A Distributed Rule Engine Based on Message-Passing Model to Deal with Big Data

Jinghan Wang, Rui Zhou, Jing Li, and Guowei Wang

Abstract—Rule engine is a good way of knowledge representation and inference. It has been extensively studied and used in artificial intelligence. However, because of its low computational efficiency and the limitation of single machine’s capacity, it cannot deal well with big data. In order to address this problem, we propose a distributed implementation of the rule engine based on message-passing model. It is designed to make use of multiple machines in a computing cluster to deal with a large amount of data in a parallel and distributed way. This paper not only describes details of the design and implementation of it, but also shows its high performance through several experiments.

Index Terms—Artificial intelligence, big data, distributed rule engine, message-passing model.

I. INTRODUCTION

Rule engines (or production systems) occupy a significant part in artificial intelligence since they are a good way of knowledge representation and reasoning. They have been widely used to build expert systems with many applications such as business management systems. A rule engine consists of three parts which are a knowledge base containing a set of facts, a rule base containing a set of rules (or productions) and an inference engine. When the rule engine starts, according to a certain match algorithm, the inference engine repeatedly tries to find rules in the rule base that could match facts that are accepted by the knowledge base. As a result we could get and execute desirable actions which are specified in the matching rules.

However, rule engines are computationally expensive and slow. When the size of problems continues to grow, the efficiency becomes much lower. In order to address these issues, lots of research works have been done in past few decades. And one of the most important contribution was the creation of Rete by Forgy in 1982 [1], which hereafter inspired lots of its improvements and modifications including Treat [2], Rete/UL [3], Rete [4], [5]. But these could still not deal well with the situation when number of rules and facts become too large under the limitation of one single computer’s capability. Some other works were about enhancing rule engines’ ability of solving problems in a parallel or concurrent way by using multi-cores or distributed systems [6]-[10]. But they either targeted on special hardware architectures thus lacked flexibility or only offered limited speedup.

Now days, the development of cloud computing offers some new potential methods to speed up rule engines when solving big data. One recent work [11] implemented a rule engine on a map-reduce based architecture. Although map-reduce technology is a popular way of dealing with big data through batch processing in cloud computing, it could not be compatible well with the rule engine involving lots of loops and iterations. Thus the improvement is limited.

In this paper, we mainly propose a new approach to implement rule engines based on a message-passing model as now the latency of communication between computing nodes in one data center has become much lower than before. The core match algorithm used is an improvement of rete algorithm whose main idea is the rete network. And we implement the network into groups of virtual interconnected computing units called processes in the message passing model, which receive and send messages to trigger the process of the match algorithm. We render the rule engine the ability to solve problems in parallel as well as to scale conveniently. So that it could make well use of the computing cluster such as a data center in the cloud environment. Finally, we implement the approach and conduct experiments to show its performance.

The paper is organized as follows. Section II provides background information of rule engines, rete algorithm and the message-passing model we use. Section III discusses how we design our rule engine. Section IV describes details of implementation. Section V is about some experiments we have conducted. Section VI concludes the paper and points out our future work.

II. BACKGROUND

A. Rule Engines

In a rule engine, one domain problem is specified as a set of tuples containing facts that are some assertions, a set of rules related to the facts and a desired or final state. Facts are held in the working memory and named working memory elements (WMEs), while rules are saved in the production memory (or rule base) and named production elements (PEs). Each rule is of the form IF-condition-THEN-action. The IF part of the rule (left-hand side or LHS) consists of a conjunction of condition elements. A condition element comes with a set of tests for attribute-value pairs of a WME, such as constant tests and consistency tests. The THEN part (right-hand side or RHS) specifies the actions that will be performed if the LHS is true.
During the run time, the rule engine executes three-phase cycle named match-resolve-fire repeatedly until either the goal is attained or no more rules can be executed. The matching phase matches The LHS’s of all rules against current WME’s to find the set of satisfied rules named conflict set. The resolution phase selects one rule for execution in the next phase using criteria including recency, specificity and priority. In the fire phase, actions specified in the RHS of the selected rules are performed. If these actions lead to changes in the working memory such as addition, deletion or modification of WMEs, the match phase starts again. The whole progress is showed in Fig. 1.

### B. The Rete Algorithm

The Rete algorithm is a highly efficient algorithm for the match phase. The elementary idea of the algorithm is to compile left hand side of rules into the rete network and then perform match with creating and passing tokens along directed edges of the network. Tokens actually are tuples of working memory elements as partial instantiations.

There are basically four types of nodes forming the rete network.

1) **Constant-test node**: These appear in the top layer of the network and perform intra-condition constant tests of condition elements. A constant test means checking whether a certain attribute has a given constant value.

2) **Memory nodes**: These include α-memory nodes and β-memory nodes. The former store intermediate results from constant test nodes as state while the latter store intermediate results from join-test nodes as state.

3) **Two-input nodes**: These appear in the lower layer and test for consistency satisfaction of distinct condition elements in two conditions. In more details, it tests whether the identical variable appearing in the two conditions is bound to the same value. Join-test node is a typical kind of two-input nodes, of which left-hand input is one β-memory node and right-hand input is one α-memory node.

4) **Terminal nodes** (or p-nodes): Each one appears at the bottom of the network and means the end of a rule. And actions in the RHS of the rule are stored there and waiting for being triggered.

Sometimes one root node is introduced in the network as a start node. An example of the rete network with rules and facts is showed in Fig. 2

When the input of the rete network makes changes to the working memory, they are introduced as tokens activating root nodes. They flow in constant-test nodes and generate tokens storing in following α-memory nodes if passing constant tests. Those new tokens then flow into two-input nodes and perform consistency tests. If passing, new tokens are generated and stored in β-memory nodes. This kind of flow goes on similarly. Finally tokens could flow into terminal nodes if all relevant tests are passed. Then corresponding rules are fire-able. Therefore the output of the network is the conflict set getting new fire-able rules. In brief, tokens flow through the rete network from the top to the bottom and activate nodes along the route. The state of the algorithm is remained in memory nodes. The rete network is a kind of data flow graph so that it has the potential to be parallelized in an easy and natural way.

### C. Message-Passing Model

Message-passing model here refers to a parallel programming model. Its fundamental conception is the isolated lightweight process, which could be created thousands even millions easily and quickly on a single
machine. Processes communicate with each other through massages. Upon receiving a message, the process executes pre-defined corresponding behaviors such as sending messages to other processes. The communication is mostly asynchronous and non-blocking. This implies that the sender could immediately continue its execution after sending a message without waiting for the message to be received. But the receiver is blocking since a message arriving earlier should be received and handled earlier than ones arriving later. Each process runs concurrently together with other processes to solve a concrete problem.

III. DESIGN

A. Mapping the Rete Algorithm on the Message-Passing Model

In general, since the Rete algorithm is mainly about the rete network consisted of different kinds of nodes, it is an intuitive way to map each node onto a single process in the message-passing model and view tokens passing through rete network as messages. Then tokens' activating nodes could be viewed as messages’ sending and receiving.

At first, when compiling rules into rete network, our work is to create corresponding processes according to types of nodes. Each process has a unique identity and keeps a private state. The identify acts as an address that other processes can send messages to. The state describes necessary relevant information about the process. The only way of getting it for other processes is through communication with messages so that each process can keep independent and isolated.

Fig. 3. Join-node process structure.

Fig. 3 shows the composed structure of a join-node process state for example. Processes present only in the form of identities.

When new facts come, the rule engine starts to run. Then messages are created and transferred among processes. Processes behave according to receiving messages. Fig. 4 shows the behavior pattern of the join-node process for example. Considering efficiency, messages of activation are designed to be asynchronous and non-blocking.

By the way, we introduce the root process as a start point. We also introduce a supervising process linking to all other working processes including one root-node process. When an agent starts on a host node, its first job is to find and connect other agents and attend the computing cluster. All agents are equal and fully functional.

Firstly, users can interact with each agent to add or delete rules and facts, start or pause the rule engine and so on. Secondly, when agents get new rules, they should compile them into rete-node processes. Some of them belong to the local subnet of rete, which is a part of the whole rete network, and others do not. Processes in different subnets could be linked with each other naturally. From the perspective of linked processes, locations are transparent when communicating with each other.

B. Distributing the Rule Engine

We design a decentralized architecture in order to distribute the rule engine among various physical computing nodes in cloud environment more easily. In more detail, on each computing node, we build an independent agent to run a full image of the rule engine. Each agent should take care of a local subset of rules, which are compiled into a subnet of rete, and a subset of facts. An agent means one supervising process and many working processes including one root-node process. When an agent starts on a host node, its first job is to find and connect other agents and attend the computing cluster. All agents are equal and fully functional.

Firstly, users can interact with each agent to add or delete rules and facts, start or pause the rule engine and so on. Secondly, when agents get new rules, they should compile them into rete-node processes. Some of them belong to the local subnet of rete, which is a part of the whole rete network, and others do not. Processes in different subnets could be linked with each other naturally. From the perspective of linked processes, locations are transparent when communicating with each other. We achieve this by developing a global identity composed by the process local identity and its host agent identity, which could act as a global communication address in the function of sending messages. Finally, when one agent gets new facts, it could make them enter the rete network through local root process or send them to or be sent to other agents or both. But no matter which agent that facts enter the rete network through, the process of match could involve other agents if necessary.

```lisp
let join-node = process (State)
loop receive
  case msg: (join-left-activation, Tokens) =>
    Amem = State.amem
    jNode = State.jNode
    Tests = jNode.tests
    Wmes = p = pass-joint-test (jTests, Tokens, Amem.wmes)
  for each Child in State.children do
    send-message (Child, (left-activation, Tokens, Wmes-p))
  msg: (join-right-activation, Wmes) =>
    Bmem = State.parent
    jNode = State.jNode
    Tests = jNode.tests
    Tokens = p = pass-joint-test (Tests, Bmem.tokens, Wmes)
  for each Child in State.children do
    send-message (Child, (left-activation, Tokens, Wmes-p))
end receive
end loop
```

Fig. 4. Behavior pattern of join-node process.

Message-passing model brings an additional benefit. Since each different process could work concurrently, we could make use of the power of multi-cores in a single machine if we could load these processes on cores evenly. However, this also brings some problems. One is that making the outcome acceptable while it might be different from that of a serial system. This problem is discussed in detail by [12]. Now, we do not focus on this problem but it deserves to be solved in our future work.
It means the match process is executed in the whole rete network. And activation and other kinds of messages are transferred between linked processes directly. Finally, a right agent should be chosen to reflect results of match-resolution-fire procedures to users.

Overall, we introduce the agent to manage the distribution of the rule engine in a decentralized and easily scalable way. But the algorithm keeps running on the level of processes rather than agents because of the global identity which hides the location. And since these processes are locations transparent when communicating, there is no great difference during the execution between a single-machine edition and a distributed edition from the perspective of algorithm if efficiency is not considered. To be noticed, the agent is the basic unit in our distributed system from the perspective of software. When agents are mapped onto the cluster of computing hardware, one computing node could load more than one agent if necessary.

C. Allocation of Rules and Facts

In our rule engine, rules are compiled into rete network consisting of linked processes. Each process represents a rete node. These processes could be evenly allocated to different agents of the distributed engine. What is more, processes from one identical rule might be in different agents. More specifically, a rule has several conditions. Each condition means several constant-test nodes with one ε memory node and they could be randomly allocated to one agent if created newly. Relationships between conditions bring β-memory nodes and join nodes. A new join node and its new parent β-memory node are allocated the same as its right-hand input α-memory node. This does well in convenience and efficiency. Other nodes such as p-nodes are allocated in a similar way.

As we discussed previously, all available agents in the system could get facts. And during the run time, facts exist in the form of working memories and tokens. Considering cost of communication, we definitely cannot transfer them between processes that might be distributed in different machines directly since they might be too large. As a result, we transfer fixed-length references instead. Each reference is global unique for acting as a key to access corresponding working memory or token. Moreover, working memories and tokens need to be accessed by processes in agents which might be at different machines. This means that these data should have several copies stored in several agents to be accessed immediately and we should ensure the data consistence and isolation. Hence, we need an available distributed memory database.

In addition, memory nodes also store working memories and tokens as intermediate results. Considering convenience of management, these data are stored in the memory database instead of processes’ states in a similar way.

At last, we introduced classification of facts. Each fact is assigned a category to. Then each rule also gets corresponding categories according to facts that match it. As a result, when we compile rules and allocate processes of one rule to different agents, we could save the allocation information as <category, agent list>. And when we get new facts, we could simply check the saved records and allocate facts to related agents that are tagged with the right categories. If no relevant records are found, facts can be allocated to agents with less load or randomly. In this way, facts with the same category are allocated to the same agents. In consequence, not only can it avoid allocating facts to all available agents that then could lead to many unnecessary tests and related messages, but also it will reduce communication between different agents.

D. Optimization

Since communication is a main performance bottleneck in a message-passing system, our optimization mainly focuses on reducing messages between rete-node processes, especially those in different agents. Different agents usually are in different machines.

First of all, we introduce collection-oriented match [13]. It means that the primary objects to be matched in the rete algorithm become collections of tuples instead of individual tuples. In other words, WMEs and tokens present in the form of collections. Each condition in rules is matched with a collection of tuples and collection-oriented instantiations are generated. Fig. 5 shows an example of collection-oriented Rete transformed from the example in Fig. 2, in which alpha memories store collections of WMEs that pass constant tests and beta memories store collection-tokens that pass consistency tests. There are two constraints. First, all tuples in the component collections are guaranteed to be mutually consistent. Second, largest possible collection-tokens need to be formed. The whole process involves breaching operation and merging operation.

We can easily find that tuple-oriented tokens can be generated by the cross product of collection-tokens’ component collections. For example, a cross product of \{W1, W2\}, \{W3\}, \{W4, W5\} could generate four tuple-oriented tokens. Then suppose each of three collections of one collection-oriented tokens contained N elements and this token will consume \((N+N+N=) O(N)\) space. But
tuple-oriented Rete will create a cross product of \( (N^*N^*N^*) \) tuple combinations as its tokens and consume \( O(N^3) \) spaces. Hence collection-oriented Rete can reduce combinations of individual tuples in classical Rete dramatically. This definitely will lead to a great reduction in the number of messages carrying tokens in our rule engine and bring much execution space and time efficiency.

We improve the form of constant-test nodes, too. In classical Rete, each condition in rules has several condition elements. Each element means several constant tests. Each test refers to a constant-test node and then means a process in our rule engine. However, this is a kind of waste since the work of a constant-test node is much simpler that the work of a join-test node and processes representing join-test nodes are much busier than those representing constant-test nodes. Hence, in order to make processes work in a more efficient way, we put all constant tests of one condition element into one constant-test node. Then, we get less constant-test processes but they could be fully utilized.

IV. IMPLEMENTATION

After surveys, we determined to implement the whole system in Erlang, a programming language naturally supporting concurrent and distributed computing in a message-passing way. It could easily create millions of processes, load them on multi-cores in balance and make these processes location transparent during the communication. Moreover, Erlang runs its program in a virtual machine as Java does, thus could get platform independence that makes the deployment of our engine easily. The following describes main modules of the whole system. We call it RUNES. And it is showed in Fig. 6.

We conduct experiments to explore the performance of RUNES and investigate the impact of variation in number of rules on the speedups. Different numbers of rules is matched

![Fig. 6. RUNES architecture.](image-url)

V. EXPERIMENTS

We conduct experiments to explore the performance of RUNES and investigate the impact of variation in number of rules on the speedups. Different numbers of rules is matched

![Fig. 7. Matching time of 1000 facts on single server.](image-url)
with a constant number of facts. As for the experiment environment, we have two servers with two Quad-Core AMD Opteron Processor 2378 and 32GB memory respectively. The operation system is CentOS 6.3.

In the first experiment, we benchmark RUNES and Drools on one single server with a focus on the time of matching all 1000 facts with different numbers of rules. Drools is one of the most popular business rule engine implementation and is implemented in Java [14]. It has a high performance because of many improvements based on rete algorithm. We simplify RUNES at this time since it does not need to work in a distributed environment. The result is showed in Fig. 7. And we can see that RUNES gains its efficiency more and more apparently when the number of rules become bigger and bigger. This is mainly because on a single machine, the message-passing model can make well use the power of multi-cores while the communication latency between cores can be neglected.

In the second experiment, we evaluate the speedup of RUNES when it is deployed to two servers instead of one server. The LAN between two servers is 1000Mbps. Both the compiling time of different numbers of rules and the matching time of 1000 facts are studied. Fig. 8 and Fig. 9 shows the result.

Above figures show that the distribution among two servers makes a negative effect on the performance. When the number of rules is small enough, the version of one server even seems to have a little higher performance. The reason is various and complex including latency of communication and additional work of keeping data consistency between different agents. However, the version of two servers still shows a steady speedup. It is apparent that time of compiling rules and time of matching facts both increases more slowly as the growing of rules. And finally the matching time of the two servers version almost becomes half of that of the one server’s version. Therefore in a distributed environment, RUNES can better play its advantages when the number of facts and rules become much large.

VI. CONCLUSION AND FUTURE WORK

In this paper, we describe a distributed rule engine named RUNES based on message-passing model to address the issue of dealing with many rules and facts efficiently. Details of design and implementation are explained. Moreover, experiments shows that RUNES indeed can make well use of multi-cores and multi-machines, especially when the number of rules and facts is very large. And considering communication, RUNES would better be deployed in a compact computing cluster with such as a data center with a high LAN bandwidth.

Of course, some problems still remain. For example, results might vary when match phase is processed in a parallel or distributed way. How can we reduce the variation or ensure these results are acceptable? How can we decrease additional overhead of passing messages and keep consistence of data between different machines? These deserve to be explored in our future work.

ACKNOWLEDGMENT

We want to thank the Supercomputing Center at University of Science and Technology of China (USTC) for their platform and technical support.

REFERENCES


**Jinghan Wang** is a master student in computer science and technology at the University of Science and Technology of China. He was born in China in 1989. He received his bachelor degree at Hefei University of Technology in 2011. His research interests include cloud computing, rule-based computing, mobile computing and distributed computing.

**Rui Zhou** is a PhD student in computer science and technology at the University of Science and Technology of China. He was born in China in 1987. He received his bachelor degree at Xidian University in 2009. His research interests include cloud computing, service-oriented computing, mobile computing and context-aware.

**Jing Li** is a professor in the School of Computer Science and Technology at the University of Science and Technology of China (USTC). He was born in China in 1966. He received his bachelor degree of engineering in computer science at USTC in 1987, and Ph.D. in computer science at USTC in 1993. His research interests include distributed systems, cloud computing and mobile computing. He is the author of a great deal of research studies published at national and international journals, conference proceedings as well as book chapters.

**Guowei Wang** is PhD student in computer science and technology at the University of Science and Technology of China. He received his bachelor degree at the University of Science and Technology of China in 2012. His research interests include cloud computing, rule-based computing, mobile computing and distributed computing.