

Parallelization of Rete-Match Algorithm for Distributed Memory Architectures

by

Mohammed A. Tayyib

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION AND COMPUTER SCIENCE

July, 1996

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

**A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600**



Parallelizing the Rete-Match Algorithm for Distributed Memory Architectures

BY

Mohammed A. Tayyib

A Thesis Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Information and Computer Science

JULY 1996

Parallelizing the Rete-Match Algorithm for Distributed Memory Architectures

BY
Mohammed A. Tayyib

**A Thesis Presented to the
FACULTY OF COLLEGE OF GRADUATE STUDIES**

**In Partial Fulfillment of the Requirements
for the degree
MASTER OF SCIENCE**

**King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
January 1997**

UMI Number: 1385298

UMI Microform 1385298
Copyright 1997, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**


UMI
300 North Zeeb Road
Ann Arbor, MI 48103


**KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN-31261, SAUDI ARABIA**

COLLEGE OF GRADUATE STUDIES

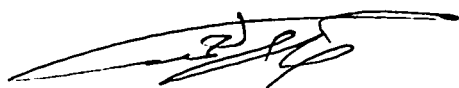
This thesis, written by Mohammed Abdui-Aziz Hassan Tayyib under the direction of his thesis Advisor and approved by his Thesis committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirement for the degree of MASTER OF SCIENCE.

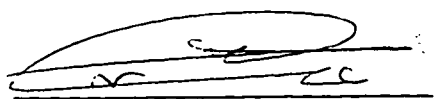
Thesis Committee


Dr. Mostafa M. Agef
Chairman


Dr. Subbarao Ghanta
Member


Dr. Muslim Bozyigit
Member


ICS Department Chairman


Dean, College of Graduate Studies



13 January 1997

ACKNOWLEDGMENT

Acknowledgment is due to King Fahd University of Petroleum and Minerals for support of this research. Acknowledgment is also due to the Exploration and Petroleum Engineering Computer Center (EXPEC) of Saudi Aramco for providing me with all the needed computing facilities to carry out this research.

I wish to express my appreciation to Dr. Mostafa M. Aref who served as my major advisor and the committee chairman who also played a major role in organizing this thesis. I also wish to thank Dr. Subbarao Ghanta for the excellent and interesting five graduate courses he taught me and the fruitful discussions we had over the past five years. These discussions had a major impact on my academic and professional life. Special thanks are also due to Dr. Muslim Bozyigit for his contribution as a thesis committee member.

Last but not least, I would like to express my thanks to all of my family and friends who gave me all the encouragement and support that I needed to carry out this research.

TABLE OF CONTENTS

	PAGE
LIST OF TABLES.....	vi
LIST OF FIGURES.....	vii
THESIS ABSTRACT	ix
CHAPTER 1: INTRODUCTION.....	1
CHAPTER 2: BACKGROUND.....	5
2.1 Production Systems	5
2.2 The Rete-Match Algorithm.....	7
2.3 Parallelism.....	12
2.4 Serializability & Concurrency Control.....	14
2.5 Previous work on Parallelizing of the Rete-Match Algorithm	17
CHAPTER 3: THE LANA-MATCH MODEL.....	24
3.1 The Lana-Match Computational Model Architecture	24
3.2 The Correctness of the Lana-Match Model	36
3.3 The Lana-Match Model Complexity Analysis.....	37
3.4 Implementing the Lana-Match Model using PVM and CLIPS	39
3.5 Advantages of the Lana-Match Model	42
3.6 Disadvantages of the Lana-Match Model.	44

LIST OF TABLES

	PAGE
TABLE 1: ACTUAL TIME MEASURED IN CASE 1.	52
TABLE 2 : SPEEDUP DATA CALCULATED FOR CASE 2.	52
TABLE 3: ACTUAL TIME MEASURED IN CASE 2.	57
TABLE 4 : SPEEDUP DATA CALCULATED FOR CASE 2.	57
TABLE 5: ACTUAL TIME MEASURED IN CASE 3.	62
TABLE 6: SPEEDUP DATA CALCULATED FOR CASE 3.	62
TABLE 7: ACTUAL TIME MEASURED IN CASE 4.	67
TABLE 8: SPEEDUP DATA CALCULATED FOR CASE 4.	67
TABLE 9: ACTUAL TIME MEASURED IN CASE 5.	72
TABLE 10: SPEEDUP DATA CALCULATED FOR CASE 5.	72

LIST OF FIGURES

	PAGE
FIGURE 1: AN EXAMPLE OF A RETE-MATCH NETWORK.....	9
FIGURE 2: A MORE COMPLEX RETE MATCH NETWORK.....	10
FIGURE 3: THE RETE-MATCH ALGORITHM OUTLINE.....	11
FIGURE 4: THE LANA-MATCH MODEL ARCHITECTURE.....	25
FIGURE 5: THE LANA-MATCH MODEL DATA STRUCTURES.....	29
FIGURE 6: THE LANA-MATCH MODEL OUTLINES (PART 1).....	33
FIGURE 7: THE LANA-MATCH MODEL OUTLINES (PART 2).....	34
FIGURE 8: THE LANA-MATCH MODEL OUTLINES (PART 3).....	35
FIGURE 9: EXAMPLE OF CASE 1.....	49
FIGURE 10: TOTALLY INDEPENDENT LIGHT ACTION RULES, ACTUAL MEASURED TIME	50
FIGURE 11: TOTALLY INDEPENDENT LIGHT ACTION RULES, SPEEDUP.....	51
FIGURE 12: EXAMPLE OF CASE 2.....	54
FIGURE 13: TOTALLY INDEPENDENT HEAVY ACTION RULES, ACTUAL MEASURED TIME	55
FIGURE 14: TOTALLY INDEPENDENT HEAVY ACTION RULES, SPEEDUP.....	56
FIGURE 15: EXAMPLE OF CASE 3.....	59
FIGURE 16: ENABLING DEPENDENT RULES WITH ADDITIONS ONLY, ACTUAL MEASURED TIME.....	60

FIGURE 17: ENABLING DEPENDENT RULES WITH ADDITIONS ONLY, SPEEDUP.	61
FIGURE 18: EXAMPLE OF CASE 4.....	64
FIGURE 19: ENABLING DEPENDENT RULES WITH LOW ADDITIONS AND DELETION , ACTUAL MEASURED TIME.	65
FIGURE 20: ENABLING DEPENDENT RULES WITH LOW ADDITIONS AND DELETION , SPEEDUP.	66
FIGURE 21: EXAMPLE OF CASE 5.....	69
FIGURE 22: ENABLING DEPENDENT RULES WITH HIGH ADDITIONS AND DELETION , ACTUAL MEASURED TIME.	70
FIGURE 23: ENABLING DEPENDENT RULES WITH HIGH ADDITIONS AND DELETION , SPEEDUP.	71

THESIS ABSTRACT

NAME: MOHAMMED ABDUL-AZIZ HASSAN TAYYIB

**TITLE: PARALLELIZING THE RETE-MATCH ALGORITHM
FOR DISTRIBUTED MEMORY ARCHITECTURES**

MAJOR FIELD: COMPUTER SCIENCE

DATE: January 1997

The Rete-Match algorithm is the main algorithm that is used to develop Production Systems. A Production System is a knowledge representation scheme in the Artificial Intelligence. 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 thesis, a parallel version of the Rete-Match algorithm for distributed memory architectures is designed, implemented, and analyzed. The implementation of this parallel version accomplished considerable speed up with respect to the number of processors over the sequential execution of the Rete-Match algorithm.

MASTER OF SCIENCE DEGREE

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

January 1997

خلاصة الرسالة

إسم الطالب : محمد عبدالعزيز حسن طيب
عنوان الدراسة : تسريع الخوارزم مطابق-ريتا باستخدام
العديد من الحواسيب ذات الذاكرة المنفصلة
التخصص : علوم الحاسب الآلي.
تاريخ الشهادة : رمضان ١٤١٧هـ - يناير ١٩٩٧م.

يلعب الخوارزم مطابق-ريتا دوراً بارزاً في مجال النظم المنتجة والتي تعتبر فرعاً من فروع الذكاء الإصطناعي، وبالرغم من كون الخوارزم مطابق-ريتا أسرع وسيلة معروفة ومستخدمة للبحث ومطابقة عناصر متعدد، إلا أن تنفيذ هذا الخوارزم على الحاسبات الآلية التقليدية يستغرق وقتاً ليس بالقليل نظراً لطبيعة المشكلة. وفي هذه الرسالة، نقدم نموذجاً مطوراً من هذا الخوارزم والذي يعتمد على استخدام العديد من الحواسيب الآلية ذات الذاكرة المنفصلة. ومن ثم تصميم، تنفيذ وتحليل النتائج. ولقد أثبتت جميع الدراسات والتحليل في هذه الرسالة إن هذا النموذج الجديد من مطابق-ريتا يؤدي أداءً متميزاً في حالة كون النظم المنتجة تتألف من قواعد تحتاج إلى وقت أكثر من الحواسيب لتنفيذها وكذلك كونها غير معتمدة على بعضها في التنفيذ، هذه الخاصية تجعل هذا النموذج المطور أكثر كفاءة وفاعلية من معظم المحاولات السابقة لتسريع الخوارزم مطابق-ريتا.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران - المملكة العربية السعودية

رمضان ١٤١٧هـ - يناير ١٩٩٧م

CHAPTER 1

INTRODUCTION

The Rete-Match algorithm is one of the main algorithms in the Production Systems field that belongs to the Artificial Intelligence. 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 thesis, a parallel version of the Rete-Match algorithm for distributed memory architectures is designed, implemented, and analyzed. This study is targeted to be a practical approach to design and implement a reliable, heterogeneous, an adaptive and efficient message passing version of the Rete-Match algorithm. This new design is also targeted to utilize run time chances for to exploit 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 new parallel version of the Rete-Match algorithm for distributed memory architectures that is presented in this thesis is being referred to as the Lana-Match model. The Lana-Match Model is an optimistic, message passing, parallel version of the Rete-Match algorithm that was specially designed to utilize the power of the parallel distributed memory architecture machines and to address the following shortcoming that was found with most of the current research:

1. Most of the work that has been reported 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) that they use to control the concurrent matching or concurrent rule firing are based on the assumptions that 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 very high degree of parallelism and also prevented them from achieving major speed up.
2. Most of these approaches were either based on shared memory architectures or were based on special topologies (i.e. tree). That is why most of the reported results (nine out of twelve) were based on simulated solutions. How well these simulated ideas perform in practice remains to be seen.

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 for a Controller Processor that maintains a Master Agenda and a Master Fact List. 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 either to be added or deleted from the Master Fact List back to the Controller. The Controller buffers the slaves 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 Master Fact List. The final conclusion will be found at the Master Fact List.

The Lana-Match model was implemented using the C Language Integrated Production System (CLIPS) and the Parallel Virtual Machine (PVM). This implementation shows a proportional speedup with respect to the number of processors. This speed up and the adaptive feature of the Lana-Match represent major improvement for the performance of the Rete-Match algorithm. Moreover, the parallel version of this algorithm that is presented in this thesis could be used as a good starting point to build real cooperating expert systems which are gaining wide acceptance and interest from major Artificial Intelligence researchers.

Chapter-2 of this thesis covers the basic concepts and the key terminology that would be needed to follow the rest of this thesis. It starts with a brief description of the production system model. This is followed by a detailed description of the Rete-Match algorithm. Next, the main key concepts and terminology for parallel computing are described followed by a summary of the main concurrency control mechanisms. Finally, this chapter is concluded with a comprehensive survey of all the previous approaches that were reported toward parallelizing the Rete-Match algorithm.

Chapter-3 contains detailed description of the architecture of the Lana-Match model followed by a comprehensive analysis of its correctness and complexity.

Finally, the advantages and the disadvantages of the Lana-Match model are summarized.

Chapter-4 summarizes comprehensive analysis for the performance of our implementation of the Lana-Match model. Chapter-5 contains the major contributions followed by a list of suggested future research topics on this subject.

CHAPTER 2

BACKGROUND

This chapter starts with a brief description of production system followed by a detailed description of the Rete-Match algorithm. Next, the main key concepts and terminology for parallel computation and Concurrency Control Mechanisms (CCM) are described. This chapter is concluded with a comprehensive survey of the previous approaches that were reported toward parallelizing the Rete-Match algorithm.

2.1 Production Systems

The term "Production System" is a heavily used term in Artificial Intelligence. Unfortunately this term may mean different things to different people. For example, Production System is a knowledge representation scheme to knowledge engineers while it is a programming language (like OPS-2 & OPS-5) for AI programmers. On the other side of the spectrum, theoreticians define production system as an abstract computation model. In this thesis, we use the term production system as a programming paradigm that is being adapted by many contemporary AI programming languages (i.e. OPS-5, CLIPS,...). This paradigm is based on the original model that

have been proposed first time by Post [5] and have undergone theoretical and application-oriented developments [5,18]. It consists of three major parts:-

1. A Rule Base is composed of a set of production rules where every rule has 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 contains lists of condition elements which 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 which could require another match.
 - [Repeat] GOTO step (a).

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 pattern. The main function of the match step is

to find the set of all legal instantiations of conditions which is called the conflict set.

The conflict resolution step decides whether the execution of the production system should halt, and then if not, it chooses one rule to be executed in the act step. There are several conflict resolution strategies. These include Depth, Breadth, Simplicity, Complexity, LEX, MEA, and Random strategies [19].

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 [24]. To get a feeling for the complexity of the matching processor, imagine a production system consists of 2000 productions and 4000 facts, where each production has 5 condition elements. This production system will perform $(4000 \times 2000 \times 5)$ matching operations at each execution cycle. The Rete-Match algorithm was first described in [19]. A complex version of this algorithm that includes an interpreter which delays evaluation of pattern as long as possible is presented in a similar report [24]. An efficient implementation of the algorithm is presented in [18-19]. 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 [19].

As in Figure 1, the Rete-Match algorithm compiles the condition elements of the production rules into a tree like network. This network is referred to as the Pattern Network. Every leaf of this tree has a list of pointers that points to the facts that matches that pattern. This list of pointers is referred to as the Alpha-Memory. The other network that joins the leaves of the pattern network to form productions is referred to as the Join Network. Every node of the join network has a list of pointers that point to the facts that match the pattern at that stage of the matching. This list is referred to as the Beta-Memory. The Rete-Match builds these two networks only once from the LHS of all the productions as these productions are being loaded into the system. All the LHSs of all the productions will be discarded after the networks are built. A more complex example network is given in Figure 2.

As in Figure 3, the pattern network and the join network form what can be described as a black-box system where all changes to the working memory pass to this black-box in the form of adding and deleting facts instructions from one side of the box which produce some more facts to be added or deleted from the system from the other side of the box. The system will be ignited by initial facts entering the system. Facts will be coming in and out of this box till the system reaches the equilibrium state in which there is no fact entering in or out of this box [60]. The

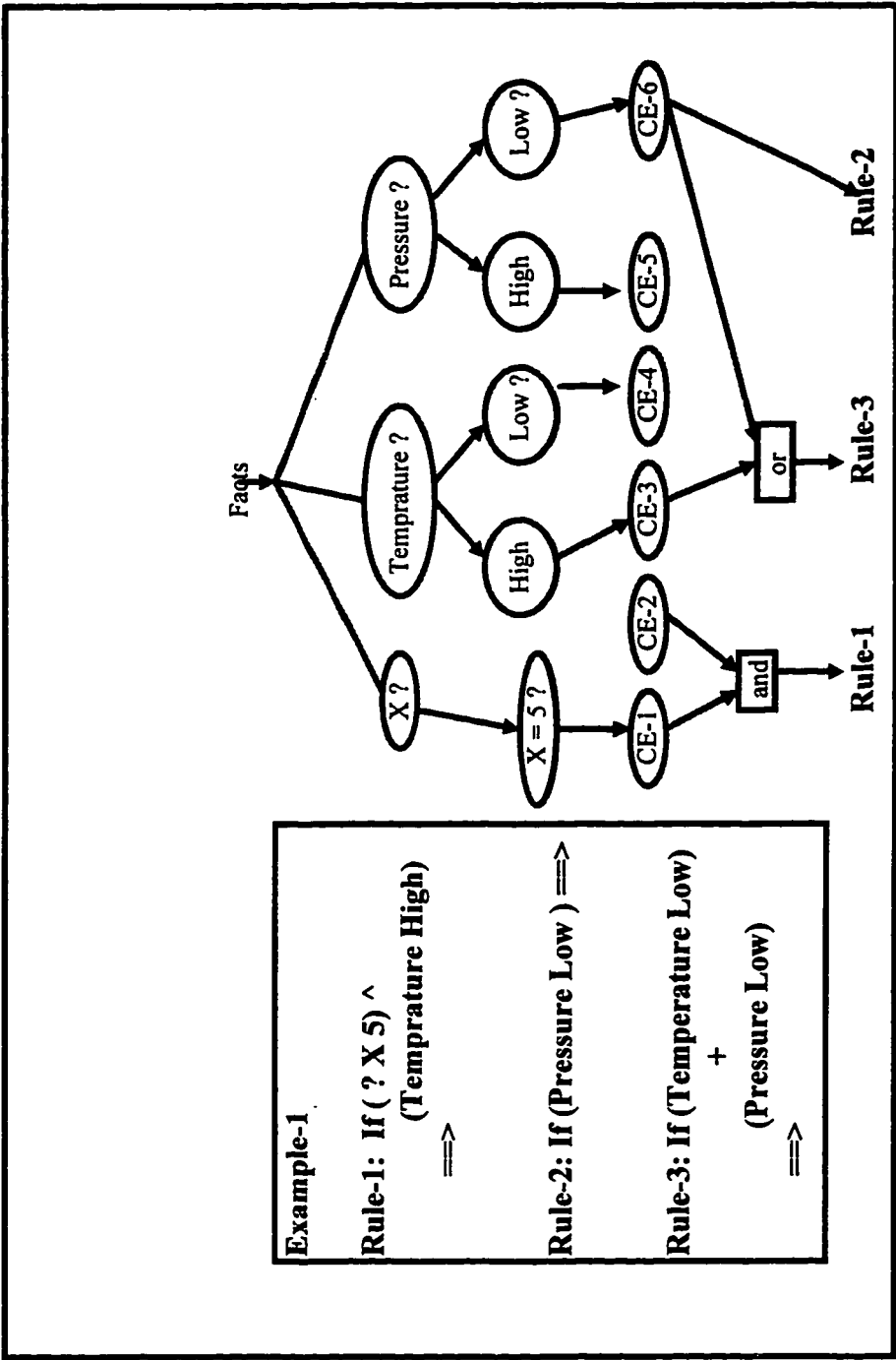


Figure 1 : An example of a Rete-Match Network

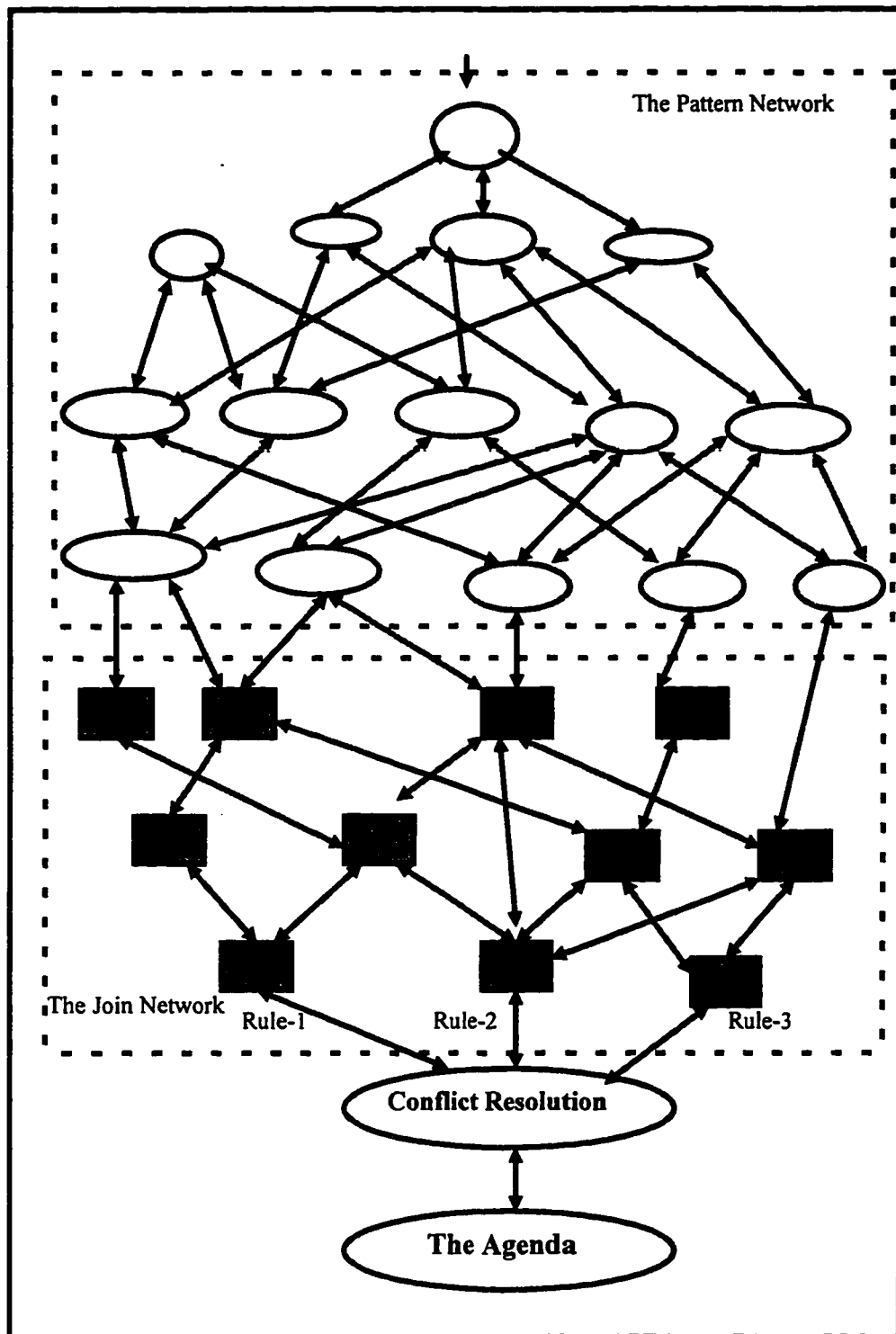


Figure 2: A More Complex Rete Match Network.

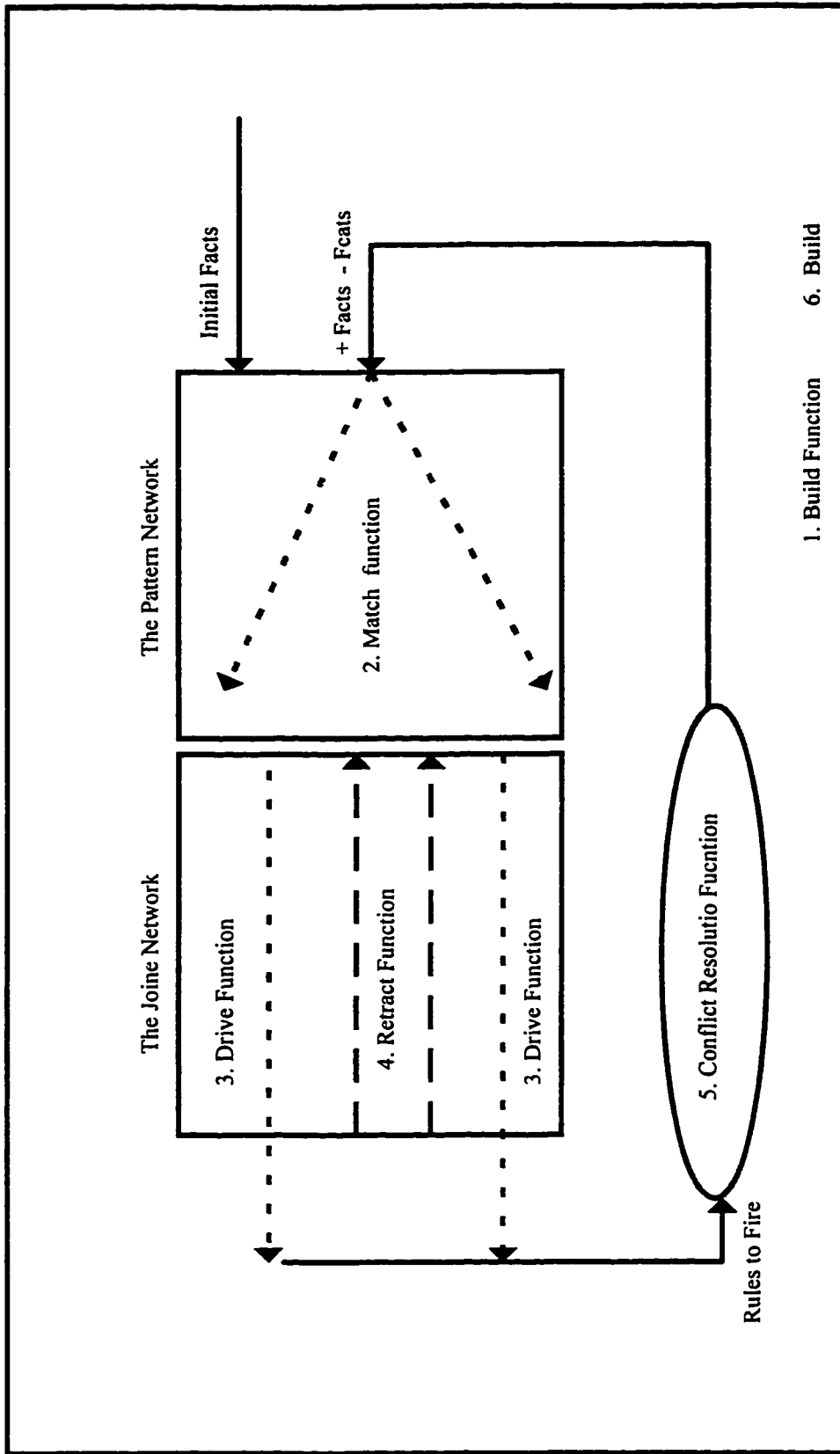


Figure 3: The Rete-Match Algorithm Outline

behavior of these functions make what is being referred to as the Rete-Match algorithm. Appendix A contains detailed English like description of the Rete-Match Algorithm in terms of description for the Build, Match, Drive, Retract, Conflict resolution and the Engine functions.

2.3 Parallelism

Dramatic increase in computing power were achieved over the past forty years. Most of it was largely due to the use of inherently faster electronic components. Unfortunately, it is evident that this trend will soon come to end because sequential computers may be approaching a fundamental physical limit on their potential computational power [3]. The limiting factor is speed of light in the vacuum. It appears that the only way around this limit (upper-bound) is to use parallelism. The idea here is that several operations are performed simultaneously, then the time taken by a computation can be significantly reduced. This is a fairly intuitive concept, and one to which we are accustomed in any organized society. We know that several people of comparable skills can usually finish a job in a fraction of the time taken by one individual. From mail distribution to harvesting and from office to factory work, our every day live offers numerous examples of parallelism through task sharing. During 1979 to 1996, parallelism has become truly attractive and a viable approach for the attainment of very high computational speeds. The arguments over single von Neumann computers versus multiprocessors, not long ago, were settled. Parallel

computing has always been criticized by referring to Grosch's law, Minsky's conjecture, and Amdahl's law [9]. Grosch's law states that the computing power of a single processor increases in proportion to the square of the cost. But this law is no longer applies to computer systems built with many inexpensive VLSI processors. Minsky's conjecture claims that because of the communications overhead between n processors, the actual performance of the parallel computer should be approximated by $\log_2 n$ instead of n . But this conjecture was formulated in the late 1960s based on experience with systems containing few processors. Only present experience overwhelmingly indicates that the actual loss in performance is much less than the loss predicted by Minsky's conjecture.

Amdahl's law states that the improvement in performance of a parallel algorithm over a corresponding sequential algorithm is limited by the fraction of algorithm that can not be parallelized. But Sandia National Laboratories has recently confirmed that almost linear speed-up has been achieved on 1024 processors hypercube for a number of practical problems. That's why, in recent years, parallel computing has grown to the point where computers of various types are both commercially available and potentially cost effective tools in a number of application areas. Appendix B contains some of the main characteristics of parallel and distributed processing.

2.4 Serializability & Concurrency Control

At the moment we think of having more than one agents to accomplish a task by sharing resources, a series of issues will start showing up to be addressed. No matter what subject domain we are talking about these issues will be similar in nature but may have different flavor at each domain. Serializability and concurrency control are the main two issues that have been studied extensively in the operating system and database subfields of the computer science discipline. In this section, we will approach this topic from the database point of view only. This is mainly due the natural similarity between knowledge and database systems. The following are definitions for few terms that will help us clarify this topic [61].

1. **Transaction:** A single executing program that would read and write data to and from the database, into a private work space, where the computation is performed. These computations will not effect the database until new values are written in to the database.
2. **Atomicity:** An atomic operation either occurs in its entirety or does not occur at all, and if it has occurred, no other operation went on during the time of its occurrence.
3. **Item:** The unit of data to which access is controlled. A relation, a tuple or a field are examples of such units.
4. **Schedule:** A schedule is the interleaved order of execution of transactions.

5. **Serialization:** The scheduling mechanism which forces transactions to run concurrently in a way that makes it appear as if they ran one at a time (serially).
6. **Serializability:** A schedule S is said to be serializable if the state of the database after the successful completion of all of its interleaved transactions will always be the same as if these interleaved transaction were executed in some serial fashion. Serializability is the most widely accepted correctness criterion for concurrency control mechanisms.
7. **Concurrency Control Mechanisms:** There is a large number of concurrency control algorithms and proposals in the literature. These mechanisms can be classified as:
 - **Pessimistic Concurrency Control:** it is based on the assumption that many (if not all) transaction will conflict with each other. Thus it will not permit a transaction to access a data item if there will be a conflicting. Which means that the execution of any operation of a transaction follows the sequence phases: validate, read, compute and write. (only update operation is considered here because its the one that causes inconsistency problems).
 - **Optimistic Concurrency Control:** it is based on the assumption that not too many transaction will conflict with one another. Thus it will permit a transaction to access a data item even if there is a conflicting transaction. Which means that the execution of any operation of a transaction follows the sequence phases: read, compute, validate and write. Although the pessimistic approach is the safest which guarantee Serializability

conditions, the optimistic approach has great potential to allow a higher level of concurrency.

- **Locking based Concurrency Control:** the main idea of locking-based concurrency control is to ensure that a shared item is accessible only by one operation at a time. This is accomplished by associating a "lock" with each item. This lock will be set by one of the transactions to prevent any other transaction from accessing it and will be reset by one of the transactions at the end of its use. There are two types of lock, Read locks and Write locks. Two Locks are said to be compatible if and only if both of them are Read locks. Only compatible locks will be allowed to be executed concurrently. Although, the basic locking mechanism guarantee mutual exclusion of accessing the shared item by conflicting transaction, it does not guarantee Serializability. With some modifications to the basic locking mechanism, Serializability conditions can be achieved. Two Phase Locking (2PL) rule is one of the modifications. This rule requires that the transaction should not release any of its locks until it is certain that it will not require any other lock. Any schedule generated by 2PL concurrency control mechanism is serializable [61].
- **Time stamp based concurrency Control:** Time stamp concurrency mechanisms do not maintain Serializability by mutual exclusion. Instead they select a prior serialization order and execute transactions accordingly. This is established by assigning each transaction a unique time stamp at its

initiation. Uniqueness and monotonicity are the two main properties of time stamp generation.

- Hybrid Concurrency Control: It is a combination of the Lock and time stamp based concurrency control mechanisms.

2.5 Previous work on Parallelizing of the Rete-Match Algorithm

The research efforts toward parallelizing the Rete-Match algorithm can be classified into the following three main categories [32].

2.5.1 Speeding up the match phase by faster sequential algorithm

Charles Forgy [18], encouraged researchers to work on Parallelizing this algorithm as he was concluding his PHD. thesis "On the Efficient Implementation of Production System" where he first introduced the Rete-Match algorithm. He also pointed that since the Rete-Match algorithm is a memory limited, this algorithm is not amenable to uniprocessor techniques like putting more data paths in the machine, pipelining the processors, or using special functional unit. His one page conclusion on Parallelizing this algorithm was the first road sign.

Miranker, D.P. [42] proposed a new version of the Rete-Match algorithm that he called TREAT. This algorithm resolves the ineffective network updates procedure that is used by Rete. In TREAT, a modify action is still implemented as a deletion and an addition, but the state of the network is stored differently. In TREAT network, the

memory nodes are eliminated. When a new fact is added, an exhaustive search through the network is made to determine the new rule binding. Minker showed that in many cases the speed-up obtained in deletion is greater than the loss in addition. Thus he concluded that TREAT obtain better performance than Rete. Minker and his group at the University of Texas at Austin have completed a new, highly optimized, C-based version of OPS-5 compiler. They reported 4-15-Fold speed-up over the Rete-Match algorithm.

Highland proposed a new version of Rete-Match algorithm that he called YES/RETE [32]. The modify action is no longer implemented as deletion followed by an addition. Instead, a new update-in-place operation is used. The update-in-place operation directly changes the attributes of a fact. For the instantiation that continue to exist after a modify action, it does not cause a network update. Only those instantiation that are affected by the modify action are updated. The YES/RETE algorithm has been implemented in KnowledgeTool and HiPER both from IBM [32]. The main difference between KnowledgeTool and HiPER is in how the program is compiled. HiPER achieve 3-11-fold speed-up over KnowledgeTool and regular Rete-Match algorithm.

2.5.2 Speeding up the match phase by parallel algorithm

Charles Forgy [24] presented a parallel version of the Rete-Match algorithm on a SIMD machine (Illiac-IV). The idea of this algorithm can be outlined as follows. First, divide the set of rules into 64 partitions, corresponding to the number of

processors in Illiac-IV. Second, for each partition, construct the Rete-Network. Third, execute the Rete-Match code sequentially on each node.

Annop Gupta [24] (a student of Charles Forgy), presented a paper on Parallelizing the Rete-Match algorithm on the DADO machine. DADO [57] is a highly parallel tree-structured architecture designed to execute production system at Columbia University by Salvatore J. Stolfo and his colleagues. It consists of a very large number (tens of thousands) of processing elements, interconnected to form a complete binary tree. Annop Gupta extended Charles Forgy parallel version of the Rete-Match to the DADO prototype machine and predicted to be able to processor 125 facts/seconds. The DADO machine was just a prototype. Reports on its performance was given by Stolfo [57, 58]. Speed up of 2-31-fold has been reported.

Researchers at Honeywell CSC proposed a parallel version of Rete-Match algorithm that is based on translating the Rete-Network into a data-flow graph that explicitly shows the data dependencies. Similarly, operations performed in the Rete algorithm are encapsulated into appropriate activities or tasks in the data-flow model which can then be executed on the available physical processing resources [24]. No performance evaluation is available.

Kemal Oflazer [24] in his thesis showed that the task of partitioning production system so that work is uniformly distributed is an NP-complete problem. He also presented a more complex heuristic method for partitioning that relies on data obtained from actual production system runs. The second part of his thesis proposes a

new parallel algorithm for performing match for production system and proposed a parallel architecture to execute it.

The Rete-Network was implemented using a similar kind of dataflow architecture PESA-I which consists of 32 special purpose processor. The simulator was built using Pascal. Only small production systems programs had been simulated. The simulated results showed that PESA-I obtained a rate of 8000 rule firing per seconds [51].

Anoop Gupta [24] explored parallelizing Rete-Match based production system programs on a simulated 32-64 shared-memory multiprocessor machine and reported the following results:-

1. The Rete-Match algorithm is highly suitable for parallel implementation.
2. The amount of speed-up available from parallelism is quite limited, about 10-fold.
3. To obtain the limited speed-up, it is necessary to exploit parallelism at a very fine granularity.
4. To exploit the suggested source of parallelism, a multiprocessor architecture with 36-64 high performance processors and special hardware support for scheduling the fine-grain task is desirable.

He concluded his thesis with the following points. His results are based on the analysis of existing programs (1980-1987) that were written with sequential implementation in mind which did not reflect the true parallelism which is to be found in programs written with parallel implementation in mind. However, he does not expect that this would have a major impact on his results.

Gupta [25] and his colleague implemented a parallel version of the Rete-Match on the Encore Multimax shared-memory multiprocessors with 16 CPUs. This approach exploits very fine grained parallelism and speed-up of 2-11-fold was achieved using 13 processor. The Rete-network was partitioned at the node level to obtain fine grain parallelism.

A simulated version of the Rete-match suitable for message-passing computing have been proposed by Acharya, Gupta, and Tambe [1], It has been simulated on the Nectar simulator, which is a message-passing computer with low message overhead. The simulation results indicated that speed 2-12-fold were achieved for three programs.

Kelly and Seseviora [30,31] have proposed a distributed version of the Rete-Match algorithm that they called Drete on a special machine CUPID that was designed to maximize the performance of Drete. The CUPID has not been implemented. It was only simulated on the CUPID simulator. The simulation indicated speed-up similar to Gupta's thesis.

2.5.3 Speeding up Production Systems by Multiple Rules Firing

Pakis [47] simulated a new programming methodology called IRIS to reduce the software complexity and to improve the parallelism in production system. IRIS reduces the software complexity by eliminating the explicit control. The simulations show that 6-90-fold speedup is possible to be achieved using the IRIS methodology.

Ishida [28] has proposed a simulated parallel programming environment consisting of analyzer that will determine the inter-instantiation data dependencies and a set of parallel language construct (i.e. rule-set and focusing mechanism). These constructs enable the programmer to group production rules into different rule sets and specify the desired conflict resolution strategy. The simulation indicated that 5.11 to 7.57 fold speedup. However this environment does not provide a mechanism for determining whether a rule set is a parallel or sequential rule-set. Instead, it is the programmer's sole responsibility to ensure that the rule instantiations in a parallel rule-set do not interfere with each other.

Schmloze [52,53] has proposed a simulated asynchronous distributed production system called PARS with the following advantages:-

1. It obtains speed up over parallel match systems by executing multiple rule instantiations simultaneously.
2. It obtains speed-up over synchronous production systems by eliminating synchronization bottlenecks.
3. It is an inexpensive solution for large-scale production system.

For a 32-processor system, PARS is almost as fast as four times. For 8-processor system, PARS is two times as fast. However the over all speed-ups are not spectacular.

Miranker et al [40,41] developed a parallel production language CREL which is syntactically identical to OPS5. A CERL program is correct if and only if all eligible serial execution paths reach correct terminal state. It is the programmer's sole

responsibility to write correct a CERL code. As the program increases in size the programmer's job becomes unmanageable.

Gamble proposed (not implemented and not simulated) a non deterministic parallel language called SWARM [21]. SWARN is a parallel language that provides formal methods for specifying production system for coding and verifying production system. No information is available yet on its performance and capability.

Kuo [35] implemented a Multiple Context Multiple Rule (MCMR) firing model. The performance of this model was measured on the RUBIC simulator and the Intel IPSC/2 Hypercube. The RUBIC is a simulator that was written in Common Lisp and currently running on a Sparc Sun workstation. Speed up of 3.35 to 19.45 fold have been obtained.

Stolfo et al. [56] implemented a prototype of the PARULEL language using Common Lisp. PARULEL capture the inherent parallelism in production programs using parallel production languages. It allows the programmer to develop parallel solutions and also enjoy actual efficient, parallel execution of the resultant code. However, it is the programmer's responsibility to write correct code which is hard to do.

CHAPTER 3

THE LANA-MATCH MODEL

In this chapter, detailed description of the architecture of the Lana-Match Model, a parallel version of the Rete-Match algorithm for distributed memory architecture, is presented followed by a comprehensive analysis of its correctness and complexity. Finally, the chapter is concluded with a summary of our implementation followed with the advantages and the disadvantages of the Lana-Match model.

3.1 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 a one or more Slave Processors (SP). Every SP communicates with the CP through two communication buffers. One of these buffers is for input and the other is for output. Figure 4 outlines the Lana-Match model architecture.

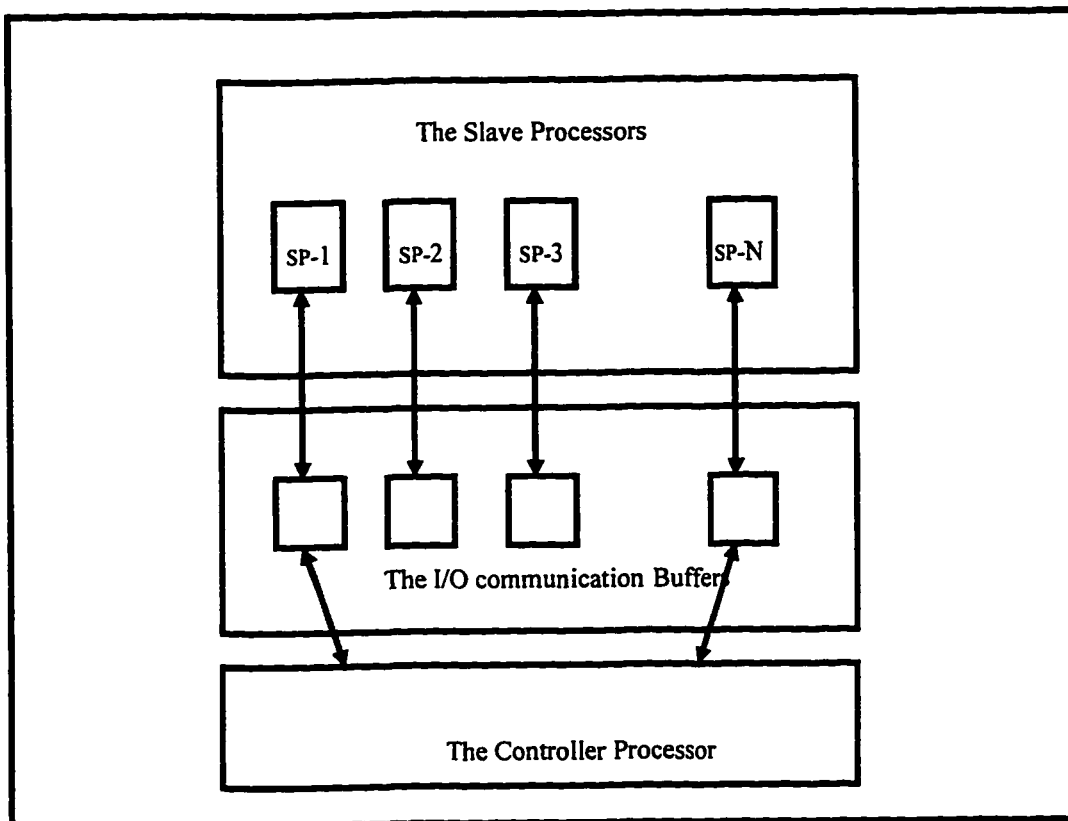


Figure 4: The Lana-Match Model Architecture

3.1.1 The Controller Processor

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

a) **The Master Rete-Network** is the pattern and join networks generated at the Controller Processor 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 Slave Processors by compiling the same rules set on each SP.

b) **The Master Facts List.** All the facts are maintained in the Master Facts List. Apart from the initial facts that are inserted directly by the user through the Controller Processor at the start-up time, all the facts that are inserted or deleted from the Master Facts List are generated at the Slave Processors. The SP receives a message from the Controller Processor 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 stamp. Asserting-facts-to or deleting-facts-from the Master Facts List activates or deactivates rules at

the Controller Processor, which are then translated to addition (or deletion) of one or more activation to/from the Master Agenda.

c) The 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. It is stamped by the Controller Processor at the creation time.
- 2) A pointer to the Slave Processor that is currently processing this activation. When the scheduler assigns this activation to an 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 which 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 Control Processor to temporarily store all the additions and deletions to the MFL. In another words, the Action Commands Zone is nothing more than a waiting zone for the results of executing an activation at a Slave Processor (SP). The reason to buffer the changes and not to apply them directly to the Master Fact List is mainly because assertion of facts can deactivate and remove some or all the 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.

d) The Action Queue guarantees serializability. It is a circular list of slots where every one of these slots is assigned to a slave processors every time an SP 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.

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

f) The Master Engine. As in Figure 5, 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 Master Engine of the model to carry out the parallel execution of the system. The Master Engine consists of the Main Loop and different control functions such as finding a free Slave Processor, scheduling an activation to a Slave Processor, rescheduling a Slave Processor and examining and committing the action commands in the same sequence as they were created.

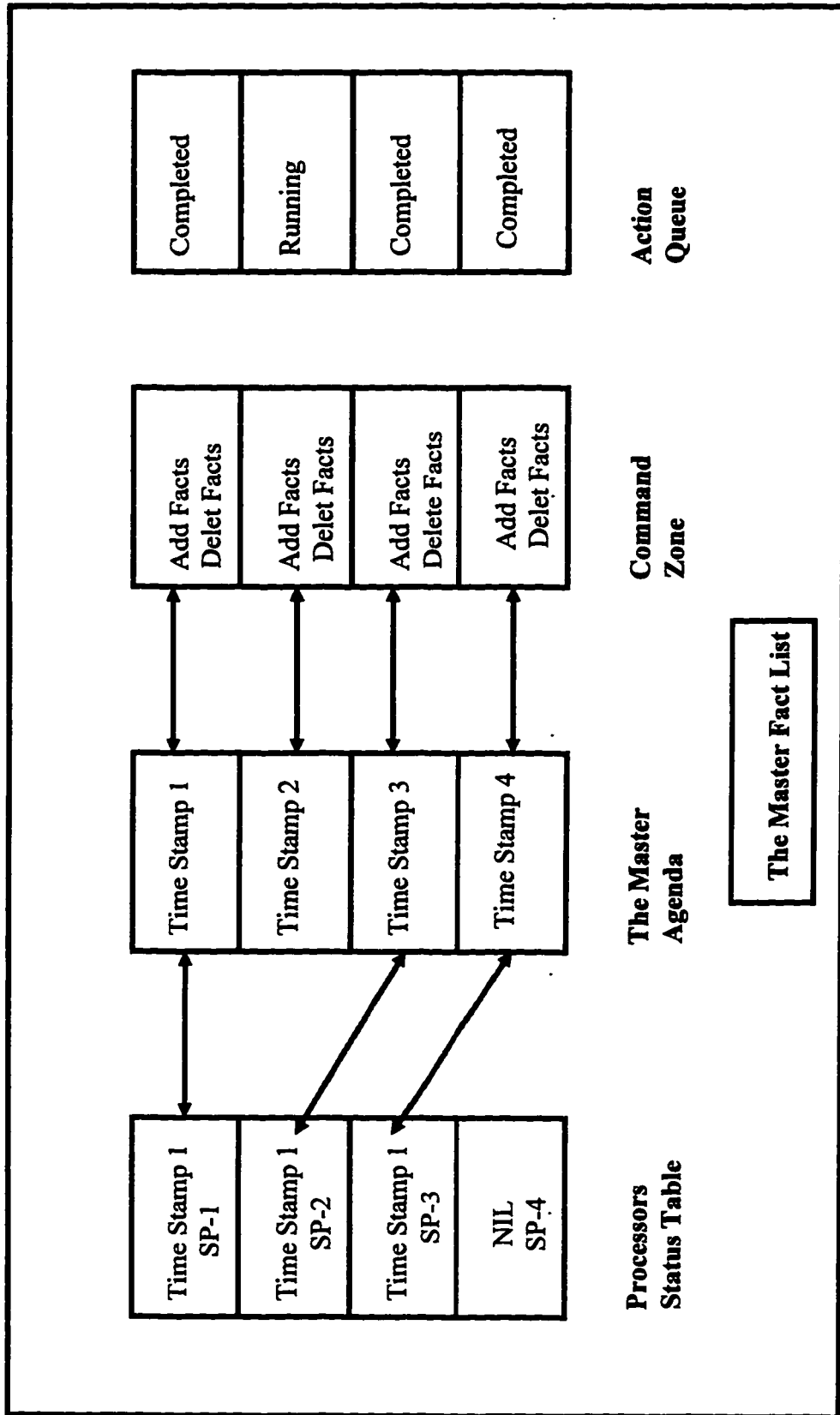


Figure 5: The Lana-Match Model Data Structures

1. **Get a Free Slave Processor Function** examines the Slave Processor Status Table and returns the first Slave Processor identification number that points to a null activation. If there is not any free Slave Processor, this function executes the Rescheduler repeatedly to search the SPTS and return the first available free SP.
2. **Schedule an activation to a Slave Processor:** Given an activation and a free Slave Processor, do the following:-
 - a) Set the corresponding entry for this Slave Processor at the Slave Processor Status Table to point to the given activation.
 - b) Set the time stamp field for this SPTS entry to the time stamp of the given activation.
 - c) Make the activation points to the Slave Processor.
 - d) Buffer all the facts that activated this activation along with its time stamp at the input buffer of the Slave Processor.
 - e) Change the status of the Action Queue entry for that activation to "Running".
3. **The Rescheduler.** At completion, every Slave Processor buffers all the facts additions and deletions that were generated while its executions along with the assigned activation time stamp at its output buffer. Every one of these facts additions and deletion 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 Slave Processor is pointing to at the Slave Processors Status Table.

If these two times match, it means that this message is a correct message and should be buffered in the action zone of the activation that is being executed at that particular Slave Processor. Then, the Rescheduler buffers this action command at the action zone of the activation that is being executed at that particular Slave Processor; makes this Slave Processor 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 Slave Processor 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 Slave Processor 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 & Commit Action Commands Function: This function examines the Action Queue starting from the beginning. If the first action on the queue is completed, execute all the action commands that are stored at the Action Zone that is pointed to by the activation that owns this slot. Next, move forward in the queue and do the same thing but stop 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 Master engine executes this loop repeatedly until the end of the Master Agenda.

LOOP until the end of the Master Agenda.

 Get a free Control Processor and schedule the top activation of the agenda to it.

 Remove this activation from the Master agenda.

END LOOP

3.1.2 The Slave Processors

The Slave Processors are the set of processors that are available for the Rescheduler to choose from. This set is sometimes referred to as the Slave Processors pool. Slave processors communicate with the Controller Processor through asynchronous message passing mechanisms. Every one of these processors has the full Rete-Match network MRN. It waits for messages from the CP asking to insert facts into its local agenda. When the CP finished sending all the facts that activates the activation that was assigned to this SP, these facts instantiate a rule at the Slave Processor. Then, the Slave Processor fires this rule and buffers all the new generated facts additions or deletions of facts to/from the MFL. Figure 6,7 and 8 outline the Lana-Match computational model in an 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
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 Remove_an_Activation (a)
  1. Set Agenda(a) = Nil;
  2. Set Command_Zone(a) = Nil;
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 wit the
  Timestamp of a (Agenda(a).timestamp);
  4. Set Action_Queue(a) = Running;
End

```

Figure 6: The Lana-Match Model Outlines (Part 1)

Function The_Rescheduler

1. Examine the output buffer of each Slave Processor;
 2. FOR each processor p which 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
- End

Figure 7: The Lana-Match Model Outlines (Part 2)

```
Function Examine_and_Commit
1. Start from the top of the Action Queue;
2. LOOP UNTIL Action_Queue(i+)=Runing;
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
```

Figure 8: The Lana-Match Model Outlines (Part 3)

3.2 The Correctness of the Lana-Match Model

The correctness of the Lana-Match Model is based on a centralized time stamp Optimistic Concurrency Control mechanism which is a full proven database technique [61]. In this thesis, this technique is extended to guarantee the correctness of parallel execution of production systems. This extension is part of the contribution of the thesis 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 MA 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.

In the Lana-Match model, correctness is accomplished by distributing the Master Agenda to the Slave Processors simultaneously and letting every SP executes its activation and passes all of its results back to the CP. These results which are either addition or deletion actions of facts to/from the Master Fact list will not be committed as the CP receives them. But instead, they will be queued in the Action Queue. As the Controller Processor (CP) is free (or needs to get a free Slave Processor), the CP executes the Examine-and-Commit function which 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 AQ. This sequence is FIFO to reflect the incremental nature of the time stamp of the activations. The Examine_and_Committ 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 Master Fact List is done exactly the same as if the activation were executed sequentially.

3.3 The Lana-Match Model Complexity Analysis.

The expected speed-up (SP) is equal to the sequential execution costs over the parallel execution costs. In another words, the SP is the ratio between the number of time units that are required for the sequential execution of all the activations of the Master Agenda on a single node and the number of time units that are required for parallel execution of all activations on more than one node. To calculate the sequential execution costs, let :

- 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 is needed to match a fact through the pattern network

and to drive it through the join network.

- T_f the average time that is required to fire a rule.

Thus, the total time that is needed to evaluate the Left-Hand-Side of a rule T_{LHS}

$$T_{LHS} = N_f * T_m \quad (1)$$

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_f . Thus, the total execution time for a rule T_{Rule} would be calculated as follows.

$$T_{Rule} = N_f * T_m + T_f \quad (2)$$

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 execution of the production system.

$$T_{seq} = N_a [N_f * T_m + T_f] \quad (3)$$

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

- T_1 the time that is needed to evaluate the Left-Hand-Side for each rule once at the Controller Processor.

$$T_1 = N_a * T_{LHS} \quad (4)$$

- T_2 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 Slave Processors.

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

- T_3 the time that is needed to fire all the rules in parallel. Firing all the rules in parallel.

$$T_3 = N_a * T_f / N_s \quad (6)$$

- T_c 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.

$$\text{Communication Cost} = N_a * T_c \quad (7)$$

- This makes the parallel execution costs as follows

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

Performance speedup is achieved when the sequential execution costs is greater than the parallel execution costs. In another words $T_{\text{parallel}} < T_{\text{seq}}$. This relation could be expanded as follows:-

$$N_s * T_m + N_s * T_m / N_s + N_s * T_f / N_s + N_s * T_c < N_s * (T_m + T_f) \quad (9)$$

This relation can be simplified as follows:

$$T_m / (N_s - 1) + N_s / (N_s - 1) T_c < T_f \quad (10)$$

However, T_m is almost constant and $(N_s / N_s - 1)$ is almost one. Thus, from this relation we can conclude that the Lana Match model performs its best speed up when the average time that is required to fire a rule T_f 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.

3.4 Implementing the Lana-Match Model using PVM and CLIPS

The two main ingredients to implement the Lana-Match model are a Rete-Match based production system package and a message passing communication mechanism. Appendix C, of this thesis summarize possible implementation alternatives for this model. In this thesis, we decided to implement the Lana-Match Model using CLIPS and PVM.

The C Language Integrated Production System (CLIPS) [59-60] that was developed at NASA's Johnson Space Center during 1985-1993 started receiving a widespread acceptance throughout the public and private sectors to be one of the most powerful Rete-Match based production system packages. It has more than 5000 users including NASA sites and branches, numerous federal bureaus, government contractors, more than 200 universities and many companies. Because of its portability, extensibility, capabilities and low-cost for both the executable and the source code, it was decided to implement the Lana-Match Model using the C Language Integrated Production System (CLIPS). The Parallel Virtual Machine (PVM) [8,10] permit a network of heterogeneous UNIX computers to be used as a single large parallel computers while handling all communications and reliability details. It provides a more efficient, powerful, reliable, popular parallel programming environment. The development of PVM started 1989 at Oak Ridge National Laboratory (ORNL). It was ported to almost every known UNIX machine. 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. For its portability and popularity, it was decided to use PVM to implement the Lana-Match Model in this thesis. To implement the Lana-Match model using CLIPS & PVM on a UNIX based environment, the following three steps were taken:

1. **Creating the Controller Processor: XCLIPS** which is an Xwindow version of CLIPS was modified to become the Controller Processor. This is accomplished by creating all the data structures that are needed by the CP such as the SPST, AQ, ME, etc. and modifying its agenda to become the Master Agenda, the fact list to become the Master Fact List and the engine to become the Master Engine. Modifying the engine is accomplished by transforming the firing mechanism to make it schedule the activations to the SPs and copy every activation to the assigned SP input buffer. Finally, the ME was made to repeatedly check the output buffers of all of its slaves and take the passed action commands and queue them in the Action Queue to be processed in the same sequence that their activations were originally created on.
2. **Creating the Slave Processor Code:** A copy of CLIPS was modified to become the Controller Processor. This was accomplished by changing the behavior of the Line Command procedure of the package from waiting for a user response to make it check repeatedly its local input buffer for messages from the CP. Moreover, this copy was modified to write the final results to its output buffer as Action Commands rather than applying it to its local fact list.
3. **Creating the communication infrastructure:** Both of these copies need to be modified to make the CP run as a PVM master and the SP to run as a slave or (worker using PVM terminology). This modification includes defining and building all the needed buffers and the necessary communication mechanisms.

3.5 Advantages of the Lana-Match Model

The Lana-Match model enjoys the following main advantages:-

1. **The Lana-Match model is Adaptive.** The Lana-Match model is a very adaptive approach in which adding a Slave Processor 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.** The Lana Match Model was designed to be a very heterogeneous model. It was designed to allow heterogeneity at three levels. First, at the network level since the Slave Processors and the Control Processors can be executed on any network as long as they can communicate via message passing. Second, at the production system level since the Slave Processor and the Control Processor can be implemented using any production system as long as it can be modified to communicate with the Controller Processor. However, the Controller Processor needs to be developed using a production system package that is based on the Rete Match algorithm. Third, at the node (or the processor) level, heterogeneous nodes that can communicate through message passing mechanism.
3. **Practicality.** The main motivation for developing the Lana-Match model is to take a practical approach to develop a parallel version of the Rete-Match algorithm that can lead to a real implementation. This need raised as we noticed that most of the research that was reported in this area was either based on a shared memory architecture or was based on a special architecture. Moreover, it

was also noticed that nine out of twelve of the previous approaches to this study were based on simulation rather than real implementation [24-26, 28-35, 40-45] . Therefore, we thought a practical real implementation approach is highly needed for this algorithm.

4. **Exploits Run Time Parallelism.** The Lana-Match Model was designed to exploit run time parallelism opportunities which 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 opportunity that can be utilized at the run time. As an example, let us say that we have only one rule to fire and 10000 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 are, at least, 10000 chances for parallelism (assuming the instantiations are independent).
5. **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 Slave Processors do not store any status. They are only created to be assigned tasks to execute and at the end, they are destroyed. It is perfectly true, that the system can start with N processors and end with less processors without effecting the final results. However, the model is not immune from the death of the CP (i.e. if the CP crush the model will be dead and will leave all the SPs behind it).
6. **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. But instead, the Lana-Match model asks the programmer to write only a set of rules and leave all the rest to be addressed by the system.

7. **Filled In the Gap Between Database and Knowledge Base Concurrency Control Mechanism.** In this thesis, 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.

3.6 Disadvantages of the Lana-Match Model.

The Lana-Match Model could be criticized with the following concerns:-

1. **The Controller Processor Can be a Memory Intensive Processor.** Since the Lana-Match Model utilizes the optimistic concurrency control mechanism, it requires high memory to store intermediate results before committing changes. However, this cost is well justified at the database area and we think it is also well justified in our approach by the speed up that it delivers.
2. **The Controller Processor Can Be a Bottle-neck.** For extremely large production systems where the Controller Processor can not keep up with the Slave Processors, some of the Slave Processors can be kept waiting. One approach to address this issue is to utilize the heterogeneity feature of the model that allows

different computers and networks to be used for implementation by executing the Controller Processor on a more powerful node while executing the Slave Processors on less powerful nodes. Another approach is to execute more than one Slave Processors at one node.

3. **Redundant Execution of the Rules.** Since the CP sends facts to the SPs to activate rules, some facts may activate more than the targeted rule. This mean we activated the same rule more than once. This contributes to the redundant execution of the rules and increases the execution time.

CHAPTER 4

RESULTS AND ANALYSIS

This chapter contains a comprehensive analysis of the performance of our implantation of the Lana-Match model. It was measured in an environment that consists of more than 500 Sun workstations that are connected through FDDI (i.e has around 100MB/Second throughput). Clusters of 2, 4, 8, 16, 32 and 64 workstations were used to measure the Lana-Match performance. Most of these workstations were Sun SPARCstation 10 model with few SPARC stations LX model. Although this network was not dedicated for these tests, all of the measurements were taken when the network was almost dedicated during weekends and after working hours. The total elapse time for the sequential execution of the Rete-Match algorithm was measured by running the rules on the original implementation of CLIPS on a SPARC station 10 workstation. This time is measured as the difference between the system time when CLIPS start processing the agenda and the system time when it complete it. The total elapsed time of the Lana-Match model was measured as the difference between the system time when the CP starts scheduling the first activation of the Master Agenda and the system time when the CP commit all the Action Commands of the last activation of the Master Agenda. The CP was running on a SPARC station 20

workstation while the SP's were running mostly on Sun SPARC stations 10 workstation.

In section 3.3, the theoretical analysis of the Lana Matching model conclude that the average time that is required to fire a rule T_f and the average time that is required by the CP to send all the facts for one activations to a SP and to receive all the generated results from executing that activation T_c , were identified as the two main parameters that influence the performance of the Lana-Match model. The Lana Matching Model perform its best when T_f is much larger than T_c . (i.e $T_f \gg T_c$). The following five test cases were designed to study the impact of these parameters on the over all performance of this implementation of the Lana-Match model and to identify the best conditions at which the Lana-Match performs the best.

- Case-1: Totally Independent Light Action Instantiations, T_f is much greater than T_c .
- Case-2: Totally Independent Heavy Action Rules, T_f is much smaller than T_c .
- Case-3: Enabling Dependent Rules with Additions, T_f is larger than T_c .
- Case-4: Enabling Dependent Rules with few Additions and Deletions, this is similar to case-3, T_f larger than T_c .
- Case-5: Enabling Dependent Rules with many Additions and Deletions, this is similar to case 3 and case 4 , except that T_f larger than T_c .

4.1 Case 1: Totally Independent Light Action Instantiations

Light Actions instantiations are rules that the majority of their actions (i.e. RHS) requires very little CPU time. In another words, Light Action Instantiations are those ones with the average number of time units that are required to fire a rule T_f is very small. Five tests were designed with 1000, 3000, 5000, 7000 and 9000 of totally independent Light Action instantiations.

This case is demonstrated by Example 1 in Figure 9. It consists of one rule which requires a pair of facts to instantiate it and performs a simple action that consists of a simple PRINT statement that prints "Hello World". The performance of the Lana-Match model was measured for this case by instantiating the given rule 1000, 3000, 5000, 7000 and 9000 times by inserting 1000, 3000, 5000, 7000 and 9000 pairs of facts at the start-up time respectively. Figure 10 shows that the time that is required to execute these instantiations on clusters of 2, 4, 8, 16, 32 and 64 processors.

Figure 11 concludes that the Lana-Match model requires more time than the sequential version of the Rete-Match algorithm if the majority of the actions of the rules has light action as its RHS. This result is mainly due to big difference between the communication costs and the saving of parallelization. In another words, $T_f \ll T_c$. Table 1 and Table 2 show the actual measured data for this case.

EXAMPLE 1

```
(defrule R1 " This is the only rule for case-1 !!!! "  
  (point1 ?x ?y )  
  (point2 ?y ?x )  
  =>  
  ( printout t "Hello world !!!!!" crlf )  
  )
```

```
(deffacts startup "Initially insert 1000, 3000, 5000, 7000 and 9000 pair of  
  facts for Test-1, Test-2, Test-3, Test-4, and Test-5"  
  
  (point1 1 2 )  
  (point2 2 1 )  
  (point1 1 3 )  
  (point2 3 1 )  
  (point1 1 4 )  
  (point2 4 1 )  
  ..... etc.....  
  )
```

Figure 9 : Example of case 1

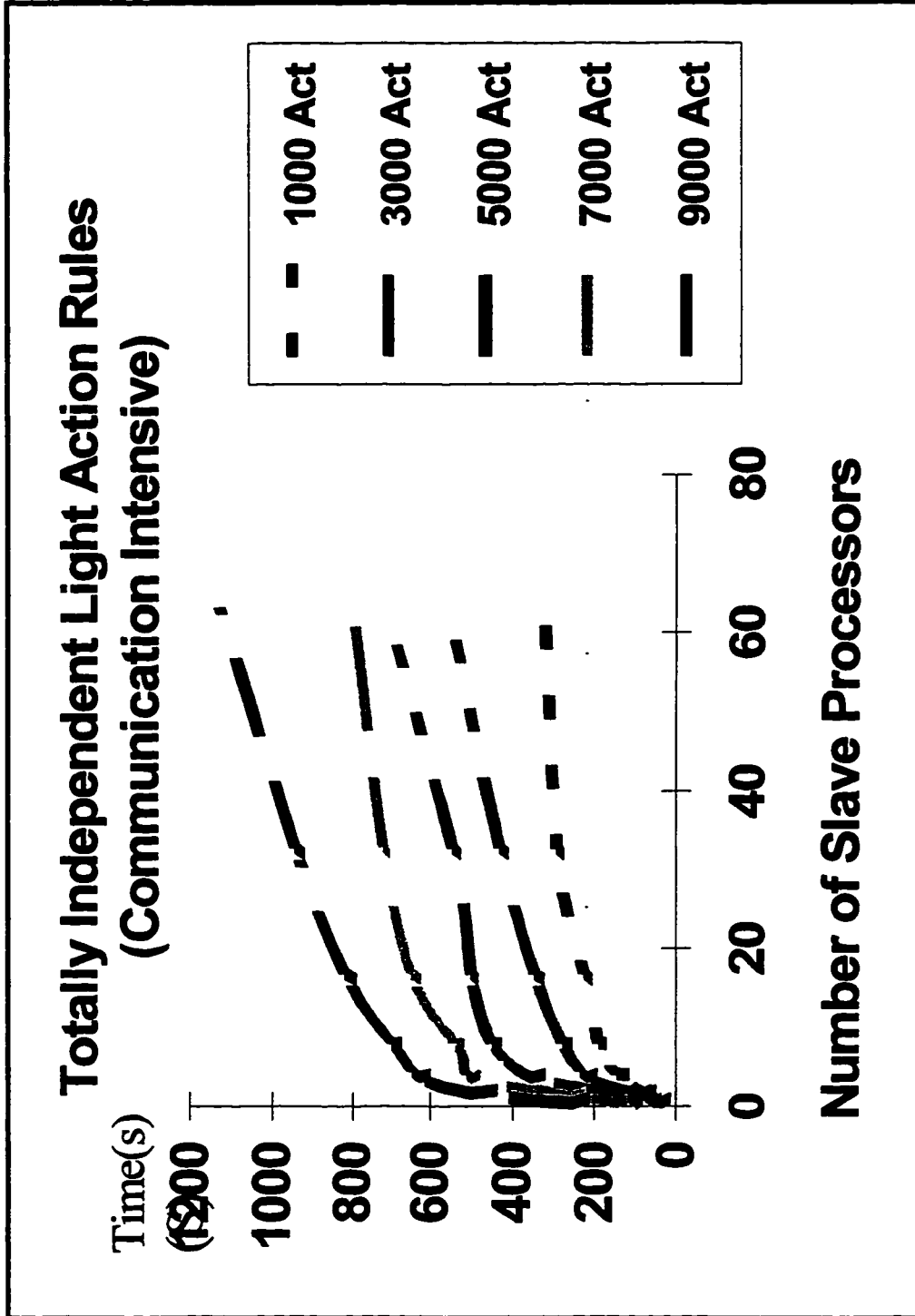


Figure 10: Totally Independent Light Action Rules, Actual Measured Time

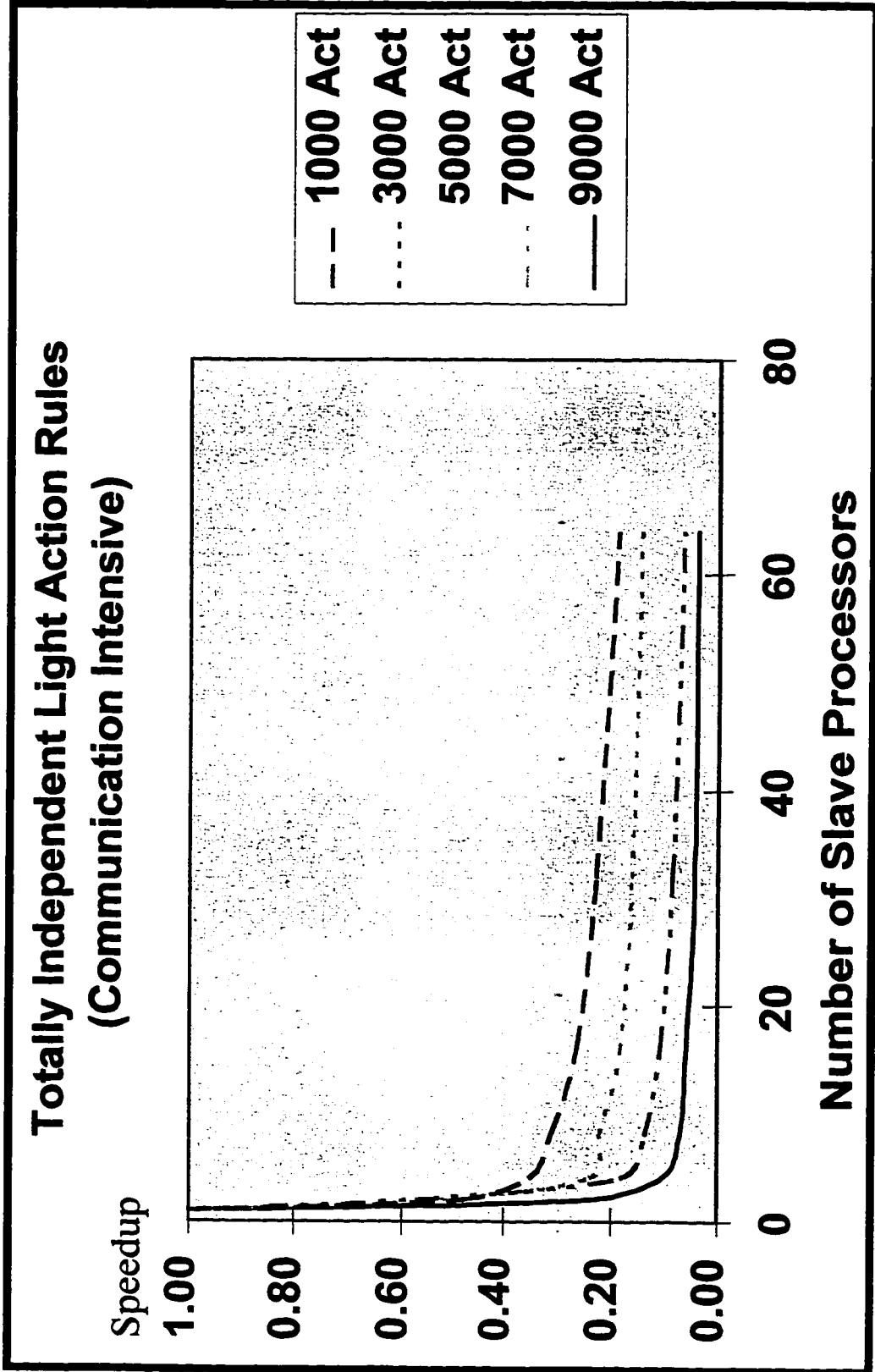


Figure 11 : Totally Independent Light Action Rules, Speedup

Table 1: Actual time measured in Case 1.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	12	35	82	114	210
2	49	61	101	217	435
4	115	191	319	474	598
8	170	255	425	513	671
16	203	321	479	622	790
32	274	409	523	702	919
64	309	539	698	786	1114

Table 2 : Speedup data calculated for Case 2.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	1.00	1.00	1.00	1.00	1.00
2	0.24	0.57	0.81	0.53	0.48
4	0.10	0.18	0.26	0.24	0.35
8	0.07	0.14	0.19	0.22	0.31
16	0.06	0.11	0.17	0.18	0.27
32	0.04	0.09	0.16	0.16	0.23
64	0.04	0.06	0.12	0.15	0.19

4.2 Case 2: Totally Independent Heavy Action Instantiations

Heavy Actions instantiations are rules that the majority of their actions (i.e. RHS) requires a lot of CPU time. In another words, Light Action Instantiations are those one with the average number of time units that are required to fire a rule T_f is very large. Five tests were designed with 1000, 3000, 5000, 7000 and 9000 of totally independent Light Action instantiations.

This case is demonstrated by Example 2 in Figure 12. It consists of one rule which requires a pair of facts to instantiate it and performs a heavy action on its RHS that consists of busy waiting of a WHILE loop statement. This WHILE statement rule makes the firing of the rule uses more CPU time. The performance of the Lana-Match model was measured for this case by conducting five different tests. These tests instantiated the given rule 1000, 3000, 5000, 7000 and 9000 times by inserting 1000, 3000, 5000, 7000 and 9000 pairs of facts at the start-up time respectively. Figure 13 shows that the time that is required to execute these instantiations on clusters 2, 4, 8, 16, 32 and 64 processors.

Figure 14 concludes that the Lana-Match model does achieve a significant speed up with respect to the number of processors in situations where the rule set consists of a very large number of independent rules were every rule has a heavy action to perform. Table 3 and Table 4 show the actual measured data for this case.

EXAMPLE 2

```

(defrule R1 " This is the only rule for case-2 !!!! "
  (point1 ?x ?y )
  (point2 ?y ?x )
  =>
  ( printout t "Hello world !!!!!" crlf )
  (while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1)))
)

(deffacts startup "Initially insert 1000, 3000, 5000, 7000 and 9000
                  Paris facts for Test-1, Test-2, Test-3, Test-4 and Test-5"
  (point1 1 2 ) (point2 2 1 )
  (point1 1 3 ) (point2 3 1 )
  (point1 1 4 ) (point2 4 1 ) )

```

Figure 12 : Example of case 2

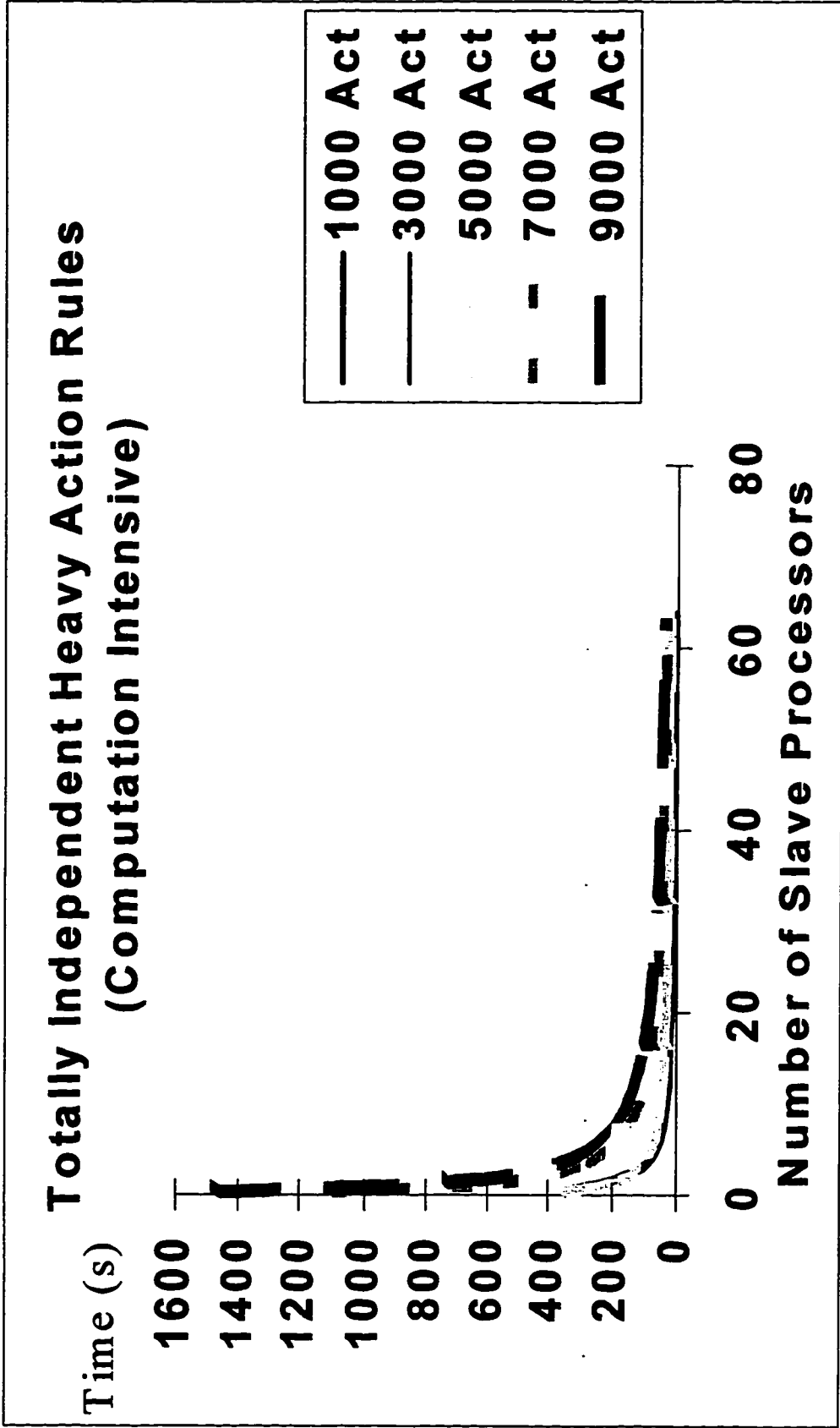


Figure 13: Totally Independent Heavy Action Rules, Actual Measured Time

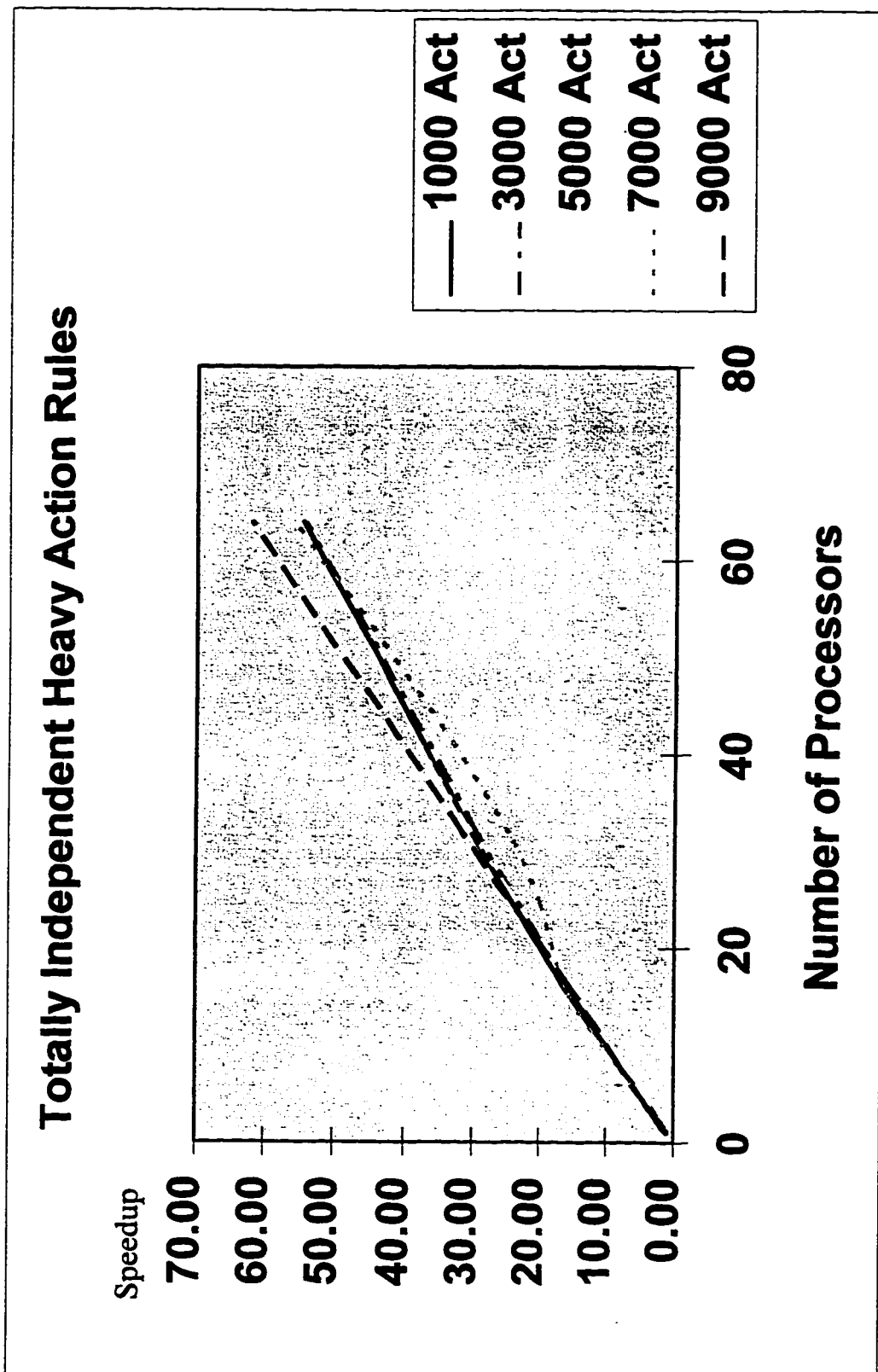


Figure 14: Totally Independent Heavy Action Rules, Speedup

Table 3: Actual time measured in Case 2.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	272	325	364	1113	1489
2	151	190	185	556	746
4	74	86	99	279	370
8	34	41	52	138	185
16	17	21	28	70	98
32	9	11	12	45	47
64	5	6	6.5	20	24

Table 4 : Speedup data calculated for Case 2.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	1.00	1.00	1.00	1.00	1.00
2	1.80	1.71	1.97	2.00	2.00
4	3.68	3.78	3.68	3.99	4.02
8	8.00	7.93	7.00	8.07	8.05
16	16.00	15.48	13.00	15.90	15.19
32	30.22	29.55	30.33	24.73	31.68
64	54.40	54.17	56.00	55.65	62.04

4.3 Case 3: Enabling Dependent Rules with Additions

Enabling Dependent Instantiations are rules that activates each others in a chained fashion in which executing an activation generates additions of facts that instantiate more activation on the agenda.

This case is demonstrated by Example 3 in Figure 15. It consists of four rules where the execution of these rules forms a chain of four consecutive instantiation sequence. The first rule requires a pair of facts to instantiate it which then inserts a pair of facts that instantiates the second rule (i.e. R2). Next, the execution of the second rule inserts a pair of facts that instantiates the third rule (i.e. R3) and so forth. The performance of the Lana-Match model was measured for this case by conducting five tests. These tests instantiate these four rules 1000, 3000, 5000, 7000 and 9000 times by inserting 1000, 3000, 5000, 7000 and 9000 pairs of facts respectively. Figure 16 shows that the time that is required to execute these instantiations on clusters 2, 4, 8, 16, 32 and 64 processors.

Figure 17 concludes that the Lana-Match model does achieve a significant speed up with respect to the number of processors. However, this speed up is not as big as the speed up in Case 2 which is mainly due to the increase on the communication costs and the increased work for the CP. Table 5 and Table 6 show the actual measured data for this case.

EXAMPLE-3

```

(defrule R1 " This is the first rule" (point1 ?x ?y )(point2 ?y ?x )
=> (assert (point3 ?x ?y) (Point4 ?y ?x) )
    (while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )

(defrule R2 "This is the second rule" (point3 ?x ?y )(point4 ?y ?x
) => (assert (point5 ?x ?y) (Point6 ?y ?x) )
    (while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )

(defrule R3 "This is the third rule " (point5 ?x ?y )(point6 ?y ?x )
=> (assert (point7 ?x ?y) (Point8 ?y ?x) )
    (while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )

(defrule R4 " This is the forth rule " (point7 ?x ?y )(point8 ?y ?x )
=> ( printout t "Hello world  !!!!" crlf )
    (while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )

(deffacts startup "Initially insert 2000, 5000, 7000 and 9000 pair of
                    facts for Test-1, Test-2, Test-3, and Test-4"
(point1 1 2 ) (point2 2 1 ) ..... etc )
(point2 2 1 )

```

Figure 15: Example of case 3

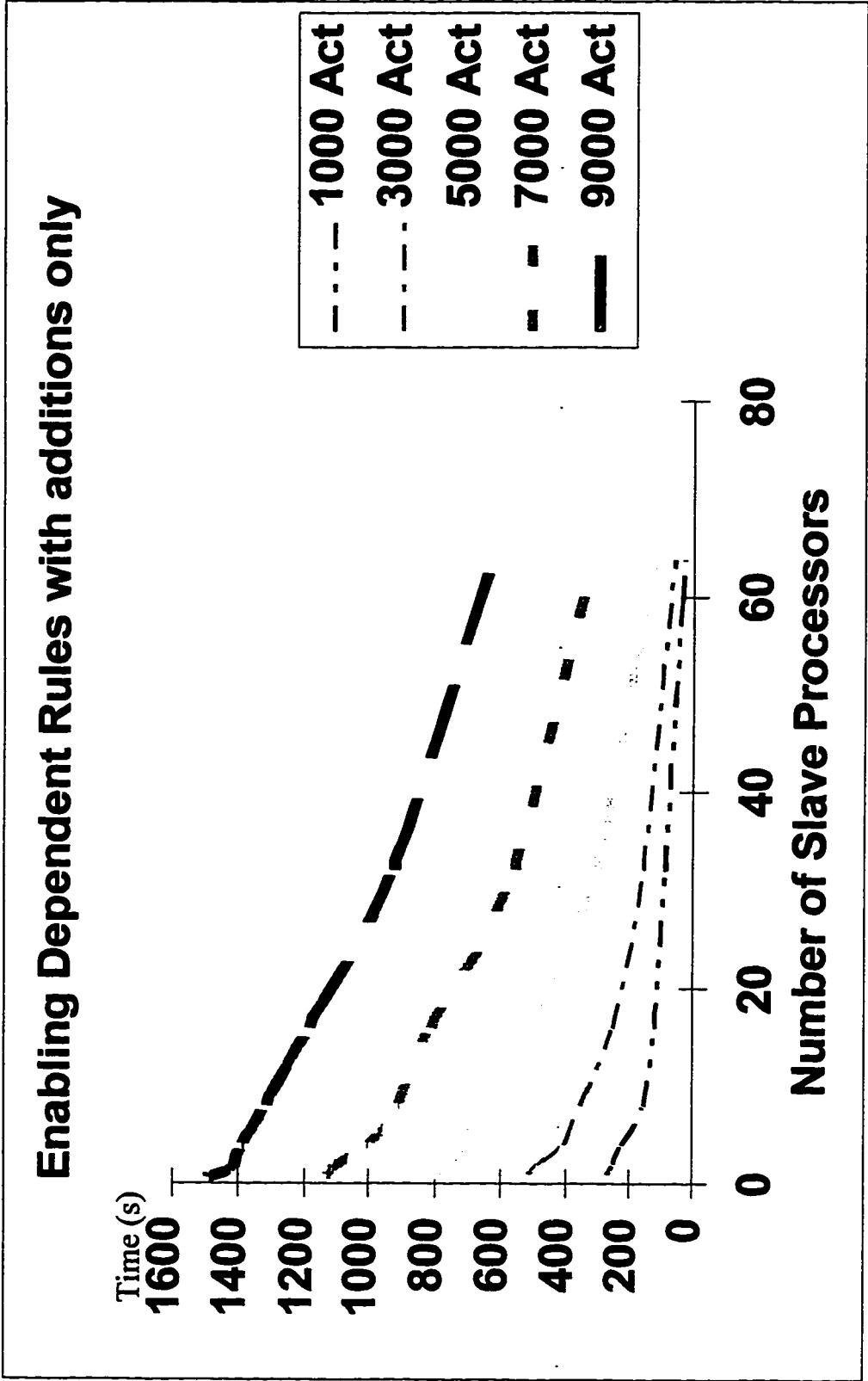


Figure 16 :Enabling Dependent Rules with additions only, Actual Measured Time

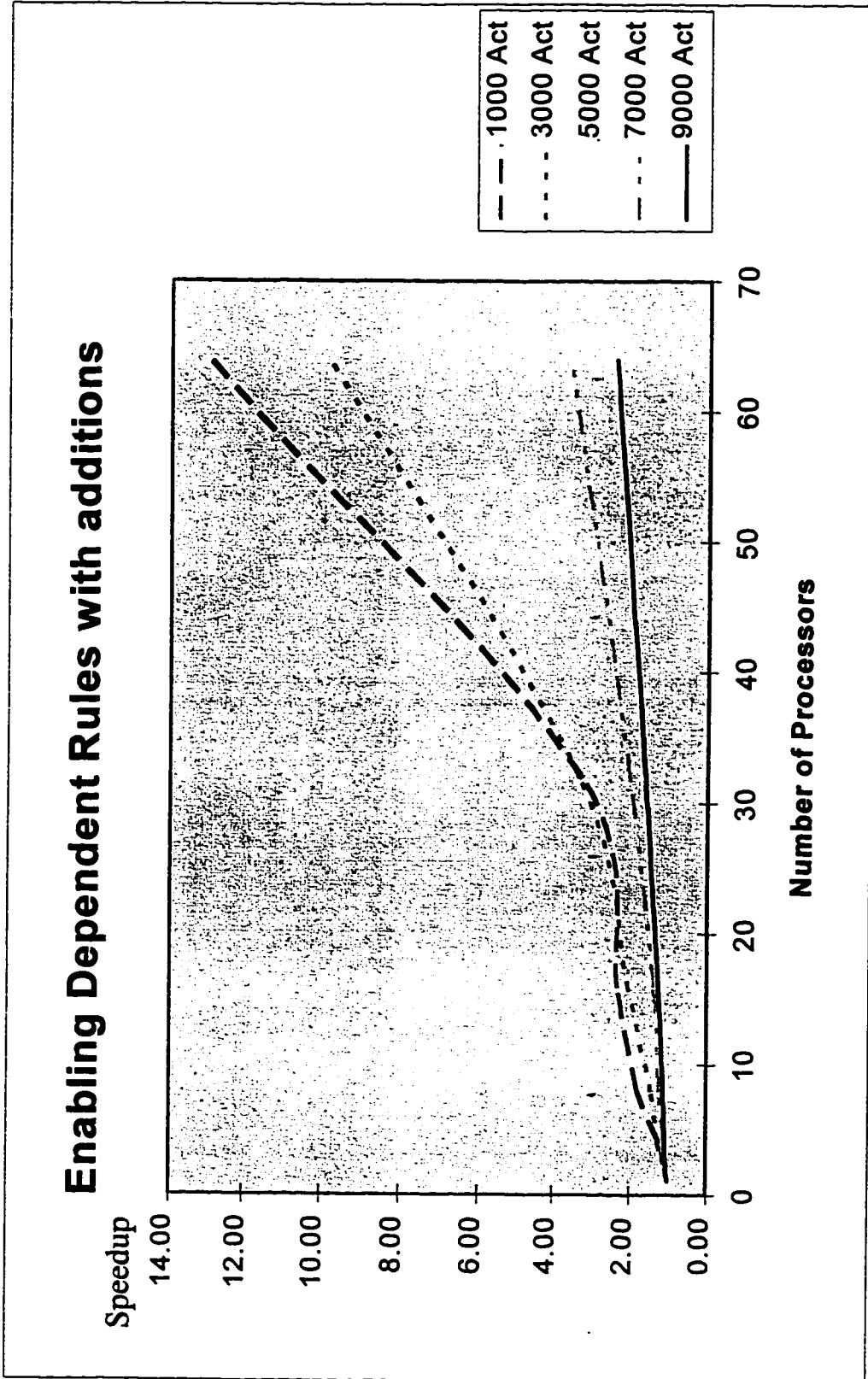


Figure 17 : Enabling Dependent Rules with additions only, speedup.

Table 5: Actual time measured in Case 3.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	272	512	809	1113	1489
2	250	482	753	1098	1403
4	223	405	712	1005	1381
8	153	354	652	905	1315
16	117	251	516	811	1182
32	83	153	301	553	922
64	21	52	111	306	617

Table 6: Speedup data calculated for Case 3.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	1.00	1.00	1.00	1.00	1.00
2	1.09	1.06	1.07	1.01	1.06
4	1.22	1.26	1.14	1.11	1.08
8	1.78	1.45	1.24	1.23	1.13
16	2.32	2.04	1.57	1.37	1.26
32	3.28	3.35	2.69	2.01	1.61
64	12.95	9.85	7.29	3.64	2.41

4.4 Case 4: Enabling Dependent Rules with Few Additions and Deletions

This case is an extension to case 3, in which Enabling Dependent Instantiations activates each others in a chained fashion by additions and deletions of facts that instantiate more or deactivate some of the activations from the agenda. However, the percentage of additions and deletion is %50 of the total number of the other actions.

This case is demonstrated by Example 4 in Figure 18. It consists of four rules where the execution of 50% of these rules does not add nor delete facts. However, the other %50 cause additions and deletions of activations from the Agenda. The performance of the Lana-Match model was measured for this case by conducting five tests. These tests instantiate these four rules 1000, 3000, 5000, 7000 and 9000 times by inserting 1000, 3000, 5000, 7000 and 9000 pairs of facts respectively. Figure 19 shows that the time that is required to execute these instantiations on clusters 2, 4, 8, 16, 32 and 64 processors.

Figure 20 concludes that the Lana-Match model performance decrease when communication costs increase between the CP and the SPs. In this case, the gain is almost negligible. This is mainly due to the increase of T_c . Table 7 and Table 8 show the actual measured data for this case.

EXAMPLE-4

```
(defrule R1 " This is the first rule" (point1 ?x ?y) (point2 ?y ?x) =>
(assert (point3 ?x ?y) (Point4 ?y ?x) ) ( retract (point8 ?x ?y) (Point9 ?y ?x) )
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(defrule R2 "This is the second rule " (point3 ?x ?y) (point4 ?y ?x) =>
(assert (point5 ?x ?y) (Point6 ?y ?x) ) (retract (point3 ?x ?y) (Point4 ?y ?x) )
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(defrule R3 "This is the third rule " (point5 ?x ?y) (point6 ?y ?x) =>
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(defrule R4 " This is the forth rule "
(point7 ?x ?y) (point8 ?y ?x) => ( printout t "Hello world !!!!!" crlf)
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(deffacts startup "Initially insert 1000, 3000, 5000, 7000 and 9000 pair of
facts for Test-1, Test-2, Test-3, Test-4, and Test-5"
(point1 1 2) (point2 2 1) ..... etc
```

Figure 18: Example of case 4

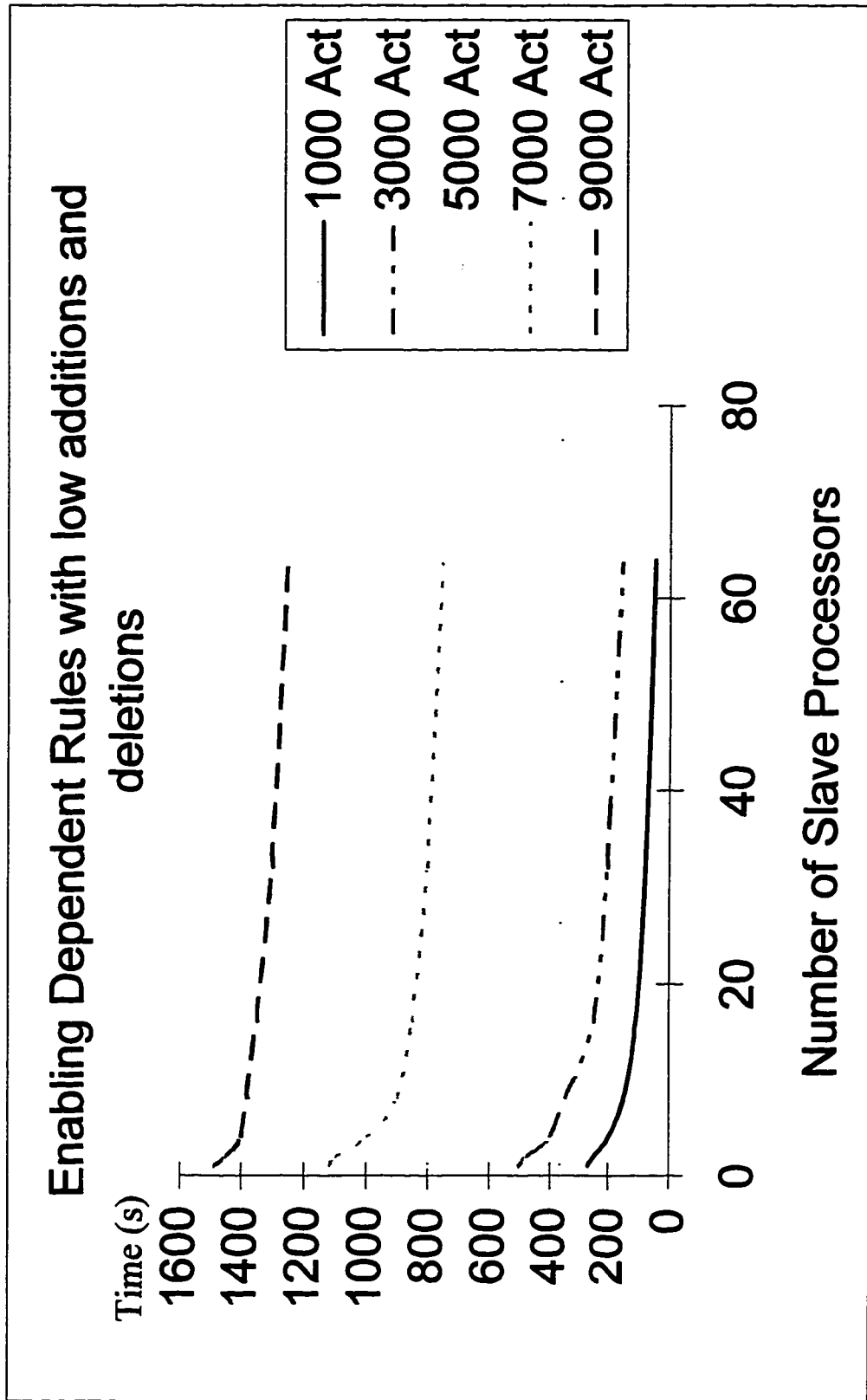


Figure 19: Enabling Dependent Rules with low additions and deletion , Actual Measured Time.

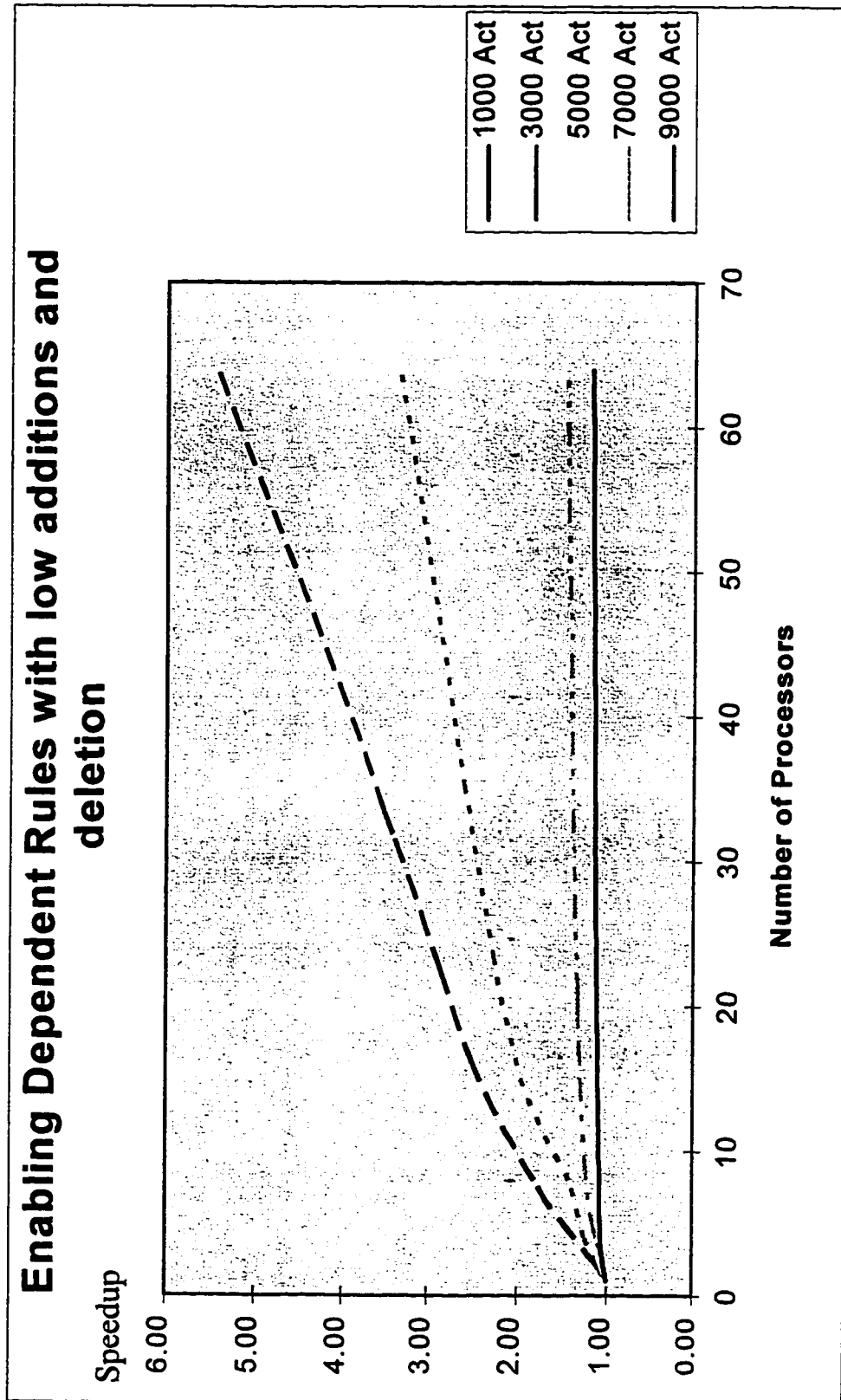


Figure 20: Enabling Dependent Rules with low additions and deletion , speedup.

Table 7: Actual time measured in Case 4.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	276	503	802	1123	1492
2	256	484	753	1105	1460
4	204	401	710	1001	1407
8	152	353	650	901	1382
16	110	259	607	855	1357
32	85	203	553	804	1309
64	52	160	509	758	1259

Table 8: Speedup data calculated for Case 4.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	1.00	1.00	1.00	1.00	1.00
2	1.09	1.04	1.07	1.01	1.03
4	1.36	1.25	1.14	1.11	1.06
8	1.81	1.43	1.23	1.24	1.08
16	2.47	2.00	1.33	1.31	1.10
32	3.40	2.50	1.45	1.39	1.15
64	5.44	3.33	1.60	1.48	1.19

4.5 Case 5 : Enabling Dependent Rules with Many Additions and Deletions

This case is an extension to case-3, in which Enabling Dependent Instantiations activates each others in a chained fashion by additions and deletions of facts that instantiate more or deactivate some of the activations from the agenda. However, ALL actions consist of additions and deletions of activations.

This case is demonstrated by Example 5 in Figure 21. It consists of four rules where the execution of ALL of these add and delete facts to/from the agenda. The performance of the Lana-Match model was measured for this case by conducting five tests. These tests instantiate these four rules 1000, 3000, 5000, 7000 and 9000 times by inserting 1000, 3000, 5000, 7000 and 9000 pairs of facts respectively. Figure 22 shows that the time that is required to execute these instantiations on clusters 2, 4, 8, 16, 32 and 64 processors.

Figure 23 concludes that the Lana-Match model is not suitable when communication costs is very high between the CP and the SPs. In this case the gain is almost negligible. This is mainly due to the large of T_c . Table 9 and Table 10 show the actual measured data for this case.

EXAMPLE-5

```
(defrule R1 " This is the first rule" (point1 ?x ?y) (point2 ?y ?x) =>
(assert (point3 ?x ?y) (Point4 ?y ?x) ) ( retract (point8 ?x ?y) (Point9 ?y ?x) )
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(defrule R2 "This is the second rule " (point3 ?x ?y) (point4 ?y ?x) =>
(assert (point5 ?x ?y) (Point6 ?y ?x) ) (retract (point3 ?x ?y) (Point4 ?y ?x) )
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(defrule R3 "This is the third rule " (point5 ?x ?y) (point6 ?y ?x) =>
(assert (point5 ?x ?y) (Point6 ?y ?x) ) (retract (point3 ?x ?y) (Point4 ?y ?x) )
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(defrule R4 " This is the forth rule " (point7 ?x ?y) (point8 ?y ?x) =>
( printout t "Hello world !!!!!" crlf)
(while (> (* ?y ?x) 0)(+ ?x ?y)(bind ?y (- ?y 1))) )
```

```
(deffacts startup "Initially insert 1000, 3000, 5000, 7000 and 9000 pair of
facts for Test-1, Test-2, Test-3, Test-4, and Test-5"
(point1 1 2) (point2 2 1) ..... etc)
```

Figure 21: Example of case 5

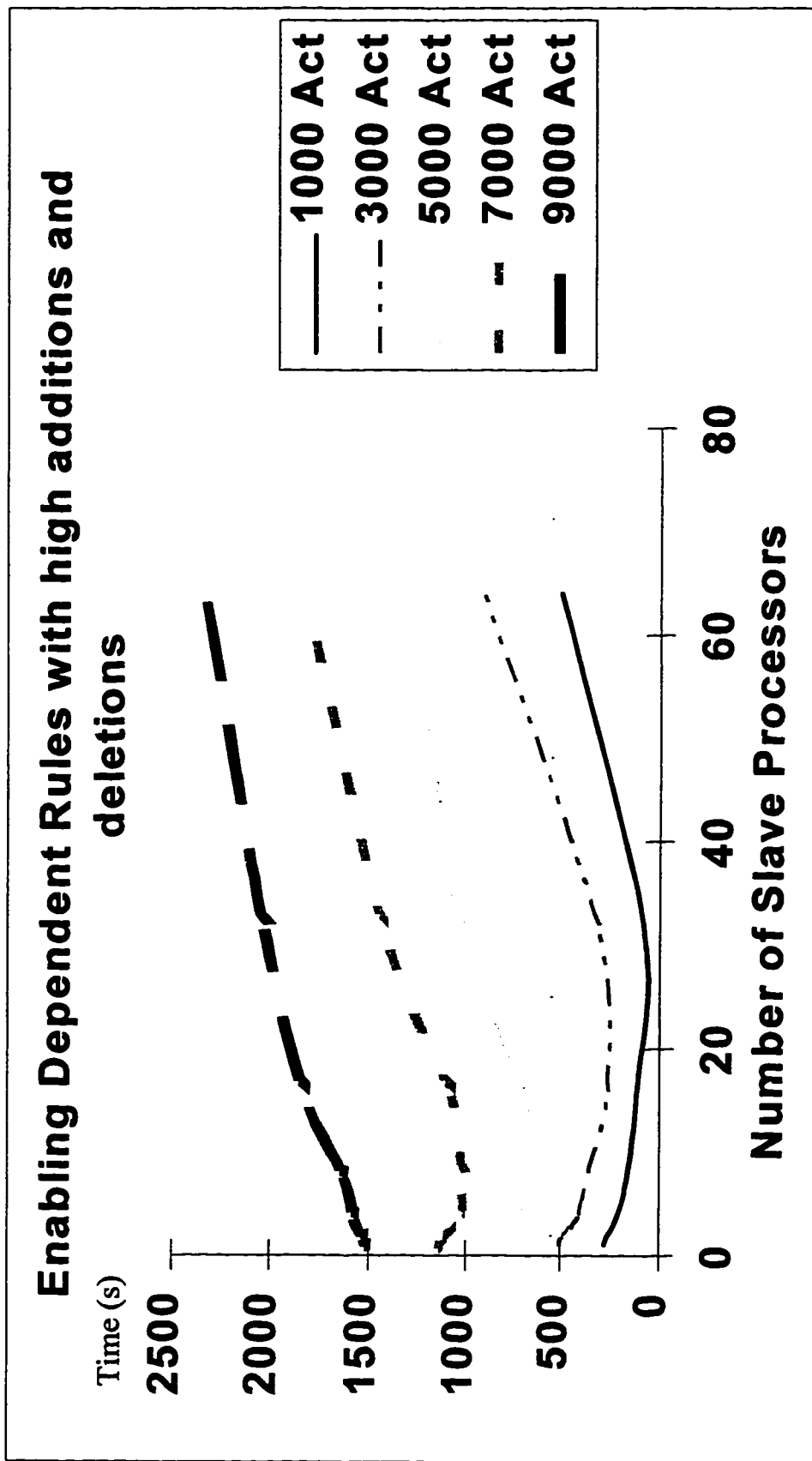


Figure 22: Enabling Dependent Rules with high additions and deletion , Actual Measured Time.

Enabling Dependent Rules with high additions & deletions

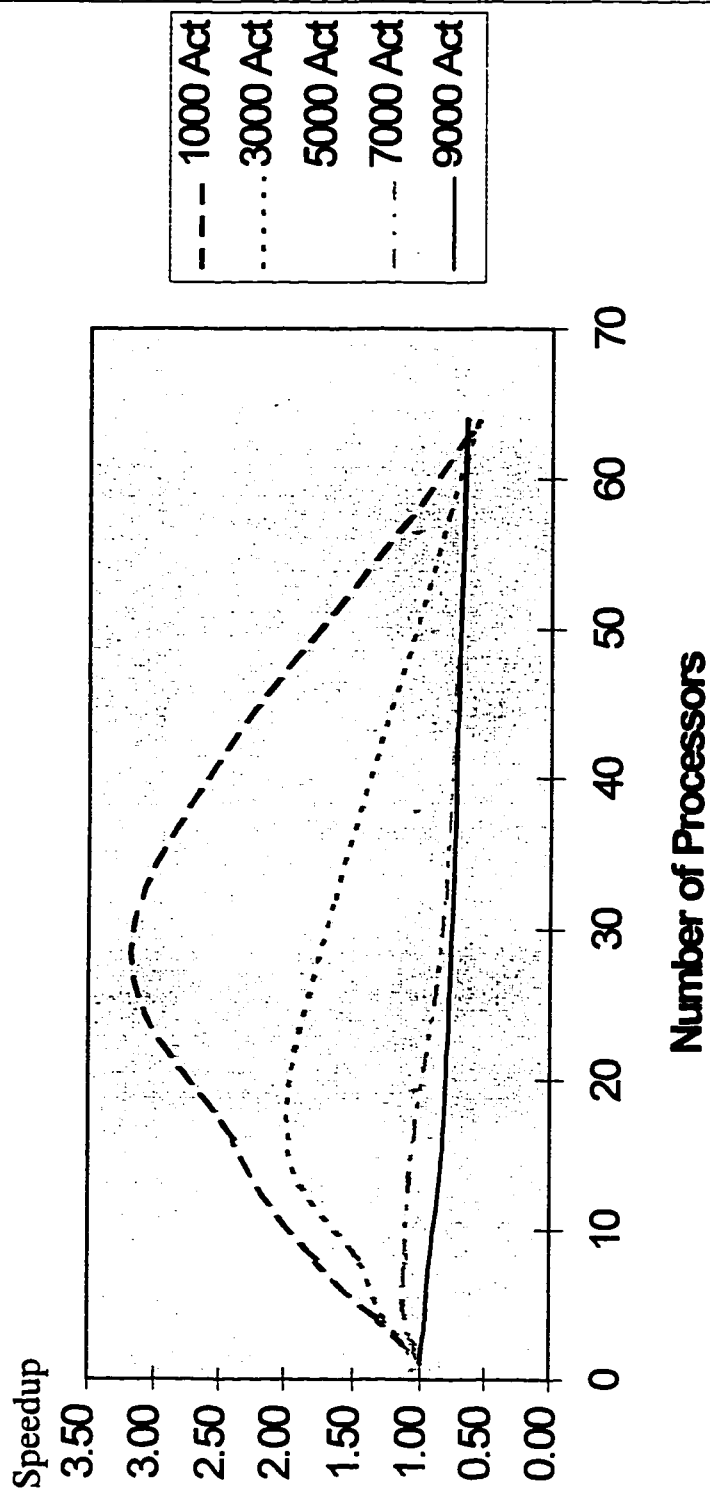


Figure 23 : Enabling Dependent Rules with high additions and deletion , Speedup.

Table 9: Actual time measured in Case 5.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	272	505	805	1114	1489
2	254	487	753	1108	1509
4	208	402	705	1003	1554
8	154	350	656	1005	1603
16	114	257	686	1054	1809
32	88	308	1000	1404	2003
64	503	909	1301	1806	2306

Table 10: Speedup data calculated for Case 5.

	1000 Act	3000 Act	5000 Act	7000 Act	9000 Act
1	1.00	1.00	1.00	1.00	1.00
2	1.07	1.04	1.07	1.01	0.99
4	1.31	1.26	1.14	1.11	0.96
8	1.76	1.44	1.23	1.11	0.93
16	2.39	1.97	1.17	1.06	0.82
32	3.10	1.64	0.80	0.79	0.74
64	0.54	0.56	0.62	0.62	0.65

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

The parallel version of the Rete-Match algorithm that is presented in this thesis for distributed memory architecture is proven to be a very practical approach which is reliable, relatively easier to implement, heterogeneous, scaleable and low communication message passing version of the Rete-Match algorithm. It utilizes run time chances for parallelism to exploit very high degree of parallelism. Another important feature of the model is that its correctness is guaranteed by the system rather than leaving it to be handled by the user, unlike most of the other attempts. The C Language Integrated Production System (CLIPS) and Parallel Virtual Machine (PVM) based implementation show that the Lana-Match model does achieve a significant speed up with respect to the number of processors in situations where the rule set consists of a very large number of independent rules where every rule has a heavy action to perform. On the other hand, the Lana-Match model requires more time than the sequential version of the Rete-Match algorithm if the majority of the actions of the rules has light action as its RHS. In another words, $T_f \ll T_c$. The Lana-Match model is not suitable when communication costs is very high between the CP and the SPs that

is mainly due to the large T_c . However, very large number of independent heavy actions rule are becoming common in some of the new production based system. This makes the speed up and the adaptive feature of the model make the results of this thesis represent major improvement to the Rete-Match algorithm.

More work and analysis is needed to automate or provide utilities that would help balance the load between the Controller Processor and the Slave Processors. The Lana-Match model has enough flexibility to make it balance its load. In this thesis, it is recommended to always keep the Controller Processor on the fastest node on the network and match its speed by distributing enough slave processors on different nodes. However, investigating the distribution of the load in a more systematic manure is recommended.

5.2 Directions for Future Research

The Lana-Match model can still gain some more improvement by further investigation of some of the following suggestions:-

1. **Cooperating Knowledge Base System:** The Lana-Match model can easily be extended to build cooperating knowledge base systems which are gaining more interest these days as a more realistic model for realizing how a group of experts cooperate to address a single problem. This is accomplished by eliminating the Controller Processor of the Lana-Match model which transforms the Lana-Match model into a non-master/salve version. The new yielding version can then be

further modified to address many of the issues that need to be addressed by cooperating knowledge base systems. We believe that most of the issues that probably need to be addressed have already been addressed in this thesis.

2. **Study the Impact of HeNCE:** As was explained in Appendix D, HeNCE which was built on the top of PVM could be an excellent tool for building the Lana-Match model. It would be worthwhile to investigate the benefits and/or drawbacks of replacing PVM with HeNCE as a building tool for the Lana-Match model. However, we anticipate that, replacing PVM with HeNCE would make the model benefit from the following planned features that HeNCE's researchers are planning to address with the new versions of HeNCE:- fault tolerance, processor migration, shadow execution facilities, toolkits for graphical assembly of concurrent application from skeleton templates, hierarchical visualization of application execution and concurrent debugging facilities.
3. **Implement a Heterogeneous Model of the Lana-Match Model:** Although the heterogeneity of the Lana-Match model is one of its main features, this feature was not utilized. In this thesis, we think that it would be worthwhile to implement and evaluate heterogeneous parallel versions of this model; either using heterogeneous knowledge bases systems languages, heterogeneous computers, and/or heterogeneous computer network.
4. **Fine Grain Parallelization of the Controller Processor:** As was explained earlier, the Controller Processor can benefit from most of the fine grain improvements ideas and research that are suggested to improve the performance of the Rete-

Match algorithm. We think it would be worthwhile to try to speed up the Controller Processor of the Lana-Match model using some of these fine grain parallelization suggestions.

Appendix A

The Rete-Match Details

The Rete-Match state machine can be described by describing the Build, Match, Drive, Retract, Conflict resolution and the Engine functions. The behavior of these function diatribes what is being referred to as the Rete-Match algorithm. Figure-1 outlines the Rete-Match machine.

The Rete-Match Functions

1) Build Function

Objective:

To build the Pattern Network and the Join Network.

Description:

The build performs the following tasks on each LHS of each production:-

1. Determine the location of each variable within the pattern.
2. Check the semantic of using these variables and report errors if exist.
3. analyze LHS to determine where variables are being bound.
4. For every field in a pattern, generates a pattern network expression.
5. For every pattern in the LHS, generates a join network expression.
6. Integrate information and expressions generated into the pattern network and the join network. As, much as possible, all expressions generated in should be integrated in the pattern network.
7. Take advantages of the potential to share common expression a among pattern and joins.

2) Match Function

Objective:

Search the pattern network to find all the patterns that match the inserted fact and store this fact at the Alpha memory of the leaf (end node) of every successfully matched pattern.

Description:

Search the pattern network (tree) in a Depth First Search fashion.

1. Set a marker at the first field of the fact and set a pointer at the root of the pattern network.
2. If the pattern node intended to match a single field

Then

If no expression is associated with the pattern node
 then increment the fact marker and go one node down
 else if (evaluate expression of this node with the current field)

Then

increment the fact marker and go one node down
 then Backtrack because of unsuccessful match.

3. If (successful match) then

Store this fact in the Alpha memory of the leaf node

Call Drive function to drive this fact to all the join nodes connected
 to this leaf pattern node.

3) Drive Function

Objective:

Update the join network when ever an added fact has successfully matched a
 pattern the pattern network.

Description

Notes:

1. When a new fact is being asserted in the join network, the Drive function handles high-level updating of the join network.
2. All Pattern nodes enter the join nodes from the RHS, but all join nodes enter other other join nodes from the LHS.
3. If the join was entered from the LHS, the partial match is stored in the Beta memory of the join. Partial matches entering from the RHS are already stored in the Alpha memory of the pattern network.

Algorithm

1. If the join being updated is a terminator join (i.e. the last join of a rule)

Then

Add an activation to the Agenda.

2. If the join being updated is a single entry join

Then

If LHS entry form a pattern node

Then

A copy of the alpha partial match is made and sent to the child joins of
 the current join using the Drive function recursively.

If LHS is from a NOT join node

Then

If (the count of facts matching the condition element) > 0

Then Increment the counter

Else Set the counter = 1

Remove this partial match from all the descendent join

3. If (the join being updated is a double entry join) Then
 If No expression is associated with the node
 Then EXP_VALUE = TRUE
 Else EXP_VALUE = CALL_FUNCTION COMPARE
 If EXP_VALUE = TRUE
 THEN
 If +RHS & +LHS THEN
 Send the concatenation of the Alpha and the Beta memory
 to all child joins of the current join using the Drive

If +RHS & -LHS & Partial match is entering from the LHS Then
 IF At the end of all memory comparison, count = 0
 Create a new partial match by adding this fact to the
 match in the Beta memory
 If +RHS & -LHS & Partial Match is entering from the RHS Then
 If count > 0 Then Count = Count + 1
 Else
 Count = 1
 Remove all partial matches that contain this fact from
 all the descendent join of the current node.

FUNCTION COMPARE
 If the join was entered from the RHS
 Then
 Compare the entering partial match to each partial match in the B-Mem
 If the join was entered from the KHS
 Then
 Compare the entering partial match to each partial match in Alpha-Mem.
 END COMPARE

4) Retract Function

Objective:

Update the join network when ever a deleted fact has successfully matched a pattern in the pattern network.

Description

It performs the opposite function as the Drive function.

5) Conflict Resolution Function

Objective:

Place a new activation on the agenda based on the current conflict resolution strategy. The agenda is the list of the rules which have their conditions satisfied and have not yet been executed.

Description

Newly activated rules are placed above all rules of lower salience and above all rules of higher salience. Among rules of equal salience, the current conflict resolution strategy is used. The current conflict resolution rules are any one of the following:-

- 1) **Depth Strategy** : Newly activated rule is placed on the top of rules of the same salience.
- 2) **Breadth strategy**: Newly activated rule is placed below all rules of the same salience.
- 3) **Simplicity Strategy**: Among rules of the same salience, newly activated rule is placed on the top of all rules with equal or higher number of comparisons that must be performed on the LHS.
- 4) **Complexity Strategy** : Among rules of the same salience, newly activated rule is placed on the top of all rules with equal or lower number of comparisons that must be performed on the LHS.
- 5) **Recency Strategy** : Among rules of the same salience, newly activated rule is placed based on its recency.
- 6) **Index strategy**: Among rules of the same salience, newly activated rule is placed based on its index.
- 7) **Random Strategy** : Among rules of the same salience, newly activated rule is placed randomly.

6) Engine Function

Objective:

Fire all rules until the agenda is empty, or rule execution errors occurred.

Description

While the agenda is not empty

**Perform all the actions of the RHS of the rule on the top of the agenda.
(add a fact or delete a fact)**

Remove this activation from the top of the agenda.

Refresh the agenda.

End While.

Appendix B

Characteristics of Parallel and Distributed Processing

No matter what sort of parallelism we are studying (i.e. massively or regular), there are some characteristics that are shared among all the parallel processing approaches. These characteristics are important in understanding and classifying these approaches. Following, is a list of most of the common characteristics[9]:

1. Granularity

The granularity of a parallel computer is the size of the units by which work is allocated to processors. It determines the size of the processors of the machine and affects the number of processors. There is always a trade-off between the size and the number of processors. Parallel computers can be classified into three grain sizes:

- a) Coarse-Grain Parallel Computers (CGPC):CGPC have small number of large and complex processors. Cray computers (i.e. 2,XMP and YMP) are good examples of coarse-grain parallel computers.
- b) Medium-Grain Parallel Computer (MGPC): MGPC use inexpensive but powerful microprocessors, such as National Semiconductor's 3200, Motorola's MC6800 and M88000 series. Most commercial parallel computers systems marked today can be considered as medium-grain parallel computers. Encore Computer's bus based Multimax, BBN's Butterfly, and Intel's iPSC hypercube are examples of medium-grain parallel computers.
- c) Fine-Grain Parallel Computers (FGPC): FGPC use large number of small and simple processors. The connection machine of Thinking Machines Corporation, the STRAN and MPP of Goodyear Aerospace are examples of fine-grain computers.

2. Throughput (or bandwidth)

It's the rate at which communication can take place between any two processors or a processor and a memory element.

3. Latency

It is the total time required for a message to go from source to destination.

4. Architectural Style

Parallel computers architecture can be divided into four styles:-

- a) Von Neumann-based style: It consists of interconnecting two or more Von Neumann type uniprocessors in variety of configurations. This style is also referred to as control-driven. Most of the parallel computers developed to date are still based on this style. Parallel computers based on this style are classified according to how they processor the program instructions and data streams as follows:-
 - Single Instruction Single Data (SISD): SISD parallel computers are the traditional Von Neumann uniprocessor. It's based on extending the horizontal microprogramming concept, which enable us to interconnect multiple uniprocessors and control them simultaneously and synchronously with very long instruction word (VLIW).
 - Multiple Instruction Single Data (MISD): At any given time , consecutive instructions of a program are in different stages of execution by advancing through pipelines of functional units in staggered fashion, one function at a time. This style include vector-array processors, pipelines processors, associative and orthogonal processors.
 - Single Instruction Multiple Data (SIMD): In this style the N processors may be assumed to hold identical copies of a single program, each processor's copy being stored in its local memory. The processors operate synchronously, at each step, all processors execute the same instruction, each on different datum.
 - Multiple Instruction Multiple Data (MIMD): This style can support multiple instruction streams by pipelines or separate complete processors. This style is further classified depending how processors and memories are connected.
- b) Dataflow style: It is based on the concept of executing program instructions as soon as their operands are ready instead of following the sequence dictated by the ordering of program instruction as in the Von Neumann style. It is also referred to as data-driven.
- c) Reduction style: It is based on the concept of carrying out instructions when results are needed for other calculations. Programs are viewed as nested applications and execution proceeds by successively reducing innermost applications according to the semantics of their respective operators unit there are no further application. This style is referred to as demanded-driven.

- d) Hybrid of dataflow and reduction: In this style, processors work first on the instruction demanded of them if the corresponding operands are ready. If the operands are not available the processor demand them from other processors while working on lower priority items.

5. Memory System

The memory system in parallel computers can be divided into two main categories:-

- a) Distributed Memory System: in this case every processor has it's own local memory. Processors could be communicating through messages passing.
- b) Shared Memory System: in this case a single memory is shared by the N processors. Through this memory processors communicate and exchange data.

Appendix C

Parallel Production Systems Correctness

This is a quick summary for the different correctness mechanisms that were used by the previous efforts to address the issue of providing parallel versions of the Rete-Match algorithm. These approaches can be grouped into two main categories:-

1. Parallel Match Algorithms

Parallel match algorithms category [32] exploits fine-grained parallelism using either one of the following two techniques:-

- Partitioning the Rete-Network and mapping the partitions onto the Multiprocessors.
- Breaking up the memory nodes and allocating them to different processors.

The main advantage of these techniques is that there is no correctness issue to be addressed. However, one major issue still needs to be addressed very carefully by these two techniques. This issue is concerned with the efficient partitioning of the network. But unfortunately, even if the network was partitioned efficiently, these techniques are still suffering from the following drawbacks:-

- a) Partitioning the network is done at the compile time for the rules which make these techniques very static. It also prevent them from utilizing run time opportunity for parallelism. This makes these techniques not very useful in situations where the set of rules change dynamically, like expert systems that adapt machine learning concepts. Another simpler case that demonstrate the short coming of these techniques, assume we have only one rule to fire and few thousands of facts that can instantiate few thousands instantiation of this rule. Although that this is a good case for parallelism, these techniques can never utilize the few thousands of chances to fire these rules in parallel.
- b) Reported results for these techniques, concluded that the parallel speed up that can be achieved is less than 10-folds regardless of the number of processor used. This makes this techniques suffer from limited (or no) scalability.
- c) These techniques do not handle the Small Cycle problem which refers to the fact that most of the production system programs affect only few nodes at each cycle. This simply make most of the available processors with not enough work to do. Although, this fact can be changed by writing parallel production system programs, but still this approach can not utilize the parallel feature of production

system. That is what make researchers prefer the multiple parallel rule fire approach.

2. Multiple Parallel Rule Firing Algorithms

On the other hand, multiple rule firing algorithms category [32] attempts to increase the availability of parallelism by parallelizing not only the match phase, but all phases of the inference cycle. It parallelize the act phase which elevate the Small Cycle problem. It also executes different inference phases concurrently which reduce the variance in processing time. However, the correctness of the algorithm is the main issue to be addressed by any multiple rule firing algorithm designers. The correctness of multiple rule firing algorithm is guaranteed by ensuring that the parallel version of this algorithm is equivalent to the sequential version of this algorithm. Researchers also identified two criterion that ensure this equivalence. These criterion's are the Compatibility and the Conversion criterion's. The following is a brief description and methods that were used to guarantees these criterion's.

First, The Compatibility Criteria

Selecting a set of rules that can be fired concurrently, based on the dependency these rules share, is the hart of these two criterion's. This means that, a set of rules instantiations is allowed to fire concurrently if they do not interfere with each other (i.e. firing a rule instantiation in that set does not prevent other instantiations from being fired). This is true if there are no inter-instantiations data dependency. A set of nonintervention rule instantiation is called a set of compatible rule instantiations. Firing a set of compatible rule instantiations concurrently would reach a state which is reachable if they are executed in some sequential order. But how can we determine rule dependencies?

There are several approaches with varying degree of computational costs and storage requirements to determine rules dependencies. However, all the reported approaches are based on analyzing the data dependency graph that represent the rules set.

A data dependency graph G where every RI_i is a Rule Instantiation (RI) that belongs to the rule set and every $RI_i \rightarrow RI_j$ is a dependency relation that can exist between any two Rule Instantiations. The data dependency graph is constructed by first analyzing the data dependencies between different pairs of rule instantiations and find out all the dependency relations that exist between them and then combining the result into a graph using some preset dependency conditions.

There are three types of dependency-relations that can exist between any two different pair of rule instantiations. The following is a brief description of each one of these relations:-

- 1) Inhibiting Dependency: Rule Instantiation RI_i is said to inhibit RI_j written $RI_i \Rightarrow RI_j$ if firing RI_i adds or deletes facts that will make RI_j is no longer satisfied (i.e. Deactivate it).

- 2) **Output Dependency:** Rule Instantiation RI_i and RI_j are said to have output dependencies written $RI_i \leftrightarrow RI_j$ if firing RI_i would delete facts that were added by firing RI_j .
- 3) **Enable Dependency:** Rule Instantiation RI_i is said to enable RI_j written $RI_i \Downarrow RI_j$ if firing RI_i adds or deletes facts that will make RI_j satisfied (i.e. Activate it).

Constructing the data dependency graph from the above dependency relations, one can use either of the following two dependency conditions:-

Pairwise Condition approach

Determining the set of compatible rules can be achieved using the following Pairwise Conditions on the set of Rule Instantiations:-

Rule Instantiation RI_i and RI_j are compatible if the following conditions are satisfied:-

- i) RI_i does not inhibit RI_j .
- ii) RI_j does not inhibit RI_i .
- iii) RI_i and RI_j do not have output dependencies.

Although the pairwise condition guarantee that ALL sequential execution orders to be equivalent to the parallel execution but it is very expensive to implement.

The Cyclic Condition approach

The strict requirements of the Pairwise Condition which requires that ALL sequential execution order to be equivalent to the parallel execution is relaxed at the cyclic condition which requires only the existence of a sequential execution that is equivalent to the parallel execution to capture more parallelism. The cyclic condition can be summarized as follows:-

A set of Rule Instantiations are compatible if they have no data dependency cycle. Rule instantiations $RI_1, RI_2, RI_3, RI_4, \dots$ and RI_n form a data dependency cycle if

$$RI_1 \Rightarrow RI_2 \Rightarrow RI_3 \Rightarrow RI_4 \Rightarrow \dots \Rightarrow RI_n \Rightarrow RI_1$$

Determining the compatible rules is done by partitioning the rule set into sets that do not form data dependency cycle.

Second, The Conversion Criteria

Since the compatibility criterion has a local view of the execution (i.e. within one cycle), the compatibility criterion is not sufficient to guarantee the correctness of the final solution. Thus the rule instantiation that may turns out to be right in one cycle may turn out to be wrong in later cycle, which can make the system reach erroneous conclusion. Therefore, the compatibility criterion is complemented by the another criterion called the Conversion Criterion. This criterion guarantee that all execution cycle converges to the right solution.

There are two approaches to guarantee the conversion criterion. The first of these approach is based on a deterministic execution model while the other approach is based on a non-deterministic computational model. Every one of these approaches is described next:

1. **Using a Non Deterministic Computational Model:** Building a parallel non deterministic computational model is based on dividing the rule set into different groups either manually [32], using context analysis or using a specialized languages. Next, different conflict resolution strategies is applied to each one of these sets. The correctness of the partitioning is left to be the programmer responsibility. Some of the reported approaches [32] give the programmer some utilities that can help him to guarantee the correctness of his partitioning. Other approaches do not. Although the non deterministic model sounds fascinating theoretically, developing and testing expert system using this approach is a nightmare.
2. **Using a Deterministic Execution Model:** Programmers who develop expert system using a deterministic computational model help the system by applying a set of rules that describe the relations among every pair of the rule set. These rules about the rules set are referred to as the Mete-Rule. These mete-rules explain to the system the relations between the rules and which rules can be executed with each other and which can not. It is the programmer's responsibility to develop and test his mete-rule. Although this approach sounds easy and more realistic for small expert systems, it is extremely difficult for large scale expert systems. Another drawback for this approach is that it does not exploit run time parallelism.

Appendix D

The Lana-Match Model Implementation Alternatives

As the old saying goes, "There are many ways to skin a cat", there are many way to implement the Lana-Match Model. The two main ingredients to implement the Lana-Match model are:-

1. A Rete-Match Based Production System Package

Selecting a Rete-Match based production system depends on the platforms and network environment that your implementation would be running on. The source code and enough knowledge of the implementation details of the selected package are also needed. The selected package would be used to implement the Controller Processor and also to implement the Slave Processor. This is accomplished by transforming a copy of the selected package into a Slave Processor and another copy into a Controller Processor. Transforming the selected package into a Slave Processor requires modifying it to receive assertion commands form the Controller Processor and also to translate all the assertions and deletions of facts that are generated from firing in rules into action commands to be send pack to the controller. On the other hand, transforming the selected package into a Controller Processor would involve major modifications to the selected package. These modifications include, changing the agenda into the Lana-Match Master Agenda, adding a Scheduler and a Rescheduler. They also include changing the package engine into the Lana-Match Main loop. The following is a list of the possible alternatives for selecting a Rete-Match based production system package:-

a) Do It Yourself!

The first option that might seems attractive is to develop a Rete-Match based production system package. However, this is not the best alternative. From experience, it was noticed that, on the average, a one man year would be needed to develop such a package. Using or buying an of- the-shelve Rete-Match based package is a better alternative.

b) Using OPS-5

The first Rete-Match package that was ever developed is OPS-2. This package was developed by Dr. Forgy [18]. It was implemented initially using LISP. However a C version of this package is also available. OPS-2 was modified later on to be the OPS-5 which was very popular package during 1970-1990.

c) Using CLIPS

The C Language Integrated Production System (CLIPS) [59-60] that was developed at NASA's Johnson Space Center during 1985-1993 started receiving a widespread acceptance throughout the public and private sectors to be one of the most powerful Rete-Match based production system packages. CLIPS has more than 5000 users including NASA sites and branches, numerous federal bureaus, government contractors, 200 universities and many companies. Because of its portability, extensibility, capabilities and low-cost for both the executable and the source code, we decided to implement the Lana-Match Model using the C Language Integrated Production System (CLIPS).

Integrating CLIPS with external functions or application is one of the most important feature that we used heavily to implant the Lana-Match Model. To help us explain the Lana-Match implementation, this section summarizes how to add external functions to CLIPS and how to pass argument to them and return values from them. All external functions must be described to CLIPS (as in figure-8) so they can be properly accessible by CLIPS programs. User-Defied function are described to CLIPS by modifying the function UserFunctions. This Function is found at the CLIPS source file main.c. The following is a C code fragment that explain this. The same regular C return() function mechanism is used to pass argument from the external function to back CLIPS. The returned argument should have the type of the external functions. The first step an external function should do to receive arguments from CLIPS is to determine the number of argument that have been passed from CLIPS to the external function. RtnArgCount() is the function that return an integer number that tells how many argument with which this external function was called. ArgCountCheck() function can be used to for error checking if a function expects minimum, maximum, or exact number of arguments and ArgRangeCheck() function can be used for error checking if a function expects a range of arguments. The next step is to receive that argument from CLIPS. Depending on the argument type, the argument can be received by any one of the functions that are described at table-2. In this thesis, most of the argument that we passed were either string or integer.

d) Many Other to Come

There are also many other packages that probably did not become as popular as OPS-5 nor CLIPS but they can still be used to build the Lana-Match Model. Many packages are still under development or is being used on a limited scale. The main issue here is the availability and the ease of the source code of the package.

```
UserFunction()
{
/*=====*/
/* Declare your C function if necessary          */
/*=====*/
int DefineFunction(function_name,function_type,function_pointer,
                  actualfunctionname);
```

```
char *function_name, *function_type, *actualfunctionname;
int (*function_pointer); }
```

2. A Message Passing Communication Mechanism

Having identified a Rete-Match based production system package to be transformed into Controller Processor and Slave Processors, the next step is to decide on the communication methods for these deferent processors. Many alternatives are available depending on wither these processor will be executed concurrently on a single faster CPU or they will distributed over more then one CPUs. If these processors (CP & SPs) will be executed concurrently on a single CPU, the regular Inter Processor Communication (IPC) mechanism can be used (i.e. Pipes, FIFOs-Named pipes, Message Queues and Sockets). If these processors will be executed on different CPUs, Remote Procedure Calls (RPC), Transmission Communication Protocol / Internet Protocol (TCP/IP), Parallel Virtual Machine (PVM), Portable Parallel Programming Paradigm (P4), HeNce or Linda can be used. Every one of these approaches has its advantages and its disadvantages depending on the environment that the implemented Lana-Match Model is targeted to be executed on. The following is a brief summary of each of these approaches and a quick summary to the advantages and disadvantages of these approaches:-

Inter Processor Communication (IPC) Level: regular Inter Processor Communication [55] mechanisms are the best alternatives, if the Control Processor and the Slave Processors are targeted to be executed concurrently on a single CPU. These mechanism are based on exchanging information across the Kernel. For Unix environment these IP mechanisms could be Pipes, FIFO named pipes, Message Queues and Sockets. However, sockets based mechanisms are the best form of the IPC mechanisms because they also allow access to the IP layer which make the code easier to be changed to run on different CPUs.

Transmission Communication Protocol /Internet Protocol (TCP/IP) Level: if the Control Processor and the Slave Processors are targeted to be executed on one or more CPUs that are networked using Transmission Communication Protocol /Internet Protocol (TCP/IP) [62], direct interfaces to TCP/IP protocol can be implemented to provide communication links between these processors. Interfaces to the TCP/IP protocol are sometimes referred to as the Network IO/ Socket interfaces. Sockets are generalizations of the a Unix like file access mechanisms that can be used to establish a communication links between a Control Processor and a Slave Processor. These links are accomplished by the following sequence:- create a socket using the "socket" system call, bind this socket to local addresses using the "bind" system call., connect this socket to a destination addresses using the "connect" system call, sending messages through the sockets can be done using the "write, written, send, or sendto" system calls, receiving data though the socket can done using "read, readv, recvfrom, or recvmsg" system calls and when finished, the socket needs to be closed. In addition to these system calls, different Unix versions provide different system calls and

library routines that perform useful functions related to network (e.g. "getpeername" to determine the address of the peer to which a socket is connected, "getsockname" returns the local address associated with a socket, "setsockopt and getsockopt" to set/get socket options, ... etc) . The main difference between a system call and a library routine is that system calls functions pass control back to the computer operating system, while library routines are just like regular programming procedures. The socket interface is becoming a very popular- and is widely supported by many vendors. Vendors who do not offer socket facilities in their operating system often provide a socket library routines that makes it possible for programmers to write applications using socket calls even through the underlying operating system uses a different set of system calls. The socket interface level is a very efficient lower level approach. However, it requires more development and maintenance costs since the programmer needs to develop and maintain all the code necessary to guarantee correct and reliable communications using the given socket system calls and library routines primitives.

Remote Procedure Call (RPC) Level: Remote Procedure Call [7] is a higher level approach than the socket based approach. It gains its power from the Protocol Description Language (PDL) and the RPCGEN compiler. In this approach, only the data structure that is needed to exchange data between the Control Processor and the Slave Processor and the different functions that would be operating on them are needed to be described using the Protocol Description Language. These data structures and functions are referred to as the as the communication protocol. The RPCGEN compiler is used to translate PDL into lower level communication calls to establish the communication mechanism. This simple abstraction makes dealing with the complexity inherited in socket programming more manageable. However, this approach is still suffering from a few problem areas such as: transparency and global variables.

PVM & P4 Level: in addition to hiding the detail of the underling network, the Parallel Virtual Machine (PVM) [10] and the Portable Programs for Parallel Processors (P4) [8,10] permit a network of heterogeneous UNIX computers to be used as a single large parallel computers while handling all communications and reliability details. Thus, PVM and P4 provide a more efficient, powerful and reliable parallel programming environment. The main differences between PVM and P4 are the origin, the efficiency, and the popularity. The development of PVM started 1989 at Oak Ridge National Laboratory (ORNL) while the P4 was developed at the Argonne National Laboratory during 1990. Although P4 is more efficient than PVM, PVM is the most popular parallel programming environment. It has been ported to almost every known machine. Moreover, most of the vendors considered PVM to be the standard parallel programming environment. For its portability and popularity, we decided to use PVM to implement the Lana-Match Model in this thesis. Parallel Virtual Machine (PVM) is a software that permits a network of heterogeneous Unix computers to be used as a single large parallel computer. The large computation

problem can be solved by using the aggregate power of many computers. Under PVM, a user can define a collection of serial, parallel, and vector computers appears as one large distributed-memory computer. PVM also supports heterogeneity at the application, machine, and network level. In other word, PVM allows application tasks to exploit the architecture best suited to their solution. It also handles all data conversion that may be required if two computers use different integer or floating point representations. Finally, the virtual machine created by PVM can be interconnected by variety of different networks.

The PVM system is composed of two parts. The first part is a daemon, called `pvm3d` that resides on all the computers making up the virtual machine. When a user wants to create a new virtual machine, he first creates a virtual machine by starting up the PVM. The PVM application can then be started from a regular UNIX prompt on any of these hosts. Multiple users can define multiple overlapping virtual machines and each user can execute several PVM application on the same PVM machine simultaneously. The second part of the system is a library of PVM interface routines which are user callable routines for message passing, spawning processors, coordinating tasks, and modifying the virtual machine. Application programs must be linked with this library to use PVM. To create a virtual machine the `pvm_myid()` needs to be called to enroll the main processor on its first call and generate a unique task id (tid). After registering this processor with the PVM, the `pvm_spawn()` is used to start `n` copies of an executable file task on the virtual machine. Finally, `pvm_exit()` is used to remove this processor from PVM. However, this will not kill this processor, but instead the processor will continue as a regular Unix processor. On the other hand, `pvm_kill(tid)` kills the task that is identified by a given task id (tid). Sending a message is composed of three steps in PVM. First, a send buffer must be initialized by calling `pvm_initsend()`. Second, the message must be "packed" into this buffer using any of the `pvm_pk*` routines. Third, the complete message then should be sent to the other processor by calling the `pvm_send()`. Two steps would be needed to receive a message through PVM. The first step is to receive the data using `pvm_recv()`. The second is to unpack the data using the `pvm_upk*` routines. The message should be unpacked the same way as it was packed. It is important to notice that what we presented here is a very brief summary for PVM. PVM is capable of a lot more functions. For full information on PVM, please refer to the post script copy of all the PVM documentation that are given with this thesis distribution software. After installing the Lana-Match software, these documents can be found at `{$LANA_RETE/pvm/docs }` directory. However the information that PVM functions that we described in this section is the most frequently used functions. which we also heavily used to implement the Lana-Match Model.

HeNCE and Linda Level: On the top of PVM and P4, two more advanced parallel programming environments are also available. These environments are HeNCE and Linda. Heterogeneous Network Computing Environment (HeNCE) [6] is an active research project. A prototyped of HeNCE is currently available. HeNCE is an X-

window based software environment designed to assist scientist in developing parallel programs that run on a network of computers. It provides the programmer with even higher level of abstraction than PVM. In HeNCE, the programmer explicitly specifies parallelism of a computation by drawing graphs. The nodes in a graph represent user defined subroutines and the edges indicate parallelism and control flow.

The HeNCE programming environment consists of a set of graphical tools which aid in the creation, compilation, execution and analysis of HeNCE programs. The main components consist of a graph editor for writing HeNCE program, a building tool for creating executable, and a trace tool for analyzing and debugging a program run. These tools are integrated into a window based programming environment. Researchers are also planning to include fault tolerance, processor migration, shadow execution facilities, toolkits for graphical assembly of concurrent application from skeleton templates, hierarchical visualization of application execution and concurrent debugging facilities. Depending on the fulfillment of HeNCE promises, we think that HeNCE would be the better alternative to implement the Lana-Match Model when HeNCE become available. Linda is a new concurrent programming mechanism that was introduced by Gelernter[9] as the forth basic concurrent programming mechanism. It differs from the three basic kinds concurrent programming mechanism of the time (i.e. monitors, message-passing and remote operations) in requiring that messages should be added in tuple from to an environment called tuple space where they exist independently until a processor chooses to receive them. This abstract tuple space environment form the bases for Linda. In this model, a processor generates an object called a tuple and place it in a globally shared collection of tuple space. Theoretically, the object remains in the tuple space forever unless it is removed by another processor. Linda consists mainly of the following four operations:-

- out(t) : This function adds a tuple t to the tuple space.
- in(t) : This function search for a tuple in the tuple space.
- eval(t) : This function is similar to out(t) with exception that the tuple argument to eval is evaluated after adding t to the tuple space.
- rd(t) : This function is similar to in(t) except that the matched tuple remains in the tuple space.

Currently two compatible prototype implementations of Linda are available on the top of P4. The first implementation takes advantages of the shared memory architecture, and the other implementation utilizes the resources of networked machines. In this thesis, we see a Linda based model to be more suitable for shared memory architecture. Thus, we don not recommend the use of Linda to implement the Lana-Match model. However, we think that the Linda model can play a major role in Parallelizing the Rete-Match algorithm for shared memory architecture.

REFERENCES

1. Achary, A.. and Tambe,M. "Production System on Message Passing Computers: Simulation results and analysis". Proc. 1989 International Conference on Parallel Processing, 1989.
2. Ahmad Ishfaq, "Semi-Distributed Load Balancing for Massively Parallel Multicomputer System ", IEEE Transactions on Software Engineering, Vol 17, No 10, October 1991, pp 987-1004.
3. Akl, Selim G, *The Design and Analysis of Parallel Algorithms*, Prentic-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1989.
4. Anderson, R. J, and L Snyder, "A Comparison of Shared and Nonshared Memory Models of parallel computers ", Proc IEEE, Vol. 79, No 4, 1990, pp 480-487.
5. Barr, A., Feigenbaum E. , *The handbook of Artificial Intelligence*. Wiliam Kaufmann, Inc. 1981.
6. Beguelin, Adam,"Graphical Development Tools for Network-Based Concurrent Supercomputing", Tech. Report for the project under contract DE-AC05-84OR21400, the U.S. Department of Energy, the applied Mathematical Science subprogram, 1990, further question questions or problems regarding HeNCE send email to hence@msr.epm.ornl.gov.
7. Bloomer, John, *Power Programming with RPC*, O'Reilly & Associates, Inc., 1991.
8. Butler, R., "Monitors, Messages, and Clusters: The P4 Parallel Programming System", National Science Foundation project grant CCR-9121875, 1991. The paper can be obtained from the following mail address: rbutler@sinkhole.unf.edu.
9. Butler, R., "P4_Linda: A Portable Implementation of Linda", National Science Foundation project grant CCR-9121875, 1991. The paper can be obtained from the following mail address: rbutler@sinkhole.unf.edu.
10. Butler, R., "User's Guide to the P4 Parallel Prog. System", National Science Foundation project grant CCR-9121875, 1991. The paper can be obtained from the following mail address: rbutler@sinkhole.unf.edu.

11. Bursky, Dave, "Communication among processors sustains fast, massively parallel computer", *Electronic Design*, Vol. 38, January 1990, PP 31-32.
12. Curran, Lawrence, "Massively Parallel Fast Work, Slow sales", *Proc IEEE*, Vol. 64, July 1991, pp 28-32.
13. Dayton Leigh, "Japan's thinking computer goes on show", *New Scientists*, 1991, pp 15-57.
14. Decegama, Angel L, *The Technology of Parallel Processing*, Vol. 1, Prentice-Hall Inc., A Division of Simon & Schuster, Englewood Cliffs, NJ 07632, 1989.
15. Douglas, C. C., "Parallel Programming Systems for Workstations Clusters", Yale University, Department of Computer Science, Research Report: YALEU/DCS/TR-975, August 1993.
16. Filperin, S. A. and L. Gravano, "Routing Techniques for Massively Parallel Communication", *Proc IEEE*, Vol. 79, No 4, April 1991, pp 488-503.
17. Fischer, James R., "Massively Parallel Computers hold key to scientific discovery, Saya Froniers 88 keynoter", *Computer*, Vol 22, March 1989, pp 98-99.
18. Forgy, C. L. *On the efficient implementations of production system*, Ph.D. thesis, Carnegie-Mellon University, 1979.
19. Forgy, C.L. Rete: "A fast algorithm for many-patter many object pattern match problem". *Artificial Intelligence*, Vol 19, 1982, pp17-37 .
20. Gabriel, Richard P., "Massively Parallel Computers: The connection machine and NON-VON", *Science*, Vol. 231, February 1986, pp 975-978.
21. Gamble. R. "Transforming rule-based programs: From the sequential to the parallel", *Proc. 3rd International Conference on Industrial and Engineering Application of AI and Expert System*, July 1990.
22. Geist, "PVM 3 User's Guide and Reference Manual", 1991, This document can be obtained from the following email address: pvm@msr.epm.ornl.gov.
23. Gupta. A., Forgy, C. , Newell, A., Weding, R., "Parallel algorithms and architecture for rule-based systems". *Proc 13th International Symposium on Computer Architecture*, Tokyo, 1986, pp 123-138.
24. Gupta, A., *Parallelism in production systems*. Ph.D. thesis, Carnegie Mellon Univ., March 1986.

25. Gupta, A., Forgy, C., Kalap, D., Newell, A., and Tambe, M.S, "Parallel OPS5 on the encore multimax". Proc. 1988 International Conference on Parallel Processing, Aug. 1988, pp 24-32.
26. Gupta, A. & Tambe, M. "Suitability of message passing for implementing production system". Proc. National Conference & AI, AAAI-88, Aug. 1988, pp 10-24.
27. Hunt, Craig, *TCP/IP Network Administration*, O'Reilly & Associates, Inc., 1992.
28. Ishida, T. "Methods and effectiveness of parallel rule firing". Proc. 6th IEEE Conference on Artificial Intelligence, IJCAI-89, 1989, pp 156-159.
29. Ishida, T., et. al., "Towards the parallel execution of rules in production system programs" . Proc 1985 International Conference of parallel Processing., 1985, pp.568-575.
30. Kelly, M., and Serviora, R, Drete- "A distributed matching algorithm". Tech. Rep.University of Waterloo, Nov. 1987.
31. Kelly, M., and Serviora, R," An evaluation of Drete on CUPID for OPS5", Proc. 11th International Joint Conference on Artificial Intelligence, IJCAI-89, 1989, pp. 84-90.
32. Kuo, S and Moldovan, D. "The state of the art in parallel production systems". Journal of parallel and distributed computing 15, 1992, pp 1-26.
33. Kuo, S.Moldovan D. and Cha. S. "Control in Production Systems with Multiple rule Firing". Proc. 1990 International Conference on Parallel Processing Vol. II. 1990, pp. 243-246.
34. Kuo, S.Moldovan D. "Performance Comparison of models for multiple rule firing". . Proc. 12th International Joint Conference on AI, AAAI-91, 1991, pp. 304-309.
35. Kuo, S. *A parallel asynchronous message-driven production system*. PH.D. thesis, University of Southern California , Sep. 1991.
36. Kuo, S, M. Miranker, D. P., and Browne, J.C., "On the performance of the CREL system" Tech Rep. Department of Computer Science. University of Texas at Austin, Feb. 1991.

37. Lea, R.M. and P. Jalowieki, "Associative Massively Parallel Computers", Proc IEEE, Vol 79, No 4, April 1991, pp 469-479.
38. Li, H and Q. F. Stout, "Reconfigurable SIMD Massively Parallel Computers", Proc of the IEEE, Vol. 79, No 4, April 1991, pp 67-81.
39. McDermott, J., and Forgy, C. "Production system conflict resolution strategies", In Watermanm, D., and Hayes-Roth, F. (eds.). *Pattern-Directed Inference Systems*, Academic Press. New Yourk, 1978.
40. Miranker, D.P., Kuo, C. and Browne. J.C. "Parallelizing transformation for a concurrent rule execution language". Tech. Rep. TR-89-30, Department of Computer Science, University of Texas as Ausin 1989.
41. Miranker, D.P., Kuo, C. and Browne. J.C. " Parallelizing compilation of rule-based programming", Proc 1990 International Conference on Parallel Processing, Vol. II., 1990, pp. 247-251.
42. Miranker, D.P. *TREAT: A new and efficient algorithm for AI production system*, Ph.D. thesis Columbia University, 1987.
43. Moldovan, D. "RUBIC: A multiprocessor for rule-based systems". IEEE Trans. System, Man Cybernet, July , 1989, pp 15-20.
44. Moldovan, D. Lee, W. and Lin. C. "SNAP: A marker-propagation architecture for knowledge processing". Tech. Rep. 90-1, Parallel Knowledge Processing Laboratory, 1990, pp 82-89.
45. Myers, Ware, "Massively parallel systems break through at supercomputing 90", Computer, Vol. 24, January 1991, pp 121-126.
46. Myers Ware , " Faster. .. Caltech dedicate world's most powerful supercomputer", Computer, Vol. 24, July 1991, pp 96-98.
47. Pasik. A. *A methodology for programming production systems and its implications on parallism*. Ph.D. thesis. Department of Computer Science, Columbia University, 1989.
48. Raschid, Louiqa, "A simulation-based Study on the Concurrent execution of rules in a database Environment". Journal of Parallel and Distributed computing 20, 1994, pp 20-42.

49. Rychener, M.D. "Production system as a programming language for Artificial Intelligence Applications", Tech. Rep. , Department of Computer Science Carnegie-Mellon University, Sep. 1976.
50. Scheafer, D.H. , "The Characterization and Representation of Massively Parallel Computing Structure", Proc IEEE, Vol. 79, No 4, April 1991, pp 461-469.
51. Schreiner, F., and Zimmerman, G. "PESA-I-A parallel architecture for production systems". Proc. 1986 International Conference on Parallel Processing, 1986, pp 23-34.
52. Schmolze, J. "An asynchronous parallel production system with distributed facts and rules". Proc. AAAI-88 Workshop on Parallel Algorithms for Machine Intelligence and pattern Recognition, St. Paul M U.S.A., Aug. 1988, pp 76-83.
53. Schmolze, J. "A parallel asynchronous distributed production system". Proc. 8th National Conference on Artifact Intelligence, AAAI-90, 1990, pp. 65-71.
54. Sipelstein, J. M. and G.E. Blelloch, "Collection-Oriented Language", Proc of the IEEE, Vol. 79, No 4, April 1991, pp 504-523.
55. Stevens, W. R., *UNIX Network Programming*, Prentice-Hall Inc., 1990.
56. Stolfo, S., " Initial Performance of the DADO2 prototype", IEEE computer, Mag. Jan., 1987, pp 75-83.
57. Stolfo, S., Dewan, H., and Wolfson. O., "The PARULEL parallel rule language". Proc. 1991 International Conference on parallel Processing, Vol. II., 1991, pp. 36-45.
58. Tanenbaum, Andrew S., *Modern Operating Systems*, Prentice-Hall International, Inc., 1992.
59. *The CLIPS Reference manual I, CLIPS : C Language Integrated Production System*, Software Technology Branch, Lyndon B. Johnson Space Center, Volume 1 and 2, Version 6, Sep 1993.
60. *The CLIPS User & Programmer's manual, CLIPS : C Language Integrated Production System*, Software Technology Branch, Lyndon B. Johnson Space Center, Volume 1,2, and 3, Version 6, Sep 1993.
61. Ulman, Jeffery, *Principles of Database and Knowledge-base systems*. Computer Science press, 1988.

62. Wilson, S., *TCP/IP and NFS Internetworking in a Unix Environment*, Michael Santifaller & Addison Wesley Publishing Co., 1991.