Lana–Match algorithm: a parallel version of the Rete–Match algorithm

Mostafa M. Aref *, Mohammed A. Tayyib

Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

Received 15 February 1997; revised 15 October 1997

Abstract

The Rete–Match algorithm is a matching algorithm used to develop production systems. Although this algorithm is the fastest known algorithm, for many patterns and many objects matching, it still suffers from considerable amount of time needed due to the recursive nature of the problem. In this paper, a parallel version of the Rete–Match algorithm for distributed memory architecture is presented. Also, a theoretical analysis to its correctness and performance is discussed. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Distributed memory architecture; Rete–Match algorithm; Performance; Correctness analysis; Production systems

1. Introduction

The Rete–Match algorithm is the fastest known algorithm for many patterns and many objects matching [1,2]. Yet, it suffers from considerable amount of time needed [3]. This paper presents a practical approach to design a reliable, heterogeneous, adaptive and efficient message-passing version of the Rete–Match algorithm (Lana–Match model) [14]. This new design is targeted to utilize run time chances for parallelism to exploit a higher degree of parallelism. Moreover, the correctness of the new model is guaranteed by the system rather than leaving it to the user that was the common practice for most of the previous attempts to parallelize this algorithm. The Lana–Match model is an optimistic, message-passing, parallel version of the Rete–Match algorithm. It is

* Corresponding author. E-mail: aref@dpc.kfupm.edu.sa

0167-8191/98/$19.00 © 1998 Elsevier Science B.V. All rights reserved.
PII: S0167-8191(98)00003-9
specially designed to utilize the power of the parallel distributed memory architecture machines and to address the following shortcomings.

1. Most of the reported work toward parallelizing the Rete–Match algorithm can be classified as pessimistic parallel match and/or parallel-rule firing models. This means, that the concurrency control mechanisms (CCM) are based on the assumptions these rules or matches will most of the time conflict with each other. Such a very strict pessimistic CCM prevented most of these approaches from exploiting high degree of parallelism and achieving major speed up.

2. Most of these approaches were either based on shared memory architectures or were based on special topologies (e.g., tree). Therefore, the reported results were based on simulated solutions.

The main idea behind the Lana–Match model is to make one or more copies of the Rete–Match network and engines running as Slave Processors (SP) for a Controller Processor (CP) that maintains a Master Agenda (MA) and a Master Fact List (MFL). The controller assigns every activated rule of its agenda to different slaves by sending all the facts that are activating that particular rule to the corresponding slave. This will activate the rule at the slave, execute the action part and send all the facts that are either added or deleted from the Master Fact List back to the Controller. The controller buffers the slave’s responses and applies them, based on the time stamp of the activation they were generated from. If an activation was deleted then its generated action commands will not be added to the MFL. The final conclusion will be found at the MFL.

The speed up and the adaptive features of the Lana–Match represent major improvement for the performance of the Rete–Match algorithm. Section 2 of this paper covers the basic concepts of production systems and the Rete–Match algorithm followed by a brief study to the previous attempts to parallelize this algorithm. Section 3 describes the Lana–Match computational model and a theoretical analysis to its correctness and performance. The advantages and the disadvantages of the Lana–Match are then presented.

2. Background

2.1. Production systems

The term ‘production system’ is a heavily used term in artificial intelligence. It is based on the original model that has been proposed first time by Post and has undergone theoretical and application-oriented development [6]. It consists of three major parts.

1. A rule base composed of a set of production rules where every rule is of the form ‘IF (conditions) THEN (actions)’ construct. The conditions part of the rule generally referred to as the left hand side (LHS).

2. A working memory is a special buffer-like data structure holding the data operated on by the program. Both the working memory and the LHS of the production rule contain lists of condition elements that are symbolic patterns.

3. An interpreter is a program that repeatedly executes the following steps:
   - Recognize: Determine which production rules evaluate to TRUE conditions for the current state of the working memory.
Select: If there are no such rules STOP; else select one of these rules.
Act: Perform the action specified by the chosen rule. This will modify the working memory by adding or deleting some data that could require another match.
Repeat: GOTO step Recognize.

This sequence of operations is called the Recognize–Act cycle. The first step of this cycle is called the matching step. The second step is called the conflict resolution step. The third step is called the act step. The match step tries to find instance of a class defined by the LHS among patterns in the working memory. This process is also called instantiating the rule. The main function of the match step is to find the set of all legal instantiations of conditions that is called the conflict set. The conflict resolution step decides whether the execution of the production system should halt, and if not, it chooses one rule to be executed in the act step.

2.2. The Rete–Match algorithm

Matching is the most time consuming step in the execution of a production system. It consumes around 90% of the total execution time for each cycle [3]. To get a feeling for the complexity of the matching process, imagine a production system consists of 2000 productions and 4000 facts, where each production has five condition elements. This production system will perform \((4000 \times 2000 \times 5)\) matching operations at each execution cycle. The Rete–Match algorithm utilizes the following facts:
1. At each execution cycle a small fraction of the working memory could be changed.
2. There are a lot of structural similarities between most of the productions that can be found in most of the production systems.

By storing results of matching from previous cycle and using them in subsequent cycles combined with performing common tests only once, the matching process can be reduced by 90% [6].

2.3. Previous work on parallelizing of the Rete–Match algorithm

The research to parallelizing the Rete–Match algorithm can be classified into three category [7].

2.3.1. Speeding up the match phase by faster sequential algorithm

Miranker et al. [9] proposed a new version of the Rete–Match algorithm (TREAT). This algorithm resolves the ineffective network updates procedure that is used by Rete. They reported 4 to 15 speed up over the Rete–Match algorithm. Highland proposed a new version of Rete–Match algorithm that he called YES/RETE [7]. HiPER achieve 3 to 11 speed up over Knowledge Tool and regular Rete–Match algorithm.

2.3.2. Speeding up the match phase by parallel algorithm

Gupta et al. [4] presented a paper on parallelizing the Rete–Match algorithm on the DADO machine. DADO is a highly parallel tree-structured architecture designed to execute production system. Gupta extended Forgy parallel version of the Rete–Match to
the DADO prototype machine and predicted to be able to process 125 facts/s. Speed up of 2 to 31 has been reported [13].

2.3.3. **Speeding up production systems by multiple rules firing**

Pasik [10] simulated a new programming methodology (i.e., IRIS) to reduce the software complexity and to improve the parallelism in production system. The simulation showed that 6 to 90 speed up is possible to achieve using the IRIS methodology. Ishida [5] proposed a simulated parallel programming environment consisting of analyzer that will determine the inter-instantiation data dependencies and a set of parallel language constructs (e.g., rule-set and focusing mechanism). The simulation indicated of 5 to 7 speed up.

Schmolze [12] proposed a simulated asynchronous distributed production system called PARS. For a 32-processors system, PARS is almost as fast as four times. For 8-processors system, PARS is two times as fast. Miranker et al. [9] developed a parallel production language CREL that is syntactically identical to OPS5. A multiple context multiple rule (MCMR) firing model is implemented [8]. The performance of this model was measured on the RUBIC simulator and the Intel IPSC/2 Hypercube. Speed up of 3 to 19 has been obtained.

2.3.4. **Other related work from the database area**

Raschid [11] presented system architecture for concurrent execution of rules (transaction) in a relational DBMS environment. Based on the DBMS Serializability they have defined a correctness criterion for concurrent execution of this system. However, it is not defined based on READ/WRITE conflict but rather it is based on conflicts between conditioned actions.

3. **The Lana–Match computational model architecture**

The Lana–Match model is asynchronous master–slave parallel computational model that consists of one Controller Processor (CP) and one or more Slave Processors (SP) (shown in Fig. 1). Every SP communicates with the CP through two communication buffers: input and output.

3.1. **The Controller Processor (CP)**

The Controller Processor (CP) is a production system that consists of a Master Rete Network, a Master Facts List (MFL), a Master Agenda (MA), an Action Queue and a Master Engine (ME). The CP is the main console through which the user can interface with the model. The user can load the rule set, insert initial facts and receive the final facts or conclusions.

3.1.1. **Master Rete Network**

Master Rete Network is the pattern and join networks generated at the CP by compiling the rules set. These two networks are created by the Build function of the
Rete–Match algorithm at the CP. At the start-up time, these two networks are also duplicated on every one of the SP by compiling the same rules set on each SP.

3.1.2. Master Facts List (MFL)

All the facts are maintained in the MFL. Apart from the initial facts that are inserted directly by the user through the CP at the start-up time, all the facts that are inserted or deleted from MFL are generated at the SP. The SP receives a message from the CP that contains the facts that activate an activation. The SP executes the activation and sends back the facts that are need to be added to (or deleted form) the MFL as a set of action commands. These action commands are queued at the Action Queue based on their time stamps. Asserting-facts-to or deleting-facts-from the MFL activates or deactivates rules at the CP, which are then translated to addition (or deletion) of one or more activation to/from the MA.

3.1.3. Master Agenda

Master Agenda is a set of activations where every one of these activations represents a rule instantiation and contains a set of pointers to the rule that it represents and all of its activating facts. Each activation maintains the following information:
1. A time stamp represents the time that this activation was added to the agenda. The CP stamps it at the creation time.
2. A pointer to the SP that is currently processing this activation. When the scheduler assigns this activation to a SP, this pointer is set to point to that SP. Otherwise, it will be set to null.
3. A pointer to the Action Commands Zone that has all the activation commands that were generated by executing this activation at the SP.

The Action Commands Zone is just a buffer that is used by the CP to temporarily store all the additions and deletions to the MFL. In other words, the Action Commands Zone is nothing more than a waiting zone for the results of executing an activation at a SP. The reason to buffer the changes and not to apply them directly to the MFL is
mainly because assertion of facts can deactivate and remove some or all of its younger activation. Since activations are executed in parallel, some of the younger activations can be completed before their elder activations. If an activation is removed from the agenda by one of its elder activation, this will remove all of the queued changes that were buffered for that activation. Allowing activations to execute in parallel and committing the changes to the Master Agenda based on their time stamps is the Lana–Match model concurrency control mechanism.

3.1.4. Action Queue

It is a circular list of slots where every one of these slots is assigned to a SP every time it is assigned to an activation. Each one of these slots has a pointer to the activation that it represents and its status. The status is set to ‘ON’ while the activation is running on the SP and it is set to ‘OFF’ whenever the SP completes the execution of that activation.

3.1.5. Slave Processor Status Table (SPST)

Every one of the SP has an entry in the SPST at the CP. This entry is set to null if the SP is idle, otherwise, it contains the status of the corresponding SP. The status of a SP is described by three entities: a pointer to the activation that is currently executing, the time stamp of that activation, the SP identification number.

3.1.6. Master Engine (ME)

As in Fig. 2, the Master Rete Network, the Master Facts List, the Master Agenda, the Slave Processor Status Table and the Action Queue are different data structures that are used by the ME of the model to carry out the parallel execution of the system. The ME consists of the main loop and different control functions such as finding a free SP, scheduling an activation to a SP, rescheduling a SP and examining and committing the action commands in the same sequence as they were created.

![Fig. 2. The Lana–Match model data structures.](image-url)
(1) **Get a free Slave Processor function** examines the SPST and returns the first SP identification number that points to a null activation. If there is not any free SP, this function executes the Rescheduler repeatedly to search the SPST and return the first available free SP.

(2) **Schedule an activation.** Given an activation and a free SP, do the following:

   (i) Set the corresponding entry for this SP at the SPST to point to the given activation.
   
   (ii) Set the time stamp field for this SPST entry to the time stamp of the given activation.
   
   (iii) Make the activation points to the SP.
   
   (iv) Buffer all the facts that activated this activation along with its time stamp at the input buffer of the SP.
   
   (v) Change the status of the Action Queue entry for that activation to ‘Running’.

(3) **Rescheduler.** At completion, every SP buffers all the generated fact additions and deletions along with the assigned activation time stamp at its output buffer. Every one of these fact additions and deletions is referred to as an action command. The rescheduler examines the output buffer of every SP and matches the time stamp of the coming action command with the current time stamp that this SP is pointing to at the SPST. If these two times match, it means that this message is a correct message. Then, the Rescheduler buffers this action command at the action zone of the activation that is being executed at that particular SP. It makes this SP pointing to null to make it free again and marks this activation as a completed activation. On the other hand, if these two times do not match, this means that while this SP is executing the activation, that activation was deactivated by its elder activations. In this case, the Rescheduler ignores the message and all other message that are coming from this SP with that mismatched time stamp. However, if all the SP output buffers are empty, the Rescheduler consider that as a free time and tries to utilize this time to examine the Action Queue and commit completed action commands based on the their time stamps.

(4) **Examine and commit action commands function.** This function examines the Action Queue starting from the beginning. If the first action on the queue is completed, it executes all the action commands that are stored at the action zone that is pointed to by the activation that owns this slot. Next, it moves forward in the queue and does the same thing but stopping at the first non-completed action even if all the activations that are queued after it were completed. This guarantees that action commands are committed in the same sequence as they were created.

(5) **The main loop.** The ME executes this loop until the end of the Master Agenda.

   ```
   LOOP until the end of the Master Agenda.
   Get a free Slave Processor and schedule the top activation of the agenda to it.
   Remove this activation from the Master Agenda.
   END LOOP
   ```

### 3.2. The Slave Processors

The Slave Processors are the set of processors that are available for the Scheduler to choose from. This set is sometimes referred to as the SP pool. SP communicate with the CP through asynchronous message-passing mechanisms. Every one of these processors
has the full Rete-Match network. It waits for messages from the CP asking to insert facts into its local agenda. When the CP finished sending all the facts that activate the activation that was assigned to this SP, these facts instantiate a rule at the SP. Then, the SP fires this rule and buffers all the new generated facts additions or deletions to/from the MFL. Fig. 3 outlines the Lana-Match computational model in English-like description.

/* THE MASTER ENGINE */
Function Get-a-free-Slave Processor;
Function Schedule-an-Activation(p,a);
Function The-Rescheduler;
Function Examine-and-Commit;
Function Remove-an-Activation (a);

/* MAIN */
LOOP Until the end of the Master Agenda
1. Get-a-free-Slave Processor(p);
2. Schedule-an-Activation(a,p);
3. Remove-an-Activation(a);
EndLOOP

Function Remove-an-Activation (a)
1. Set Agenda(a) = Nil;
2. Set Command-Zone(a) = Nil;
End

Function Get-a-free-Slave Processor;
1. Examine the Slave Processors Status Table and returns the first SP that points to Nil;
2. If there is not any, then execute the Rescheduler;
End

Function Examine-and-Commit
1. LOOP UNTIL Action-Queue(i)=Running;
2.1 Apply Command-zone(i) to the master agenda;
2.2 For each rule activation that is generated; Add that activation to the bottom of the master Agenda with the current timestamp;
2.3 For each activation that is deleted from the Agenda;
   Set PST(a) = Nil;
   Remove-activation(a);
   Command-Zone(a) = Nil;
End LOOP
End

Function Schedule-an-activation(p,a)
1. Set PST(p).proc --> Agenda(a);
2. Set PST(p).time = Agenda(a).timestamp;
3. Send all the facts that activated Agenda(a) to the Input buffer of the Slave processes p along with the Timestamp of a
(Agenda(a).timestamp);
4. Set Action-Queue(a) = Running;
End

Function The-Rescheduler
1. Examine the output buffer of each Slave Processor;
2. FOR each processor p that completed its Activation (i);
   2.1 Read the fact additions and deletions;
   2.2 Read the timestamp;
   2.3 IF the timestamp = PST(p).timestamp;
      Write all the facts addition and deletions that is generated by i to the Command-Zone(a);
      Set PST(i) --> Nil; Set Action-Queue(a) = Completed;
EndIF
End FOR
3. Return if at least one Slave Processor was freed; Otherwise
   3.1 Call Examine-and-Commit;
   3.2 Call the Rescheduler;
End

/* THE SLAVE PROCESS */
1. Examine the Input buffer repeatedly when an activation is assigned;
   1.1 Read the facts from the buffer;
   1.2 Read the timestamp;
   1.3 Insert these facts into its local fact list;
2. FOR each generated Activation
   2.1 Execute the Action of the Activation;
   2.2 Write all generated facts additions and deletion to the output buffer of the Slave Processor;
2.3 Write the timestamp of the activation to the output buffer;
End FOR

Fig. 3. The Lana-Match model outlines.
4. The theoretical analysis of Lana–Match model

4.1. The correctness of the new matching model

The correctness of the Lana–Match model is based on a centralized time stamp optimistic concurrency control mechanism that is a fully proven database technique. In this paper, this technique is extended to guarantee the correctness of parallel execution of production systems. This extension is part of the contribution of the paper as an improvement that fills in the gap between database and production system techniques. This mechanism was implemented as follows: at the creation time, the CP stamps every activation that it adds to the Master Agenda with an increment integer. This integer number is stored as part of the activation and will be used to identify this activation and all the results that is generated from it.

This correctness is accomplished by distributing the Master Agenda to the SP simultaneously and letting every SP executes its activation and passes all of its results back to the CP. These results that are either addition or deletion actions of facts to/from the MFL will not be committed as the CP receives them. But instead, they will be queued in the Action Queue. As the CP is free (or needs to get a free SP) the CP executes the Examine-and-Commit function. This function examines and executes the action commands that are queued at the Action Queue starting from the top of the queue. Then, it executes the completed activations in the same sequence as they were queued in to the Action Queue. This sequence is FIFO to reflect the incremental nature of the time stamps of the activations. The Examine-and-Commit function continues in the same fashion and stops at the first non completed activation. This makes sure that although the results of younger activations were obtained at the SP, these results will not be committed until all its elder activations were completed. This guarantees that the final effect to the MFL is done exactly the same as if the activation were executed sequentially.

4.2. The Lana–Match model complexity analysis

The expected speed up (SUP) is equal to the sequential execution costs over the parallel execution costs. In other words, the SUP is the ratio between the time that are required for the sequential execution of all the activations of the Master Agenda on a single node and the time that are required for parallel execution of all activations on more than one node. Several values were assigned to calculate the sequential execution costs.

\[ N_a \] the number of activations that were actually fired
\[ N_f \] the average number of facts per a rule instantiation
\[ T_m \] the average time that are needed to match a fact through the pattern network and to drive a fact through the join network
\[ T_f \] the average time that are required to fire a rule
Thus, the total time that is needed to evaluate the Left-Hand-Side of a rule $T_{LHS}$

$$T_{LHS} = N_l \cdot T_m. \quad (1)$$

Then, the time that is needed to fire a rule equals to the total time that is needed to evaluate the Left-Hand-Side $T_{LHS}$ and the Right-Hand-Side $T_i$. Thus, the total execution time for a rule $T_{Rule}$ would be calculated as follows.

$$T_{Rule} = N_l \cdot T_m + T_i \quad (2)$$

This makes the total execution time on a single node ($T_{seq}$) equals to the total execution time of all the activations that are fired during the exaction of the production system

$$T_{seq} = N_s \cdot [N_l \cdot T_m + T_i]. \quad (3)$$

On the other hand, to calculate the parallel execution cost $T_{parallel}$ let:

- $T_1$ be the time that is needed to evaluate the Left-Hand-Side for each rule once at the CP.

$$T_1 = N_s \cdot T_{LHS} \quad (4)$$

- $T_2$ be the time that is needed to re-evaluate the Left-Hand-Side for each rule that is assigned to all the SPs and let $N_s$ be the number of SPs.

$$T_2 = T_1 / N_s \quad (5)$$

- $T_3$ be the time that are needed to fire all the rules in parallel.

$$T_3 = N_s \cdot T_i / N_s \quad (6)$$

- $T_c$ be the average time that the CP needs to send all the facts for one activation to an SP and to receive all the generated results from executing that activation.

communication cost $= N_s \cdot T_c. \quad (7)$

- This makes the parallel execution costs as follows.

$$T_{parallel} = T_1 + T_2 + T_3 + \text{communication cost} \quad (8)$$

Performance speed up is achieved when the sequential execution costs are greater than the parallel execution costs. In other words $T_{parallel} < T_{seq}$. This relation could be expanded as follows:

$$N_s \cdot T_m + N_s \cdot T_m / N_s + N_s \cdot T_i / N_s + N_s \cdot T_c < N_s \cdot (T_m + T_i). \quad (9)$$

This relation may be simplified to:

$$T_m / (N_s - 1) + (N_s / (N_s - 1)) \cdot T_c < T_i. \quad (10)$$

However, $T_m$ is almost constant and $(N_s / (N_s - 1))$ is almost one. Therefore, the Lana–Match model achieves its best speed up when the average time $T_c$ that are required to fire a rule $T_i$ is much greater than the average time that the CP needs to send all the facts for one activations to an SP and to receive all the generated results from executing that activation.
5. Advantages and disadvantages of the Lana–Match model

The Lana–Match model enjoys several main advantages.

1. The Lana–Match model is adaptive. Adding a SP is accomplished easily just by copying the Rete–Match network on that node and declaring that node to be ready for the CP.

2. Heterogeneity at two levels. First, at the network level since the SP and the CP can be executed on any network as long as they can communicate via message-passing. Second, at the production system level since the SP and the CP can be implemented using any production system as long as it is based on the Rete–Match algorithm.

3. Exploits run time parallelism. The Lana–Match model was designed to exploit run time parallelism opportunities that are more fruitful than compile time analysis that was taken by ten out of the twelve reported studies in this area. Compilation time analysis does not utilize many of the parallelism opportunities that can be utilized at the run time. As an example, let us say that we have only one rule to fire and 10,000 run time instantiations for that rule. In this case, compilation time analysis does not foresee any chance for parallelism while it is obvious that there may be 10,000 chances for parallelism.

4. Reliability. The Lana–Match model was designed to have a great immunity from losing a node on the network. This is mainly true because the SP do not store any status. It is perfectly true, that the system can start with N processors and end with fewer processors without affecting the final results. However, the model is not immune from the death of the CP.

5. No interaction from the programmer. The Lana–Match model does not leave any responsibility on the programmer to develop parallel production systems that are as correct as their sequential versions.

6. Filled in the gap between database and knowledge base concurrency control mechanism. In this paper, we demonstrated this by showing the applicability of some of the techniques of distributed database serializability theory and concurrency control mechanisms to solve the parallel rule firing production systems.

The Lana–Match model could be criticized with the following concerns.

6. Conclusion and future research

This paper presented a new parallel version of the Rete–Match algorithm for distributed memory architectures. This model is an adaptive, heterogeneous, practical
and reliable parallel model that contributed a major improvement to the original Rete–Matching algorithm. In the next phase of our project, this model will be implemented and the performance of our implementation will be analyzed and reported. For this study, we will use the following two main ingredients to implement the Lana–Match model that are a Rete–Match based production system package and a message-passing communication mechanism. For the first ingredient we will use the C Language Integrated Production System (CLIPS) and for the other we will use the parallel virtual machine (PVM). This is mainly due to the following reasons.

1. The CLIPS was developed at NASA’s Johnson Space Center during 1985–1993. It has more than 5000 users including NASA sites and branches, numerous federal bureaus, government contractors, more than 200 universities and many companies.

2. The PVM permits a network of heterogeneous UNIX computers can be used as a single large parallel computer while handling all communications and reliability details. It provides a more efficient, powerful, reliable, popular parallel programming environment. The PVM system is composed of two parts: a demon that resides on all the computers of the virtual machine and a library of PVM interface routines. These routines are user callable routines for message-passing, spawning processors, coordinating tasks, and modifying the virtual machine.

Acknowledgements

The authors wish to acknowledge King Fahd University of Petroleum and Minerals (KFUPM) for utilizing the various facilities in preparation of this paper.

References