ICOT Technical Memorandom: TM-1130

TM-1130

Strategy Management Shell on a Parallel Machine

by
Y. Kohda & M. Maeda (Fujitsu)

November, 1991

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Strategy Management Shell on a Parallel Machine

Youji Kohda, Munenori Maeda
International Institute for Advanced Study
of Social Information Science
FUJITSU LABORATORIES LTD.
1-17-25 Shinkamata, Ota-ku
Tokyo 144, Japan

{kohda,m-maeda}@iias.flab.fujitsu.co.jp

Extended Abstract

Parallel processing has great potentiality, and it looks for the places where it can show the maximum strength. Multi-tasking capability is desirable in parallel processing as in sequential processing. In single-tasking, it is necessary to rebuild parallel programs for each parallel machine to extract the maximum power of the parallel machine. It is even more difficult to design a suitable load-balancing strategy in multi-tasking environment. Static scheduling will fail in multi-tasking environment, since jobs are entered at any time and interfere with each other. Moreover a parallel machine is a dynamically operating electronic complex rather than a simple automaton, and this also makes static scheduling incompetent. We use the power of multi-tasking to tune up parallel programs for a specified parallel machine. For the purpose of tuning up, we can plan a series of experiments to find suitable strategies. However we have to confront a vast search space of possible strategies, if we plan a straightforward search.

Strategy Management Shell is a specially designed multi-tasking shell to make the search feasible. A parallel machine has lots of PEs (Processing Elements) and they have wasteful idle time. The shell utilizes the idle time to make the search in parallel and repeatedly. It also utilizes a well-chosen "representation" for describing load-balancing strategies, since search space is considerably reduced by choosing the appropriate representation conforming to the target.

We have prototyped a Strategy Management Shell in KL1 on multiPSI. KL1 is a concurrent logic programming language. The concurrent nature is suitable to describe the task management, and the logic feature is suitable to describe the strategy management. The process of KL1 is a goal, and a goal conveys all the necessary environments explicitly. The program of KL1 is a set of clauses, and clauses are highly independent of each other. Hence both processes and programs are easily migrated in KL1, and the shell makes the most use of the fact.

A task is really a job but running under a different load-balancing strategy. The shell may

create several tasks for a job. The shell manages a database which records the load-balancing performance of strategies tried in the past for every job. It can pick up the best strategy among the strategies recorded in the database, and a task with the strategy is executed as *the leading task*. Similarly, it can suggest several experimental strategies and several tasks are executed as *experiments* under the strategies. When these tasks end, the performance of each strategy is recorded in the database, which will be consulted later. A list of strategies can be given by users, and then the shell picks up the best strategy among them. Otherwise some learning algorithm is used to extract a "success rule" from the set of superior strategies. Experimental strategies will be generated using the success rule with low cost. Another candidate is genetic algorithm, which can produce an experimental strategy by gathering locally optimized decisions from a pair of strategies. The shell works as follows:

- When a job is entered by a user and the parallel machine is not overloaded, the shell starts the leading task and several experimental tasks of the job. The result of the job is the one that is returned first from one of the tasks.

- When a job is entered by a user and the parallel machine is overloaded, the experimental tasks already entered are canceled by the shell, and then the leading task of the job gets started. This avoids unnecessary wait caused by the shell's experiments.

- When the parallel machine is not overloaded and can afford more tasks, the shell voluntarily picks up a job in turn from the database and starts several experimental tasks of the job. It makes use of the idle time of PEs for further experiments.

The representation of a load-balancing strategy is separated from program text, to make it easy to try various strategies without inspecting the text. It consists of two descriptions: how to partition a program into segments and how to assign PEs to a task. A program is partitioned into several segments and a segment is a unit of program transfer. (Duplication between segments may occur.) A subset of PEs is assigned to each task. (A PE may be assigned to different tasks at the same time.) The subset gives the load-boundary of a task; segments can be transferred only to the PEs in the subset. This avoids PE's wasteful possession of local memory with unused segments. In general, PE offers a "meeting spot" to processes and their program. In sequential processing, the only solution is to move necessary segments to the central PE where processes wait. In parallel processing, processes may be moved to other PEs instead of segments when they fail to meet.

PEs in an assigned subset are numbered and the numbering makes a "forwarding route" in the subset. Segments are forwarded by the shell along the route, and processes are forwarded by underlying mechanism along the same route. With the cooperation of the shell and the underlying mechanism, processes are load-balanced, following the necessary segments:

- When a PE is overloaded with lots of processes, the shell forwards a segment from the PE to the successor PE, expecting to distribute the processes.

- When a PE is not overloaded and can afford more processes, the shell forwards a segment from the predecessor PE to the PE, expecting to gather processes in one PE.

- When a necessary segment is not found for a process on a PE, underlying mechanism forwards the process to the successor PE of the PE.

We rely on the fact that programs in KL1 can be partitioned into so small pieces that we can expect the transfer cost is not expensive. Otherwise all the segments can be delivered to all the PEs along the forwarding route before the execution, and the action of forwarding segments can be simulated by enabling or disabling segments with masks. Program transfer should be a part of task execution, while program transfer is often out of consideration in batch environment.