Suitability of Message Passing Computers for Implementing Production Systems

Anoop Gupta
Dept. of Computer Science
Stanford University
Stanford, CA 94305

Milind Tambe
Dept. of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Two important parallel architecture types are the shared-memory architectures and the message-passing architectures. In the past researchers working on the parallel implementations of production systems have focussed either on shared-memory multiprocessors or on special purpose architectures. Message-passing computers have not been studied. The main reasons have been the large message-passing latency (as large as a few milliseconds) and high message reception overheads (several hundred microseconds) exhibited by the first generation message-passing computers. These overheads are too large for the parallel implementation of production systems, where it is necessary to exploit parallelism at a very fine granularity to obtain significant speed-up (subtasks execute about 100 machine instructions). However, recent advances in interconnection network technology and processing node design have cut the network latency and message reception overhead by 2-3 orders of magnitude, making these computers much more interesting. In this paper we present techniques for mapping production systems onto message-passing computers. We show that using a concurrent distributed hash table data structure, it is possible to exploit parallelism at a very fine granularity and to obtain significant speed-ups from parallelism[1].

1. Introduction

Production systems (or rule-based systems) occupy a prominent place in the field of AI. They have been used extensively in the attempts to understand the nature of intelligence as well as to develop expert systems spanning a wide variety of applications. Production system programs, however, are computation intensive and run slowly. This slows down research and limits the utility of these systems. In this paper, we examine the suitability of message-passing computers (MPCs) for exploiting parallelism to speed-up the execution of production systems.

To obtain significant speed-up from parallelism in production systems it is necessary to exploit parallelism at a very fine granularity. For example, the average number of instructions executed by subtasks in the parallel implementation suggested in [10] is only about 100. In the past, researchers have explored the use of special-purpose architectures and shared memory multiprocessors to capture this fine-grained parallelism [10, 16, 17, 18, 11, 21]. However, the performance of MPCs for production systems has not been analyzed. Considering MPCs is important, because MPCs represent a major architectural and programming model in current use. Previously, the communication delays in the MPCs made them impossible to be used for the purpose of exploiting fine grained parallelism. However, recent developments in the implementations of MPCs [3], have reduced the communication delays and the message processing overheads by 2-3 orders of magnitude. The presence of these new generation MPCs such as the AMETEK-2010 [19] makes it interesting to consider MPCs for implementing production systems.

This paper is organized as follows. Section 2 describes the OPS5 production system and the Rete matching algorithm used in implementing it. Section 3 describes recent developments in the MPCs and presents the assumptions about their execution times which we will use in our analysis. Section 4 presents our scheme for implementing OPS5 on the MPCs. We then evaluate its performance and compare it with other parallel implementations of production systems.

2. Background

2.1. OPS5

An OPS5 [2] production system is composed of a set of *if-then* rules called productions that make up the *production memory*, and a database of temporary assertions, called the *working memory*. The individual assertions are called working memory elements (WMEs), which are lists of attribute-value pairs. Each production consists of a conjunction of condition elements (CEs) corresponding to the *if* part of the rule (the left-hand side or LHS), and a set of actions corresponding to the *then* part of the rule (the right-hand side or RHS).

The CEs in a production consist of attribute-value tests, where some attributes may contain variables as values. The attribute-value tests of a CE must all be matched by a WME for the CE to match; the variables in the condition element may match any value, but if the variable occurs in more than one CE of a production, then all occurrences of the variable must match identical values. When all the CEs of a production are matched, the production is satisfied, and an instantiation of the production (a list of WMEs that matched it), is created and entered into the *conflict set*. The production system uses a selection procedure called *conflict-resolution* to choose a production from the conflict set, which is then *fired*. When a production fires, the RHS actions associated with that production are executed. The RHS actions can add, remove or modify WMEs, or perform I/O.

The production system is executed by an interpreter that repeatedly cycles through three steps: *match*, *conflict-resolution*, and *act*. The matching procedure determines the set of satisfied

[1]This research was sponsored by Encore Computer Corporation, Digital Equipment Corporation and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Air Force Avionics Laboratory. Anoop Gupta is supported by DARPA contract MDA903-83-C-0335 and an award from the Digital Equipment Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Encore Computer Corporation, Digital Equipment Corporation and the Defense Advanced Research Projects Agency or the US Government.

productions, the conflict-resolution procedure selects the highest priority instantiation, and the act procedure executes its RHS.

2.2. Rete

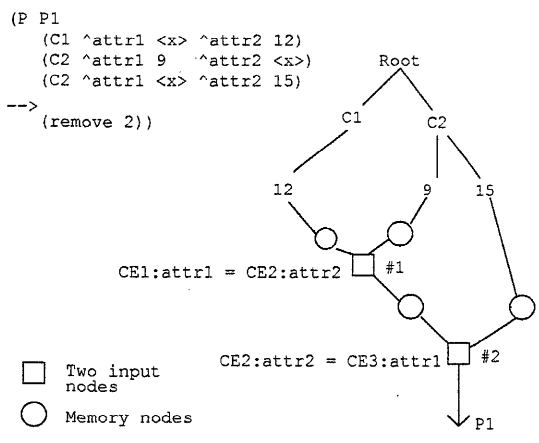
Rete [7] is a highly efficient match algorithm that is also suitable for parallel implementations [9]. Rete gains its efficiency from two optimizations. First, it exploits the fact that only a small fraction of working memory changes each cycle by storing results of match from previous cycles and using them in subsequent cycles. Second, it exploits the commonality between CEs of productions, to reduce the number of tests performed.

Rete uses a special kind of a data-flow network compiled from the LHSs of productions to perform match. The network is generated at compile time, before the production system is actually run. The entities that flow in this network are called *tokens*, which consist of a *tag*, a *list of WME time-tags*, and a *list of variable bindings*. The tag is either a + or a – indicating the addition or deletion of a WME. The list of WME time-tags identifies the data elements matching a subsequence of CEs in the production. The list of variable bindings associated with a token corresponds to the bindings created for variables in those CEs that the system is trying to match or has already matched.

There are primarily three types of *nodes* in the network which use the tokens described above to perform match:

1. *Constant-test nodes*: These are used to test the constant-value attributes of the CEs and always appear in the top part of the network. They take less than 10% of the time spent in Match.
2. *Memory nodes*: These store the results of the match phase from previous cycles as state. This state consists of a *list* of the tokens that match a part of the LHS of the associated production. This way only changes made to the working memory by the most recent production firing have to be processed every cycle.
3. *Two-input nodes*: These test for joint satisfaction of CEs in the LHS of a production. Both inputs of a two-input node come from memory nodes. When a token arrives from the *left memory*, i.e., on the left input of a two-input node, it is compared to each token stored in the *right memory*. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action occurs when a token arrives from the right memory. We refer to such an action as a *node-activation.*

Figure 2-1 shows the Rete net for a production named P1.



```
(P P1
   (C1 ^attr1 <x> ^attr2 12)
   (C2 ^attr1 9   ^attr2 <x>)
   (C2 ^attr1 <x> ^attr2 15)
-->
   (remove 2))
```

Figure 2-1: The Rete network.

3. Message-Passing Computers and Assumptions

MPCs are MIMD computers based on the programming model of concurrent processes communicating by *message passing*. There is no global *shared* memory and hence communication between the concurrent processes is explicit as in Hoare's CSP [12], though not necessarily synchronous. The early MPCs such as the Cosmic Cube [20] had a high network latency of about ~2 millisecond (ms) and a high overhead of message handling of about ~300 microseconds (µs). As a result, it was impossible to exploit parallelism at the fine granularity of 50-100 µs as is necessary in production systems.

Recent developments in MPCs such as *worm-hole routing* [4] have reduced the network latencies to 2-3 µs and the use of special processors such as the MDP (Message Driven Processor) [5] can potentially reduce the message reception overhead by an order of magnitude. With today's VLSI technology, it is possible to construct MPCs with thousands of processing nodes and hundreds of megabytes of memory [3]. Thus very fine grain parallelism can now be exploited easily with the MPCs.

This raises the issue of whether production systems can be implemented efficiently on the MPCs to give good speedups, which we analyze in detail in this paper. For the purpose of this analysis, we assume a *32-ary 2-cube architecture* (1024 nodes), with a *4 MIPS* processor at each node similar to the MDP. The various times that required for our analysis are as follows. The latency of wormhole routing is given by

$$T_{wh} = T_C(D + L/W)$$

Where —

T_C	Channel Delay, assumed to be 50 nanoseconds (ns), as in [3].
W	Channel Width, assumed to be 16 bits.
L	Length of the message in bits.
D	Distance or number of hops traveled by the message. If two processing nodes are selected at random in a k-ary n-cube, then number of hops is $n*(k^2 - 1)/3k \approx 22$ for our 32-ary 2-cube.

We assume that the MDP is driven by a 100 ns clock and that the time to execute a send (broadcast) command is

$$T_s = (5 + N*Q) \text{ clock cycles.}$$

where a message of Q words is to be sent to N sites [5]. The overhead of receiving messages is assumed insignificant [5]. Thus there are two delays associated with a message: T_s in transmission, T_{wh} in its communication.

4. Mapping Rete on the MPC

In this section we describe our mapping of Rete on the MPCs. We draw heavily from our previous work with the *PSM* implementations of production systems on shared-memory multiprocessors [9, 10, 21].

One possible scheme for implementing OPS5 on the MPCs arises from viewing Rete in an *object-oriented* manner, where the nodes of Rete are objects and tokens are messages. This scheme maps a single object (node) of Rete onto a single processor of the MPC. However, there are two serious problems: (1) The mapping requires one processor per node of the Rete net, and the processor utilization of such a scheme is expected to be very low; (2) Often, the processing of a WME change results in multiple activations of the same Rete node, which in the above mapping would be processed sequentially on the same PE, thus causing that PE to be a bottleneck.

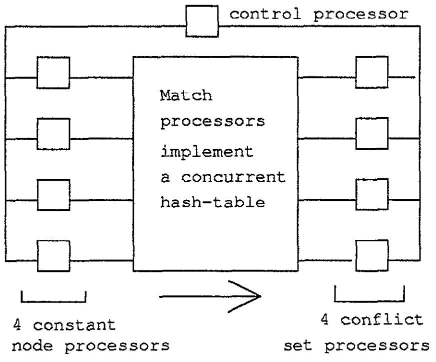

control processor
Match processors implement a concurrent hash-table
4 constant node processors
4 conflict set processors

Figure 4-1: A high level view of the Mapping on the MPCs.

To overcome the limitations of above mapping, we propose an alternative mapping, a high-level picture of which is shown in Figure 4-1. At the heart of this mapping is a *concurrent distributed hash-table* [6] data structure that enables fine-grain exploitation of concurrency. The details are described later in this section. As shown in the figure 4-1, the parallel mapping consists of 1 *control processor*, 4 *constant-node processors*, 4 *conflict set processors*; and the rest are *match processors*. The constant-test nodes of the Rete net are divided into 4 parts and assigned to the constant-node processors. The match processors perform the function of the rest of the Rete net. The conflict-set processors perform conflict-resolution on the instantiations sent to them. Subsequently, they send the best instantiation to the control processor. The control processor is responsible for performing conflict-resolution among the best instantiations, evaluating the RHS and performing other functions of the interpreter.

As mentioned in Section 2.2, most of the time in match is spent processing two-input node activations. Hashing the contents of the associated memory nodes, instead of storing them in linear lists, reduces the number of comparisons performed during a node-activation and thus improves the performance of Rete. One hash table is used for all left memory nodes in the network and the other for all right memory nodes. The hash function that is applied to the tokens takes into account (1) the variable bindings tested for equality at the two-input node, and (2) the unique node-identifier of the destination two-input node. This permits quick detection of the tokens that are likely to pass the equal variable tests.

In our mapping, to allow the parallel processing of (1) tokens destined for the *same* two-input node and (2) tokens destined for different two-input nodes, the hash tables buckets storing the tokens are distributed among the PEs of the processor array. In particular, a small number of corresponding buckets from the left and right hash tables are assigned to each *processor pair* in the array -- the left-buckets to the left processor and the right buckets to the right processor. (Note that when processing a node activation, the left and right buckets at only one index need to be accessed.) This mapping is pictorially depicted in Figure 4-2. There is one restriction on the communication with the processor-pair — it can only be done through the *left-processor*. Allowing communication with both left and right processors can result in creation of duplicate tokens leading to incorrect behavior, and it does not gain as much in concurrency.

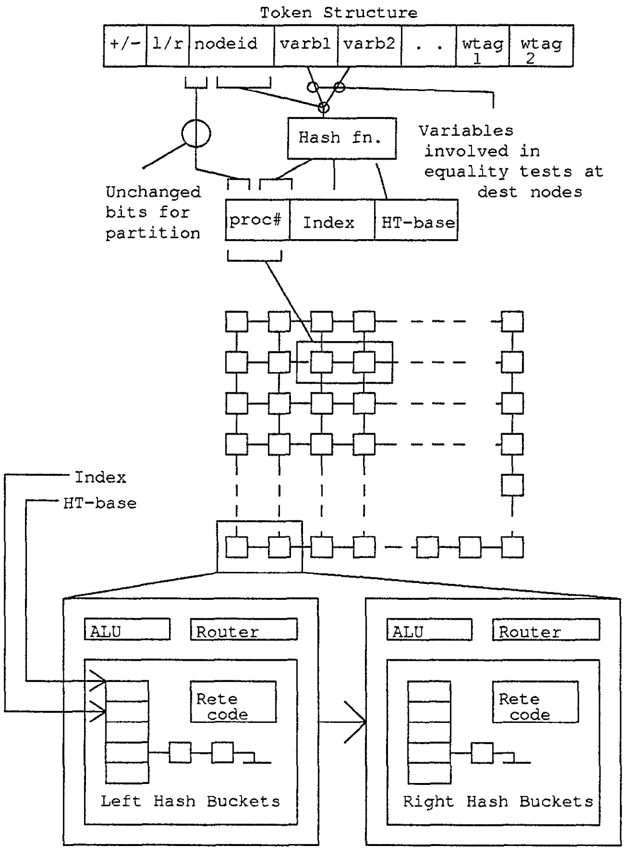


Figure 4-2: The detailed mapping.

A processor-pair together performs the activity of a single *node activation*. Consider the case when a token corresponding to the left-activation of a two-input node arrives at a processor-pair. The left processor immediately transmits the token to the right processor. The left processor then copies the token into a data-structure and adds it to the appropriate hash-table bucket. Meanwhile, the right processor compares the token with contents of the appropriate right bucket to generate tokens required for successor node activations. The right-processor then calculates the hash value for the newly created tokens, and sends each token to the processor pair which owns the buckets that it hashes to. The activities performed by the individual processors of the processor pair are called *micro-tasks*, and all the micro-tasks on the various processor pairs are performed in parallel.

The performance of this scheme depends on the discriminability of hashing. Two observations can be made in this respect:

1. Hashing is based on equality tests in CEs and 90% of the tests at two input nodes are equality tests [9].

2. The locks on the hash tables in the PSM implementations have not been seen to be bottlenecks [10, 21].

Thus hashing is not expected to be a problem in general. However, in certain production systems, a large number of two-input nodes do not have any tests. For such nodes, various schemes as proposed in [1], can be used to introduce discriminability into the tokens generated. Furthermore, when the compiler does come across nodes which cannot be hashed, it can assign a larger number of processors for that pair of buckets, (since all the tokens would end up in a single pair of buckets) thus breaking up the processing.

The code for the Rete net is to be encoded in the OPS83 [8] software technology. With this encoding, large OPS5 programs (with $\approx$ 1000 productions) require about 1-2 Mbytes of memory — a problem, since each MPC processor has only 10-20 kbytes of local memory. We therefore use two strategies to save space:

1. Partition the nodes of Rete such that each processor evaluates nodes from only one partition. This partitioning is easily achieved if the hash function preserves some bits from the node-id. To avoid contention, nodes belong to a single production are put into different partitions.
2. One cause of the large memory requirement is the in-line expansion of procedures. We can instead encode the two-input nodes into structures of 14 bytes, indexed by the node-id. A small performance penalty of loading the required information into registers is then paid in the beginning of the computation.

The system's overall operation is as follows:

1. The control processor evaluates a WME change and transmits it to the constant node processors.
2. The constant node processors match the WME with the constants in the CEs. The result of this match is tokens that have bindings for the variables in matched CEs. These tokens represent individual node activations and are sent to appropriate processor pairs.
3. The following steps are then repeated by the processor-pairs until completion of match:
    - Split the node-activation into micro-tasks and perform them in parallel.
    - Count the number of successor tokens generated due to this token; if no successors are generated, then send an acknowledgement (ack) message to this processor pair's activator.
    - Accept ack messages from the successors. If accounted for all successors of a token, send an ack message to the activator.

Detecting termination in a distributed system is a complex problem in itself [15]. The ack messages provide an easy and reasonably efficient method of informing the conflict-set processors about the completion of the match. Thus after the processing of the last activation in the current match cycle, a single stream of ack messages flows back, finally to the control processor, which then informs the conflict set processors that the match is completed.

5. Performance Analysis

We now evaluate the MPC implementation using the measurements on the Rete net from [9].[2] The point of the analysis is to establish that the MPCs will provide good speedups compared to other previously proposed parallel implementations, rather than to estimate the exact performance that will be obtained on a real machine.

One of the important numbers for this analysis is the time spent in the processing of one node activation. Using that, we can estimate the time for a micro-task. A node activation is identical to a *task* on the PSM, which takes 200 μs on a 1 MIPS processor [10]. Measurements of the number of instructions executed indicate that about 50% of that time is spent in updating the hash bucket and 50% in performing tests with tokens in opposite memory. We therefore assume that on our 4 MIPS processor, performing a micro-task will take about 25 μs, which is 200 μs * 1/4 (due to processor speed) * 0.5 (due to partitioning of the node-activation into micro-tasks).

Since the processor-pairs communicate via tokens, we also need to calculate the overhead of a token message. The length of a token-message is dependent on the number of variable bindings and the number of WME timetags carried by the token. There are on average four variable bindings per production [9]. The number of WME timetags is dependent on the number of CEs in a production. Assuming the number of CEs to be (M = 5) for the moment, we use the token-structure in Figure 4-2 to estimate 42 bytes of information per token. The overhead of sending the token message will therefore be equal to $T_s = (5 + Q * N)$ clock cycles, with Q = 42/4 words and N = 1 processor (see section 3). Substituting, we get $T_s \approx 1.6$ μs. The communication delay T_{wh} is given by $T_C(D + L/W)$. This communication will be between a random pairs of processors. Therefore, D = 22. We have assumed T_C to be 50ns and W to be 16. Our L is 42 * 8 = 336 bits. Substituting, we get $T_{wh} \approx 2.2$ μs. The total delay will be therefore 1.6 + 2.2 = 3.8 μs per token message between processor-pairs.

We can now estimate the cost of one match cycle. The steps below correspond to the algorithm in the previous section.

Step 1: The WME changes are transmitted to the 4 constant-node processors. The cost of addition of a WME is as follows: The average WME consists of 24 attribute value pairs, which can be encoded in 24 bytes for attributes + 24 words for the values = 30 words. Broadcasting this WME takes T_s = (5 + 30 words * 4 processors) clock cycles i.e., 12.5 μs.

For the communication delay, T_{wh}, D = 1 since the constant node processors are one hop away from the control processor. The value of L is 30 words * 32 bits/word = 960 bits; W = 16 and the value of T_C is fixed at 50. Substituting, we get $T_{wh} \approx 3.1$ μs. Thus the total time spent in communication during WME-addition is 15.6 μs.

For deleting a WME, only the timetag of the WME to be deleted is passed on to the constant-node processors. Calculating T_s and T_{wh} in a similar fashion, we get the total time spent in delete to be 1.1 μs. There is an average of 2.5 WME changes per cycle. Assuming equal proportions of adds and deletes, the cost of the first step is 1.25(1.1 + 15.6) $\approx$ 21 μs.

[2]We do not analyze the conflict-resolution and action parts of the match since these take less than 10% of the time in a serial implementation. Since we have divided up the conflict set and pipelined the action part with the match, these should take even less time than that. In case they do become bottlenecks, various schemes discussed in [9] can be used to reduce their overheads.

Step 2: The constant tests are now evaluated. Assuming that the constant tests are implemented via hashing, there are 20 constant-node activations per WME change [9]. On average, each partition will have 5 activations per WME change. Thus about (5 * 2 / 4 MIPS) ≈ 2.5 μs are spent in matching the constant nodes. A token structure is then generated and bindings are created for the variable(s) of the CEs which passed the tests. Measurements [9] show that there will be about 5-7 such tokens generated per WME change, which we assume to take 20 μs. This whole operation of processing a WME-change by a constant-node processor is therefore estimated to take about 22.5 μs. For the 2.5 WME-changes, (22.5 * 2.5) ≈ 56 μs will be spent in processing the constant nodes and generating the initial tokens in a cycle. The generation of these tokens is pipelined with sending the tokens to the match processors.

Step 3: The processor-pairs perform the rest of the match. The node-activation typically go to different processor-pairs, and are processed in parallel. Therefore, the total time to finish the match is determined by the longest chain of dependent node-activations, since the micro-tasks in the chain have to be processed sequentially. On an average, the chain will be generated after 50% of the initial tokens in a cycle have been generated. A constant-node processor takes 56 μs to generate all the initial tokens; therefore, we assume that the initial token generating the long chain will be created after 28 μs. Including the constant-node processors, let the longest chain be of length M = 5.

When a token arrives at the left processor, it is immediately transmitted to the right processor. For this transmission, T_s is still 1.6 μs. But, T_{wh} = 50(1 channel + 42 * 8/16) = 1.1 μs. Thus, after a token arrives at the left processor, it will take 1.6 + 1.1 = 2.7 μs to reach the right processor. The right processor will take 25 μs to finish the micro-task. It will then take 3.8 μs for the successor token to reach its destination. Thus, the time to complete a micro-task is 25 + 2.7 + 3.8 = 31.5 μs. A chain of length 5 will therefore take 31.5 * 4 + 28 μs (due to the constant nodes) = 154 μs. (Similar analysis could be done if the successors are generated by the left processor).

The ack messages are propagated back through the node activation chain, after the last activation is processed. It is 1 word of information and so we estimate T_{wh} = 1.2 μs and T_s = 0.6 μs. Assuming that the ack is processed in 1 μs, the time spent in the chain of ack messages is (M = 5) * (1 + 1.2 + 0.6) = 14.0 μs. Adding all the numbers together, we get the time for MPC to match to be approximately 154 + 14 + 21 = 189 μs.

A production system generates 200 micro-tasks on an average/cycle, and therefore a uniprocessor will take 200 * 25 = 5000 μs per cycle. Using this we get about *26 fold* speedup for the above system with the longest chain of M = 5. This is ~60% of the maximum parallelism exploitable on an ideal multi-processor at this granularity. Our calculations show that the speedups is ~14 fold if M = 10 and ~9 fold if M = 15. Again, this is ~60% of the maximum available parallelism. This is comparable with the estimate of 60% exploitable parallelism in shared memory multiprocessors at the node-activation level [9]. This coarser grain node-activation level parallelism can be exploited on the MPCs by allocating both the left and right buckets to one processor. Our calculations show that the micro-task based scheme is capable of exploiting 1.5 time more speedup than a scheme to exploit the node-activation level parallelism.

6. Discussion

Comparing the MPC implementation to a shared memory multi-processor implementation, we see that the principle advantage of the MPC implementation is the *absence* of a centralized task-scheduler, which can be a potential bottleneck. As shown in [9], in shared-memory implementations, a slow scheduler forces saturation speedup with relatively small number of processors, irrespective of the inherent parallelism in the system. However, the MPC implementation suffers from a static partitioning of the hash tables. It is possible that distinct tokens, which could potentially be processed in parallel, are processed sequentially because they hash to the same processor pair. Such a possibility does not arise in the shared-memory implementation, since the size of the hash table is independent of the number of processors.

Another tradeoff to be considered is between processor utilization and the number of processors. With a higher number of processors, the processor utilization will be low, but the message contention in the network will be reduced. As the number of processors is reduced, processor utilization will be improved; but again, this will also increase the hash table contention. Thus there are some interesting tradeoffs involved in moving towards the MPCs.

A mapping similar to one proposed in this paper has been used to implement production systems on the simulator for *Nectar*, a network computer architecture with low message passing latencies [13]. These simulations show that good speedups can be obtained by implementing production systems on MPCs with low latencies [22]. The simulations also indicate that the constant node processors can quickly become bottlenecks if the initial tokens are not generated and sent fast enough. In our current implementation, we have hashed the constant nodes to take care of such a possibility. If the constant node processors continue to be bottlenecks inspite of this, then schemes proposed in [22] can be used to remove them.

Finally, we would like to reiterate the importance of mapping production systems on MPCs. Current production systems offer limited (10-20 fold) parallelism [9]. We have shown that the MPCs are capable of exploiting this limited parallelism. However, production systems with more inherent parallelism are getting designed [14]. In such production systems, the parallelism is expected to be much higher [21]. For such production systems, it becomes necessary to analyze easily scalable architectures such as the MPCs for their implementations.

7. Summary

Recent advances in interconnection network technology and processing node design have reduced the latency and message handling overheads in MPCs to a few microseconds. In this paper we addressed the issue of efficiently implementing production systems on these new-generation MPCs. We conclude that it is indeed quite possible to implement production systems efficiently on MPCs. At a high level, our mapping corresponds to an object oriented system, with Rete network nodes passing tokens to each other using messages. At a lower level, however, instead of mapping each Rete node onto a single processor, the state and the code associated with a node are distributed among the multiple processors. The main data structure that we exploit in our mapping is a concurrent distributed hash-table that not only allows activations of distinct Rete nodes to be processed in parallel, but also allows multiple activations of the same node to be processed in parallel. A single node activation is further split into two micro-tasks that are processed in parallel, resulting in very high expected performance.

Acknowledgements

We would like to thank H. T. Kung for questioning our assumptions about shared memory architectures. We would like to thank Charles Forgy, Brian Milnes, Allen Newell and Peter Steenkiste for many useful comments on earlier drafts of this paper. We would also like to thank Kathy Swedlow for technical editing.

References

[1] Acharya, A., Kalp, D., Tambe, M.
Cross Products and Long Chains.
Technical Report, Carnegie Mellon University Computer Science Department, In preparation.

[2] Brownston, L., Farrell, R., Kant, E., Martin, N.
Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.
Addison-Wesley, 1985.

[3] Dally, W. J.
Directions in Concurrent Computing.
In *Proceedings of ICCD-86.* October, 1986.

[4] Dally, W. J.
Wire Efficient VLSI Multiprocessor Communication Networks.
In *Stanford Conference on Advanced Research in VLSI.* 1987.

[5] Dally, W. J., Chao, L., Chien, A., Hassoun, S., Horwat, W., Kaplan, J., Song, P., Totty, B., Wills, S.
Architecture of a Message-Driven Processor.
In *International Symposium on Computer Architecture.* 1987.

[6] Dally, W. J.
A VLSI Architecure for Concurrent Data Structures.
PhD thesis, California Institute of Technology, 1987.

[7] Forgy, C. L.
Rete: A fast algorithm for many pattern/many object pattern match problem.
Artificial Intelligence 19:17-37, 1982.

[8] Forgy, C. L.
The OPS83 Report.
Technical Report 84-133, Carnegie Mellon University Computer Science Department, May, 1984.

[9] Gupta, A.
Parallelism in Production Systems.
PhD thesis, Carnegie Mellon University, March, 1986.

[10] Gupta, A., Forgy, C. L., Kalp, D., Newell, A., Tambe, M. S.
Parallel OPS5 on the Encore Multimax.
In *Proceedings of the International Conference on Parallel Processing.* August, 1988.

[11] Hillyer, B. K. and Shaw, D. E.
Execution of OPS5 production systems on a Massively Parallel Machine.
Journal of Parallel and Distributed Processing 3:236-268, 1986.

[12] Hoare, C. A. R.
Communicating sequential processes.
Communications of ACM 21(8):666-677, 1978.

[13] Kung, H. T., Steenkiste, P., Bitz, F.
The Nectar computer architecture.
Personal Communication.

[14] Laird, J. E., Newell, A., & Rosenbloom, P. S.
Soar: An architecture for general intelligence.
Artificial Intelligence 33:1-64, 1987.

[15] Mattern, F.
Algorithms for distributed termination detection.
Journal of Distributed Computing 2:161-175, 1987.

[16] Miranker, D. P.
TREAT: A New and Efficient Algorithm for AI Production Systems.
PhD thesis, Columbia University, 1987.

[17] Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
PhD thesis, Carnegie-Mellon University, March, 1987.

[18] Schreiner, F. , Zimmerman, G.
Pesa-1 — A Parallel Architecture for Production Systems.
In *International Conference on Parallel Processing.* 1987.

[19] Seitz, C., Athas, W., Flaig, C., Martin, A., Seizovic, J., Steele, C., Su, W.
The Architecture and Programming of the AMETEK 2010 Multicomputer.
In *Hypercube concurrent computer and applications.* 1988.

[20] Sietz, C. L.
The Cosmic Cube.
Communications of ACM C-33(12), 1984.

[21] Tambe, M. S., Kalp, D., Gupta, A., Forgy, C. L., Milnes, B., Newell, A.
Soar-PSM/E: Investigating match parallelism in a learning production system.
In *Proceedings of the PPEALS-88.* 1988.

[22] Tambe, M., Bitz, F., Steenkiste, P.
Production Systems on the Nectar: Simulation Results and Analysis.
Technical Report, Carnegie Mellon University Computer Science Department, In preparation.