993

© 1993, ICOT

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

ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems

Abstract

Distributing computational load to many processors is a critical issue for efficient program execution on multiprocessor systems. Finding a good load distribution algorithm is one of the most important research topics for parallel processing. Tools for evaluating load distribution algorithms are very useful for this kind of research. This paper describes a system called ParaGraph that gathers periodical statistics of the computational and communication load of each processor during program execution, in both the higher level of programming language and lower level of implementation, and presents them graphically to the user.

1. Introduction

In the Japanese Fifth Generation Computer Systems Project, parallel inference systems have been developed for promoting parallel software research and development. The system adopts a concurrent logic programming language KL11) as the kernel and consists of a parallel inference machine, PIM2) and its operating system, PIMOS3).

For efficient program execution, the computational load must be appropriately distributed to each processor. On scalable loosely-coupled multiprocessor systems, load balancing and minimization of communication overhead are essential, but become more difficult compared to tightly-coupled systems as communication costs increase. Although many load distribution algorithms have been developed4), 5), none have been sufficient to execute every program effectively. Finding a good load distribution algorithm is one of the most important research topics for parallel processing.

Tools for evaluating load distribution algorithms are very useful for this kind of research. The objective of the ParaGraph system is to help programmers design and evaluate load distribution algorithms on loosely-coupled multiprocessor systems. ParaGraph gathers profiling information during program execution on the parallel inference machine, PIM, and displays it graphically based on the X window system6).

Many performance displays have been devised for utilization, communication, and task information7)-8). For example, graphical meters7) represent processor-utilization and graphical animation on a processor configuration map8) represents interprocessor-communication of message-passing programs. Such specialized views provide an intuitive feeling for dynamic behavior, but it is difficult to determine where the performance bottlenecks are. Because the excution of parallel programs often raise complex phenomena, simple observation of each phenomena can not provide full information needed to detect performance bottlenecks. For example, suppose that when tasks are not mutually independent and must communicate with each other closely. The program is less efficient because of communication overhead. But graphical meters may show processors work hard, although most of processing time must have been consumed on message-handling. In this case, it is useful to compare the activity of processors with frequencies of sending and receiving messages along execution time. Thus, bottlenecks are often determined by comparing with some pieces of profiling information each other. In ParaGraph system, every kind of profiling information can be displayed based on three common axes to be easy to compare. Because such profiling information can be viewed as having three axes: what, when, and where.

In Section 2, how load distribution can be described in KL1 on PIM are described. Section 3 describes the implementation of the ParaGraph system and graphical representation of program execution, and Section 4 discusses how useful graphical displays are to detect performance bottlenecks with examples of various programs.

The contents of this paper partially overlap the subject of a previous paper9).

2. Load Distribution Algorithms

2.1 Load distribution in KL1

The parallel inference machine runs a concurrent logic programming language called KL11), 3), 10). A KL1 program consists of a collection of guarded Horn clauses of the form:

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

where H, G_i, and B_i are atomic formulas. H is called the head, G_i, the guard goals, and B_i the body goals. The guard part consists of the head and the guard goals and the body consists of body goals. They are separated by the commitment operator "|". 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. This is denoted as P/N.

The guard goals wait for instantiations to variables (synchronization) and test them. When the guard part of one or more clauses succeed, one of those clauses is selected and its body goals are called. These body goals communicate with each other through their common variables. If variables are not ready for testing in the guard part because the value has not been computed yet, testing is suspended.

In addition to the above basic mechanism, there is a mapping facility which includes load distribution specification. The programmer can annotate the program by attaching pragmas to the body goals to specify a processor (specified by Goal@node(Proc)). The programmer must tell the KL1 implementation which goals to execute on which processors.

```
next_queen(N,I,J,B,R,D,BL):- J>0, D=0 |
  BL = [BL0,BL1],
  R = [R0,R1],
  BL0 = [get(Proc)|BL2],     processor specification
  try_ext(N,I,J,B,R0,D,BL2)@node(Proc),
  next_queen(N,I,J-1,B,R1,D,BL1).
```

Fig. 1 A sample KL1 program.

Figure 1 shows a part of a KL1 program. If the goal next_queen/7 is committed to this clause, its body goals are called. The goal try_ext/7 has a processor specification, and it is to be executed on processor number "Proc". This processor number can be dynamically computed.

2.2 Design Issues

Load balancing derives maximum performance by efficiently utilizing the processing power of the entire system. This is done by partitioning a program into mutually independent or almost independent tasks, and distributing tasks to processors. Many load balancing studies have been devised, but they are tightly

coupled to particular applications. Therefore, programmers have to build load distribution algorithms for their own applications.

To distribute the computational load efficiently, the programmer should keep in mind the following points. Since load distribution is implemented by using goals, the programmer should understand the execution behavior of each goal. When goals are executed on a loosely-coupled multiprocessor, the programmer should investigate the load on individual processors and the communication overhead between processors.

For evaluating load distribution algorithms, tools must provide many graphic displays for the programmer to understand the computational and communication load of each processor in both the higher program and lower implementation levels. No single display and no single profiling level can provide the full information needed to detect performance bottlenecks.

3. System Overview

3.1 Gathering Information

To statistically profile large-scale program execution, KL1 implementation provides information gathering facilities, low-level profiling and higher-level profiling. KL1 implementation provides these facilities as language primitives, to minimize the undesirable influence to the execution behavior of programs. These facilities have been implemented at the firmware level. The profiling facilities are summarized as follows.

1) Low-level profiling

Profiles the low-level behavior of the processor, such as how much CPU time went to the various basic operations required for program execution.

2) Higher-level profiling

Profiles the higher-level behavior of the processor, such as how many times each piece of the program was executed.

To minimize the perturbation, the gathered profiling information resides in each processor's local memory during program execution, and after execution, ParaGraph collects this information and converts into some standard form.

Since profiling information is automatically produced by the KL1 implementation, programmers do not have to modify the application programs.

3.1.1 Low-level Profiling

The basic low-level activities can be categorized into computation, communication, garbage collection, and idling. Computation means normal program execution such as goals' reductions and suspensions, communication means sending and receiving inter-processor messages, garbage collection means itself, and finally, idling means doing nothing.

The processor profiling facility measures how much time went to each category for each processor. Such information can be periodically gathered to show gradual changes of behavior. The profiling facility can also measure frequencies of sending and receiving various kinds of interprocessor messages11)-12).

- A *throw_goal* message transfers a KL1 goal with a throw goal pragma to a specified processor.
- A *read* message requests for some value from the remote processor when a clause selection condition

requires it.

- An *answer_value* message replies to a read message when the request value becomes available.
- A *unify* message requests body unification (giving a value to a variable).

3.1.2 Higher-level Profiling

KL1 provides a mechanism for grouping goals and controlling their execution in a meta-level. This mechanism can be considered to be an interpreter for the KL1 language. It also provides profiling facility at a higher level than processor profiling. Low-level profiling gathers a number of important statistics from many aspects that help analyzing performance bottlenecks, but it provides no information on where in the program is the root of such a behavior.

To correlate execution behavior with a portion of the program, higher-level profiling measures how many times goals associated with each predicate are reduced or suspended (due to unavailability of data required for reduction). Transition of behavior can be observed by periodically gathering the information.

3.2 Graphic Displays

The profiling information can be viewed as having three axes: what, when, and where. In sequential execution, "where" is a constant and the "when" aspect is not important, since the execution order is strictly designated. Therefore, simple tools like gprof provided with UNIX[Note]) suffice. However, all three axes are important when parallel execution is concerned.

If such massive information is not presented carefully, the user might be more confused than informed. Therefore, ParaGraph provides graphic displays based on three axes. We named each representation using the terms "What," "When," and "Where." The term "What" is the visualization target corresponding to the type of profiling information such as low-level processor behavior, higher-level processor behavior, and interprocessor message frequencies. The term "When" and "Where" indicate time expressed by a cycle number and the processor number respectively.

Figure 2 shows the graphic displays of ParaGraph. These displays are execution behavior of all solution search program of N queen problem.

Every type of profiling information can be easily displayed with the views described below with a menu-oriented user interface such as the bottom-right window in Figure 2. If the window size is too small to display everything in detail, coarser display aggregating several cycles or several processors together is possible to see the overall behavior at a glance. Scrolling on the vertical and horizontal directions are also possible if details are to be examined. It is also possible to display only selected "What" items.

Note: UNIX is a trademark of AT&T Bell Laboratories.

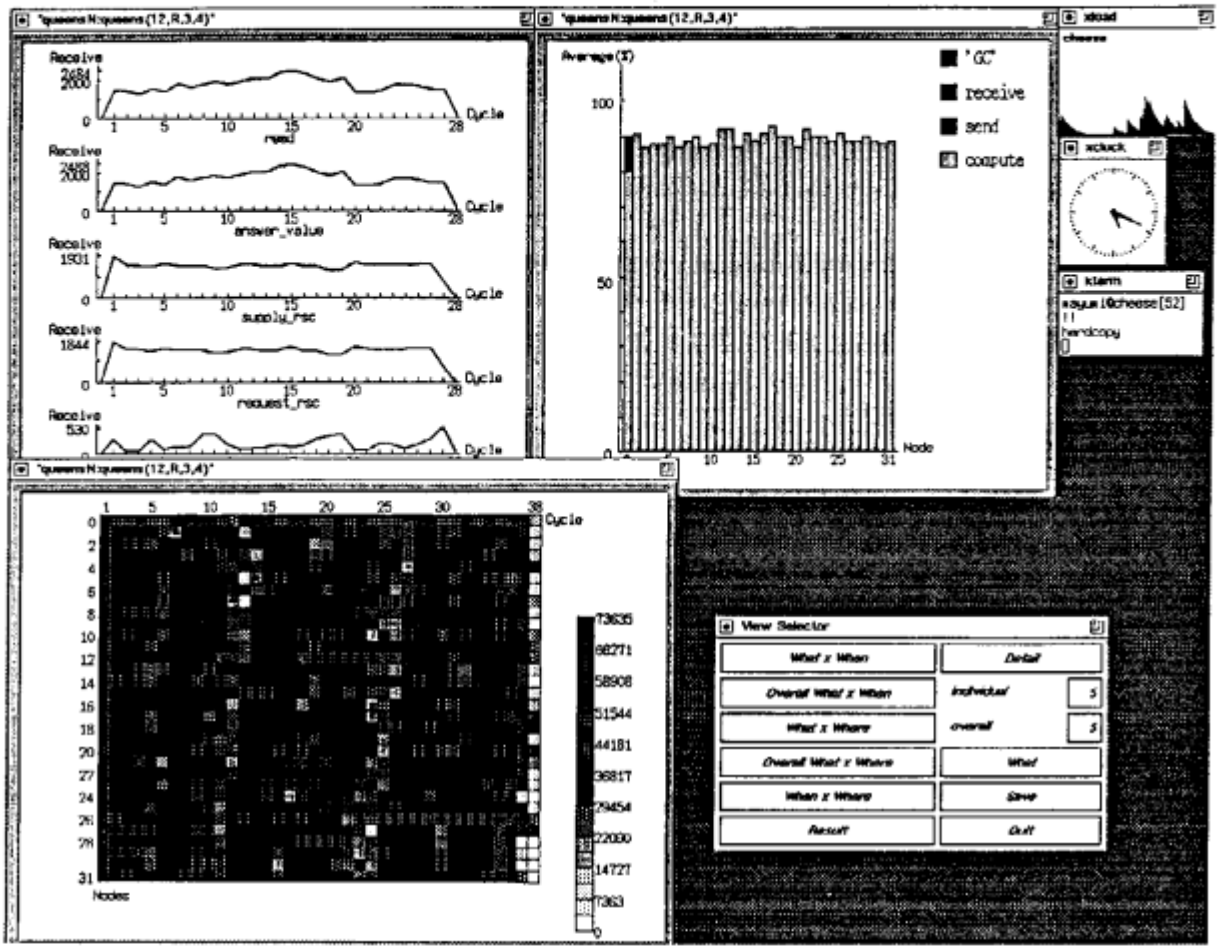

Fig.2-Examples of graphic displays: a What×When view (top-left), an overall What×Where view (top-right), and a When×Where view (bottom-left) and a menu-oriented user interface (bottom-left).

3.2.1 A What×When View

There are two kinds of views in terms of "What" and "When" items. One is a What×When view which shows the behavior of each "What" item during execution. A graph is displayed of a "What" item in order of the total volume. The x axis is the cycle numbers, and the y axis is the rate of processor utilization, the number of messages, and the number of reductions or suspensions corresponding to the type of profiling information. Since every graph is drawn with the same scale on the vertical axis, it is easy to compare with "What" items.

The other is an overall What×When view which shows the behavior of all "What" items during execution. Each "What" item is stacked in the same graph and displayed by a line. The y axis represents the average rate of processor utilization, the total number of messages, and the total number of reductions and suspensions corresponding to the type of profiling information.

These views are helpful for example, if a program has sequential bottlenecks such as tight synchronization. In this case, the number of goal reductions will be down at some portion during program execution. Such a problem will be detected easily by observing program execution.

The top-left window in Figure 2 shows received message frequencies on all processors with What×When view. In this window, four kinds of receiving message frequencies are displayed on each graph. These messages are displayed in order of the total number of received messages. The other messages are displayed by scrolling vertically.

From this, we know that each received message frequency on all processors is less than 2,500 times/an interval (an interval is 2 second). As this program is divided mutually independent subtasks, communication message frequency is very low.

3.2.2 A When×Where View

A When×Where view shows the behaviors of all "What" items on each processor. Each processor is displayed with various color patterns that indicate volume. The relationship between color patterns and volume are shown in the bottom right corner. The darker the pattern, the busier the processor. Volume means the rate of processor utilization, the number of messages, and the number of reductions or suspensions that correspond to the type of profiling information. It's also possible to display only selected "What" items instead of all of them.

The bottom-left window in Figure 2 is a When×Where view. The x axis is the cycle number, and the y axis is the processor number. This view displays the execution behavior of all goals on a 32-processor machine. The color patterns indicate the number of reductions. The relationship between the number of reductions and color pattern is displayed on the bottom right corner.

From this, we know that the work load on each processor was well balanced, and this program was executed about 70,000 reductions/an interval on each processor at each moment in time.

3.2.3 A What×Where View

There are two kinds of views in terms of "What" and "Where" items. One is a What×Where view which shows the load balance of each "What" item on each processor. A bar chart is displayed of a "What" item in order of total volume. The x axis represents the processor numbers, the y axis represents the rate of processor utilization, the number of messages, and the number of reductions or suspensions that correspond to the type of the profiling information. All bar charts are drawn with the same scale on the vertical axis, so it is easy to compare with the volume of each "What" item.

The other is an overall What×Where view which shows the load balances of all "What" items on each processor. Each "What" item is stacked in the same bar chart and displayed by a certain color pattern. The y axis represents the average rate of processor utilization, the total number of messages, and the number of total reductions or suspensions that correspond to the type of profiling information. The relationship between each category and color pattern is displayed on the top-right corner.

The top-right window in Figure 2 shows the low-level behavior of the processor with an overall What×Where view. In this window, each categories of low-level behavior is displayed with several color pattern.

From this, the average of computation took more than 80% of total execution time, and the average of communication on processor No. 0 was about 10%, and the others were less than 5%. Since processor No.

0 collected answer values from the others, it took higher average. Thus, this view shows most of the processors run fully, and this example program was executed very efficiently on each processor.

4. Examples

This section discusses which views to use to view various performance bottlenecks. For efficient program execution on multiprocessor systems, the following phases are usually repeated until a solution is reached: a) a program is partitioned into subtasks, b) the subtask is mapped to each processor dynamically, and c) each processor runs subtasks while communicating with each other.

Various problems are often encountered when executing a program on multiprocessor systems. We will show how graphic displays in both the higher program and lower implementation levels are helpful with performance problems.

4.1 Uneven Partitioning

When the granularity between subtasks is very different, it is useful to observe the low-level processor behavior with a When×Where view and the higher-level processor behavior with a What×Where view. From the When×Where view, we will find which processors run fully and which are idle. From the What ×here view, we will determine which goals caused the load imbalances.

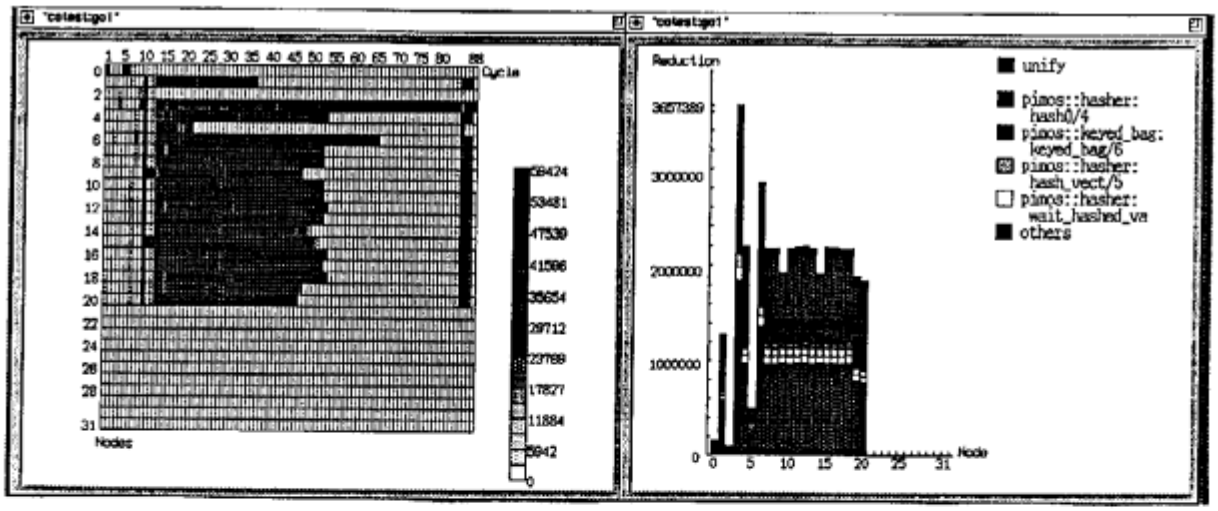

Fig.3-The low-level processor behavior (left) and execution behavior of goals (right).

The left window in Figure 3 shows the low-level behaviors on each processor with a When×Where view, while the right window in Figure 3 shows the higher-level behaviors of the same processors with a What×Where view on a 21-processor machine. An example program is a logic design expert system which generates a circuit based on a behavior specification. The strategy of parallel execution is that first, the system divides a behavior specification into sub-specifications, next designs subcircuits based on the sub-specifications on each processor, and finally gathers partial results together and combines them.

The When×Where view suggests that most of processors run almost equally, but processors No.3 and

No. 6 run fully, and processors No 0, No. 2,and No. 5 were idle. The What × Where indicates the which goals were executed on each processor.

From this, we know that processors No. 3 and No. 6 were allocated very complicated tasks, and processors No.0, No.2, and No.5 were allocated very tiny tasks, that is, uneven partitioning of behavior specification must cause a bottleneck in performance.

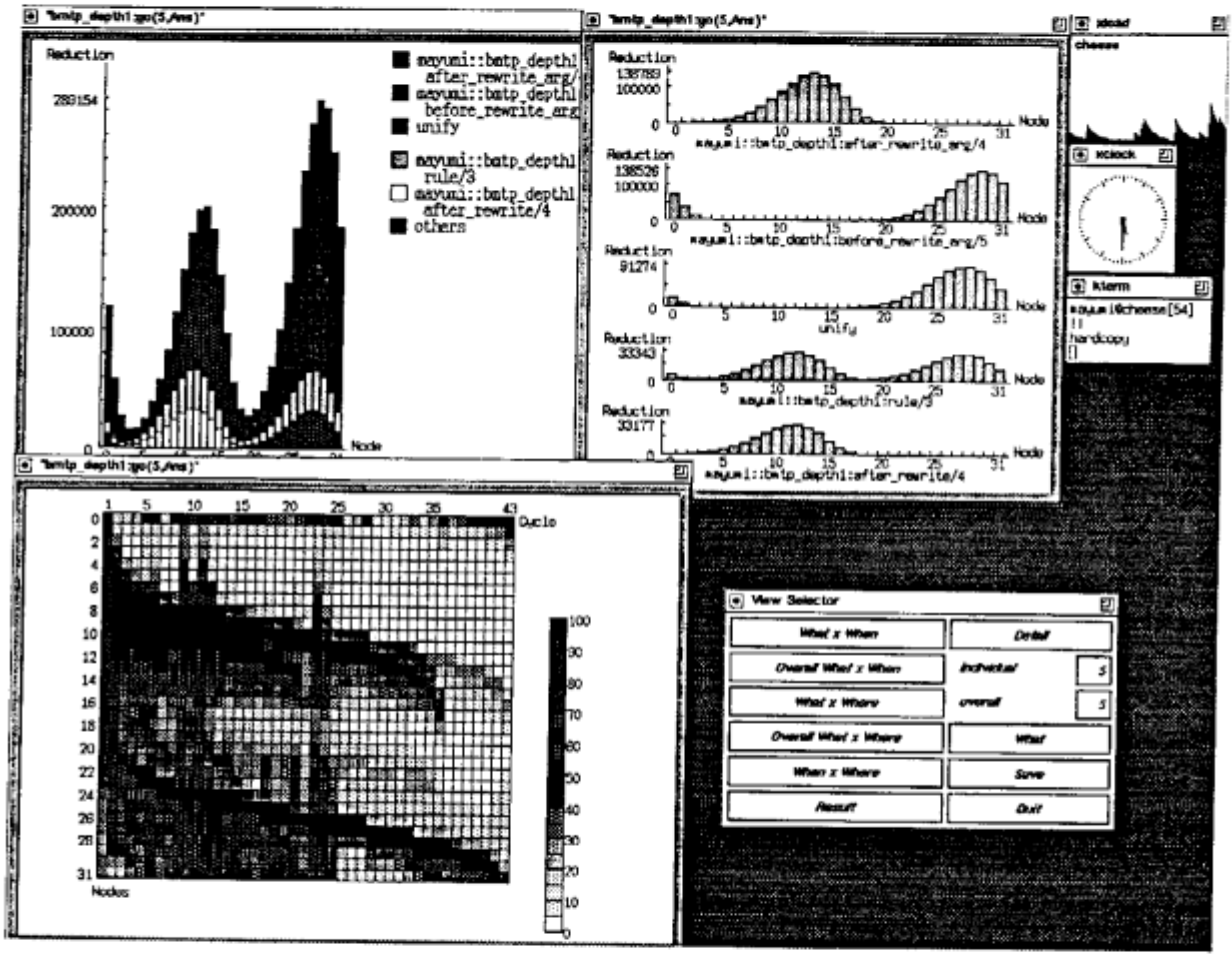

Fig.4-Low-level processor behavior (bottom-left), the load balances of all goals (top-left), and the load of each goal (top-right).

4.2 Load Imbalance

If a mapping algorithm has problems such as allocating subtasks to the same processor, it is useful to observe low-level behavior of the processor with a When × Where view and higher-level behavior with an What × Where view. From the When × Where view, we see which processors run fully or which are idle, and from the What × Where view, we see the load balance of each goal. Using both views, we can determine how to distribute the goals that are imbalanced to each processor.

The bottom-left window of Figure 4 shows low-level behavior of the processor with a When×Where view, the top-left window and the top-right window show the higher-level behavior of the processor with an overall What×Where view, a What×Where view respectively.

An example program is a part of the theorem prover which evaluates whether an input formula is a tautology. The strategy consists of 2 steps: 1) convert an input formula to clause form (i.e, conjunctive normal form), 2) evaluate its clause form and determine whether it is a tautology.

The step 1 is executed in parallel as follows. First, main task partitions an input formula into subformulas. Second, it generates subtasks to convert subclause forms, and finally, distributes subtasks to many processors dynamically. These steps are repeated recursively until subformulas are converted to subclause forms. The step 2 is executed in sequential on processor No.0.

The When×Where view of the bottom-left window suggests that only certain processors (processor No.6-15 and No.23-31) run fully and that the others were mostly idle. The overall When×Where view of the top-left window also suggests most of the goals were executed on the same processors, especially the number of reductions of top five goals were higher than the other goals.

We can check the load of each goal on each processor from the What×Where view of the top-right window. These goals were executed on certain processors and were the cause of the load imbalances. From this, we have to change its mapping algorithm to be flatten the shape, to use all processors efficiently.

4.3 Large Communication Overhead

When subtasks are not mutually independent and must communicate with each other closely, the program is less efficient because of communication overhead. In this case, the low-level behavior of the processor with an overall What×Where view and frequencies of sending and receiving messages with a What×Where view are helpful. From the overall What×Where view, we will learn how much time has been consumed on message handling for each processor, while the What×Where view shows us what kind of messages each processor has sent or received.

Figure 5 displays an execution behavior of an improved version of the program described in section 4.2. The left window shows the load balances of all goals on a 32-processor machine with an overall What×When view. This view shows that the work load on each processor was balanced in overall execution, but was not efficient because of large communication overhead. It will be proved from low-level behavior of the processor with an overall What×Where view shown in the right window.

Figure 6 shows the same program execution as Figure 5. The left window shows the receiving and sending message handling time rate with What×Where view, the right window shows the frequencies of four received inter-processor messages with a What×When view.

The right window of figure 5 suggests the load average on each processor was about 80 - 85%, but the average of computation on each processor was about 20%. Most of the processing power was consumed sending and receiving message handling time more than 60% of total execution time.

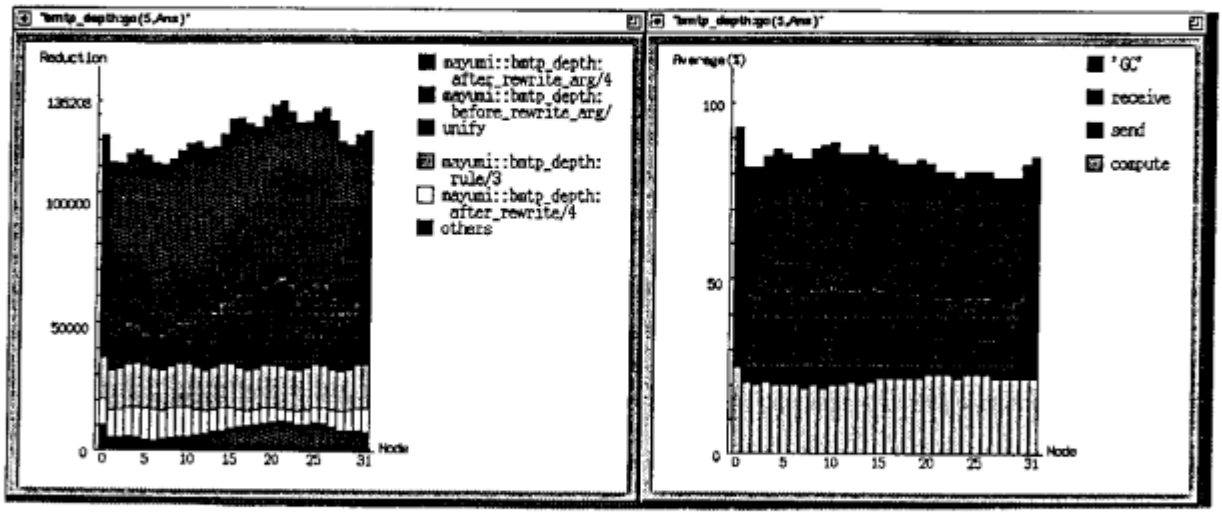

Fig.5-The load balances of goals (left) and low-level processor behavior (right).

The left window of figure 6 shows the message handling time on each processor at each moment in time was almost equally, right window in figure 6 shows that the read message was received about 180,000 times, answer_value message was about 170,000 times, unify message was 100,000 times, and throw_goal message was about 66,000 times per interval on all processors. The tasks generated in this program communicated with each other closely among processors as compared with the result of N queen's message frequencies (see the top-left window of figure 2).

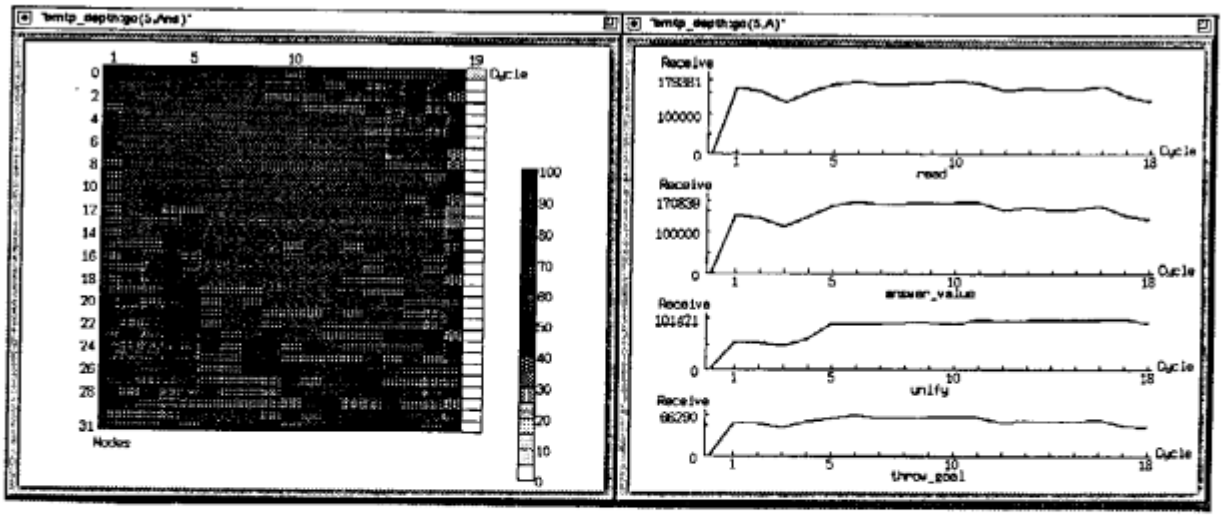

Fig.6-Low-level processor behavior about message handling (left) and message frequencies (right).

From this, we know that as work loads are distributed more and more, it becomes easier to balance work loads on each processor, but communication overhead also increases and performance is thus lowered. As a

result, we have to redesign or improve how to divide into subtasks. Because the generated subtasks that were not mutually independent caused such a problem we mentioned above.

5. Conclusion

We developed the ParaGraph system on parallel inference machines to provide graphic displays of processor utilization, interprocessor communication, and execution behavior of parallel programs. Experiments with various programs have indicated that graphic displays are helpful in dividing work loads evenly and determining where the bottlenecks are on multiprocessor systems.

We released a version last year as a tuning tool of PIMOS, but have experienced some problems. In the future, we will improve the system considering the following points.

First, real-time performance visualization tools are needed. Although displaying execution behavior in real-time perturbs the program being monitored, it is useful not only in early tuning but also in debugging such as detecting deadlock status and infinite loops. To develop such a tool, low overhead instrumentation techniques and new displays that are easy to understand for programmers appearing in real-time must be devised.

Second, tools which can visualize the portion of the performance bottlenecks directly are needed. Massively parallel machines that have thousands of processors and programs for long runs produce a large amount of profiling information, but it is difficult to process or display for simple expansion of our system because of a vast quantity of information. To solve such problems, analysis techniques indicating bottlenecks directly will be needed. We will study automatic analysis techniques and graphical displays of its result (we call this *bottleneck visualization*). One such approach is critical path analysis13), which identifies the path through the program that consumed the most time.

Acknowledgments

The work described in this paper was done under ICOT (Institute for New Generation Computer Technology) contract as a part of the R&D of the Fifth Generation Computer Systems Project.We thank all researchers of ICOT and other companies who tested our tool. We also thank K. Nakao and H. Kubo who helped us to develop this tool.

References

1) Ueda, K., and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine. The Computer Journal, 33, 6, pp.494-500 (December 1990).

2) Goto, A., Sato,M., Nakajima,K., Taki, K., and Matsumoto, A.:Overview of the Parallel Inference Machine (PIM) Architecture. Proc. Fifth Generation Computer Systems 1988, 1, Tokyo, pp. 208-229.

3) Chikayama, T., Sato, H., and Miyazaki, T.:Overview of the Parallel Inference Machine Operating System (PIMOS). Proc. Fifth Generation Computer Systems 1988, 1, Tokyo, pp. 230-251.

4) Furuichi, M., Taki, K., and Ichiyoshi, N.:"A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Program on the Multi-PSI". ICOT TR-526, Tokyo, ICOT Research Center, December 1989.

5) Kimura, K., and Ichiyoshi, N.:Probabilistic Analysis of the Optimal Efficiency of the Multi-Level Dynamic Load Balancing Scheme.Proc. Sixth Distributed Memory Computing Conference, 1989.
6) Scheifler, R. W., and Gettys, J.:The X Window system, ACM Trans. on Graphics, 5, 2, pp. 79-109 (1986).
7) Malony, A. D., Reed, D.A., and Rudolph, D.C.:"Integrating Performance Data Collection, Analysis, and Visualization". Performance Instrumentation and Visualization, 1st ed., N.Y., ACM Press, 1990, pp. 73-97.
8) Heath, M. T., and Etheridge, J.A.:Visualizing the Performance of Parallel Programs. IEEE Software, pp. 29-39, September 1991.
9) Aikawa, S., Kamiko, M., Kubo, H., Matsuzawa, F., and Chikayama, T.:ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. Proc. Fifth Generation Computer Systems 1992, 1, Tokyo, pp. 286-293.
10) Ichiyoshi, N.:"Research Issues in Parallel Knowledge Information Processing". ICOT TM-0822, Tokyo, ICOT Research Center, November 1989.
11) Nakajima, K., Inamura, Y., N. Ichiyoshi, Chikayama, T., and Nakashima, H.:Distributed Implementation of KL1 on the Multi-PSI/V2". Proc. Sixth International Conference on Logic Programming, 1989.
12) Nakajima, K., and Ichiyoshi, N.:"Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI". ICOT TR-531, Tokyo, ICOT Research Center, February 1990.
13) Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S., and Torzewski, T.:IPS-2: The Second Generation of a Parallel Program Measurement System. IEEE Trans. Parallel and Distributed Systems, 1, 2, pp. 206-217 (April 1990).