Rule partitioning versus task sharing in parallel processing of universal production systems

by HEE WON
SUNY at Buffalo
Amherst, New York

ABSTRACT

Most research efforts in parallel processing of production systems have focused on the rule partitioning method. This method cannot be successfully applied to the universal production systems that allow different inference mechanisms, different scopes of WM (working memory) testing, different rates of WM changes, and which do not use empirical data for partitioning.

Parallel memory configuration is essential for memory intensive applications such as production systems. Maximum parallelism cannot be achieved without sufficient memory bandwidth.

A new parallel processing method that can run the universal production systems on a parallel memory configuration is proposed, and is compared with the rule partitioning method.
INTRODUCTION

Three main issues will be addressed: how to efficiently execute parallel production systems, the importance of parallel memory for the parallel production system, and the necessity of architecture that is general enough to be applicable to universal production systems.

Most parallel production systems try to partition the set of rules by analyzing the parallel executability of the set. However, a rule can be characterized differently, depending on where it is used. A rule can be involved in more than one task. The optimal partitioning of rules changes as the task changes. Characterizing a rule according to a task is necessary in exploiting the maximum parallelism.

The main computation of the production systems is to match the condition elements to the data base. It requires sifting vast amounts of information. Thus parallel memory is the key factor determining the success of parallel processing. However, most parallel production systems ignore the necessity of parallel memory. Parallel CPUs cannot be fully utilized if the information needed is not readily available because of limited memory bandwidth. Without parallel memory, the true parallel processing cannot be achieved in the production system.

In this paper, a set of heterogeneous processors are loosely coupled to form a multiprocessor system rather than a special parallel processing system such as a tree machine. This configuration is more practical and less expensive because it can be easily built by connecting already existing computers rather than building a new computer system.

Another deficiency we can find in most parallel production systems is the lack of universality. Most proposed systems are based on a very narrow domain. Some proposals are only good for forward inference chaining while others specialize in backward chaining. More restrictions are imposed on its applicability by requiring empirical data to produce the optimal mapping of the rule to the parallel processor.

A task sharing concept is introduced to solve the problems mentioned above. To execute the concept efficiently, a parallel memory system is advocated.

PRODUCTION SYSTEMS

Production systems consist of three basic components: a set of rules, a data base, and an interpreter for the rules.

A rule consists of a conjunction of condition elements called the left hand side (LHS) and a set of actions called the right hand side (RHS). A set of rules called productions make up the production memory (PM). Rules and productions are interchangeably used in the following discussion.

The data base contains facts and assertions. The rules act upon this data to update the state of knowledge by modifying the data base. The data base is called working memory (WM).

The interpreter may be seen as a select-execute loop in which one rule applicable to the current state of the data base is chosen and then executed. Its action results in a modified data base, and the select phase begins again. A well known OPS5 interpreter goes through the following phases in the select-execute cycle:

1. MATCH: Determines which productions are satisfied in the current state of the working memory.
2. CONFLICT RESOLUTION: One of the matched productions is selected for execution based on some predefined criteria.
3. ACT: The actions specified in the RHS of the selected production are executed.

PARALLEL PRODUCTION SYSTEMS

Many parallel production systems have been proposed with the goal of accelerating the rule firing rate of each cycle. Most methods partition the rules and assign each partition to different processors. Each partition can be either a rule or a set of rules that are not usually affected by the same set of working memory changes. Common characteristics of the proposed algorithms are: (1) Partitioning is made in compile time, and (2) Rule partitions are disjoint subsets. The method can be termed “Rule Partitioning” in the sense that the interdependency and the possibility of parallel execution between rules are analyzed and used as criteria of the partitioning. Thus this method tries to find the global optimum.

Another approach will be to find the local optimum. Execution of the production systems consists of tasks. The exploitation of parallelism within a task is the main idea of the task sharing algorithm. But the sequence of tasks is determined in run time. Therefore careful consideration should be given to how to make efficient run time scheduling.

ASSUMED ARCHITECTURE AND UNIVERSAL PRODUCTION SYSTEMS

The following discussion is based on a very general and realistic processing environment.

A loosely coupled heterogeneous parallel processing architecture is assumed. Each processor can store large programs and execute them independently. Each processor varies in its speed and size. Some processors might have very powerful floating point ALUs while the others can have specialized I/O
units. One of the processors is used for user interface. The processors communicate through a simple bus structure.

Universal production systems are to be run on the assumed architecture. In the universal production systems, the inference mechanism can be forward or backward, or both. The condition element (LHS) of each rule requires different processing times, different resources, and different scopes of WM testing. Some condition elements might require floating point operation while others might need user's response to complete the evaluation. Global WM tests can be accommodated in the universal systems. The action (RHS) part can also have varieties of choices. The rate of change in WM can be massive or limited. The number of rules affected by a change in WM might vary to a great degree.

**RULE PARTITIONING**

In general, the rule partitioning method proceeds as follows:

1. Assign rules that are likely to be active at the same time to different processors.
2. Assign the WM elements corresponding to the condition elements of the allocated rules to the same processor. The WM allocation does not necessarily lead to disjoint WM partitions.
3. Repeat the following until no further rule can be fired.
   a. For each change in WM, broadcast the change to all processors.
   b. All processors where local WM has been changed conduct the match process and report results to the control processor.
   c. The control processor identifies a single rule for execution, and the action part becomes the next change in WM.

This partitioning method suffers from several significant problems in executing the universal production systems on parallel processing systems:

1. Predicting optimal partitioning is impossible in the universal production systems. For example, optimal partition for the forward chaining method is no longer optimal for the other inference methods.
2. Balancing the load amongst the processors requires predicting the processing time of each individual production in addition to the parallel executability.
3. Parallel executability might adversely effect the attention mechanism. Attention does not stay focused when a condition element shared by two rules which belong to different tasks is activated.
4. Compile time scheduling does not reflect the dynamic behavior which is necessary to make the true optimal rule partitioning.
5. Run time scheduling takes too much overhead because the entire PM and WM should be rearranged in every cycle.

**TASK SHARING**

**Processing Grain**

Different levels of processing can be defined in the production systems.

1. System level: Several independent production systems can be integrated to build one large production system.
2. Task level: Execution of a production system is made of tasks such as a series of goals or a sequence of WM changes.
3. Rule level: Each individual rule can be found under the task level.
4. Match level: Evaluating a rule involves matching condition elements to WM elements.
5. Selection level: Selecting a rule to fire requires conflict resolving task.

Parallel executability can be found within a level or across the levels. In system level, parallelism can be found if more than one production system can be concurrently activated to solve the problem of the larger production system. At the task level, sequential execution should be enforced if the next task cannot be determined before the current task is completed. One the other hand, parallel execution can be possible at the task level if more than one task can be initiated simultaneously. Most efforts for parallel production systems have focused on the rule level because each rule can be evaluated independently. Very fine grain parallelism can be found within match level, selection level, or action level. Additional parallelism can be found across these levels.

How to exploit the existing parallelism depends on how the processing element is assigned to the available resources. In system level, parallel processing can be achieved by cooperatively executing individual production systems assigned to the different processors. The parallel processing method in the task level is discussed in the following section.

**Task Sharing**

First the difference between Rule Partitioning and Task Sharing should be made clear. In Rule Partitioning, the rules themselves are partitioned and assigned to different processors in compile time. Thus only small portions of rules are assigned to each processor. In Task Sharing, no partition is made amongst the rules. Each processor has entire PM and WM. However every processor shares the execution of the task by running different rules. The decision as to which processor executes which rule is made dynamically. This dynamic scheduling can be done without much overhead because no transference of rules across the processors is necessary. The way in which every processor can access the entire PM and WM eliminates the overhead involved in rearranging the PM and WM. The price to pay is, of course, the duplication of PM and WM in every processor.
**Task Sharing Algorithm**

Heterogeneous processors connected to a bus are the underlying architecture on which this algorithm is based. The main idea of this algorithm is to let each processor autonomously schedule its task. The coordination of the system is maintained by the fact that all the processors follow the same scheduling strategy, and the scheduling activity is broadcasted to all other processors if it is necessary.

Even distribution of the load to the processors should be carefully planned. For a given task, a set of rules should be assigned to the processors in an optimal way. The relationship between the ability of the processor and the computational need of the rule will determine how well a set of the rules will map onto a parallel processor. Thus the abilities of the processors and the availability of each processor should be known to the algorithm. The algorithm also should be intelligent enough to identify the computational need of each individual rule. For example, special condition elements such as user interface elements should be identified in compile time so that the information is readily available in run time. The algorithm can be described as follows.

1. **Task Identification**: For a given task such as establishing a goal in backward chaining or executing a match phase for each WM change in forward chaining, every processor identifies the relevant set of rules and forms a task table. The task table holds the scheduling information and the condition element matching information.
2. **Initial Task Scheduling**: All the processors simultaneously execute the first batch of scheduling. Each rule is characterized by its special need such as user input or floating point operation or special I/O operation. For example, a rule with floating point operation condition elements should be assigned to a processor having floating point ALU. The assignment of two rules with identical characteristics to two identical processors is determined by its rule order and the processor order which are known throughout the system. The characteristic or the rule order of the current set of rules can be identified consistently throughout the processors so that consistency of scheduling can be maintained despite its autonomous scheduling strategy. Every processor identifies its rule to execute in addition to identifying the assignment of the rest of the rules to the other processors.
3. **Match**: Repeat the following until all the relevant rules for the current task are matched.
   a. Every processor independently matches the rule assigned to it.
   b. Any processor having finished its match process identifies the next rule to execute, and broadcasts its results of matches (fail or success and/or its instantiated variables) with the new scheduling information.
   c. As soon as the message is received by all processors, every processor updates its task table. A possible scheduling conflict between two processors should be resolved by some arbitration method.
4. **Conflict Resolution**: This process can be executed independently and concurrently by every processor. The results will be the same because every processor has the same set of matched rules and the same conflict resolution strategy.
5. **Act**: The same action is taken autonomously and consistently throughout the system.

Further overlapping between phases of the selection-execution cycle can be achieved. When some of the processors are busy in matching the last portions of the active rules, the idle processors can proceed to the next phase so that the conflict resolution can be completed as soon as the last result of the match phase is available.

**RULE PARTITIONING VS. TASK SHARING**

1. **Load balance**: In rule partitioning, it is hard to balance the load among the processors. Balancing the load for one cycle might conflict with the balance requirement of the next cycle. Balancing the load for the forward chaining does not coincide with that of backward chaining. Moreover, evenly distributing the same number of active rules to the processors does not guarantee the load balance because execution time of each rule varies. In the worst case, the parallel processor might degrade to the serial processor when all the active rules reside in the same processor from cycle to cycle.

   In task sharing, optimal load balance is always maintained by dynamically assigning an even share of rules to the processors. A processor can process several short rules while another processor processes a long rule. A processor can grab another rule to process as soon as it has finished processing the current rule. Adaptive scheduling can be achieved by matching the need of a rule with the ability of a processor. Thus this method always guarantees optimal load balance regardless of unpredictable execution environment or execution time differences among rules.

2. **Communication**: In match phase, two different kinds of communications can be observed: one for testing WM, the other for reporting the matched results. In rule partitioning, communication is necessary for testing WM because entire WM is available within the same processor. In rule partitioning, it depends on how to allocate WM. If no duplication of WM is allowed, then quite a large amount of communication is required to test the WM elements which are not available within the same processor. The situation is aggravated if global WM test is required. To eliminate communication of this kind necessitates the duplication of part of the entire WM.

   In reporting the matched result, the rule partitioning has advantage over the task sharing. Rule partitioning needs to report the results of successful matches (or the results of part of successful matches if only local maximum needs to be considered in the conflict resolution phase). In task sharing, all the results of the matches with the scheduling information need to be broadcasted to the other processors.

   Rule partitioning needs to broadcast the actions of se-
lected rule(s) after resolving the conflict while task sharing does not because every processor can resolve the conflict autonomously.

3. Universality: Three different criteria can be used to measure the universality of the algorithm: hardware, software, and application. The method should be applicable to different hardware configurations and be able to take advantage of it. It should allow dynamic software environments. It should be flexible enough to adopt new applications without difficulty. None of these important issues have been addressed in the rule partitioning methods published so far. However, the task sharing method considers all of these issues. Hardware specification can be integrated into the scheduling scheme to take advantage of it. Users can choose or change any inference mechanisms at any time without degrading the system performance. New applications can be run efficiently without extracting empirical statistics or analyzing rule dependency in advance.

4. Fault Tolerance: The task sharing method has fault tolerance which the rule partitioning method does not have. The task sharing method has achieved fault tolerance while achieving parallel processing. Each processor keeps the record of execution status relevant to the current task as well as its own copy of the scheduling mechanism, and the entire PM and WM and interpreter. This complete distribution strategy eliminates need for any hardware element like a control processor for scheduling purposes. Any partial breakdown of the system does not effect the recovery of the overall system.

5. Hardware Requirement: Three different configurations can be envisioned in the multiprocessor system. At the low end multiple CPUs are connected to a single memory. This method might suffer from severe memory contention when that method is applied to a memory intensive application such as a production system. In the middle we can find multiple CPUs connected to shared memories via interconnection networks. This method might relieve some of the memory contention. But this solution is still far from being universal. At the high end we can find the architecture on which task sharing is based. Each processor has its own memory which can hold the entire PM and WM. A single bus is used to connect the multiple processors. The only traffic on this bus will be the broadcasting of the matched result. All other information is readily available in each processor system including the scheduling algorithm. This method might be expensive. However, for a memory intensive application like the production systems, the importance of parallel memory outweighs that of parallel CPUs.

EXTENSION OF THE ALGORITHM

The algorithm is based on the assumption that all of the PM and WM can be stored within a processor. But only part of PM and WM can be brought into the internal memory as the volume of PM and WM grows. Every processor has its own disk where the entire PM and WM resides.

The question is: how to efficiently bring in the necessary PM and WM for a given task. It takes too much time for every processor to search the entire PM and WM residing in disk. One way of solving this problem is to partition the search area. Every processor can search for different portions of PM and WM without disk interference because every processor has its own disk under its control. After bringing in different portions of PM and WM into different processors, exchanging portions of PM and WM throughout the system is necessary for every processor to have entire PM and WM relevant to the current task. Again duplication of PM and WM in disks can help to decrease the disk search time.

A possible improvement is to decrease the communication overhead in exchanging portions of PM and WM. Partial exchange, like transferring only special rules to special processors and keeping the rest as local tasks, can decrease the communication overhead at the expense of imperfect load balancing. For example, a rule requiring user response must be assigned to the processor having a user terminal while the others can be executed in any processors.

CONCLUSION

This paper has pointed out the limitation of the rule partitioning method. The method searches for the global optimum by analyzing the parallel executability of the rules in compile time. The task sharing method has been proposed to analyze the parallelism in the context of tasks. The local optimum of a given task can be found by analyzing parallel executability in run time. This local optimum should be used to map the productions to the parallel processors in run time.

Parallel memory is essential to execute memory intensive applications such as production systems. Parallel CPUs can access the same information simultaneously because each processor has its own memory where the entire PM and WM is available. Parallel CPUs cannot be fully utilized without parallel memory. This duplication of PM and WM for every processor can decrease the overhead incurred by the run time scheduling method.

The universal system is needed in a growing area like the production systems. A special architecture based on a special algorithm will suffer from its shortcoming as the field expands. The task sharing scheme on the heterogeneous processors with parallel memory is general enough to afford the universal production systems.

REFERENCES