A Multiprocessor Architecture for Production System Matching

Michael A. Kelly, Rudolph E. Seviora

Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

This paper presents a new, highly parallel algorithm for OPS5 production system matching, and a multiprocessor architecture to support it. The algorithm is based on a partitioning of the Rete algorithm at the comparison level, suitable for execution on an array of several hundred processing elements. The architecture provides an execution environment which optimizes the algorithm's performance. Analysis of existing production systems and results of simulations indicate that an increase in match speed of two orders of magnitude or more over current implementations is possible.

1. Introduction

The recent popularity of expert systems in a variety of application areas demonstrates their value as problem solving tools. This technology, however, does not come without a price; executing expert systems is computationally very expensive.

Expert systems are often written using production languages such as OPS5[Forg81] and OPS83[Forg83]. Production system execution consists of repeatedly matching the conditions of IF-THEN rules to the problem solving state and firing the most appropriate rule - which in turn alters the problem solving state. A majority of the computational expense is in the matching phase of the production cycle.

Efforts have been made to define customized processors to speed matching but invariably bus bandwidths and device speeds limit their performance. Several multiprocessor designs have been put forward to deal with the amount of computation that matching requires. They too provide limited benefit, but more due to algorithmic considerations than the boundaries imposed by physics; the inherent granularity of the match operation does not allow effective use of more than a small number of processors.

The paper describes a new partitioning of the established matching algorithm for OPS5, leading to a much higher potential for parallel execution than previous versions. An architecture to support this new algorithm is presented along with some initial simulation results. The simulations show that a high degree of parallelism can be effectively exploited.

2. Requirements of Matching - The Rete Algorithm

The match phase of a production cycle consists of a many-pattern/many-object matching of rules to problem state information. The condition pattern of a rule is a set of, possibly interdependent, condition elements. The problem state is represented by a set of independent working memory elements. Matching results in a set (the conflict set) of rule instantiations which are rules whose conditions have been satisfied by a particular set of working memory elements. Conflict resolution consists of choosing the most appropriate rule instantiation for firing in the act phase of the production cycle.

A very efficient matching algorithm, the Rete Algorithm[Forg82], takes advantage of two observed characteristics of expert system execution to speed matching. One is that the set of rules for a particular application will have many similarities in their condition patterns. The other is that the problem state changes slowly, i.e. firing a rule changes only a small subset of the working memory. The efficiency of this algorithm is the reason it was chosen as the basis for the parallel matching algorithm described in the next section.

The Rete algorithm requires compiling the condition patterns of a system's rules into a network of condition and memory nodes. An example of this compilation is shown in Figure 1.

Working memory elements are fed into one end of the network and filter through to emerge from the other end as entries to the conflict set. The memory nodes store partial match information as the match proceeds. This means that when a rule is fired, only the changes it makes in the working memory need be presented to the network. A new conflict set results from applying the changes indicated by the network's output to the set which existed before the rule was fired.

Matching, as described, consists of a set of node activations. Each node activation means receiving and storing a token (representing a piece of partial match information) generated by a previous node activation.

Rule Conditions:

1) (C1 ^attr1 12 ^attr6 <= 7)
   (C2 ^attr2 > 5)
   (C4)

2) (C1 ^attr1 12 ^attr2 <X>)
   (C3 ^attr3 <X>)
   (C4)

3) (C2 ^attr2 > 5 ^attr3 <Y>)
   (C4 ^attr1 <Y> ^attr3 >= <Y>)

Rete Network:

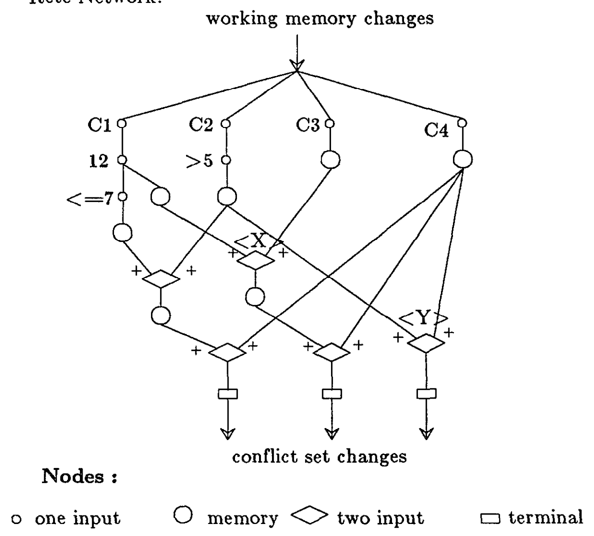


Figure 1: Example Rete Network

An attribute value is extracted from the new token and is compared against corresponding attribute values extracted from each of the complementary tokens already stored at that node. Successful comparisons, as defined by the node condition, result in the generation of new partial match information, in the form of tokens, sent to subsequent nodes.

3. Sources of Parallelism

One potential source of parallelism in the execution of production systems is in the act phase, which directly affects match execution. Normally, working memory changes caused by a rule firing are introduced to the matching mechanism separately. If these changes, typically two or three, are processed in parallel, an added degree of parallelism of two to three can be realized.

A second, more extensive source of parallelism is in the match phase via a partitioning of the Rete algorithm. One method considered is to execute node activations in a Rete network as individual processes. The degree of parallelism available is roughly estimated to be equal to the number of two-input node activations caused by a working memory change, about 35 [Gupt83], since these activations represent the bulk of the processing required. A scheme involving this approach is discussed in [Stol84] and [Gupt84]. The overall expected speedup from this proposal is small because individual node activations can represent a considerable amount of processing. Data from [Gupt83] shows that a two-input node activation involves comparing a token with an average 10 - and up to 870 - others.

The above discussion suggests that, if the advantages of the Rete algorithm are to be exploited, a partition involving entities smaller than nodes is required. A possible solution is to partition at the level of token-token comparisons. A node is split into several copies, each associated with a single token from the original memory nodes. Each node copy and its single token, from either a left side or right side memory, represents a separate process. As a group, the independent copies perform the same match function as the complete network. As individuals, they can be distributed over a large set of independent processing elements. This distribution poses some unique communication problems but offers a high degree of parallelism, in the order of 350 (the number of node activations expected times the average number of token comparisons to be done for each).

Partitioning at the comparison level is being considered (in somewhat dissimilar forms) in other research projects, [Gupt86] and [Ramn86], both of which agree that substantial speedup is possible.

The relatively high potential for comparison level partitioning is the reason it was chosen for the architecture described here. A difficulty of this approach is the definition of a partitioning algorithm that both preserves match correctness and keeps communication overhead to a manageable level. Also of concern when dealing with multiprocessors is the problem of load balancing.

The next section presents the partitioned match algorithm and a dynamic load balancing algorithm.

4. A Fine Grain Match Algorithm

Two algorithms are presented in this section. The first algorithm is a distributed Rete algorithm based on a discrimination network slightly different from the one employed by the original matching algorithm. It distributes the network condition information at the node level, and the match state information at the token level. This results in 'comparison level' distribution of match processing, suitable for a system containing a large number of processing elements.

The second algorithm operates in cooperation with the first to manage the workload at each processing element. It redistributes portions of the match process between match phases to ensure a balanced load over the entire set of processing elements.

4.1. A Distributed Rete Algorithm

This algorithm is targetted for a machine consisting of a large number of processing elements, each with its own local storage. The one input nodes in the network can be distributed over a set of processors without alteration since they use no stored data. A process representing a one input node receives working memory changes and transmits tokens representing the ones that have passed the node's constant tests. In order to partition the rest of the network, it is necessary to provide separate memories for each node. An example network where this has been done is shown in Figure 2. For this network, a two input node and its

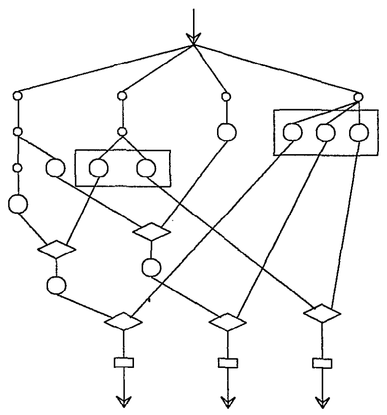


Figure 2: Network from Figure 1 with Split Memory Nodes

two associated memories can be processed independently. To isolate individual tokens for comparison, the two input nodes are separated into a number of node copies, each associated with a single token from one of the original memory nodes. A node copy contains comparison information for a left- or right-handed test, destination information for new tokens it forms, and some status information. Figure 3 shows how a single node with its left and right memories is split up. Duplication and splitting of memory nodes increases the storage required by a factor of approximately three.

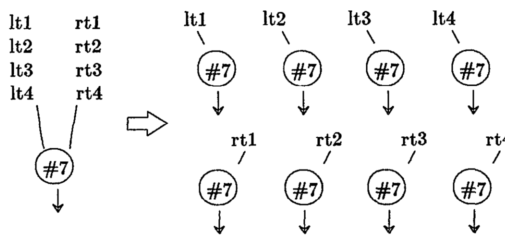


Figure 3: Example of Node Splitting

Processing of the node/token pairs resulting from two input nodes with two positive inputs is done as one of two operations : 1) If a token arrives at a node on the opposite side to its stored one, the node test is performed and may result in the generation of a new token. 2) If a token arrives at a node on the same side as the stored token, the node's response will depend on whether the token represents the addition (positive) or deletion (negative) of a piece of partial match information. A negative of the stored token will cause the node/token pair to be deleted. A new positive token will cause the generation of a new node copy to which the new token is attached. One of the existing node copies, the 'generative' copy, is made responsible for this so that only one new copy is generated. (The generative node copy is not deleted if a negative of its token arrives, so that node information is not lost.)

Processing of Not nodes, which have one positive input and one negative input, is a little more difficult because of the asymmetry involved, and the fact that node responses are based on information about the entire contents of the token memory on the negated side. This dependence on information that does not exist at any one node copy is overcome by making a stipulation about the communication system over which tokens will be transported. The stipulation is that tokens will always reach their destinations in the order they were generated. There are several ways this could be performed; one is to force all tokens over a single, linearizing channel. The architecture chosen includes this characteristic, as is discussed in the next section of this paper.

Node operation is as follows: node copies storing tokens from the positive side of the original two input node operate similarly to the node copies from nodes where both inputs are positive. The exception being that new tokens are generated with the opposite polarity to the ones received i.e. the addition of a new token on the negative side causes the deletion of a token previously generated by the positive side, and vice versa. To deal with the negated input, node copies storing tokens from the negative side of the original node are placed in the path of tokens generated by their positive counterparts. If a positive-side node copy generates a new positive token, the negative-side copy will receive it and has the option of generating a cancelling token if it contains information that negates it. The negating token will always arrive at destinations later than the positive token since it was generated later. This ensures the correctness of the cancelling operation.

Terminal nodes can be eliminated since their function is effectively one of addressing rule instantiations to the conflict set, which can be performed by the two input nodes at the bottom of the Rete network.

This algorithm has the inherent property that it is relatively unaffected by the ratio of program size to data size. That is, activity over a large number of Rete network nodes involving few tokens each is similar to activity concentrated at a few nodes associated with many tokens each. Another advantage is that node copies involve only one token each and so they represent similar amounts of processing. Nodes are also self managing in that the sizes of their images (in terms of the number of copies which exist) change to reflect the total amount of processing they require.

4.2. A Dynamic Load Balancing Algorithm

An architecture involving a large number of processing elements implies that each is small; local memory areas can become easily overloaded. The small size of process entities, node/token pairs, allows them to be moved between processing elements to alleviate this problem. The following algorithm describes a method for spreading node copies over a large set of processors while insuring that a correct match takes place:

When a generative node copy receives a positive token on the same side as its own stored token, it generates a new, marked, copy of itself to hold the new token. The marked copy becomes immediately active. At the end of the match phase, the marked copy is passed to another processor. The first new copy created by the generative copy becomes the new generative copy at the beginning of the next production cycle, while the original copy ceases to be generative. This action causes active nodes in the Rete network to continually diffuse away from busy areas of the processor array. If a processing element becomes filled with too many marked copies before the match phase is complete, it can force a premature end-of-cycle. This process involves halting the match until all marked node copies are passed between processors. This alleviates the memory shortage at the over-full processor. The Match phase continues after this pause as if a new production cycle was starting. The frequency of these forced end-of-cycle pauses is related to the loading of processing elements, resulting in a graceful decline in performance of the match phase as loading increases.

5. Architecture

Two issues must be addressed in defining an appropriate architecture for the algorithms of the previous section.

One is the organization of processing elements to be used. This is based on the communication required between processors for the algorithms described. The second consideration is the internal structure of a processing element. These are influenced by the requirements of both the match operation and the communication systems defining the organization of processors.

5.1. Organization of Processing Elements

After considering several alternatives, a processor organization consisting of a single uniform layer of processing elements was chosen. There are three main reasons for this choice. First, a single array of processing elements allows effective response to node activation requests which occur simultaneously at various depths in the network. This means a fast response time to widely used match information. Secondly, the amount of data to be stored at any one node in the Rete network varies from system to system, and dynamically as a system runs. Allowing all node information to be spread over all processing elements avoids the memory balancing problems that a segmented system might incur. And thirdly, nodes involving negated rule conditions imply examining the node's entire set of tokens before a response can be made. Using a single array of processing elements, and the communication system described below, makes a solution to this problem compatible with non-negated node activity.

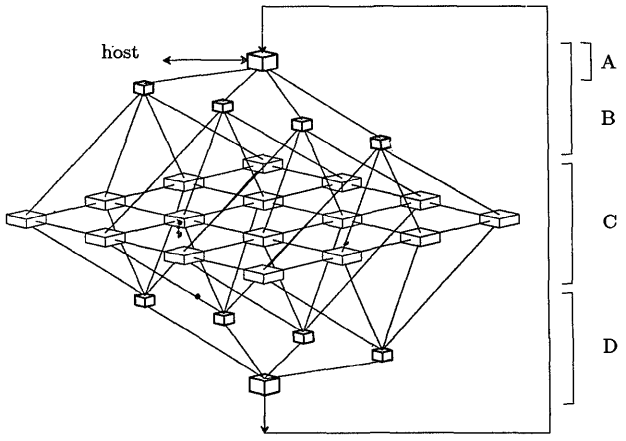
An effective way to send match information to all processing elements quickly is with a broadcast system in the form of a word-width distribution tree. Such a system provides a high bandwidth path but avoids high fanout at any one node. It also allows some asynchronism of data flow between various parts of the tree. This asynchronism helps accommodate varying processing speeds at different processing elements. A similar structure with data flowing from leaves to root can be used to collect responses (new tokens) from the processing elements. Individual processing elements will respond infrequently but ,as a whole, the array will produce responses roughly equal in volume to the original input. This means that the collection tree requires a root with bandwidth equal to that of the broadcast tree but limb bandwidths can reduce toward the processing elements, terminating in serial connections. Responses may be required by other processing elements and so the roots of the two trees are joined, creating a data path loop through the processing elements. Responses that are meant for the host, i.e. conflict set changes, are separated out and redirected at this joining point.

The broadcast/collection system described provides a path for match information flow as well as a point of contact for the host which executes the Conflict Resolution and Act phases of the production cycle. The load balancing algorithm also requires communication between processing elements, but the broadcast system is inappropriate for this. Unlike match information which must be sent to all processing elements, a packet carrying a new node/token pair is only required at one - the one that will store it. The broadcast system is not suited to this type of transfer since it treats all destinations equally. Also, node/token packets have no particular destination; short trips over a local, low bandwidth communication network will suffice. The design considered here uses a square network, connecting a processing element to each of its four closest neighbours (this degree of interconnect may be reconsidered based on simulation results). Figure 4 shows an example of the organization of processing elements chosen for 16 elements. (The local communication mesh is completed with links between processing elements on opposite edges, forming a uniform toroid.)

The broadcast system uses a word-width path while the local links are serial. The number of local links, and the expected number of links traversed by a node/token packet before seating, give the local communication system an effective bandwidth roughly equal to the broadcast system bandwidth.

5.2. Processing Elements

Each processing element must perform a number of duties during a production cycle. They include:



A Filter/Interface Node

B Broadcast Network

C PE Array and Local Network

D Collection Network

Figure 4: Processor Organization

1) receiving and filtering match information from the broadcast system

2) performing match operations on received token and stored tokens

3) transmitting new match information to the collection half of the broadcast system

4) receiving and storing (or passing on) node/token packets from the local communication system, and

5) generating new node/token packets when necessary

This set of responsibilities varies widely in processing requirement. It also contains few constraints on simultaneous execution. For these reasons, the structure chosen for the processing elements is a central processor (and ROM) which performs the various algorithm operations, and a set of state machines to handle data transfers over the I/O ports. Figure 5 shows a block diagram of a processing element. The state machines can perform data transfers independently but

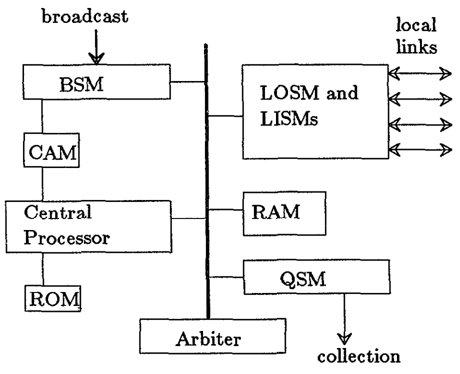


Figure 5: Processing Element Contents

are coordinated by the central processor. The central processor also performs memory management of a common RAM area using a two level interrupt system. A bus arbiter controls access to the common RAM bus on a priority basis.

The BSM is a state machine connected to the broadcast port. Its responsibility is to store relevant incoming tokens into the local RAM. A CAM, accessible to both the central processor and the BSM, contains the IDs of nodes which have copies in local RAM. The CAM supplies present/not-present responses to the BSM. The central processor has full access to the CAM which also contains, as data, pointers to the node copies in RAM. The QSM, when triggered by the central processor, transmits new match information onto the collection half of the broadcast system. The LOSM, when triggered by the central processor, transmits new node/token packets onto one of the local communication links. The LISMs either store incoming node/token packets, or pass them through the processing element, depending on the amount of free space available.

It is estimated that a processing element, less memory blocks, is about the same complexity as a simple 8-bit microprocessor.

6. Simulations

A first pass design of the architecture has been completed and a register transfer level simulator has been written. The purpose of simulations is two-fold :

1) The detail of simulation allows the verification of the algorithms, using a small problem. They also provide some initial performance values, for the small problem considered.

2) The simulations also provide accurate timing information for more elaborate simulations involving larger problems, which will not be done at the register transfer level.

6.1. The Simulator

The form of the simulator allows the simulation of 1, 2, 4, 8, and 16 processing element arrays. (The 4x4 array was the largest simulated due to the computational expense of such detailed simulations.)

The fairly mature CMOS technology available at the University is assumed: Processor clock speed is 10 MHz. ROM, RAM and CAM access times are 200, 250, and 500 nS, respectively.

In the simulations, the conflict resolution and act phases do not take place. Changes to the working memory are fed into the array as if a rule firing had taken place, and response time is observed.

The simulation subject is a program loop performed by a single production. The production contains four condition elements and two actions; the condition elements contain two constant values along with the class type, and one or two variables each (all typical values).

The characteristics of execution are :

1) A rule firing causes 2 working memory changes.

2) Each working memory change causes 3 (all successful) one-input node activations.

3) Each working memory change causes 6 two input node activations. And

4) The match phase generates two changes to the conflict set.

Simulations were performed using a large enough RAM area to avoid memory shortages.

6.2. Simulation Results

Simulations for arrays using 1, 2, 4, 8, and 16 processors were performed until the cycle time stabilized - a steady state was reached. Table 1 shows the match times recorded for these simulations. The match phase execution time for the problem simulated running on a VAX 11/780 using an OPS83 compiler is 3.26 mS. (The cycle time on the VAX was 3.62 mS. The conflict resolution and act phases of the cycle consisted of a simple 'for' loop containing a 'fire 1' statement. It was assumed that this would take a maximum of 10% of the cycle time. This corresponds to a match time of approximately 3.26 mS.)

# of Processors	Execution Time (mS)
1	4.238
2	2.524
4	1.627
8	1.144
16	1.000

Table 1: Match Execution Times

7. Discussion

The results of these simulations show that the level of parallelism exploited by the distributed Rete algorithm is approximately four. The theoretical limit is 6 to 8 (from the number of tokens at each level), ignoring the effects of the one input nodes. Node level parallelism for the simulated rule is 1 to 2 (from the widths of the two layers of two input nodes in the network), again ignoring the effect of the one input nodes. The advantage of the distributed Rete algorithm increases with program size. As discussed, the level of parallelism available in an average production system is in the order of 300. Another consideration is variation in the number of tokens stored at the nodes in a network. This has little impact on the distributed Rete algorithm but has a strong effect on the execution of an algorithm based on node level parallelism.

Another matter to be considered in analyzing the results of the simulations is the set of technology parameters used. They are based on a particular process available at the University, and trail the leading edge by a factor of about 3. The simulation serves to compare the effects of varying the processor array size; the comparison of match times shown versus the VAX processing time could be improved by using more advanced technology parameters.

Also available from the simulator is an all but complete set of software for a processing element. This has allowed a second analysis of instruction set requirements. It is apparent that some improvement is possible in this area as well.

With some additions to the model, further simulation and analysis will determine the effects of such things as larger overall system size, and the possibility of multiple rule firing systems. These will impact on the size of processing array required, and so the requirements of the communication systems - particularly the broadcast/collection system. Also to be determined is the performance penalty associated with production systems that approach the available memory size.

Investigations are also under way to extend the matching capabilities of the architecture to those of OPS83 which includes simple function calls. One approach that may be taken is the use of a non-homogeneous array, using the processing elements described along with arithmetic processing elements for the execution of mathematical functions. This will add to the flexibility of the programming environment at a minimum of cost in match execution time.

8. References

[Forg81] Forgy, C.L. "OPS5 Users Manual", Technical Report CMU-CS-81-135, Carnegie-Mellon University 1981.

[Forg82] Forgy, C.L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence 19, pp. 17-37, September 1982.

[Forg85] Forgy, C.L. "OPS83 User's Manual and Report", Production Systems Technologies Incorporated, 1985.

[Gupt83] Gupta, A., Forgy, C.L. "Measurements on Production Systems", Technical Report CMU-CS-83-167, Carnegie-Mellon University, 1983.

[Gupt84] Gupta, A., "Implementing OPS5 Production Systems on DADO", Technical Report CMU-CS-84-115, Carnegie-Mellon University 1984.

[Gupt86] Gupta, A., Forgy, C.L., Newell, A., Wedig, R., "Parallel Algorithms and Architectures for Rule-Based Systems", Proceedings 13th International Symposium on Computer Architecture, pp. 28-37, 1986.

[Ramn86] Ramnarayan, R., Zimmermann, G., Krolikoski, S., "PESA-1: A Parallel Architecture For OPS5 Production System", Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, pp. 201-205, 1986.

[Stol84] Stolfo, S. J., "Five Parallel Algorithms for Production System Execution on the DADO Machine", Proceedings National Conference on Artificial Intelligence, pp. 300-307, 1984.