Studies on the Parallel Inference Engine PIE64

Kentaro SHIMADA*, Yastuo HIDAKA*, Jun'ichi TATEMURA*, Hanpei KOIKE** and Hidehiko TANAKA***

SYNOPSIS

PIE64 is a parallel inference machine dedicated to the committed choice language Fleng. PIE64 has sixty-four processing elements, which are called Inference Units (IUs), and two independent interconnection networks which have an automatic load balancing facility. Each IU has an inference processor named UNIREDI, two network interface processors, and a management processor. In this paper, we describe the system of PIE64 including the Fleng programming environment and the architecture of the Inference Unit. The Fleng programming environment involves the multiwindow debugger system named HyperDEBU. We also discuss the parallel management kernel for PIE64.

1 Introduction

Committed choice languages, such as Fleng and GHC, are parallel logic programming languages which have a semantics suitable for parallel execution, and can be easily implemented on parallel machines. In addition, they provide a much programming power owing to the logic based semantics. In order to construct both a parallel machine of high performance and an effective programming environment, we adopted Fleng and have developed the parallel inference machine PIE64 and the multiwindow debugger HyperDEBU.

This paper is organized as follows. Section 2 briefly explains the committed choice language Fleng, and section 3 explains the general architecture of PIE64. Section 4 describes the inference processor UNIREDI, and Section 5 describes the parallel management kernel for PIE64. And then section 6 presents the multiwindow debugger HyperDEBU. Section 7 concludes this paper.

2 Fleng

A Fleng program is a set of declarations of horn clauses like:

\[ \text{Head} := \text{Goal}_1, \text{Goal}_2, \ldots, \text{Goal}_n \]

The left side of := is called the head, and the right side is the body.

* Graduate Student, Fac. of Eng., University of Tokyo
** Lecturer, Fac. of Eng., University of Tokyo
*** Professor, Fac. of Eng., University of Tokyo

Execution begins when top query goals are given. For each goal, one of the clauses whose head is unifiable with the goal is selected, and then the goal is reduced to the body part of the clause. Execution is the repetition of this process and is the unit of parallel processing.

A variable in a goal is a single assignment variable, and the binding to an undefined variable is always done in the body part. If heid unification specifies a particular value of an undefined variable, execution of the goal will be suspended. When the value is determined later by another goal, the suspended unification is resumed.

3 Parallel Inference Engine PIE64

The most fundamental issues in parallel processing are said to be communication latency and synchronization cost. Besides these issues, parallel management, e.g. load distribution and scheduling, also become important in highly parallel processing. Although parallel management does not cause so much overhead in low-parallel processing, it causes inevitable overhead in high-parallel processing, because the granularity must be fine in order to get sufficient parallelism.

In order to absorb such overhead effectively, we have adopted a composite architecture for the processing element of PIE64. This composite architecture consists of three kinds of processors which bear responsibility for "computation", "communication and synchronization" and "management". PIE64 consists of such 64 processing elements and two interconnection networks. A processing element of PIE64 is called an
3) multi-context processing, 4) a dedicated instruction set to execution of Fleng programs.

4.2 Multi-context processing

The internal pipeline of UNIREDDII is shared multiple instruction streams (contexts). UNIREDDII executes them concurrently, and which contexts should be executed is determined cycle by cycle. In other words, UNIREDDII acts as a pipeline-shared MIMD processor. UNIREDDII executes four contexts concurrently.

The aims of the multi-context processing are twofold. 1) To reduce pipeline interlocking caused by overlapping executions of instructions (intra-processor effect). 2) To reduce the cost of process switching due to remote memory access (inter-processor effect). The second effect is an especially important feature for a processing element of parallel machines.

One difference between the multi-context processing mechanism of UNIREDDII and other processors which adopt cycle-by-cycle multi-threading mechanism is that UNIREDDII can execute even one context continuously at every cycle when it can not execute the other contexts, while other processors can only execute different contexts in continuous cycles. Therefore they suffer from degradation of the instruction issue rate as the number of executable contexts decrease. UNIREDDII has a pipeline interlocking mechanism to enable continuous execution of even one context to keep fairly high performance under that condition.

4.3 Instruction Set

The instruction set of UNIREDDII is specialized to execute Fleng. In addition, its all instructions are executed at a single cycle to enable cycle-by-cycle context switching. The most characteristic instruction of UNIREDDII is the dereference instruction. It dereferences links of a variable and gets the value of the variable. In a conventional implementation of such an instruction, it may require multiple cycles to complete its operation because links of a variable can be longer than one.

In the implementation of UNIREDDII's dereference instruction, it jumps to itself so that it will be re-executed later to complete its remained acts when all acts of it can not be done at a single cycle. Meanwhile instructions of the other contexts are executed to compensate for delayed slots caused by that jumping.
Table 1: the execution time of the sample programs for UNIREDD and SPARC

<table>
<thead>
<tr>
<th>programs</th>
<th>execution time [μSec]</th>
<th>UNIREDD</th>
<th>SS-ELC*</th>
<th>SS-ELC/UNIREDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>append 100</td>
<td>0.193</td>
<td>0.160</td>
<td>1.20</td>
<td>0.829</td>
</tr>
<tr>
<td>reverse 30</td>
<td>0.539</td>
<td>0.810</td>
<td>1.50</td>
<td>1.47</td>
</tr>
<tr>
<td>sort 50</td>
<td>0.819</td>
<td>1.20</td>
<td>1.58</td>
<td></td>
</tr>
<tr>
<td>8 queens</td>
<td>70.4</td>
<td>111</td>
<td>1.30</td>
<td></td>
</tr>
</tbody>
</table>

*Sparesation ELC, 33MHz clock with 64KByte cache

By that scheme, we implemented all instructions as single cycle ones in spite of being dedicated to execute Fleng.

4.4 Simulation Results

As for the performance evaluation, we have tested Fleng compiled code for UNIREDD by the register-transfer level simulation and compared with the code for SPARC on a workstation which uses one SPARC processor (Sparesation ELC, 33MHz clock with 64KByte cache). Table 1 shows the execution time of the sample programs for both processors. In that table, UNIREDD is about fifty percent faster than the SPARC processor except in the append 100 program in which UNIREDD can not make use of the multicontext processing mechanism to reduce the pipeline interlocking. The overall advantage of UNIREDD is about thirty percent and, if we normalize the device technology by the clock rate (10MHz for UNIREDD and 33MHz for the SPARC processor), we can say that UNIREDD is about four times faster than the SPARC processor.

5 Architecture of Parallel Management Kernel for PIE64

In this section, we describe the management processor's parallel management kernel. The parallel management kernel is a kind of operating system kernel, and performs low level system management. In particular, we attach much importance to “parallel management”, which is the special management necessary for parallel processing, as shown in Fig. 2. Higher level functions of operating systems, e.g. programming environment, are not included in the functions of the kernel.

5.1 Parallel Management

In order to distinguish parallel management clearly from ordinary services like resource management, we define it as management which is needed for each thread in parallel processing.

As the granularity of parallel processing becomes finer, processing cost for parallel management grows rapidly. While ordinary management aims to offer some service in response to explicit requests, the purpose of parallel management is not to offer service in response to a request, but to reduce execution time of a program. Therefore, even if there is no explicit request, parallel management must support efficient parallel execution from behind.

Among several kinds of parallel management, load distribution and scheduling are most important, because they have great effects on execution efficiency.

Load distribution is to determine partitioning of a program into threads, and to determine assignment of threads to processing elements (PEs). The requirements for it are: (1) extraction of parallelism, (2) load balancing, and (3) reduction of communication.

Scheduling is to determine execution order of threads assigned to the PE. The requirements for it are: (1) reduction of synchronization cost, (2) avoidance of heap memory exhaustion, (3) Parallelism extraction, and (4) priority designation by a user.

5.2 Parallel Management Kernel

The parallel management kernel is executed by MP on each HI, and it performs low-level system management. Its organization is shown in Fig. 3. The kernel utilizes heap consumption information obtained from the heap memory management and load information of other HIs from NIP, so as to bring the best load distribution and scheduling.

In PIE64, load distribution is handled at three stages, namely, static partitioning by the compiler, dynamic partitioning by
the parallel management kernel), and dynamic assignment by the interconnection network. Among them, the dynamic partitioning is performed according to information only obtained at the run time, e.g. parallelism. All of the requirements for load distribution can be successfully attained with this three-stage strategy.

The scheduling strategy of the kernel introduces dynamic priority and respite time in starting execution of a thread. Dynamic priority is based on the parallelism at run time, and the remaining quantity of the heap memory. Its purpose is to release a programmer from being worried about resource exhaustion caused by explosive parallelism, and to increase parallelism quickly when it is insufficient. Respite time was introduced to utilize data dependency which is partly detectable in the compiler, for reducing cost of suspension and context switching.

5.3 Preliminary Evaluation Result
We have had preliminary results of a comparison between static and dynamic partitioning. If parallelism greatly exceeds the number of PEs, simple dynamic partitioning with little overhead is more effective than sophisticated static partitioning. However, if parallelism and the number of PEs are of comparable size, dynamic partitioning becomes dramatically ineffective. We concluded that the most promising method is a composite method which provides the merits of both static and dynamic partitioning.

6 Multiwindow Debugger
HyperDEBU

6.1 Design of Debugger for Fine-grained Highly Parallel Programs
A sequential program has only one thread of execution, which can be debugged with a sequential interface. On the other hand, a parallel program has multiple complicated control/data flows which are considered to be high-dimensional information. If a sequential interface is used to debug a parallel program, the bottleneck between a programmer and the program makes it difficult to examine and to manipulate the execution of the program. Therefore, a high-dimensional interface is necessary to debug a parallel program.

Since a user compares a model represented by a debugger
with the expected behavior of the program when he/she debugs a program, the debugger must provide a view of the kind he/she wants. Accordingly, the debugger must provide views which have flexible levels and aspects of abstraction.

We developed a multiwindow debugger HyperDEBU which provides windows flexible enough for programmers to examine and manipulate complicated structure composed of multiple control/data flows. Tracing links on a window which displays information of a program execution, a user can get an expected window.

HyperDEBU consists of the following windows: (1) toplevel-window, (2) process-windows (a) TREE view (b) I/O tree view (c) GOAL view, (3) structure-windows. Figure 4 shows an overview of HyperDEBU.

6.2 Features of HyperDEBU

6.2.1 Various Views for Bug Locating

Toplevel-window and Process-windows For the needed flexibility of levels of views from global to local, HyperDEBU provides a toplevel-window and process-windows.

A toplevel-window, which provides a global view, is opened initially, and a user provides initial goals into it. The user can examine and manipulate execution of a program in the global scope with this window. Moreover, the user can get a process-window to examine detail of any process displayed on the toplevel-window.

A process-window, which can be opened for any process, enables examination and manipulation of the process. To locate bugs, a user can get a subprocess as another window from this process.

Moreover, HyperDEBU has a structure-window which provides a data-level view. A user can get it from any data displayed on windows of HyperDEBU. Since all of the data on HyperDEBU are updated as the program runs, a user can examine state of the program dynamically.

Three Views of Process-window A process-window enables bugs to be located efficiently using the flexible levels and aspects of abstraction of the process model. The process-window has three windows as views of the process model: TREE view, I/O tree view, and GOAL view. They show an execution tree (control flow relationship of the computation), a tree of input-output causality (data flow relationship of the computation), and a set of goals (snapshot of the computation), respectively. They enable flexible examination from multiple aspects. A user can get a subprocess as a window from their windows.

6.2.2 Program Visualization

The global view of the toplevel-window visualizes execution of a Fleng program. Execution of a Fleng program is represented by visualizing the two flows: control flow (goal reduction) and data flow (guard and unification). Their histories are displayed on TREE view, and I/O tree view. However, visualizing all goals and data is hard to comprehend. The toplevel-window visualizes only processes with respect to some particular goals and only some particular data-flows. A user needs to specify what is "particular", depending on the user's intention. HyperDEBU provides "breakpoints" to specify it.

Control-flow The toplevel-window visualizes processes with respect to some particular goals to provide a global view. Figure 5 shows an overview of the toplevel-window. Each process is displayed as a rectangle. A color of the rectangle indicates the state of the process. A nest of rectangles indicates the relation between a process and its subprocess. Clicking a rectangle generates a new process-window for this process.

By observing creations and state transition of processes, and by observing modification of data in the arguments of topgoals, a user can comprehend the execution of Fleng program correctly and easily.
6.2.3 Breakpoints for Parallel Execution

We extend “breakpoints” as a debugger’s knowledge given by a user before the execution of the program. The debugger uses this information to control the execution, visualization and static debugging. Breakpoints are specified as pairs of “point” and “direction”. There are directions as follows: pause (to stop a goal there), process (to visualize a process with respect to a goal), notree (not to keep execution history) and stream (to visualize a data as a stream).

6.2.4 Browsing Program Code

To comprehend static information of Fleng programs, HyperDEBU has a prototype of the program code browser, which helps a user: (1) setting breakpoints (2) making static debugging (3) correcting a source code.

7 Conclusion

We have developed the parallel inference machine PIE64 including the inference processor UNIREDI, and investigate some aspect of the parallel management kernel for PIE64. We have also developed the multiwindow debugger HyperDEBU for Fleng. We are now developing other kinds of programming environment, such as a performance debugging tool, and some parallel applications on PIE64.

References


