Implementation of Multiple Rule Firing Production Systems on Hypercube

Steve Kuo and Dan Moldovan
skuo@gringo.usc.edu and moldovan@gringo.usc.edu
DRB-363, (213) 740-9134
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-1115

Abstract

The performance of production programs can be improved by firing multiple rules in a production cycle. In this paper, we present the ***multiple-contexts-multiple-rules (MCMR) model*** which speeds up production program execution by firing multiple rule concurrently and guarantees the correctness of the solution. The MCMR model is implemented using the RUBIC parallel inference model on the Intel iPSC/2 hypercube. The Intel iPSC/2 hypercube is chosen because it is a cost-effective solution to large-scale application. To avoid unnecessary synchronization and improve performance, rules are executed asynchronously and messages are used to update the database. Preliminary implementation results for the RUBIC parallel inference environment on the Intel iPSC/2 hypercube are reported.

1 Introduction

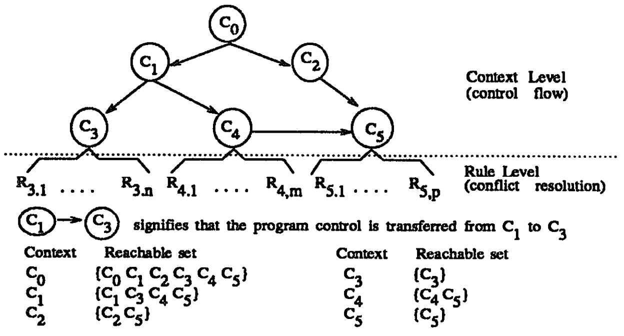
The multiple rule firing production systems increase the available parallelism over parallel match systems by parallelizing not only the match phase, but all phases of the inference cycle. To speedup the derivation of correct solutions by multiple rule firing, two problems - the ***compatibility problem*** and the ***convergence problem*** - need to be addressed. The compatibility problem arises from the data dependences between production rules. If a set of rules does not have data dependence among themselves, they are said to be ***compatible*** and are allowed to fire concurrently. The convergence problem arises from the need to follow the problem solving strategy used in a production program. If the problem solving strategy is ignored, then two tasks may be executed out of sequence or two actions for the same task may be executed in the wrong order resulting in an incorrect solution.

There are three approaches to address the compatibility and the convergence problems. The first approach considers only the compatibility problem and resolves it by data dependence analysis [3] [6] [8]. Both synchronous and asynchronous execution models have been proposed. In these models, rules which are compatible are fired concurrently in a production cycle. Because the convergence problem is not addressed in these models, the problem solving strategy for a production program may be violated when multiple rules are fired simultaneously. The second approach addresses the compatibility and the convergence problems by developing parallel production languages. CREL [6] and Swarm [2] are two such languages. Production programs written in these languages do not use control flow or conflict resolution to ensure that the right rules are fired. Instead, production rules are fired as soon as they are matched. The correctness of these parallel production programs is guaranteed by showing that for any arbitrary execution sequence the correct solutions are always obtained [1]. A potential hurdle for CREL and Swarm is the possible difficulty to prove the correctness of a large production program. In addition, to be able to fire production rules as soon as they are matched may not be the same as being able to fire multiple rules concurrently. These questions will be answered when the benchmark production programs have been translated into CREL and Swarm programs and their performance measured.

The multiple-contexts-multiple-rules (MCMR) model presented in this paper represents a third approach. The MCMR model addresses the compatibility problem by data dependence analysis. It addresses the convergence problem by analyzing the control flow in a production program to maintain the correct task ordering. In a production program, a complex problem can be solved by dividing it into smaller tasks until they are easily solved. These tasks are called *contexts* and each context is solved by a set of *context rules*. The MCMR model improves the performance of a production program by activating multiple contexts and firing multiple rules concurrently. It guarantees the correctness of the solution by determining the conditions under which multiple contexts and multiple rules can be activated and fired. To capture the maximum parallelism, the production rules and the working memory are distributed when the MCMR model is implemented on the Intel iPSC/2 hypercube. The sequential inference cycle is also replaced with the RUBIC parallel inference cycle. The RUBIC parallel inference cycle executes rule in different nodes asynchronously and updates the working memory by message passing. The performance of production programs on iPSC/2 is measured.

2 Multiple-Contexts-Multiple-Rules Model

To resolve the compatibility and the convergence problems successfully, one needs to understand how problems are solved in production programs. In general problem solving, a complex problem is usually solved by stepwise refinement. It is divided into smaller and smaller subproblems (or tasks) until they are easily solved. If other subproblems need to be solved before solving a subproblem, the program control is transferred from one subproblem to another. Production programs solve complex problems in exactly the same way. First, the production rules in a production program are divided into subsets of rules, one subset for each subproblem. A subset of rules is called a *context* and each individual rule in the subset is called a *context rule*. Every context rule in the same context has a special *context WME*. Rules in different contexts have different context WMEs. A programmer can control which context is active by adding and removing the context WMEs. Context rules are divided into *domain rules* and *control rules*. Domain rules address the subproblem associated with the context and conflict resolution is used to select the right rule to fire. If other subproblems need to be solved before solving a subproblem, the control rules transfer the program control to the appropriate contexts by modifying the context WMEs. By analyzing the control rules, the control flow between different contexts can be determined. The problem solving strategy and the control flow diagram for an example production program is shown in Figure 1.



C_0
C_1
C_2
C_3
C_4
C_5
Context Level (control flow)
$R_{3.1}$ .... $R_{3.n}$ $R_{4.1}$ .... $R_{4,m}$ $R_{5.1}$ .... $R_{5,p}$
Rule Level (conflict resolution)
$C_1 \rightarrow C_3$ signifies that the program control is transferred from C_1 to C_3
Context Reachable set
C_0 $\{C_0\, C_1\, C_2\, C_3\, C_4\, C_5\}$
C_1 $\{C_1\, C_3\, C_4\, C_5\}$
C_2 $\{C_2\, C_5\}$
Context Reachable set
C_3 $\{C_3\}$
C_4 $\{C_4\, C_5\}$
C_5 $\{C_5\}$

Figure 1: Control in Production Program

The multiple-contexts-multiple-rules model resolves the compatibility and the convergence problems at two levels: the *rule* level and the *context* level. At the rule level, the MCMR model resolves the compatibility problem by data dependence analysis. A set of rules is allowed to fire concurrently and are said to be *compatible* or *serializable* if executing them either sequentially or concurrently, the same state is reached. This is the case if there are no data dependences among rules in the set. The data dependence analysis is performed at compile time to construct a parallelism matrix P $= [p_{ij}]$ and a communication matrix C $= [c_{ij}]$. Rules R_i and R_j are compatible if $p_{ij} = 0$; they are incompatible if $p_{ij} = 1$. The communication matrix C is used for communication purpose when production rules are partitioned and mapped into different processing nodes in a message-passing multiprocessor. Rules R_i and R_j need to exchange messages to update the database if $c_{ij} = 1$; they do not need to if $c_{ij} = 0$.

The MCMR model resolves the convergence problem at the rule level by dividing contexts in a production program into three different types: (1) *converging* contexts, (2) *parallel nonconverging* contexts and (3) *nonconverging* or *sequential* contexts. A context C is a converging context if starting at a state satisfying the initial condition INIT for that context, all execution sequences result in states satisfying the post condition POST for that context [1] [2]. Otherwise context C is a nonconverging context. The conflict resolution can be eliminated for a converging context because all execution sequences converge to the correct solution. Compatible rules can be fired simultaneously within a converging context without error. This is because firing a set of compatible rules concurrently is equivalent to executing them in some sequential order and all execution sequences reach the correct solution for a converging context (for proof, see [5]). For a nonconverging context, conflict resolution must be used to reach the correct solution. The performance of a nonconverging context can be improved by parallelizing its conflict resolution. A parallel nonconverging context is a nonconverging context whose conflict resolution is parallelizable and as a result multiple rules may be selected. The conflict resolution for a sequential context is not parallelizable and only sequential execution is possible. By dividing contexts into different types and applying the correct execution model for each type, the compatibility and the convergence problems are resolved at the rule level.

The MCMR model resolves the compatibility and the convergence problems on the context level by analyzing the control flow diagram to determine which contexts are allowed to be active at the same time. These contexts are called *compatible contexts.* Two contexts are compatible if their reachable sets do not intersect and rules in the two reachable sets do not have data dependences (for proof, see [4]). The reachable set for a context C_i is the set of contexts which are reachable by following the directed arcs in the control flow diagram starting from C_i. Context C_i is included in its own reachable set. The reachable set for context C_1 for the example production program in Figure 1 is $\{C_1, C_3, C_4, C_5\}$.

The production rules are analyzed at compile time to generate the compatibility context matrix $CC = [cc_{ij}]$. Two contexts C_i and C_j are compatible and are allowed to be active at the same time if $cc_{ij} = 0$; they are incompatible if $cc_{ij} = 1$. The programmer then consults the CC matrix and modifies the production rules if needed so that only the compatible contexts will be activated concurrently in production program execution. In this way, the compatibility and the convergence problems are resolved on the context level.

3 RUBIC Parallel Inference Model

Because the traditional rule based inference cycle is sequential in nature and does not reflect well with the MCMR model, a new parallel inference model has been developed. This new model is called the *RUBIC parallel inference model* and is shown in Figure 2. It consists of seven phases: *match*, *local conflict resolution (LCR)*, *context-wide conflict resolution (CCR)*, *context-wide compatibility determination (CCD)*, *act*, *send-message* and *receive-message*.

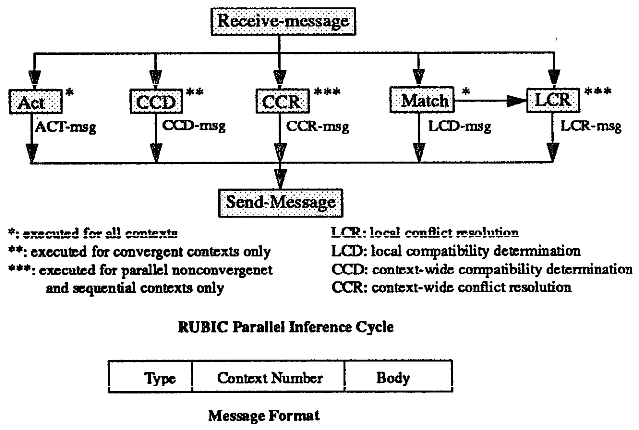

Receive-message
Act *
CCD **
CCR ***
Match *
LCR ***
ACT-msg
CCD-msg
CCR-msg
LCD-msg
LCR-msg
Send-Message
*: executed for all contexts
**: executed for convergent contexts only
***: executed for parallel nonconvergenet and sequential contexts only
LCR: local conflict resolution
LCD: local compatibility determination
CCD: context-wide compatibility determination
CCR: context-wide conflict resolution
RUBIC Parallel Inference Cycle
Type
Context Number
Body
Message Format

Figure 2: RUBIC Parallel Inference Cycle and Message Format

Before the start of production program execution, the production rules and the working memory are distributed among the processing nodes. Each processing node compiles the rules into a Rete network. During the execution of production programs, each node executes the RUBIC parallel inference cycle asynchronously and communicates with each other by messages. The action each node takes depends on its incoming message. If processing node $Node_i$ receives two messages telling it to execute the match phase for context C_1 first then the CCD phase for context C_2, it would execute the match phase for C_1 first and then the CCD phase for C_2. At the same time, $Node_j$ may be executing the CCR phase for context C_3 and the act phase for context C_4 depending on its incoming messages. Each outgoing message is prepared and sent to the appropriate nodes during the send-message phase. When each node finishes processing a message, it enters the receive-message phase. If there are messages waiting to be processed or newly arrived messages, the oldest message is examined and the appropriate action is taken. If no message is available, the node loops and waits for an incoming message. By allowing each node to execute in an asynchronous, message-driven fashion, the variance between execution times for different processing nodes is reduced and the performance is improved.

We also need to be able to detect the termination of a production program, which occurs when there is no rule matched; i.e. all tasks have been completed. By requiring that the context WME for a context be removed at the completion of that context, a production program concludes its computation when no context WME exists . This is easily implemented with a special termination rule whose LHS condition is satisfied when no working memory element is present in the working memory and whose RHS action consists of an explicit halt command. The implementation of the RUBIC parallel inference model on Intel iPSC/2 hypercube is explained below in more detail.

RUBIC Parallel Inference Model

Match: The Rete algorithm is used to match the production rules. However, at the end of the match phase, different actions are executed depending on the types of contexts. For rules in convergent contexts, the processing node skips the local conflict resolution phase and enters the send-message phase immediately. During the send-message phase, the rule numbers of the matched rule, not rule instantiations, are sent by messages to the designated CCD node for compatibility determination. Different converging contexts have different designated CCD nodes allowing the compatibility determination phases for compatible contexts to be executed asynchronously.

For rules in parallel nonconverging and sequential contexts, each processing node immediately executes local conflict resolution phase at the conclusion of the match phase.

Local Conflict Resolution: The local conflict resolution phase is executed for the sequential and the parallel nonconverging contexts, but is skipped for the convergent contexts. For the sequential context, each node uses the conflict resolution strategy to select the local dominant rule instantiation. At the end of local conflict resolution phase, the send-message phase is called to send this dominant rule instantiation to the designated CCR processor for context-wide conflict resolution. For parallel nonconverging context, the local conflict resolution phase is parallelized to select possibly multiple rule instantiations. The node then enters the send-message phase and sends these instantiations to the designated CCR node. Each parallel nonconverging context and sequential context has a different CCR node allowing the context-wide conflict resolution to be executed asynchronously for different contexts.

Context-wide Conflict Resolution: The context-wide conflict resolution is not executed for converging contexts. For a sequential context, a context-wide dominant rule is selected from local dominant rules. The send-message phase is called to send the context-wide dominant rule to all nodes containing rules in that context. For a parallel nonconverging context, the context-wide conflict resolution is parallelized and multiple rule instantiations may be selected. The send-message phase is again called to send messages to all nodes containing rules in that context.

Context-wide Compatibility Determination: The context-wide compatibility determination phase is not executed for parallel nonconverging and sequential contexts. For a converging context, a rule is chosen arbitrarily as the dominant rule. To conserve storage space, the parallelism P matrix is broken down into smaller P_i matrices, each for one context. The CCD node consults its P_i matrix to select a set of compatible rules. At the end of the CCD phase, the send-message phase is called to deliver messages to all processors containing rules in that context.

Act: Each node examines the incoming messages and executes the right-hand-side of the selected rules. The communication C matrix is used to send messages between processors updating and keeping the WM consistent. If $c_{ij} = 1$, then a message needs to be sent to PE_j if rule R_i is fired. If $c_{ij} = 0$, no message needs to be sent.

Send-Message: The send-message phase deals primarily with the communication protocol for iPSC/2. It is implemented as a distinct module to make the final code more portable to other message-passing machines. A message consists of three components: a message type, a context number and the body of the message. A message can be of the type LCR, LCD (local compatibility determination), CCR, CCD and ACT. Even though LCD stands for local compatibility determination, there is no such operation. It is used to distinguish different types of messages. The type of a message dictates the action to be performed by the receiving node. The context number indicates the context in which the action should be executed. The body of a message contains information such as the local dominant rule, the set of selected rules and RHS actions. The message format is shown in Figure 2.

Two matrices are needed to send messages: the allocation matrix A $= [a_{ij}]$, and the communication matrix C. The allocation matrix contains the partitioning information. It is used to send LCR and LCD messages from nodes containing rules in a given context to the designated context node. It is also used to send the CCR and CCD messages back to the original nodes. The communication matrix is used to send ACT messages to different nodes to maintain a consistent working memory.

Receive-Message: All incoming messages are stored in a queue. The receive-message phase is called when the node finishes its current message. If there are newly arrived messages, these messages are enqueued and the first message is dequeued and appropriate action is taken depending on its type. If the queue is empty and there is no newly arrived messages, the node loops waiting for an incoming message.

Program Termination: A special termination rule is used to halt the program execution. This rule is satisfied when there is no context WME in the working memory. Its execution causes an explicit halt command to be sent to all nodes which terminates the program execution.

4 Results

Table 4.1 Test Production System

Program	Description
A	Tournament: scheduling bridge tournaments
B	Toru-Waltz16: implementing the Waltz's edge labelling algorithm
C	Cafeteria: setting up cafeteria
D	Snap-2d[1]: a two-dimensional semantic network
E	Snap-TA: verifying the eligibility of TA candidates
F	Hotel: modeling hotel operations

Table 4.2 Sequential Simulation Results

	Production Programs					
Measurements	A	B	C	D	E	F
# of rules	26	48	94	574	574	832
# of sequential cycles = α	528	207	493	590	1175	5115

Table 4.3 Simulation Results using the RUBIC Parallel Inference Model

	Production Programs					
Measurements	A	B	C	D	E	F
Are parallel solutions correct	yes	yes	yes	yes	yes	yes
Max # of rules fired per cycle	120	14	18	15	106	81
Max # of contexts activated per cycle	1	1	12	3	3	35
Ave # of contexts activated per cycle	1	1	6.10	2.50	2.62	9.18
# para-cycles = β	85	65	78	66	125	263
$Speedup_{\alpha/\beta} = \alpha/\beta$	6.21	3.18	6.32	8.94	9.40	19.45

In this section, we first present the simulated speedups obtainable for production programs using the RUBIC parallel inference cycle, and then present the performance achieved on the Intel iPSC/2 hypercube. A logic-level simulator has been developed to measure the theoretical speedups by assuming that there are an infinite number of processors. This simulator is written in Common LISP and is currently running on a Sun Sparc workstation. Six test production programs developed at USC, CMU and Columbia have been simulated and their performance measured. To measure the performance of production programs on iPSC/2 hypercube, we have added the necessary message-passing protocols to the simulator codes. Because of the large memory requirement of the LISP program, we were only able to run 8 nodes concurrently. We have finished measuring the performance of two test programs and are in the process of measuring the rest.

[1]Snap is a simulator for semantic network array processor under development at USC [7].

4.1 Simulation Results

Six test production programs developed at USC, CMU and Columbia have been simulated under two models: the sequential execution and the RUBIC parallel inference model. By analyzing the simulations results, we can verify the validity of the MCMR model and measure its speedups. Table 4.1 lists and describes the six test programs. Because the test programs range from small programs to large programs with varying degree of parallelism and contain all three types of contexts, they represent a good mix of production programs.

The sequential and MCMR simulation results are listed in Table 4.2 and 4.3 respectively. All six test programs reached the correct solution when they were executed using the the RUBIC parallel inference model; therefore the validity of the MCMR model was verified. For Tournament and Toru-Waltz16, only one context was activated at a time. They obtained speedups of 3.18 and 6.21-folds by firing multiple rules to exploit the available parallelism within a context. On the other hand, Cafeteria, Snap-2d, Snap-TA and Hotel were able to exploit the parallelism across different contexts by activating multiple contexts simultaneously. As a result, they achieved speedups from 6.32 to 19.45-folds. This indicates that there is considerable parallelism in production programs and the RUBIC parallel inference model can effectively capture the available parallelism.

4.2 Hypercube Performance Results

The RUBIC parallel inference cycle has been implemented on the Intel iPSC/2 hypercube with 8 nodes. We have run two test programs, Cafeteria and Hotel, using the static partitioning-by-context scheme. When rules are partitioned by context, all rules in the same context are mapped to the same processing

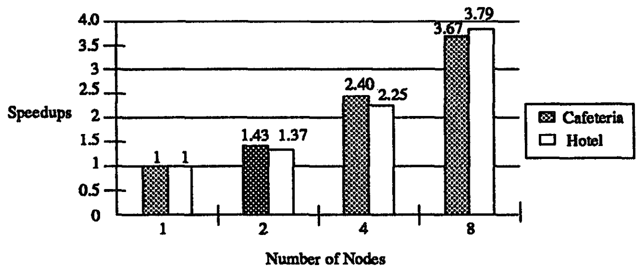


Figure 3: Performance for Partition-by-Context

node. The performances for Cafeteria and Hotel on the Intel iPSC/2 hypercube are shown in Figure 3.

Cafeteria and Hotel were able to achieve good speedups when rules are partitioned by contexts. Note, due to the timing limitations of LISP on iPSC/2 only the real time was measured. The theoretical speedup for Cafeteria is 6.32 and it achieved a speedup of 3.67 for 8 nodes. The theoretical speedup for Hotel is 8 (only 8 nodes are available) and it achieved a speedup of 3.79. Since only eight nodes were available, the upward bound for speedup is eight. For this reason, the performances for Cafeteria and Hotel were quite close. But if more nodes were available, we expect the performance for Hotel would continue to increase while the performance of Cafeteria would begin to top off.

Even though good speedups were obtained by allocating rules in a context to the same node, it is not clear whether this is the best partition. We are developing a partitioning algorithm which uses simulated annealing for rule allocation. This algorithm will read in the run-time information and uses the parallelism and the communication matrices to estimate the computational and the communication costs for each partitioning. To accomplish this goal, we are also in the process of extending the timing functions to provide better run-time information.

5 Summary

In this paper, we have presented the multiple-contexts-multiple-rules (MCMR) model which guarantees the correctness of the obtained solution when multiple rules are fired. Six test programs have been simulated under the MCMR model and all six programs reached the correct solutions. Speedups of 3.18 to 19.45-folds have been obtained for these programs which indicate to us that there are considerable parallelism in production programs.

To implement the MCMR effectively on the Intel iPSC/2 hypercube, the RUBIC parallel inference cycle has been developed. Production programs Cafeteria and Hotel have been executed on the iPSC/2 hypercube and obtained good performance. The theoretical speedup for Cafeteria is 6.32-fold, and it obtained a speedup of 3.67-folds 8 nodes. The theoretical speedup for Hotel is 8-folds (only 8 nodes are available) and it achieved a speedup of 3.79-fold. The hypercube performance results indicate that there are considerable parallelism in production programs and the RUBIC parallel inference model can effectively capture the available parallelism.

References

[1] Chandy, K. M., Misra, J. *"Parallel Program Design: A foundation."* Addison Wesley, Reading, Massachusetts, 1988.

[2] Gamble, R. *"Transforming Rule-based Programs: form the sequential to the parallel."* Third Int'l Conference on Industrial and Engineering Applications of AI and Expert Systems, July 1990.

[3] Ishida, T. el at *"Towards the Parallel Execution of Rules in Production System Programs".* International Conference on Parallel Processing, 1985, 568-575.

[4] Kuo, S., Moldovan, D., Cha, S. *"Control in Production Systems with Multiple Rule Firings."* Technical Report No. 90-10. Parallel Knowledge Processing Laboratory, USC.

[5] Kuo, S., Moldovan, D., Cha, S. *"A Multiple Rule Firing Modele - The MCMR Model."* Technical Report. Parallel Knowledge Processing Laboratory, USC.

[6] Miranker, D.P., Kuo, C., Browne, J.C. *"Parallelizing Transformations for A Concurrent Rule Execution Language."* In Proceeding of the International Conference on Parallel Processing, 1990.

[7] Moldovan, D., Lee, W., Lin, C. *"SNAP: A Marker-Propagation Architecture for Knowledge Processing."* Technical Report No. 90-1. Parallel Knowledge Processing Laboratory, USC.

[8] Schmolze, J. *"A Parallel Asynchronous Distributred Production System."* Proceeding of Eight National Conference on Artificial Intelligence. AAAI90. Page 65-71.