Parallelization of artificial intelligence algorithms

Undergraduate Thesis

Submitted in partial fulfillment of BITS F423T Thesis

By

Akhilesh Sudhakar

2013A7TS173P

Under the supervision of

Prof. Sundar S Balasubramaniam

Professor, Dept. of Computer Science and Information Systems,
Birla Institute of Technology and Science, Pilani

Birla Institute of Technology and Science, Pilani (Rajasthan)
(December, 2016)

Acknowledgements

I am thankful to the Dean, Academic Research Division and Head of the Department, CSIS BITS-Pilani for giving me this research opportunity. I thank my thesis supervisor, Prof. Sundar B., for his guidance and insight into research. His constant mentorship has helped me delve into the topic at length. I also thank the staff of the Department of Computer Science for their help and support.

Certificate

This is to certify that the thesis entitled, "Parallelization of Artificial Intelligence Algorithms" and submitted by Akhilesh Sudhakar, ID No. 2013A7TS173P, in partial fulfillment of BITS F423T (Thesis) embodies the work done by him under my supervision.

Prof. Sundar S Balasubramaniam
Professor,
Dept. of Computer Science and Information Systems,
Birla Institute of Technology and Science, Pilani

Abstract

Thesis Title: Parallelization of Artificial Intelligence Algorithms

Supervisor: Prof. Sundar S Balasubramaniam, Professor, Dept. of Computer Science and Information Systems, Birla Institute of Technology and Science, Pilani

Semester: First

Session: 2016-17

Name of Student: Akhilesh Sudhakar

ID No.: 2013A7TS173P

Abstract:

Rule-based systems are a fundamental form of Artificial Intelligence. An artificially intelligent agent requires knowledge about the world, to function. Rules are a simple representation of this knowledge. A rule specifies an action to be taken, if a particular set of conditions are satisfied. It is quite possible that even for less complex agents, a vast number of rules are needed to be matched, resolved and fired. Parallelizing these three phases can reduce execution times drastically, allowing for performing more tasks in the same time. This thesis explores the parallelization of the rule matching phase and the rule firing phase. For the matching phase, a parallel version of the RETE matching algorithm has been explored, while the firing phase uses an interference analysis between rules, followed by a rule selection process. This thesis evaluates the effectiveness of parallelizing the matching phase and the firing phase separately, as well as combined together. This work assumes an OPS5-like production system.

Index Terms – *Data dependency, rule based systems, expert systems, parallel processing, OPS5*

Contents

Chapter 1

Introduction

An OPS5-like rule based system works on a match-recognize-act principle. This thesis is divided into 4 major parts. Firstly, an OPS5 like system is implemented in C language. The choice of C as the language of implementation is briefly discussed. Secondly, the 'match and recognize' phase of the system is parallelized. The sequential RETE algorithm is used as the matching algorithm, which is then parallelized. Analysis is done regarding why the RETE algorithm has been chosen as the matching algorithm, why it is suitable to be parallelized and results are presented by simulating this over rulesets. The dependence of the performance of this algorithm on the type of rules presented to it is also discussed. Thirdly, the 'act' phase of the system is parallelized. This involves implementing a parallel firing system. This parallel firing system is comprised of:

1) Interference analysis:
   Given that no order of execution is assumed while writing rules in an OPS5-like system, it is possible that sequential execution of a rule after the other might give different results from parallel execution of both of them. When such a situation arises between two rules, there is said to be interference between them.
2) Firing algorithm:
   This is an algorithm that picks multiple rules to be fired in parallel. This algorithm makes use of certain facts it can learn about the rules during compile-time itself. The rest of the input to this algorithm comes from the run-time activity.

Fourthly, a combined simulation of parallel RETE and parallel firing algorithm is performed and results are analyzed.

Chapter 2
Background

2.1 OPS5:

The OPS5 production language consists of two types of entities- working memory elements and production elements.

Working memory elements (WMEs) are 'objects' or 'class instances' whose attributes represent the state of the world at any given time.

Production elements (PEs) are rules are written in an "if-then" format, where each production has a left-hand-side (LHS) and a right-hand-side (RHS). The LHS has a set of condition elements that have to all be true (an AND of different condition elements) for the rule to be satisfied. Condition elements are positive (+) and negative (-). Positive condition elements are satisfied when an object from the WME matches their criteria, while negative condition elements are satisfied only when there is no object from the WME that matches their criteria. The RHS represents an action to be taken when the LHS is satisfied. This action might or might not change the states of the WMEs. Changes to WMEs can either be additions, deletions or modifications

A rule is said to have an instance or be instantiated when a WME matches its LHS. The combination of attribute values of such a WME is called an instantiation of the rule.

In OPS5-like systems, a three-phase cycle is executed:

1) **Match phase**- The LHS of each PE is compared with every WME and all possible executable instances of rules are collected into a conflict set.
2) **Select phase**- Executable instances from the conflict set are selected to be executed and passed to the next phase.
3) **Act phase**- The RHS of the selected instantiated rules are executed, which may cause changes to the WMEs

2.2 Sample Ops5 Program:

Working memory:

```
#1 1 [NIL] (VALUE ^DATA 1)

#2 2 [NIL] (VALUE ^DATA 42)

#3 3 [NIL] (VALUE ^DATA -4)

#4 4 [NIL] (VALUE ^DATA 1 ^TYPE NUMBER ^POSITIVE TRUE)

#5 5 [NIL] (VALUE ^DATA 77 ^POSITIVE TRUE)
```

Production elements:

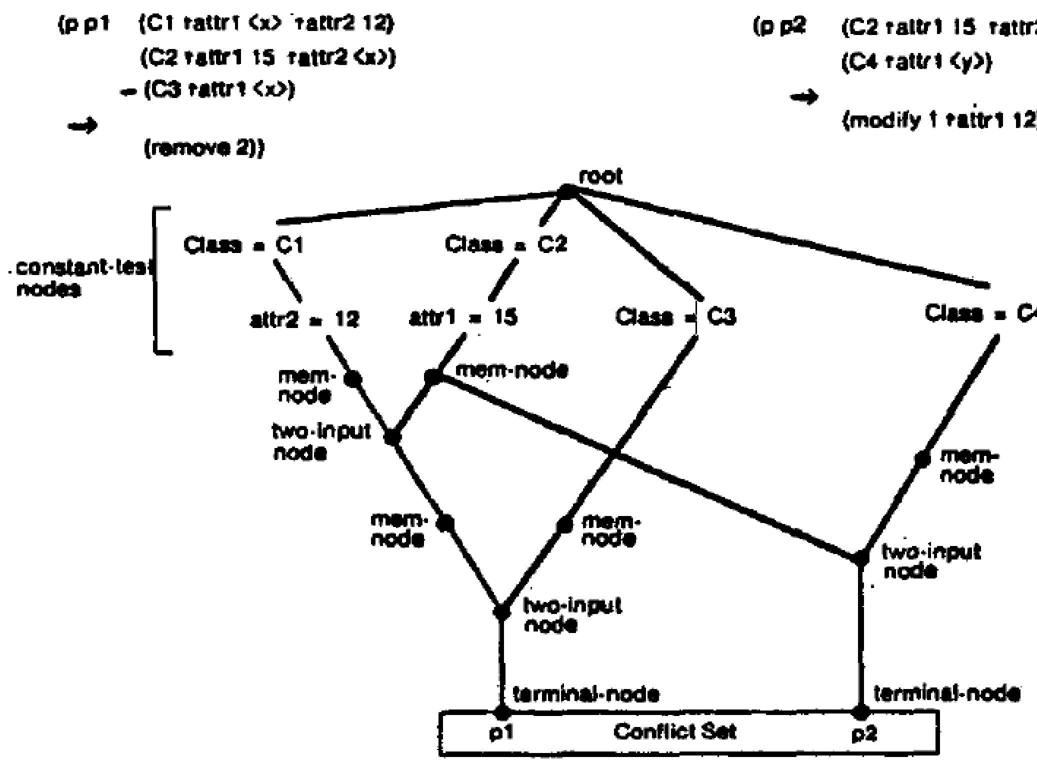
```
 (p rule-1 (begin) (value ^data < 0 ^positive NIL) --> (modify 2 ^positive FALSE)
)

(p rule-2 (begin) (value ^data > 0 ^positive NIL) --> (modify 2 ^positive TRUE) )

(p rule-3 (begin) (value ^data { > 0 }) - (value ^data > ) - (value ^positive NIL) -->
(write |Largest value: | (tabto 20) (crlf)) (remove 1) (remove 2) (make normal-
values) )

(p rule-4 (normal-values) (value ^data ) - (value ^data > ) --> (write (tabto 20)
(crlf)) (remove 2) )

 (p rule-4-specific (normal-values) (value ^data ^positive TRUE) - (value ^data > )
--> (write (tabto 20) (crlf)) (remove 2) )
```

(Source: http://www.cs.gordon.edu/local/courses/cs323/OPS5/ops5.html)

2.3 A matching algorithm- RETE:

While this report assumes that the reader has knowledge about the RETE algorithm presented in [1], a brief description has been presented below for convenience.

The RETE algorithm makes use of a data flow network that it builds at compile time. It passes ‘tokens’ through this network, where each token points to the WMEs that match subsets of condition elements in the LHS of production rules. A token contains changes to working memory element that will be caused by the current match iteration.



Source: [3] (A sample RETE network)

```
⟨−(Expression   ↑Name Expr41   ↑Arg1 Y   ↑Op +   ↑Arg2 Y)⟩
⟨+(Expression   ↑Name Expr41   ↑Arg1 2   ↑Op *   ↑Arg2 Y)⟩
```

Source: [1] (A sample token)

Tokens flow through network nodes, which are of the following types:

1) **Constant nodes:** These nodes check if the WMEs that are pointed to by tokens satisfy constant-valued attributes condition elements.
2) **Memory nodes:** These store tokens that match previous cycles. Notionally, they store tokens matching with certain condition elements, which are a part of the entire LHS of a production rule.
3) **Two- input nodes:** These nodes take both their inputs from memory nodes and test if the tokens at both inputs jointly satisfy certain condition elements.
4) **Terminal nodes:** Each production rule has one terminal node. All tokens that reach the terminal node have completely satisfied the LHS of the corresponding production. From terminal nodes, tokens are used to make changes to the WMEs.

The RETE algorithm is parallelized for performance enhancement.

Chapter 3

Related work

3.1 OPS5 in C:

An existing implementation of OPS5 in C, known as C5, is said to have been built by AT&T according to [2], but it does not appear to be easily accessible online.

3.2 Parallel matching:

The original uni-processor sequential RETE algorithm is used in some OPS5 implementations as the matching algorithm [1].

The idea for the parallel RETE algorithm that has been implemented as one part of this thesis has been borrowed to some extent from [3], the difference being that while the above paper uses a bus architecture of shared memory processors, the thesis implements it as a distributed system with each node being a multiprocessor.

Another parallel variant of the RETE algorithm has been discussed by Cao et. al., in [4]. In this MapReduce implementation, a master passes sub-rules to all the workers in the Map stage, after which it passes facts which are picked up by workers in a queue. These facts are then matched with the sub rules and the results are sent back to the master. In the Reduce stage, the master collects all the information from the slaves and decides which rules can be fired accordingly.

Stolfo et. al., prototyped a large-scale parallel computer, DADO, which is a VLSI-based tree-structured machine in [5], on which parallel rule matching software could be deployed [5]. The algorithm operates by assigning a rule to a processing element at a fixed level within the tree, making the matching time independent of the number of productions in the system. To the level just below this, a single PE is assigned an object from the WMEs. Selection of individual rules from the conflict set is also parallelized. Furthermore, Stolfo proposes five different parallel algorithms [6] that can all be run on the DADO machine, each with their respective pros and cons.

In [7] the authors propose a parallel version of RETE for distributed systems. In this setup, the master maintains a 'fact list' or working memory, and assigns each activated rule to a slave. It then sends all facts that are responsible for activating that rule to the slave, which in turn activates the rule at the slave end and hence, changes WMEs. The slave then sends back changes to the master, after which the WM is updated.

More recently, Peters et. al., published work on scaling parallel rule-based reasoning across multiple GPUs [8]. They parallelize RETE by making important distinctions between alpha and beta nodes in the match-graph, with respect to workload partitioning. An alpha node has no parents and represents exactly one rule term while a beta node has two parents and links two or more single rule terms of one rule. To compute matches of a beta node, a number of threads are spawned which are equal to the number of matches of a parent node. Beta matches of all beta nodes at a particular depth are computed in parallel. Special methods to simplify the processing of negative condition elements have been addressed by Mahajan et. al., while parallelizing RETE in [9].

Renaux et. al., have built PARTE (Parallel RETE) in C++, a system for gesture recognition with soft real time guarantees in [10]. This system notionally treats the RETE network as a DAG which is a task dependency graph for the matching phase of the algorithm. In this graph, every 'actor node' waits only for its predecessor node to send it information and it sends information to its successor.

3.3 Parallel firing:

The major work in parallel firing has been published by Ishida. His analysis is based on dependency graphs, for instance, in [11]. In other work, Nieman proposes a system of parallel rule firing which trades off improved performance with the fact that serializability might not be guaranteed for interfering rules in all situations. Further, interference analysis at run-time is expensive and this is done away with by introducing a locking mechanism on the working memory. In this context, 'UMass Parallel OPS5' has been prototyped which parallelizes the entire match-recognize-act cycle. Most work in parallel firing has been published exclusively by Ishida et. al., in [12], [13], [14], [15], [16].

The general approach in most of his work has been to build data dependency graphs between PEs and WMEs in order to perform interference analysis. Further these works also lay emphasis on separating compile-time and run-time analysis.

Chapter 4

OPS5-like production system in C

Most OPS5 like implementations have been experimented with in C-based or LISP-based languages. The choice of C as the language of implementation is because Open MPI has been used for message passing across systems and OpenMP has been used for shared memory multiprocessing on multiprocessors, and both these provide best support for C. Further, C is faster than most higher lever languages and it is in the interest of accuracy of speedup results that the overheads which come with using more abstracted language be kept minimal.

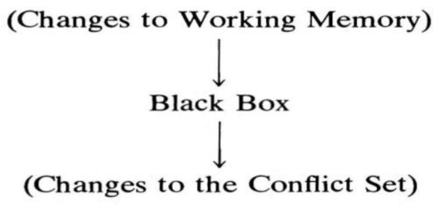
Chapter 5

Parallel rule matching

This section first discusses some details as to the nature of the RETE algorithm and why it is suitable to be parallelized. Some of the ideas in this section have been borrowed from [3].

5.1 State-saving nature of the RETE algorithm:

The RETE algorithm is state-saving in the sense that it saves the state of previous computations in order to avoid recomputing from scratch in every iteration. It is this characteristic of the algorithm that makes it better than non-state saving algorithms. According to an estimate by Gupta et. al in [3], non-state saving algorithms will have to overcome an inefficiency factor of around 120 if they have to compete with the time taken by a state-saving algorithm for OPS5 like production systems. The only input to the RETE algorithm is the changes to the WMEs and the only output from the algorithm is the changes to the conflict set.



For, instance let us assume we are talking about a particular production rule 'p' with condition elements CE1, CE2.. CEn. The algorithm saves state by storing WME tuples that match CE1, CE2… CEn individually for each CE. It also stores tuples that match certain combinations of CEs. For instance, one implementation of RETE stores tuples that match CE1 and CE2 jointly. This information is then used during the running of the algorithm to find tuples that match this combination along with CE3. This new set of tuples is then refined to select only those tuples that match CE4 too, and so on.

5.2 Why RETE is suitable for parallelism:

The algorithm is modeled on a graph-like structure, starting from a root node, branching out and recombining at different nodes. The data flow through this network lends itself to a parallel model- even a superficial examination would give one ideas of running the different branches on different processors, etc. Further, many previous simulations and studies show good speedups upon parallelizing RETE, and these results have been documented in [17], [18], [19], [20].

5.3 Parallelizing RETE:

From the match network in Fig (RETE graph), two design choices of parallelization arise:

1) **Parallelizing production matching:**
   In this model, the matching of different productions to produce instantiations is done parallely. Crudely, this would mean something to the effect that each parallel processing element would be allotted a path starting from the root node to the corresponding terminal production node in the network and would output the final tokens. The advantage of using such a model would be that it can easily be deployed on a generic distributed computing setup as the different production rules are independent of each other. This would mean that not much communication would be necessary between different computers and also that overheads would not be a bottleneck. The advantage of this model lies in the fact that it is not mandatory to have a multiprocessor shared memory architecture to deploy it. Further, one does not have to worry about memory contention issues.

2) **Parallelizing node-activations:**
   In this model, each node in the RETE network would be allowed to parallelize the processing of the tokens that pass through it. Further, it would also be able to parallely handle multiple tokens that have arrived at it, simultaneously. This would mean that multiple tokens would be flowing through the graph at the same time. A description of such a method can be found in [21].
   The advantage of using this approach can be seen upon observing some statistics and measurements regarding the number of rules fired in a production system on an average, the number of changes to working memory, etc. These have been elaborated upon in [21] and [22].

5.4 Choice of approach:

Parallelizing production matching is the approach that has been chosen. This is because it is easier to implement, can be modeled on more accessible hardware, is more generic in nature and is not dependent on the nature of the PEs and WMEs. The tradeoff, however, is that one can achieve theoretical lower time bounds by using the other approach discussed above.

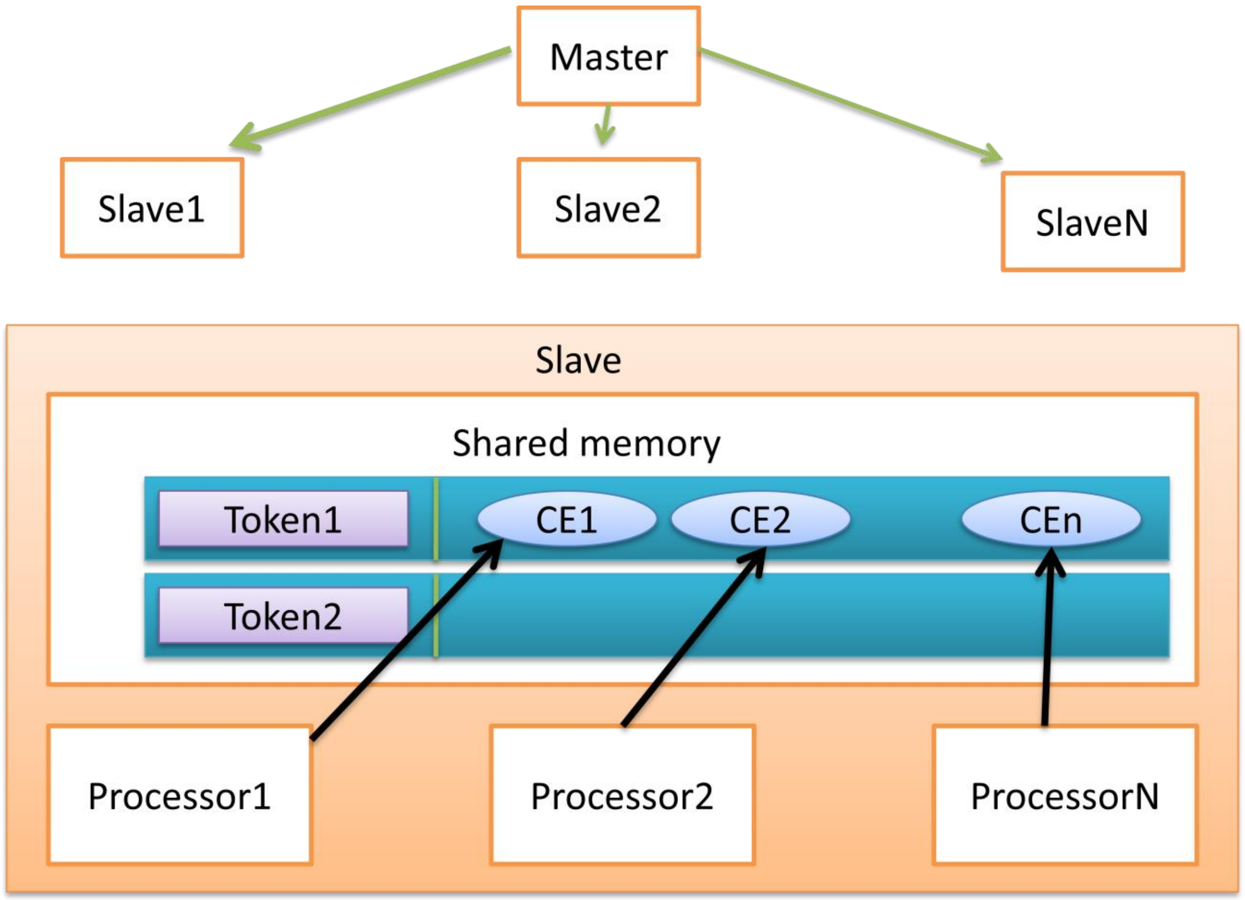
5.5 Details of parallelization:

- **Hardware**:
  - Intel Core i5-4200U CPU @ 1.60GHz 2.30GHz
  - Number of cores per system – 2
  - Number of systems available – 11
- **Language and APIs:**
  - C language
  - OpenMPI for message passing
  - OpenMP for shared memory multiprocessing
- **Definitions:**
  - Split-node: A node that either has two parents or more than two children. All other nodes have one or zero parents and one or zero children
  - No-split-path: A path which starts and ends with a split-node or ends with a terminal node, but does not have any split-nodes between the start and the end node

5.6 Parallel matching model:

- Distributed model with multiprocessing at each node
- Input: Set of changes to WME
- Output: Set of changes to conflict-set
- Initially, a copy of the RETE network is made on all the slaves, and the master

5.7 Schematic Diagram:



5.8 Algorithm:

Master (distribute tasks):

- Send current tokens to slaves
    - A slave is allocated a no-split-path, along with a set of tokens, and has to process the nodes between the start and the end on the no-split-path
- Receive set of filtered tokens from slaves
- Once the master receives the tokens from a slave:
    - Free the slave for further tasks

- If the slave has finished processing on a terminal node, then the tokens are put into the conflict set
- Else, if the slave has finished processing on a split-node:
  - If the node has 2 parents (Two-input node), wait for the other parent's input from the corresponding slave process. Once this input has arrived, allocate the no-split-path starting from this node to a free slave
  - If the node has more than one child, allocate free slaves to each child's no-split-path

Slave (run allocated task on shared memory):

- Master places tokens into shared memory
- Each token is given an incremental ID, Tid
- At each processor:
  - (Obtain-step) From shared memory, obtain the first unprocessed condition element (CE) of the current token (T) being processed
  - (Evaluate-step) Evaluate the CE as follows:
    - If the CE has multiple parts (an AND or an OR set):
      - Compare the Tid of the current token in the shared memory with the token being processed before evaluating each part
      - If there is a mismatch, reject the token and go back to Obtain-step. A mismatch indicates that the token has been rejected by another processor
    - If the token fails to match the criterion of a CE, delete the token from shared memory and go back to obtain step
  - If the CE obtained is the last unprocessed CE of the last token:
    - Perform evaluate-step with this CE
    - Communicate the contents of the shared memory back to the master
    - This will be the set of all tokens that have successfully met all the conditions on the no-split-path allocated to this slave

Note: Each CE in the shared memory has an associated status field which indicates if the CE is being processed by another processor or not.

5.9 Results of parallel matching

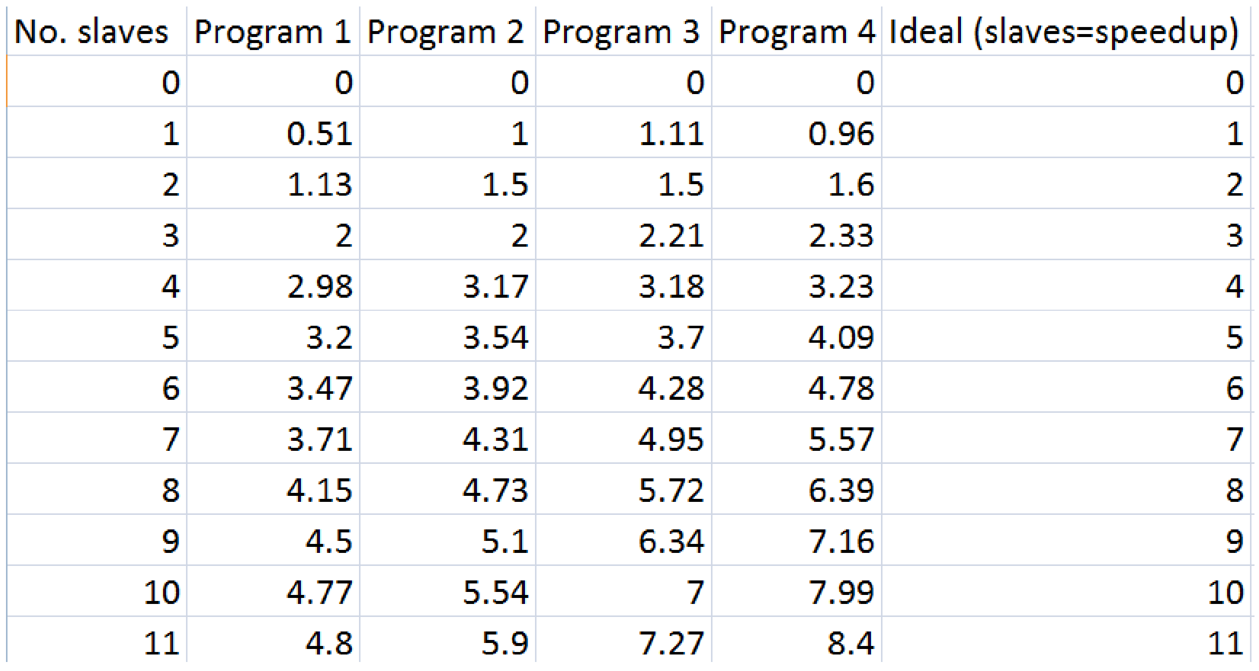
The following table shows the speedups gained by the parallel matching algorithm, as the number of slave nodes increases. These values have been obtained by using the formula:

Let $\mathbf{t_{seq}}$ = Time taken to run the sequential RETE algorithm

$\mathbf{t_{par}}$ = Time taken to run the parallel RETE algorithm

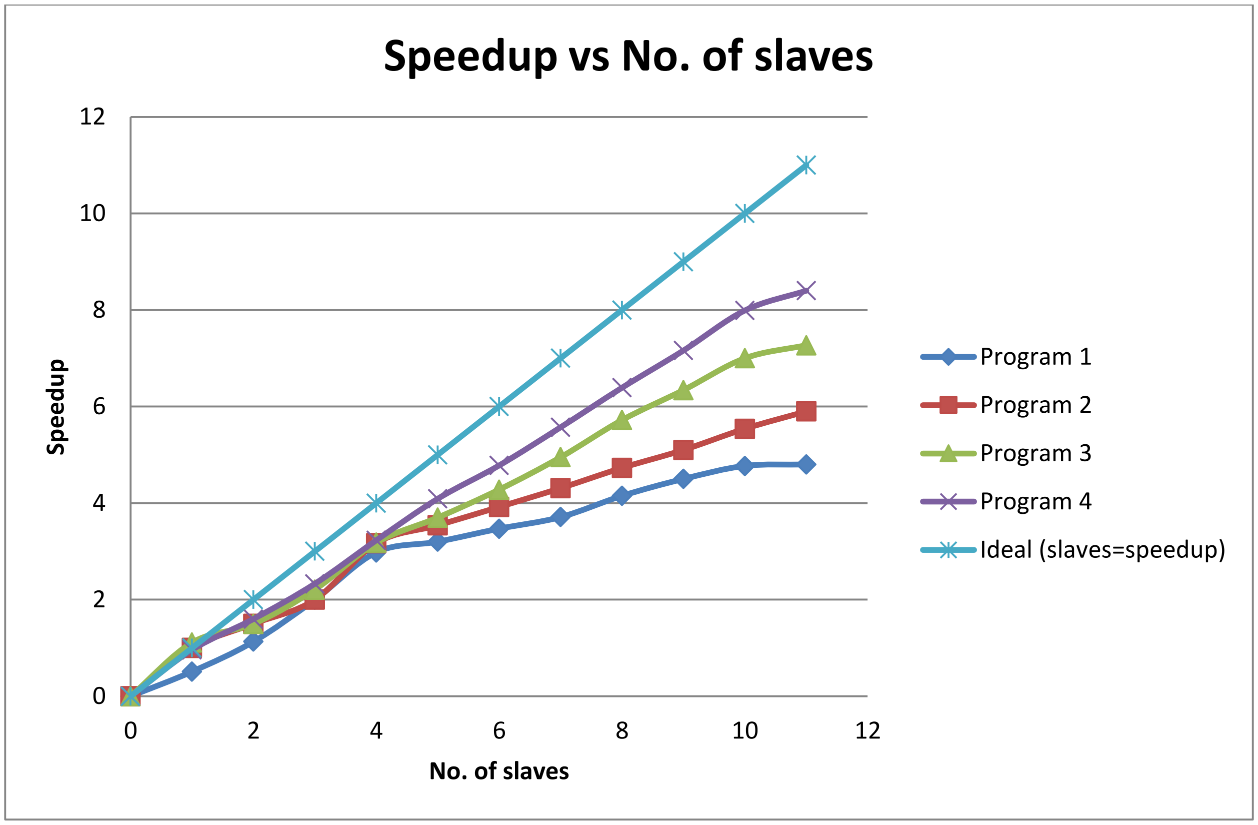
$$\textbf{Speedup} = \mathbf{t_{seq}} \,/\, \mathbf{t_{par}}$$

As the speedups are heavily dependent on certain metrics of the program, such as average number of rules, size of working memory, average number of firings per rule, average number of changes to working memory per cycle, etc., 4 different programs have been used to observe speedup trends. Efforts have been made to vary these metrics among the 4 programs in order to observe corresponding trends in speedup.



No. slaves	Program 1	Program 2	Program 3	Program 4	Ideal (slaves=speedup)
0	0	0	0	0	0
1	0.51	1	1.11	0.96	1
2	1.13	1.5	1.5	1.6	2
3	2	2	2.21	2.33	3
4	2.98	3.17	3.18	3.23	4
5	3.2	3.54	3.7	4.09	5
6	3.47	3.92	4.28	4.78	6
7	3.71	4.31	4.95	5.57	7
8	4.15	4.73	5.72	6.39	8
9	4.5	5.1	6.34	7.16	9
10	4.77	5.54	7	7.99	10
11	4.8	5.9	7.27	8.4	11

The following graph shows the above values graphically:



Chapter 6
Parallel Rule firing

The major work in this area has been published by Ishida et. al.., in [11] in which an a compile-time as well as a run-time analyzer checks dependencies between rules that can interfere with each other and makes sets of rules that can be fired parallely without conflict issues. However, this system has faced criticism because of its pessimistic assumptions about the number of rules conflicting with each other. This means that a very high degree of parallelism cannot be exploited by such an approach. Further, there is a differentiation between paired-rule conditions and all-rule conditions. Paired-rule conditions are easier to detect as they involve checking dependencies between pairs of rules, whereas all-rule conditions are more time consuming as they detect interference between all rules together, and this means performing the expensive task of finding connected components in a large graph of connections between PEs and WMEs. However, the catch is that paired-rule conditions, because of their limited exposure, forbid simultaneous firing of certain rules that in reality, can be fired together. These can be found only by applying all-rule conditions.

Due to the above-mentioned issues, an alternate approach has been adopted in this thesis. This approach has been borrowed from [7]. This model is a distributed model, with a master controller process (CP) and many slave processes (SPs). From the previous stage of the algorithm (parallel matching), the master receives the conflict-set, or more specifically, the set of all changes to the conflict set. Hence, the master now has a set of rules with the corresponding matches of WMEs that should instantiate the rules. In the Lana-model, these rules from the conflict set (also called 'master agenda (MA)'), combined with their corresponding WME matches are called 'activations'. The matching WMEs that are part of each activation are called 'facts'. The RHS that is fired upon using facts to activate rules is called 'action'. The working memory at any point of time is called a 'master fact list (MFL)'.

The working of the algorithm involves the master sending each activation along with its corresponding facts to a slave, which executes the action corresponding to that activation and returns the consequent changes to the WM, as tokens to the master. After ensuring that certain concurrency control is maintained, the master uses these returned tokens to update the MFL (or in other words, the WM). The activations that have been processed by slaves are deleted from the Master Agenda.

6.1 Additional data structures and implementations:

Master Agenda (MA): Contains activations, each pointing to the corresponding rule and the set of facts that activate it. Each activation also contains:

1) Time stamp of the time when the activation was added to the MA
2) Pointer pointing to the SP that the activation was allotted to, and set to NULL when not allocated to any SP yet
3) Pointer pointing to the Activation Commands Zone containing WM (or MFL) updation tokens generated by executing this activation at the SP

Action Commands Zone (ACZ): Buffer containing all the changes to MFL generated by a particular SP. This buffer is necessary and changes cannot be directly committed to the MFL by SPs, as concurrency control needs to happen.

Action Queue (AQ): Circular list, where each block in the list corresponds to an SP. A block consists of:

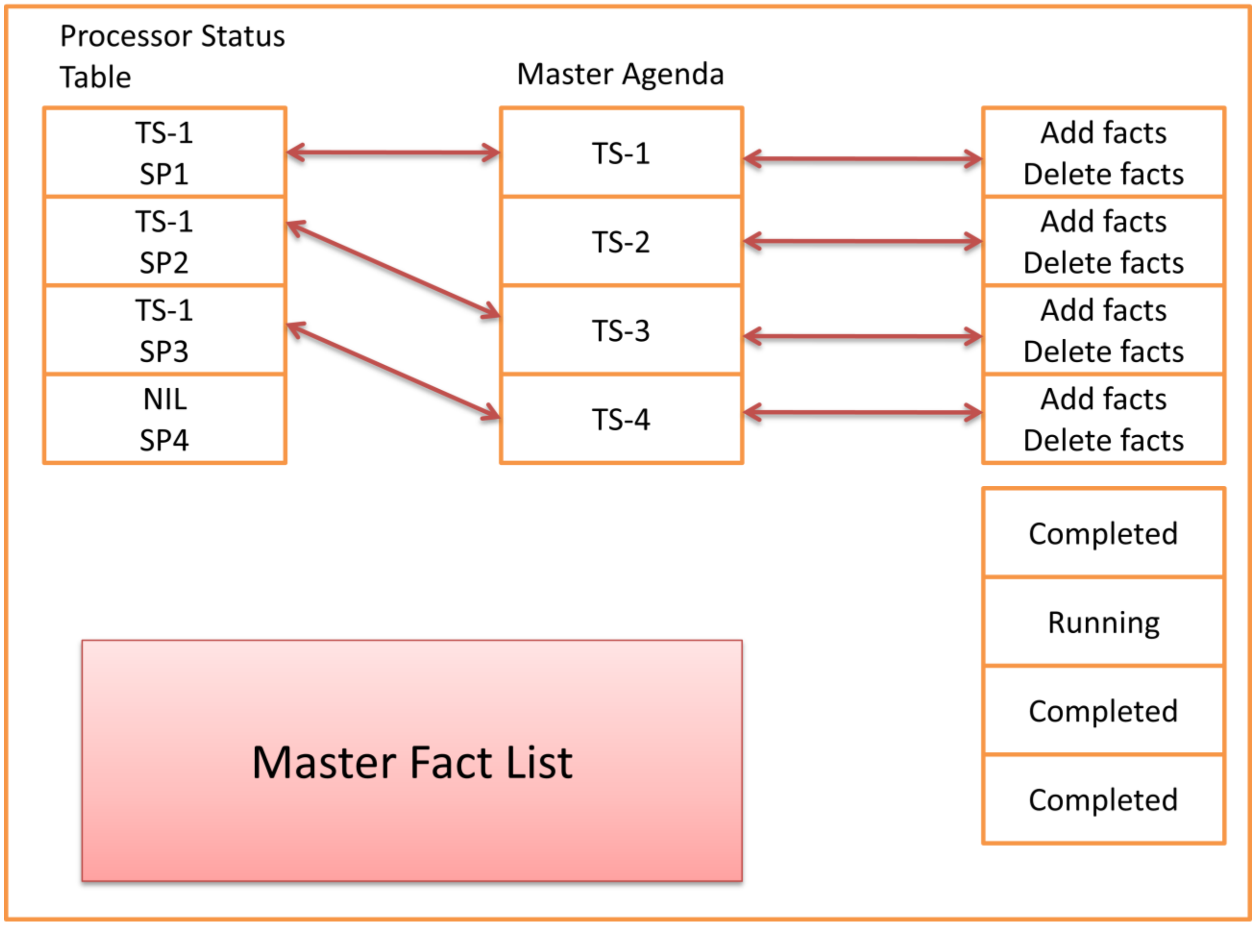
1) Pointer to corresponding activation
2) Status- ON when SP is executing activation and OFF when SP completes its execution

Processor Status Table (PST): Table holding statuses of SPs. The status is NULL if the SP is idle, else it contains:

1) Pointer to activation being executed
2) The activation's timestamp
3) Unique ID of the SP

6.2 Details of parallelization:

Schematic diagram:



Master side (this is executed on loop by removing each activation executed from the MA, and until the MA runs out of activations):

- Find the first free SP by scanning the PST. If none is available, run the rescheduler.
- If next activation is 'a' and free SP is 'p'
    - Point p to a and make the entry for p in the PST
    - Set the timestamp of the entry in the PST to the timetamp of a
    - Point a to p
    - In the input buffer of p, place all facts that led to the a becoming an activation
    - In AQ, set a's status to running

- **Rescheduler:**
    - Check the SPs' output buffers and compare timestamp of action command at output buffer with timestamp of the SP in the PST
    - If timestamps match
        - Implication: Correct message.
        - Place action command in the ACZ.
        - Point the SP to null
    - Else
        - Implication: Older activation deactivated this activation
        - Ignore this and all such message from this SP with wrong timestamp
        - If all output buffers are empty:
            - Implication: Idle time for master
            - Use idle time to perform check-and-commit action in the AQs
- **Check-and-commit:**
    - Assume the first block in the AQ is b
    - Do until b is a non-completed action, updating b to be the next block each time:
        - Let z be the set of all actions stored in the ACZ and pointed to by the activation that resides in block b
        - Execute all the actions in z
    - (the termination condition for this loop enforce committing of actions in the exact ordering in which they were created)

Slave side (run this loop forever):

- Upon request from master, fill in local facts (received from master) into input buffer
- Use facts to activate activations
- Fire activations and fill generated changes to WM in the output buffer
- Wait for request from master

6.3 Results of parallel firing

The following table shows the speedups gained by the parallel matching algorithm, as the number of slave nodes increases. These values have been obtained by using the formula:

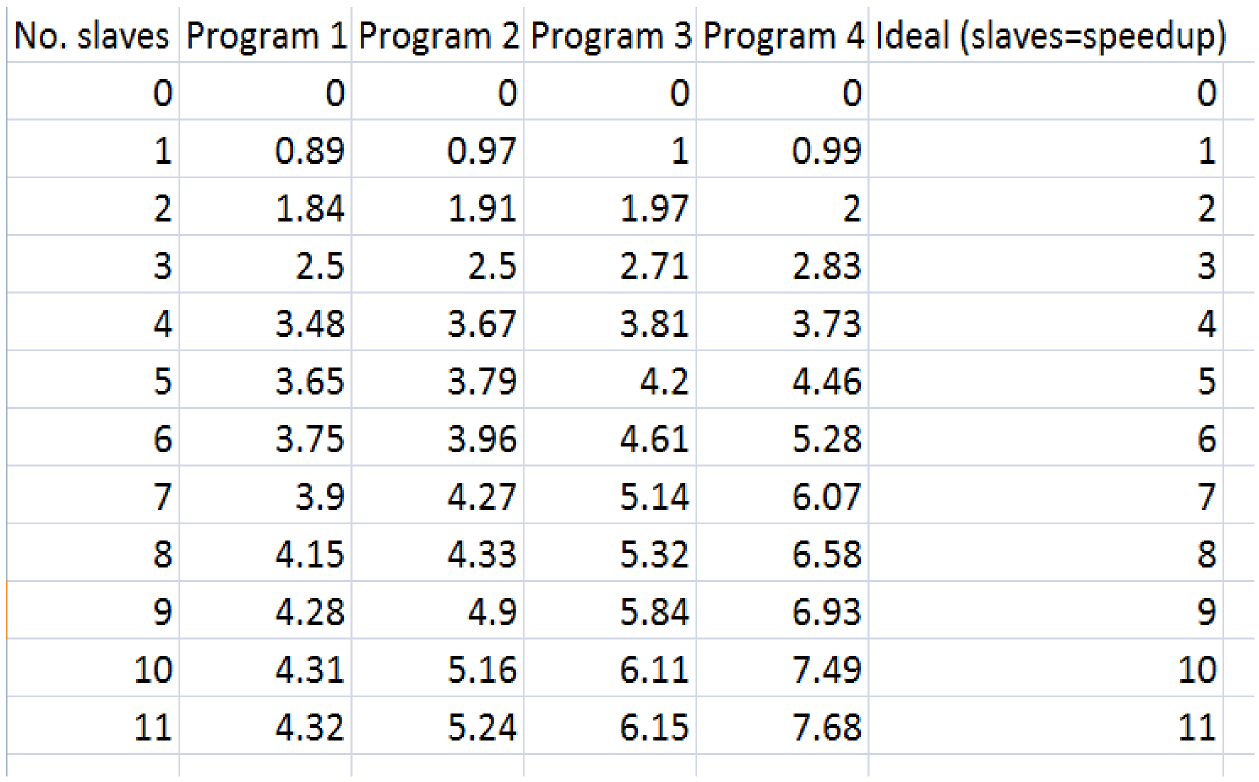
Let :

t_{seq} = Time taken to run the sequential firing algorithm

t_{par} = Time taken to run the Lana parallel firing algorithm

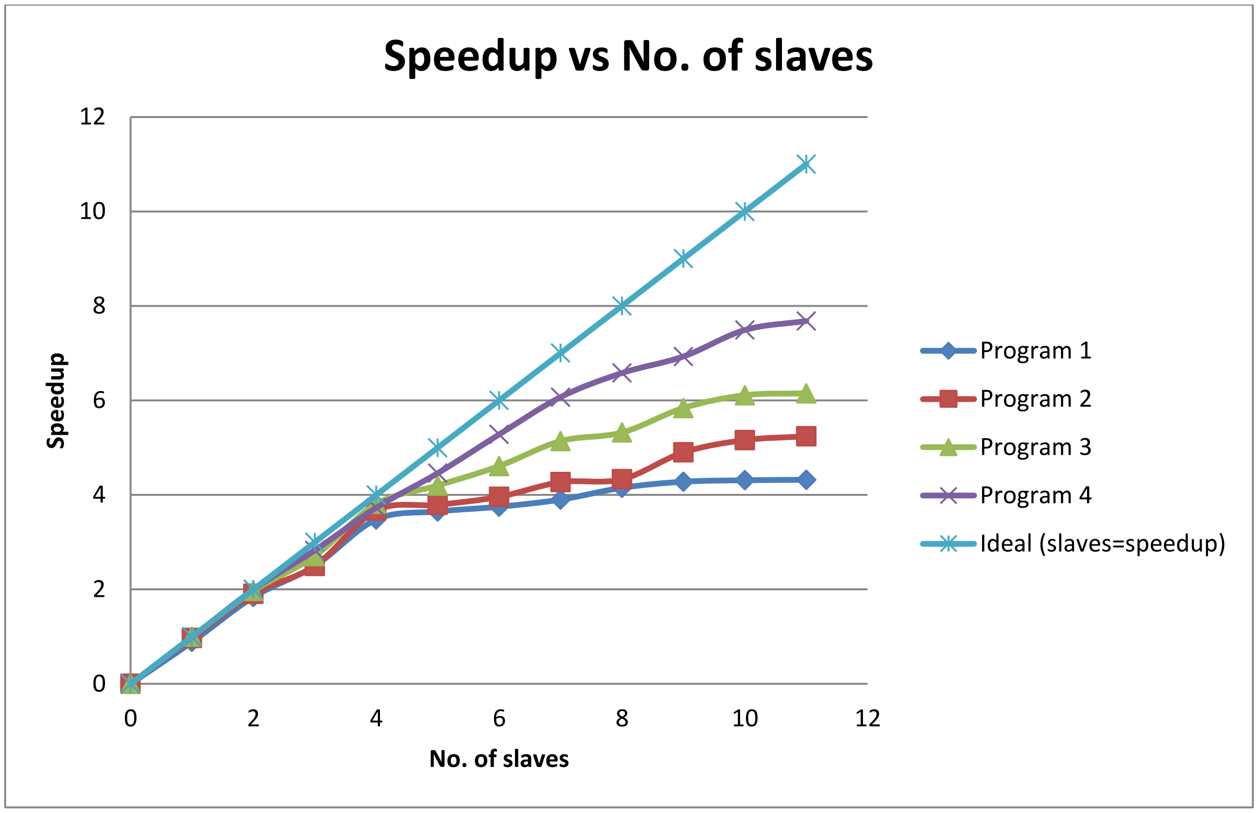
$$\textbf{Speedup} = t_{seq} / t_{par}$$

As the speedups are heavily dependent on certain metrics of the program, such as average number of rules, size of working memory, average number of firings per rule, average number of changes to working memory per cycle, etc., 4 different programs have been used to observe speedup trends. Efforts have been made to vary these metrics among the 4 programs in order to observe corresponding trends in speedup.



No. slaves	Program 1	Program 2	Program 3	Program 4	Ideal (slaves=speedup)
0	0	0	0	0	0
1	0.89	0.97	1	0.99	1
2	1.84	1.91	1.97	2	2
3	2.5	2.5	2.71	2.83	3
4	3.48	3.67	3.81	3.73	4
5	3.65	3.79	4.2	4.46	5
6	3.75	3.96	4.61	5.28	6
7	3.9	4.27	5.14	6.07	7
8	4.15	4.33	5.32	6.58	8
9	4.28	4.9	5.84	6.93	9
10	4.31	5.16	6.11	7.49	10
11	4.32	5.24	6.15	7.68	11

The following graph shows the above values graphically:



Chapter 7
Discussion and conclusions

The parallel matching gives speedups in the range of 4.5-8.5, when run on the specified hardware. The parallel firing gives speedups in the range of 4-7.5.

However the parallel firing speedups seem to saturate within the above specified range (the graph starts flattening out), while parallel matching speedups still show an upward trend, meaning they could show better speedups if the number of processing elements are increased. A combination of both these algorithms give results very similar to the parallel matching results. The reasons for the above are possibly because:

- Statistically, literature has shown that the matching phase is the bottleneck in the match-select-act cycle, accounting for over 90% of the time taken
  Let:
  T_{par_match} = total time taken for the parallel matching algorithm
  T_{par_fire} = total time taken for the parallel firing algorithm
  T_{seq_match} = total time taken for the parallel matching algorithm
  T_{seq_fire} = total time taken for the parallel firing algorithm

  Speedup = Total time taken for parallel matching and firing/
  Total time taken for sequential matching and firing

  $$= \frac{(Tseq_{-match} * 0.9) + (Tseq\_fire * 0.1)}{(Tpar\_match * 0.9) + (Tpar\_fire * 0.1)}$$

- Whereas the act phase is of lesser significance when it comes to the time taken, because the number of matching comparisons is far higher than the number of rules actually fired
- This is also obvious from the fact that the LHS of all rules have to be matched in every iteration, while the RHS of only the satisfied rules will be fired

These results are also very heavily dependent on the nature of the program being analyzed itself, and hence 4 sample programs have been presented, each with different firings per cycle, match rates, number of production rules and working memory sizes.

Chapter 8

Future scope

8.1 Parallel matching:

Finer parallelism can be achieved than what has been presented as part of the thesis. For instance, a condition element having AND or OR sub-elements can further be parallelized by submitting each part of the AND chain or the OR chain to a processor and preempting the other processors accordingly. Further, program rewriting can also be employed at compile time, where the AND conditions can be linearized into separate condition elements and be treated as separate nodes in the RETE network, while OR conditions can spawn separate branches in the network.

Given the availability of a multiprocessor system with a large number of multiprocessors connected in a bus-like fashion, with each processor having its own private cache and access to a common shared memory, then one can achieve node-level parallelism in the network, with each node evaluating multiple tokens in parallel. This might give better speedups, as the number of node activations scales proportional to the number of changes to the working memory. This would also eliminate the upper bound on speedups caused by large variations in the processing requirements of each production rule. Further, the algorithm would not be constricted to work on productions independently and if there are common test conditions that have to be checked among many productions, then the node checking these conditions would be evaluated only once and the results shared between all the production rules.

8.2 Parallel firing:

A more involved analysis of the rule interferences could be performed. This could mean approaches such as isolating the rules that can be fired parallely by representing rules and corresponding working memory elements as a graph, followed by finding strongly connected components in this graph as separately parallelizable rulesets.

References

[1] Forgy C. L., “Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem”, *Artificial Intelligence 19* , 1982.

[2] Kowalski, Thaddeus J., Levy, Leon S, “Rule-Based Programming”, pp. 6, 1996.

[3] Gupta A., Forgy C. L., Newell A., Wedig R., “Parallel Algorithms and Architectures for Rule-Based Systems” , ISCA '86 Proceedings of the 13th annual international symposium on Computer architecture, pp. 28-37, 1986.

[4] Cao B., Yin J., Zhang Q, Yanming, “A MapReduce-based architecture for rule matching in production system”, 2nd IEEE International Conference on Cloud Computing Technology and Science, 2010.

[5] Stolfo S. J., Miranker D. P., “Dado: A Tree-Structured Architecture for Artificial Intelligence Computation”, Annual Review of Computer Science, Vol. 1, pp. 1-18, 1986.

[6] Stolfo S. J., “Five parallel algorithms for production system execution on the dad0 machine”, AAAI-84 Proceedings, 1984.

[7] Aref M. M., Tayyib M. A.,“Lana–Match algorithm: a parallel version of the Rete–Match algorithm”, Parallel Computing 24, pp. 763–775, 1998.

[8] Peters M., Brink C., Sachweh S., Z¨undorf A., “Rule-based Reasoning on Massively Parallel Hardware” SSWS'13 Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems, Vol. 1046, pp. 33-48, 2013.

[9] Mahajan M., Kumar V. K. P., “Efficient parallel implementation of RETE pattern matching”, Journal of Computer Systems Science and Engineering, Vol. 5, Issue 3, pp. 187-192, 1990.

[10] Renaux T., Hoste L., Marr S., Meuter W. D., “Parallel Gesture Recognition with Soft Real-Time Guarantees”, Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, pp. 35-46, 2012.

[11] Ishida T., "Parallel Rule Firing in Production Systems", IEEE Transactions on Knowledge and Data Engineering, Vol. 3, Issue 1, pp. 11-17, 1991.

[12] Ishida T., "Parallel Firing of Production System Programs. *IEEE" Transactions on Knowledge and Data Engineering*, Vol. 3, No.1, pp. 11-17, 1991.

[13] Ishida T., Gasser L., Makoto Y., "Organization Self-Design of Distributed Production Systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 2, pp. 123-134, 1992.

[14] Ishida T., "An Optimization Algorithm for Production Systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 6, No. 4, pp. 549 - 558, 1994.

[15] Ishida T., *Parallel, Distributed and Multiagent Production Systems.* Lecture Notes in Artificial Intelligence 878, Springer-Verlag, 1994.

[16] Ishida T., Sasaki Y., Nakata K., Fukuhara Y., "A Meta-Level Control Architecture for Production Systems", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No.1, pp. 44-52, 1995.

[17] Gupta A., "Parallelism in Production Systems", *Ph.D. Th., Carnegie-Mellon University*, 1985.

[18] Hillyer B. K., Shaw D. E., "Execution of OPS5 Production Systems on a Massively Parallel Machine", *Journal of Parallel and Distributed Computing,* Vol. 3, Issue 2, pp. 236-268, 1984.

[19] Miranker D. P., "Performance Estimates lbr the DADO Machine: A Comparison of Treat and Rete. Fifth Generation Computer Systems", *ICOT*, 1984.

[20] Oflazer K., "Partitioning in ParalleI Processing of Production Systems", *Ph.D. Th., Carnegie-Mellon University*, 1985.

[21] Forgy C., Gupta A., "Preliminary architecture of the CMU production system machine" *Hawaii International Conference on Systems Sciences*, pp. 194-200, 1986.

[22] Forgy C., Gupta A., "Measurements on Production Systems", *Technical Report CMU-CS-83-167* , 1983.