NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Implementation of Production Systems on Message Passing Computers: Techniques, Simulation Results and Analysis

Milind Tambe, Anurag Acharya
Anoop Gupta

20 April 1989
CMU-CS-89-129 3

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, Ohio 45433-6543; and by the Encore Computer Corporation. 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 the Encore Computer Corporation, the Digital Equipment Corporation, the Defense Advanced Research Projects Agency or the US Government.

Implementation of Production Systems on Message-Passing Computers: Techniques, Simulation Results and Analysis

Milind Tambe and Anurag Acharya
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Anoop Gupta
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The two important parallel architecture classes are the shared-memory architectures and the message-passing architectures. In the past researchers working on the parallel implementations of production systems have focused 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-handling 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 reduced the network latency and message-handling 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. We present simulation results for this mapping, and show that message-passing computers can provide good speedups for production systems. We perform a detailed analysis of the problems in the proposed implementation and suggest solutions for some of the problems. The results presented in this paper will guide our real implementation of production systems on a message-passing computer.

University Libraries
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Table of Contents

1. Introduction
2. Background
    2.1. OPS5
    2.2. Rete
3. Proposed Mapping on MPCs
    3.1. The Base Mapping
    3.2. Variations of the Base Mapping
4. Simulator Details
5. Results and Analysis
    5.1. Speedups Obtained and the Impact of Overheads
    5.2. Increasing the Obtained Speedups
        5.2.1. Small Cycles
        5.2.2. Uneven Distribution of Tokens
6. Summary and Future Work
Acknowledgements
References

List of Figures

Figure 2-1: An OPS5 production and its instantiation. 3
Figure 2-2: An example production and its network. 4
Figure 3-1: A high level view of the Mapping on the MPCs. 7
Figure 3-2: The micro-task mapping of Rete on the MPC. 7
Figure 3-3: Modified version of the mapping used for the simulations. 9
Figure 4-1: Trace input to the simulator. 11
Figure 5-1: Speedups with zero message-passing overheads. 13
Figure 5-2: Speedups with varying overheads: Rubik (top), Tourney (middle), Weaver (bottom). 15
Figure 5-3: Unsharing nodes of the Rete network. 16
Figure 5-4: Speedups with the unsharing. 17
Figure 5-5: Distribution of tokens in two independent cycles for Rubik. 18
Figure 5-6: Speedups with the copy and constraint. 20

List of Tables

Table 5-1: Breakdown of message-processing overheads into send and receive times. 13
Table 5-2: Tokens in the sections of the three programs. 16

1. Introduction

Production systems (or rule-based systems) occupy a prominent place in the field of AI. They have been used extensively in efforts to develop expert systems spanning a wide variety of applications (e.g. R1/XCON [28], SPAM [29], Weaver [20]), as well as to understand and model the nature of intelligence (e.g. ACT* [1], Soar [25]). However, production system programs are extremely computation intensive and slow. This limits the utility of these systems both in research and application environments. Further research and development in production systems will require enlarging the production-memory (knowledge bases) in these systems [3, 31], which will only exacerbate the problem of long run times. Therefore, efficient implementations are critical for continued research in production systems.

This paper focuses on the use of parallelism for speeding up 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 implementation suggested in [13] is about 100. The early message-passing computers (MPCs), such as the Cosmic Cube [34], had high network latencies of ≈ 2 millisecond (ms) and high message-handling overheads of ≈ 300 microseconds (μs). This made it impossible to use them for the purpose of exploiting fine grained parallelism. As a consequence, researchers have concentrated on evaluating special-purpose architectures and shared memory multiprocessors as production system engines [6, 13, 17, 30, 32, 33, 36]. However, recent developments such as *worm-hole routing* [7, 8], have reduced the network latencies to 2-3 μs and special-purpose message-handling hardware such as the *Message Driven Processor* [9] will potentially reduce the message-handling overhead by an order of magnitude. As a result, MPCs have become a suitable target architecture for implementing production systems. As an additional motivation, we expect that, in near future, production systems for perceptual and sensory tasks will be built [31]. This, in conjunction with new applications in fields such as robotics [24], is expected to cause a significant increase in the available parallelism. Also, researchers are investigating formalisms that permit more explicit expression of parallelism in production systems [18, 19]. These formalisms are expected to increase the available parallelism. Therefore, it is necessary to evaluate easily scalable architectures. These considerations have prompted us to investigate the use of MPCs as production system engines.

In this paper, we first present a mapping of production systems on MPCs[1]. We show that using a distributed hash-table, it is possible to exploit parallelism at a very fine granularity. We then present simulation results for this mapping, which show that the mapping can provide good speedups. The particular simulator used is based on the

[1]This paper is an extended version of [14].

implementation of Nectar [2]. Nectar is a MPC with low network latencies and low message processing overheads under development in the Computer Science Department at CMU. We intend to use these results to guide our actual implementation on the Nectar.

The paper is organized as follows: Section 2 provides background information on production systems. Section 3 describes the proposed implementation. Section 4 describes the assumptions made about the execution times. Section 5 presents the simulation results and their analysis. Finally, Section 6 concludes the paper with a summary of the results and comments about future work.

2. Background

In this paper, we concentrate on the parallelism available in OPS5 [5] and OPS5-like languages [23]. These languages are widely available and are extensively used by AI researchers. All the systems mentioned in the previous section are implemented in OPS5-like languages. Section 2.1 describes the basics of OPS5 production systems. Section 2.2 describes the *Rete* matching algorithm. Rete is the standard algorithm used for existing uniprocessor and parallel processor implementations of OPS5 and Soar. It also forms the basis for the implementation proposed in this paper.

2.1. OPS5

An OPS5 production system is composed of a set of *if-then* rules, or productions, that constitute the *production memory* (*PM*) and a set of data-items, called the *working memory* (*WM*). The data-items in WM, called *working memory elements* (*wmes*), are structures with a fixed set of named access functions, called attributes, much like Pascal records. Each production is composed of a set of patterns or *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)*. Figure 2-1 shows an OPS5 production and its instantiation (defined later). The production has three CEs and one action element. (In the figure, only the text in lower case is part of the OPS5 syntax; the rest are our comments to help the reader understand OPS5 syntax.)

A CE is composed of a set of tests for a wme's attribute-value pairs. These tests are of two types: *constant tests* and *equality tests*. A constant test checks whether a particular attribute of the wme has a given constant value which is either a symbol or a number. For instance, the test *^color blue* in the production in Figure 2-1 is a constant test. An equality test binds a variable (syntactically any symbol enclosed in angled brackets: e.g <var>) to the value of a particular attribute and checks whether the bound value is consistent with all other values bound to the same variable within the scope of the production. A CE may be optionally negated, i.e., preceded with a dash (—). To satisfy a non-negated CE, a wme must satisfy all the tests specified in the CE. A negated CE is staisfied, if there is no wme

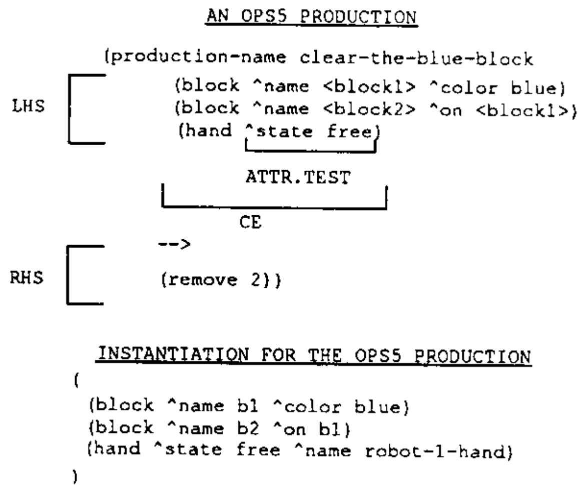


Figure 2-1: An OPS5 production and its instantiation.

that satisfies all its test. If each of the non-negated CEs of a production, matches a wme and none of its negated CEs matches a wme, then the production is *satisfied*. The set of wmes that conjunctively satisfy a production is referred to as an *instantiation* of the production. Figure 2-1 shows an instantiation. The set of all instantiations, active at any given time, is called the *conflict set*.

An OPS5 program is executed by an interpreter that cycles through three phases: match, resolve and act (the MRA cycle). The match phase of the MRA cycle updates the conflict set with all the new instantiations. The resolve phase uses a selection procedure called *conflict resolution* to choose a single instantiation from the conflict set, which is then *fired*. When a production instantiation fires, the RHS *actions* associated with the production are executed and the instantiation is removed from the conflict set. RHS actions may add wmes to WM, delete wmes from WM, perform input/output, or call a user-defined function.

2.2. Rete

The Rete matching algorithm [10] is a highly efficient algorithm for the match phase that has been shown to be suitable for parallel implementation [13, 16, 36]. The algorithm gains its efficiency from two sources. First, it stores the partial results of match from previous cycles for use in subsequent cycles, exploiting the fact that only a small fraction of WM changes on each cycle. Second, it attempts to perform tests that are common to CEs of multiple productions only once, by *sharing* them in a directed acyclic graph structure, called the *Rete network*.

The algorithm performs match using a special kind of data-flow network that is compiled from the LHS of productions, before the production system is actually executed. An example production and the network for this production is shown in Figure 2-2.

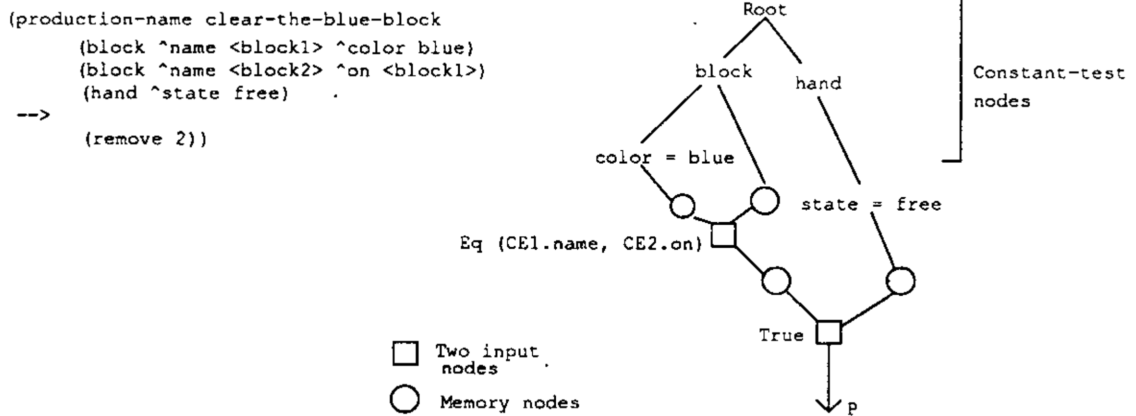


Figure 2-2: An example production and its network.

This data-flow network passes items called *tokens* across its arcs between its nodes. Tokens are *partial instantiations* of productions. They consist of a *tag*, a *list of wme IDs*,[2] and a *list of variable bindings*. The tag is either a + (plus) or a – (minus) indicating the addition or deletion of the token. The list of wme IDs identifies the wmes matching a subsequence of the CEs of the production. The list of variable bindings associated with a token corresponds to the bindings for the variables in those CEs that the algorithm has already matched.

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

1. *Constant-test nodes*: These are used to perform the constant tests of the CEs and always appear in the top part of the network.

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 WM by the most recent production firing have to be processed every cycle. As shown in Figure 2-2, memory nodes appear on both the inputs of a two-input node.

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* (and vice versa when a token arrives from the right memory). All token pairs that have consistent variable bindings are bound into new tokens and flow down the link to the left-inputs of successors of the node.

[2] Each wme has a unique ID assigned to it.

At the beginning of the MRA cycle, the changes to WM flow down the network from the root (see Figure 2-2). They match the constant-test nodes, generating tokens that get stored in memory nodes. These tokens then flow into the two-input nodes. The combined activity of storing a token in a memory node and performing the following two-input node test to generate successor tokens is referred to as a *two-input node activation* (or just an activation). An activation can be *left* or *right* depending on the memory in which the token is stored.

3. Proposed Mapping on MPCs

In general, the match phase of the MRA cycle is the fundamental computational bottleneck in a production system. So, concentrating our effort on speeding up match is critical. Therefore, our mapping focuses on the match phase. The first subsection presents our base mapping. The second subsection presents some variations of the base mapping. One of these variations is used in our simulations.

3.1. The Base Mapping

One possible, perhaps intuitive, scheme for implementing production systems on the MPCs arises from viewing the Rete match algorithm 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 WM change generates multiple activations of the same node of the Rete network resulting in a serialization of their processing. As a consequence, the processor handling these activations becomes a bottleneck.

To overcome these limitations, we propose an alternative mapping based on a distributed hash-table [7]. To be able to adequately describe this mapping, we need to briefly digress into a discussion of the role of hash-tables in the Rete match algorithm.

The Rete algorithm spends most of its time in processing two-input node activations [13]. Therefore, its performance can be effectively improved by reducing the computation performed during a two-input node activation. A major time-consuming factor in this computation is the comparison of an incoming token with the tokens in the opposite memory (the right memory is the opposite memory of a left-token, and vice versa). If the tokens in the memory nodes are stored in a hash-table, instead of a linear list, the number of comparisons performed during a two-input node activation is reduced, greatly improving the Rete performance.

Our proposed implementation is based on two *global* hash-tables — one for all the left memories and the other for all the right memories. Instead of being stored in linear lists at individual memory nodes, tokens are hashed and stored in (or deleted from) buckets of these hash-tables. The hash function applied to the tokens uses (as parameters), the *node-id* of the destination two-input node, and the *values* bound to the variables that are tested for equality at the destination node. Therefore, tokens flowing into a two-input node with the same values bound to the variables hash to the same index. For a left (right) token with a hash-bucket index of K, the two-input node needs to search only the right (left) hash-table bucket with index K, eliminating the need for a search through its entire right (left) memory. Thus, when processing a node activation, the left and right buckets at only one index need to be accessed.

At an abstract level, the computation in the two-input nodes of the Rete network can now be described as operations on the two global hash-tables. Tokens are generated from a pair of corresponding left-right hash-buckets and sent to some other pair(s) of hash-buckets. Tokens keep moving from bucket to bucket, until the end of the match phase. Hashing may reduce the number of token comparisons (at two-input nodes) by as much as a factor of 10 [17]. Therefore, hashed memories are the data-structure of choice for Rete implementations.[3]

A high-level picture of our mapping is presented in Figure 3-1. As shown in the figure 3-1, the parallel mapping partitions the available processors into one *control processor*, a small set of *constant-test processors*, a small set of *conflict-set processors*, and a large set of *match processors*. The constant-test processors perform the constant tests specified by the Rete net. The constant nodes of the Rete net are partitioned and assigned to the constant-test processors. The match processors perform the variable tests. The conflict-set is partitioned and assigned to the conflict-set processors. They select the best instantiation and send it 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.

The match processors perform the function of Rete's two-input nodes using the hash-table described previously. The range of hash indices is partitioned among the match processors (both left and right hash-buckets with the same index are assigned to the same processor). Tokens destined for different hash-buckets can be processed in parallel, therefore, distributing the hash-table among processors allows parallel processing of tokens detsined for the same two-input node as well as tokens destined for different two-input nodes.

In particular, the partitions of the hash-tables are assigned to a *processor pair*: the left buckets to the left processor

---

[3]Discrimination trees might be an alternative

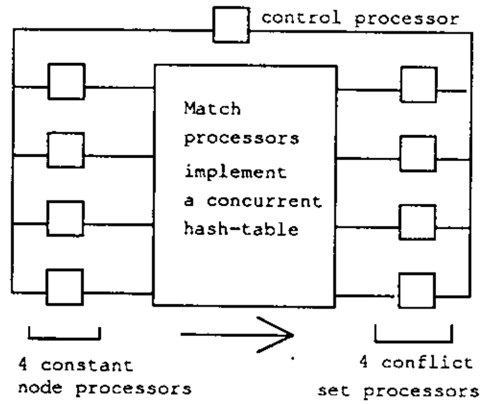


Figure 3-1: A high level view of the Mapping on the MPCs.

and the right buckets to the right processor. This mapping is shown in Figure 3-2. The communication with the processor pair is restricted to the left processor. Allowing communication with both the left and the right processors can result in creation of duplicate tokens leading to incorrect behavior, and it does not gain as much in concurrency[4].

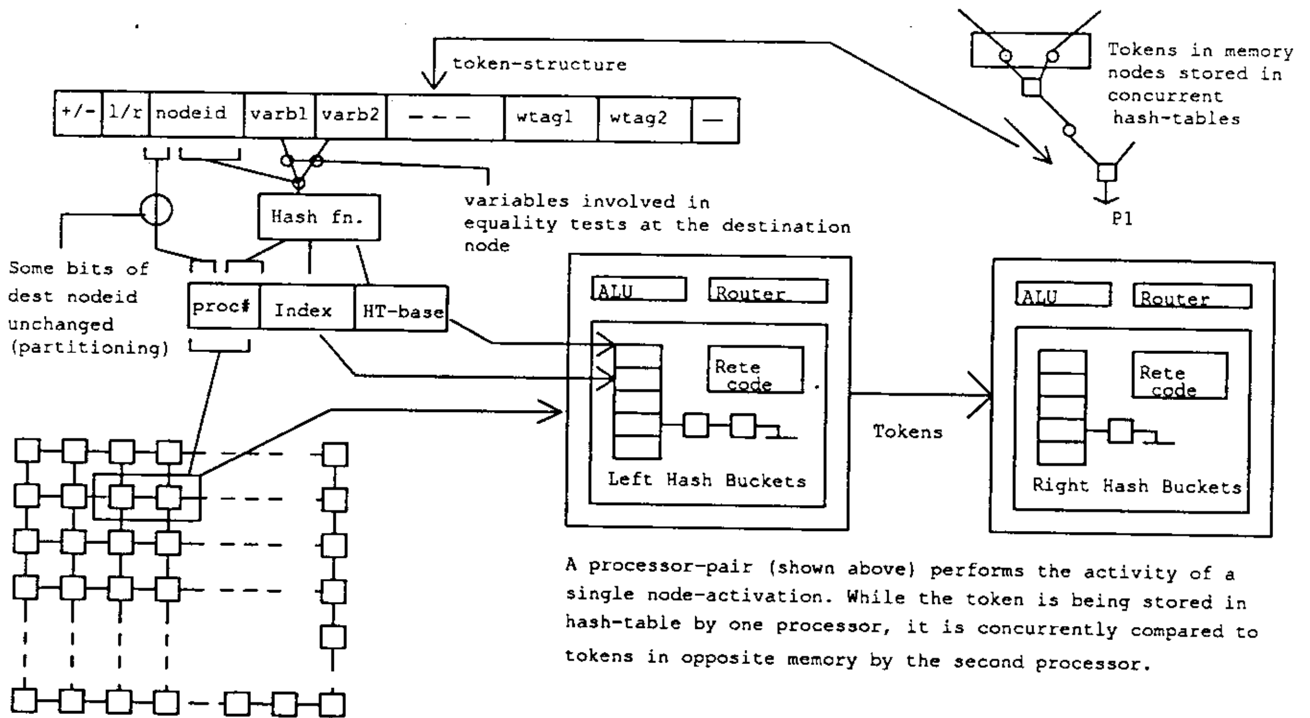


Figure 3-2: The micro-task mapping of Rete on the MPC.

[4]For a discussion of the problem of duplicate token generation, see [13].

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 that owns the buckets to which it hashed. 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 code for the Rete net is to be encoded in the OPS83 [11] software technology. With this encoding, large OPS5 programs (with ≈ 1000 productions) require about 1-2 Mbytes of memory, a potential problem, since a message-passing processor may have only 10-20 kbytes of local memory [8]. Two strategies can therefore be used to save space in the actual implementation:

- 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 belonging to a single production are put into different partitions.
- 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.

3.2. Variations of the Base Mapping

Depending on the communication overheads and the number of processors available, some variation of the scheme presented in the previous subsection may turn out to be more suitable for a particular MPC. Three such variations are described below:

1. *Processor-pairs:* In our mappings we have employed a processor-pair to evaluate a single node-activation. If the number of processors is small and the processor utilization is important, it is possible to allocate both the left and right buckets to a single processor instead.
2. *Constant-node processors:* The control processor in the mapping presented above broadcasts wmes to some designated constant-node processors. However, these processors could become bottlenecks, if the communication overheads are comparatively high. If so, broadcasting wmes to all processors at the beginning of the MRA cycle would be preferable. All the processors then evaluate the constant test nodes and generate all the initial tokens in a match phase.

3. *Conflict-set processors:* If the number of processors available is limited, the control processor could be used to process the conflict-set as well. The instantiations could be directly sent to the control processor. The contol processor would then be in charge of both the resolve and the act phases.

Initially the Nectar architecture [2][5] will have only a limited number of processors (32). Consequently, for our simulation studies we assume the following variation (see Figure 3-3) of the mapping presented previously.

The figure shows a *control processor*, responsible for performing conflict-resolution on the instantiations, evaluating the RHS, and performing other functions of the OPS5 interpreter. The match processors implement the distributed hash-table described previously. The match procedure proceeds as follows:

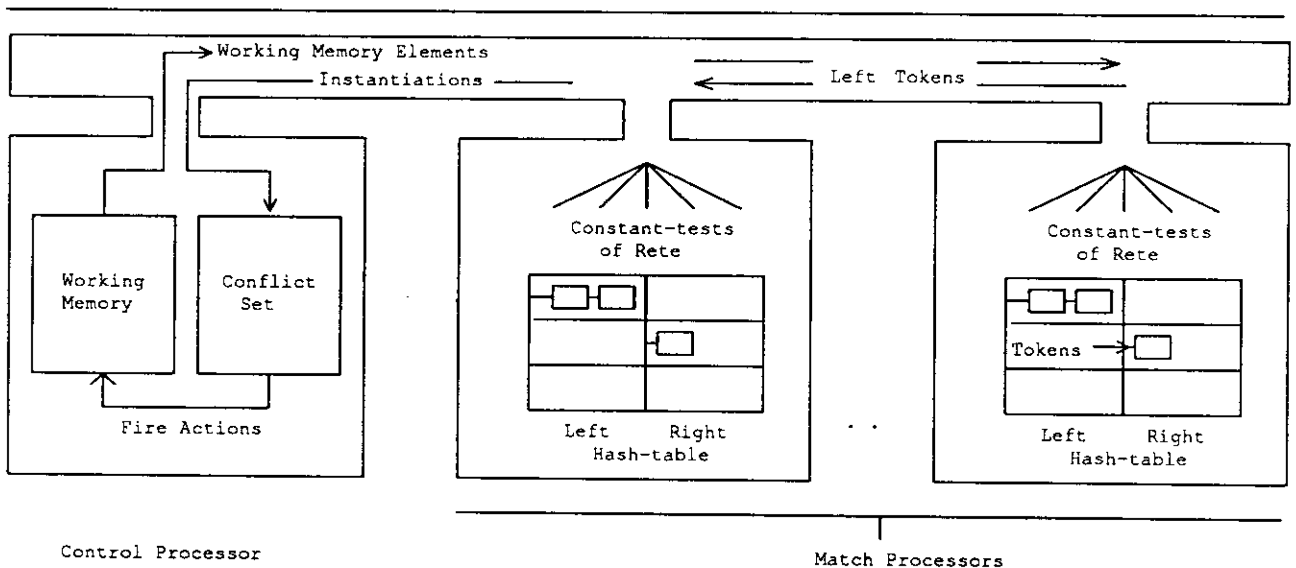


Figure 3-3: Modified version of the mapping used for the simulations.

1. At the beginning of MRA cycle, the control processor broadcasts one packet (containing all the wmes generated in the previous cycle) to all match processors.

2. All the match processors perform all constant tests. The tokens generated from the constant tests generate node-activations, most of which are right-activations (see Figure 2.2). These tokens are then hashed; if the required hash-bucket is not resident in the processor's local memory, then no further processing is done on the token (Recall that the range of hash indices is partitioned and assigned to the match processors). Else, the processor moves on to step 3.

3. For each hash-table activation for a bucket in the match processor's local memory, the token is added

[5]Nectar is the target of our real implementation

to (or deleted from) the bucket. The token is then matched with tokens in the corresponding hash-bucket in the opposite hash-table. If new tokens are generated, they are hashed and sent to the processor that owns the corresponding bucket. Recall from Section 2.2 that the tokens generated at two-input nodes result only in left activations.

4. Tokens received by a processor are processed exactly as in step 3.

5. All production instantiations are sent back to the control processor. The control processor starts the match phase of a new cycle when the current cycle ends.

In step 2, work is duplicated across the match processors — all the processors evaluate all the constant tests. However, since the constant tests take a very small amount of time, this duplication of effort is not a significant problem. Further reduction in proportion of time spent in constant tests is also possible [13], e.g., implementing constant-tests using hashing can speed up the constant-tests by a factor of 5.

It has been shown [36] that while the right activations are evenly hashed, left activations can be fairly unevenly distributed. To balance the computation, the mapping used in the simulation employs two different granularities:

1. The tokens generated directly by the wmes (mostly right-activations) on a processor are grouped together and processed as a single unit (see step 3 previously).

2. The left-tokens are processed as separate units (see step 4 previously).

Thus, the low variance section (tokens directly generated by wmes) is processed at a coarser granularity for lowering communication overheads, while the high variance section (left-tokens) is processed at a finer granularity for load balancing. In general, parallel programs tend to work with fixed granularity (for instance, Kruskal and Smith [22] assume fixedness in their definition of granularity). Thus, this modification illustrates an alternative strategy with multiple granularities.

4. Simulator Details

The simulator used was developed by the Nectar group [4]. A detailed trace of the activity of the hash-table used for the Rete network, corresponding to the actual production system runs, is input to the simulator. Figure 4-1 shows a small fragment of the trace that is fed as an input to the simulator.

Given this input, the simulator builds a big hash-table like data-structure. Each *processor* gets its partition of the hash-table. Using this data structure, the algorithm from the previous section is simulated. The simulator uses the following estimates for the cost of node-activations:

```
Hash bucket = 27
  input token: nodeid = 27, node type = AND, direction = add, left
  output token: nodeid = 121, hash bucket = 77
                nodeid = 121, hash bucket = 143

Hash bucket = 10
  input token: nodeid = 15, node type = AND, direction = delete, right
  output token: --

Hash bucket = 25
  input token: nodeid = 150, node type = NOT, direction = add, right
  output token: nodeid = 75, hash bucket = 77
```

Figure 4-1: Trace input to the simulator.

- The time to evaluate all of the constant test nodes: 30 μs
- The time to add or delete one left token: 32 μs
- The time to add or delete one right token: 16 μs
- The time spent to compare a token with tokens in opposite memory (per new token generated): 16 μs

The particular times mentioned above were obtained using the profile data from our previous implementations [16, 36]. (We also experimented with some variations in the ratio of the time to process the left and right tokens. These variations did not make more than 5 - 10% difference in the results.) The cost of processing the constant-test nodes is obtained using estimates from [13] — assuming that the constant-test nodes are implemented using hashing.

Like any other simulation, ours has its share of inaccuracies. First, tokens could be generated in different orders in parallel and serial implementations. The trace driving the simulator is obtained from a uniprocessor, and thus does not always accurately model the activity of a parallel processor. However, this problem is not expected to be serious, since in our previous implementations [16, 36], a change in the order of token generation has not been a problem (a maximum variation of 10%). Second, we do not simulate *termination detection*. In distributed systems, termination detection has been widely studied (for instance, see [27]). Investigations of the impacts of the various termination detection schemes on our implementation and the selection of the most suitable scheme will be the subject of future work.

5. Results and Analysis

This section reports on the results and analysis of our experiments. Previous research has analyzed parallelism in production systems in detail at the levels of complete programs and individual MRA cycles. We used the results from this research to structure our experiments. Thus, rather than determining speedups available for some particular production systems, we simulated several characteristic sections of production system execution. These sections were identified on the basis of previously reported research. This allows us to focus our attention on known and suspected bottlenecks. Moreover, concentrating on small sections allowed us to analyze the behavior of the production systems at a finer intra-cycle level. For the purposes of the simulation, three traces were chosen. The traces are from the production systems used in [16]. The traces we used are:

- *Good speedups* (**Rubik**): This section was taken from the execution trace of Rubik, a program to solve the Rubik's cube. The section represents four consecutive cycles.

- *Small cycles* (**Weaver**): This section was taken from the execution trace of Weaver, a program to perform VLSI routing. Small cycles are those with few (100 or less) tokens in them. As shown in [36], small cycles are special cause of concern, since they limit speedups. The section represents four consecutive small cycles.

- *Cross-product* (**Tourney**): This section was taken from the execution trace of Tourney, a program to do scheduling for a tournament. (A cross-product is generated when a token arriving at a two-input node finds a large number of matching token in the opposite memory; thus generating a large number of successor tokens.) One cycle with a heavy cross-product was chosen from this program. This particular cycle has a large number of non-randomized tokens, tokens that hash to the same bucket. This uneven distribution of tokens is a major problem for parallelization [16]. Four small cycles that surround the cross-product cycle were also included in the section to provide some points of comparison for the cross-product cycle.

In the rest of this paper, we refer to these sections by the name of the programs from which they were taken — this should not be construed to mean that the numbers presented are for the entire programs. Approximately 200 runs were simulated. Each simulation run required from half an hour to six hours on a SUN 3/260. The parameters for the simulations were the number of processors employed and the communication delays.

5.1. Speedups Obtained and the Impact of Overheads

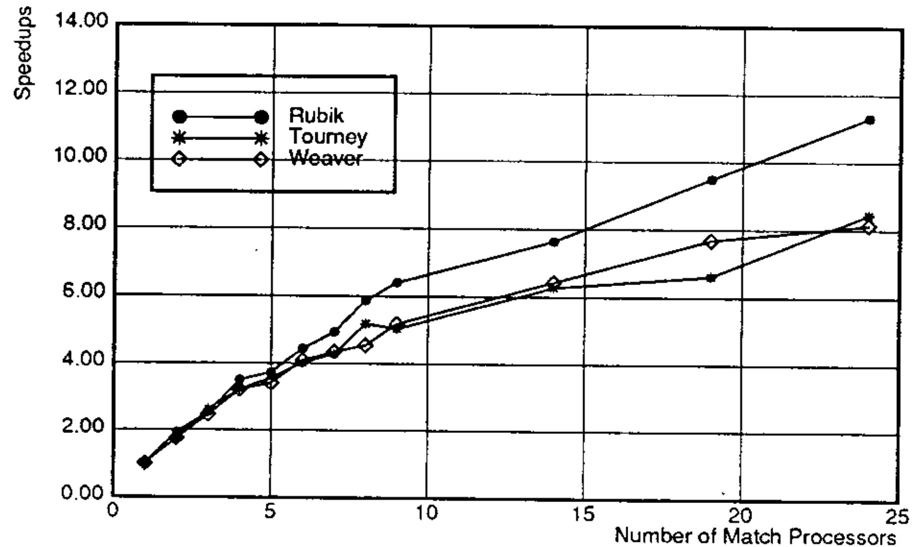


Figure 5-1: Speedups with zero message-passing overheads.

Figure 5-1 shows the speedups obtained for the three systems with zero interconnection network latency and zero message-processing overhead. The buckets were distributed to the processors in a round robin manner. This graph shows the best possible speedup, given the proposed implementation. As expected, Rubik has the largest overall speedup. The results from runs simulating a single match processor with zero communication overheads are used as the base cases for calculating all the speedups presented in the paper.

For the rest of the runs, we kept the interconnection network latencies set to the value provided by the Nectar group (0.5 μs) and varied the value of the message-processing overhead to study the impact of communication overhead on the speedups. The settings that we used are listed in Table 5-1.

Runs	Send overhead	Receive overhead	Total overhead
Run 1	0 μs	0 μs	0 μs
Run 2	5 μs	3 μs	8 μs
Run 3	10 μs	6 μs	16 μs
Run 4	20 μs	12 μs	32 bμs

Table 5-1: Breakdown of message-processing overheads into send and receive times.

Figure 5-2 shows the impact of increasing the message-passing overheads in the three sections of Rubik, Tourney and Weaver. The (0 μs) graphs are the ones shown in Figure 5-1 with zero overheads. The impact of message-passing overheads are comparatively low in Rubik (loss of 30% of speedup), somewhat higher in Tourney (loss of

about 45% of speedup), and even higher in Weaver (loss of up to 50% of speedup)[6]. This difference is explained by Table 5-2, which shows the right and left activations for each trace. Recall that only the left activations need to be communicated to other processors. Since Rubik has a smaller percentage of left activations (of all activations), it is not affected as much as Tourney and Weaver which have much larger percentages of their activations as left activations.

Interestingly, there are dips in the speedup graphs showing a *decrease* in the speedup with an *increase* in the number of processors employed. This shows that the partitioning of the hash-tables could result in an uneven distribution of the processing load. Section 5.2 analyzes this effect in detail.

Given the large number of tokens generated, it is interesting to note that the interconnection network was mostly (97-98% time) idle. This is explained by the small delay (0.5 μs) associated with the interconnection network. Thus, for our mapping, the interconnection network is not a bottleneck.

5.2. Increasing the Obtained Speedups

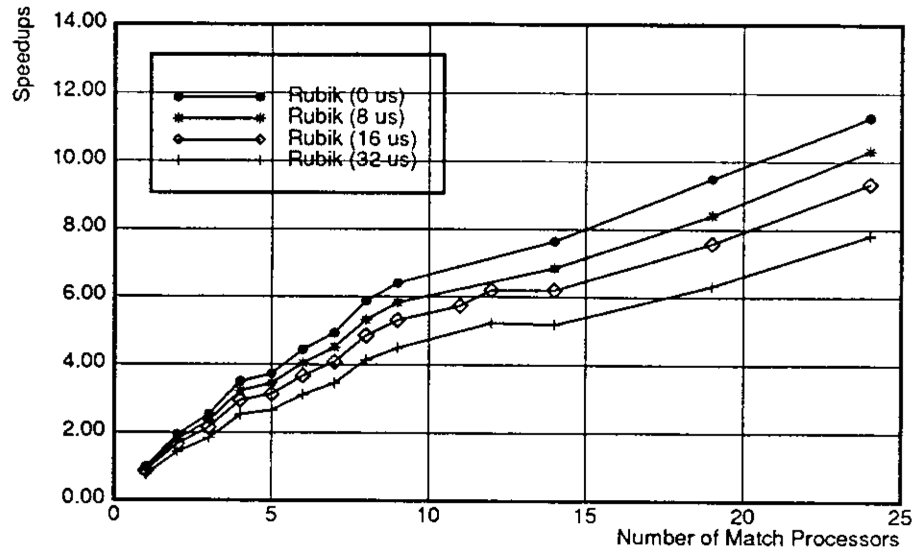
The simulations in the previous section have shown that up to 8-12 fold speedups are available in the three characteristic sections. These speedups are comparable to those achieved in these sections on our shared-bus implementation [21]. However, the large number of tokens and low contention in the interconnection network suggests that a larger amount of parallelism should be available. This is a problem that the earlier analyses have not been able to explain completely. In this subsection, we investigate some of these problems and present some methods of improving the speedups.

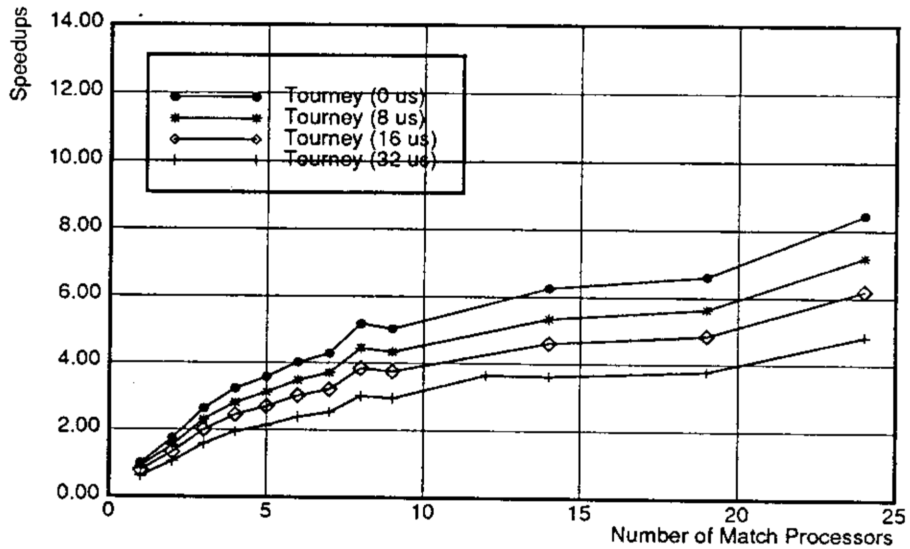
5.2.1. Small Cycles

We had chosen the traces from Weaver to investigate the phenomenon small cycles [16]. One problem that we identified in these cycles is the slow generation of successor tokens. There are very few right activations in these cycles, as seen from Table 5-2. However, there are only three left-activations that generate a majority (120 out of about 150) of the activations in one of the cycles. A processor that generates such large number of successors becomes a bottleneck, since it takes about 16 μs to generate every successor. It is possible to avoid this multiple successor problem by transforming the Rete network node that generates the successors[7]. The transformation splits the bottleneck node so that the generation of the successors can proceed in parallel. This can be achieved by one or more of the following methods.

[6]The speedups for Tourney are somewhat overestimated. Due to the poor hashing discrimination, a large number of tokens hash to a few buckets. Token deletion therefore requires more time to search through these buckets than the constant time assumed by the simulator.

[7]We have assigned 16 μs to all successors, irrespective of how these were produced. This may exaggerate some of the results, since in general, some successors may be produced much faster than others, especially if the successors share tests at the two-input node. However, the techniques shown here are applicable whenever a large number of successors are generated from a single hash-bucket site.





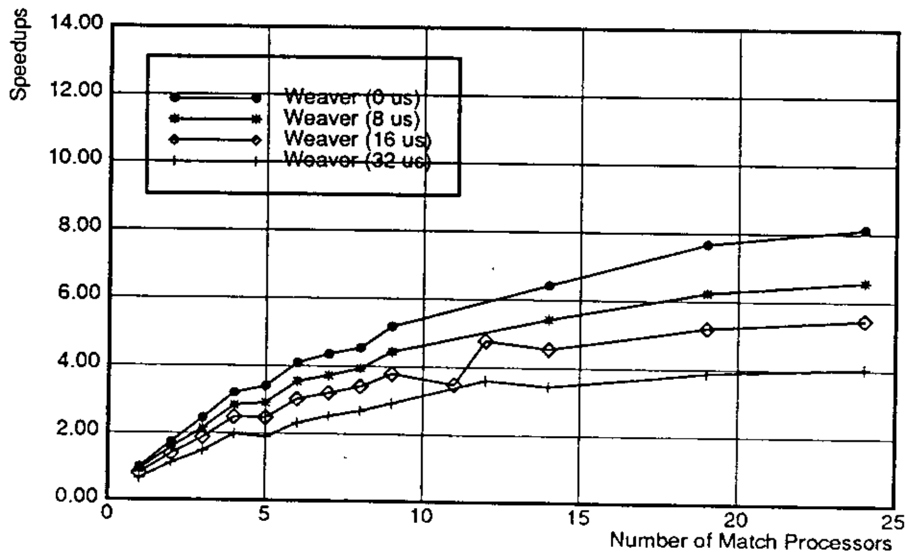


Figure 5-2: Speedups with varying overheads: Rubik (top), Tourney (middle), Weaver (bottom).

Program	Left activations	Right activations	Total activations
Rubik	2388 (28%)	6114 (72%)	8502
Tourney	10667 (99%)	83 (1%)	10750
Weaver	338 (81%)	78 (19%)	416

Table 5-2: Tokens in the sections of the three programs.

1. *Unsharing the Rete network nodes* where multiple tokens are generated. For instance, in Figure 5-3, the two outputs O1 and O2 share the two-input node formed by joining conditions I1 and I2. If we unshare in the manner shown, the outputs for O1 and O2 are generated independently, thus removing the bottleneck of generating a large number of tokens from one site. Some work is duplicated; but, given the limited number of tokens generated in a small cycle, this duplication should not be a problem. The effect of sharing in general is quite limited (a factor of 1.1 to 1.6 in the running time) [13, 30]. Therefore, such unsharing should not be a problem in general.

2. *Introducing one or more dummy nodes* in the Rete network. This is quite similar to the introduction of dummy-nodes in ([13], Chapter 4). These dummy elements can divide the successors into 2-4 parts.

3. *Applying copy and constraint* [35] on the node that generates a large number of successors. Multiple copies of the given node are made, each of which generates part of the activations generated by the original node.

Figure 5-4 shows the impact of using the first scheme on speedup achieved for Weaver. As might be seen, there is a substantial improvement.

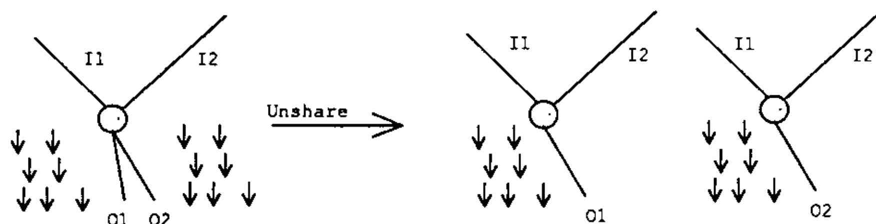


Figure 5-3: Unsharing nodes of the Rete network.

The problem of small cycles could possibly be solved in yet another way, especially for systems with high communication overheads. It is possible to identify the productions affected in small cycles and process all the tokens associated with matching the production on a single processor. Since such cycles do not possess much

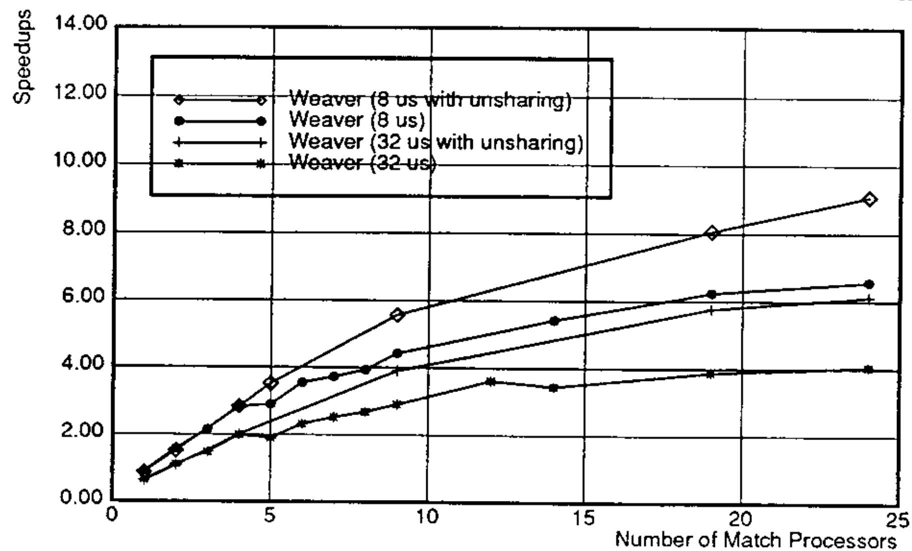


Figure 5-4: Speedups with the unsharing.

parallelism, avoiding the communication overheads seems to be a useful strategy. This would add another level of granularity to the mapping. Though the different granularities are decided *a priori*, the mapping would seem to converge to the variable granularities approach promoted in [15].

5.2.2. Uneven Distribution of Tokens

The simulation results show that the average *idle time* of the processors increases with increasing number of processors. This suggests an uneven distribution of tokens. Figure 5-5 shows the distribution of left tokens in two cycles for Rubik. The x-axis marks the processor number and the y-axis marks the number of activations processed by a processor in a cycle (for instance, processor number 1 processed ≈ 20 tokens in both cycles). The aggregated distribution of the tokens, even within the four MRA cycle sections used, is more or less even; however, the distribution of tokens at the level of an individual MRA cycle is quite uneven.

In addition to uneven distributions across the processors in both the cycles, we see a peculiar behavior viz. the processors busy in one cycle are seen to be idle in the next cycle and vice versa. The graphs for Tourney are quite similar. The distribution in Weaver is much more even.

The uneven distribution of tokens to processors occurs in two different ways:

1. *Poor distribution of active buckets to processors*: Although buckets are distributed equally among all processors, only some of these buckets are active, i.e., process any tokens. These active buckets get distributed poorly among processors: multiple buckets on a single processor together process a high number of tokens, while all the buckets assigned to some other processor are inactive. Both Rubik and

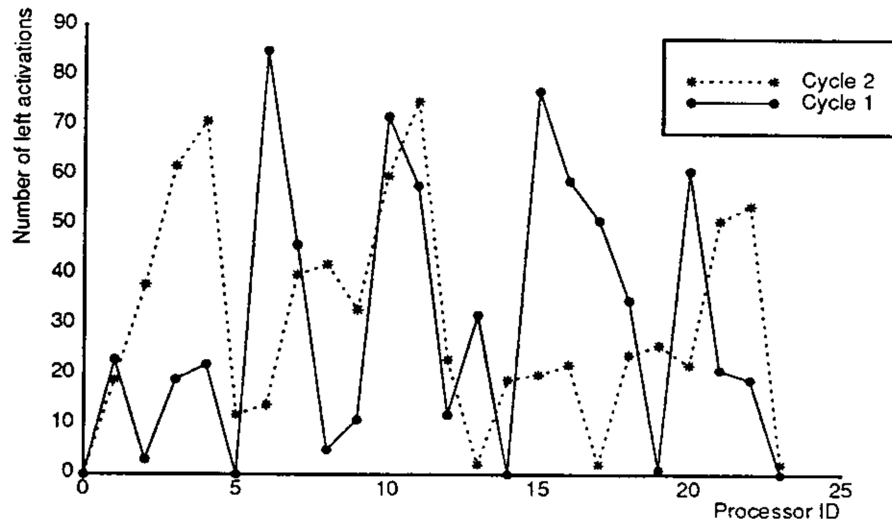


Figure 5-5: Distribution of tokens in two independent cycles for Rubik.

Tourney seem to suffer from this problem. Typically, this problem occurs for the buckets belonging to the left hash-table.

2. *Poor distribution of tokens to buckets*: A single bucket on a processor processes a high number of tokens. This is in turn caused by:

   - *Cross-products with no hashing*: A cross-product refers to a large number of activations of the same two-input node from its left input [13]. If there is no variable tested at the two-input node (as happens in the trace taken from Tourney), then the hashing scheme cannot discriminate between any of the tokens and all the tokens hash to the same bucket.

   - *Multiple-modify-effect*: This refers to the tokens generated in Rete when multiple wmes matching a single production are modified all at once. A modify action is implemented as a delete action followed by an add action. Due to the state-saving nature of the Rete-net, when a wme is deleted, all the successor left-tokens (containing the wme) have to be deleted. The add action then regenerates all the tokens — these tokens are identical to the earlier tokens, except that they differ in the value of the newly added wme. With multiple-modify actions, the delete-add cycle continues; generating multiple tokens headed for the same bucket, half of which are adds and half are deletes. This is a hitherto unsuspected effect and was uncovered by the detailed nature of the analysis performed.

The poor active-bucket distribution suggests a redistribution of the hash-table buckets so that the active buckets

are distributed evenly among processors. In assigning buckets to processors, we had originally assumed a round robin strategy. The distribution in Figure 5-5 indicates the failure of this strategy to distribute active buckets evenly among processors. A *random* distribution of the buckets to the processors was tried as an alternative, but failed to provide a significant improvement.

The failure of the random distribution of buckets to improve the performance led us to build a simple probabilistic model of this distribution. The model assumed that only a fraction of the total number of buckets are active, and that each active bucket gets only a single activation. This models allowed us verify some (perhaps intuitive) conclusions about the distribution. These conclusions are:

1. The probability of getting a completely even or totally uneven active bucket distribution is very low (< 1%). It is more likely for the bucket distribution to be somewhere in between.

2. Increasing the proportion of active buckets for the same total number of buckets increases the probability of even distributions. Typically, a large proportion of right buckets is active; hence, they get distributed evenly.

3. With an increasing number of processors, the probability of uneven distribution increases, i.e., the probability of the bucket distribution allowing linear or near-linear speedup decreases. This is part of the reason why the speedups seen in the previous section do not scale well with an increase in the number of processors.

What is the impact of the uneven distribution? To answer this question, we used an *off-line greedy* algorithm to distribute buckets among processors. This algorithm was provided a detailed trace of the activity in each bucket, (this would not be available to the actual distribution algorithm). A greedy algorithm was chosen since determining the optimal distribution of buckets to processors is a version of the multi-processor scheduling problem, a well-known NP-complete[8] problem [12]. Therefore, computing an optimal distribution (given 100 or more active buckets) could be time consuming. The greedy algorithm provided us a series of distributions, one per cycle. These distributions have a very low variance and hence should be close to the optimal distribution.

The new uniform distribution improved the speedups by a factor of 1.4. Thus, with a better distribution algorithm, we could obtain an improvement of a factor of up to 1.4 in performance. A potential solution for this distribution problem is dynamic (run-time) load balancing. However, our problem is complicated by the presence of *state* of the Rete net, i.e., the presence of tokens in the hash-buckets. A token cannot be sent to an arbitrary

[8]This assumes that the tokens in different buckets are independent of each other. If we also take into account the dependence of tokens, then the problem is a version of the precedence-constrained scheduling problem with unequal length tasks, which is a NP-hard problem.

processor, as its target hash-bucket is present only on a particular processor. Also, moving hash-buckets around to change the token distribution is too costly. Thus, it appears that we cannot take advantage of the existing dynamic load balancing schemes; and possibly, better static load distribution by source-level transformation of the production systems may be the only method for improving the performance.

The second problem for uneven token distribution was the poor distribution of tokens to buckets. The problem of poor token distribution occurs due to inability of the hashing function to discriminate between tokens going to the same two-input node. This can be solved by the copy and constraint mechanism. This is a source-level transformation which splits the culprit productions into multiple copies, each matching only a part of the data the original production matched. This allows additional discrimination to be introduced in the hashing function: the tokens belong to different productions and hence different destination node-ids, and hence hash to different buckets. Figure 5-6 shows the speedups achieved with this mechanism[9].

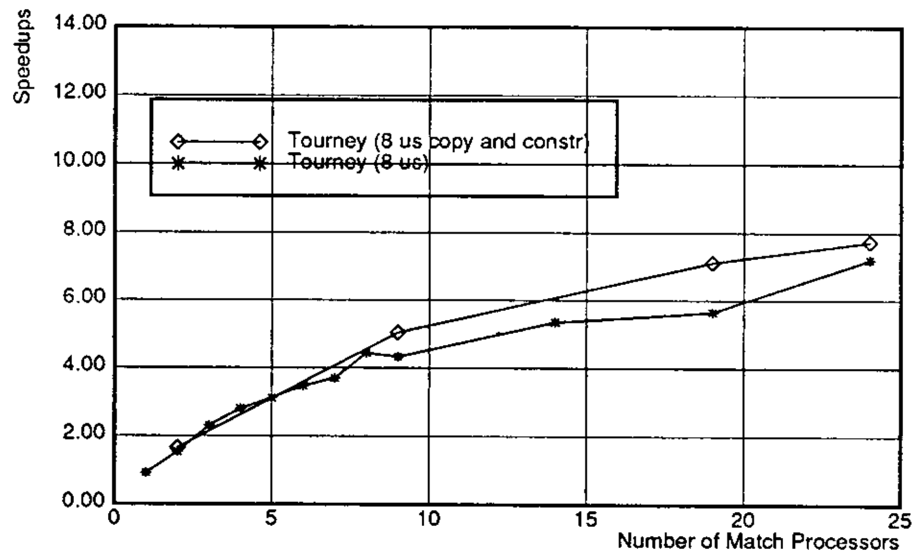


Figure 5-6: Speedups with the copy and constraint.

Although uniform distribution of tokens to processors will improve the performance of the system, it cannot completely alleviate the problem of sublinear speedups in the system. In particular, the even distribution does not take into account the precedence constraints among the tokens, caused by the structure of the Rete net. Methods for eliminating some of the precedence constraints appear in [13, 36], and we plan to adopt one of these methods in our final implementation. However, not all the constraints can be eliminated — since they arise from the structure of the Rete algorithm.

[9]As noted earlier, the speedups without copy-and-constraint are somewhat overestimated. Therefore, we do not see a big increase in the speedup.

The discussion above can be summarized in terms of some of the tradeoffs involved in message-passing and shared-bus mappings of production systems. The shared-bus mapping maintains some centralized task-queues and the hash-tables in the shared memory. The principle advantage of the the distributed mapping is the absence of these centralized task-queues, which can be a potential bottleneck. However, the distributed mapping suffers from static partitioning of the hash-table; this may lead to a poor distribution of active buckets to processors. We saw that distinct tokens, heading for distinct hash-buckets may be routed to the same processor; this causes the tokens to be processed sequentially. The problem does not arise in the shared-memory implementation, since the hash-table is not partitioned among processors. However, the problem shown in Tourney, of uneven distribution of tokens to buckets, is a serious problem even for shared-memory implementations, i.e., if multiple tokens are hashed to the same bucket, they are executed sequentially both on the distributed memory and the shared-memory implementation (to process a token, the entire hash-bucket needs to be accessed exclusively).

6. Summary and Future Work

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 showed that using a concurrent distributed hash-table data structure, it is possible to exploit parallelism at a very fine granularity and to obtain significant speedups from parallelism. We presented simulation results and showed that MPCs can successfully be used as production system engines. For a number of processors, comparable to our shared-bus implementation [16], the MPCs provide a comparable speedup in the simulated sections of the production systems. But since the speedups on the MPCs are obtained with substantially faster processors than the shared-bus implementation, the total execution time has been speeded up by a much larger factor.

Before concluding, we would like to reiterate the importance of mapping production systems on the MPCs. First, we expect that in the near future, production systems for perceptual and sensory tasks will be built [31]. This, in conjunction with new applications in the fields such as robotics [26], is expected to cause a significant increase in the available parallelism. Second, researchers are exploring methods of developing formalisms that permit more explicit (or implicit) expression of parallelism in production systems [18, 19, 25]. These formalisms are also expected to increase the amount of available parallelism. Hence, it is necessary to analyze easily scalable architectures such as the MPCs for implementing production systems.

The mapping presented in this paper may be thought of as being near the center of a continuum of mappings. At one extreme, we have the mapping of hash-tables replicated on all processors. These copies of the hash-tables need

to be consistent with each other, which would require continuous updates among the various copies. At the other extreme is a mapping with a single master-copy of the hash-table (on a single processor). All the remaining processors perform match using this single hash-table, generating contention for the processor owning the hash-table. To address some of the load-balancing issues associated with our current mapping, we intend to investigate the possibility of moving toward one or the other end of this continuum.

We plan to use these results to guide our implementation of OPS5 on Nectar. Nectar is scheduled to be operational in early 1989 and we expect to complete the first cut implementation by Fall 1989.

Acknowledgements

We would like to thank H. T. Kung for questioning our assumptions about shared memory multi-processors. We would like to thank Francois Bitz for building the simulator and Peter Steenkiste for lively discussions. We would like thank Paul Rosenbloom and Allen Newell for many helpful comments on earlier drafts of this paper. We would also like to thank Kathy Swedlow for her technical editing.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Air Force Avionics Laboratory and by the Encore Computer Corporation. 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 the Encore Computer Corporation, the Digital Equipment Corporation, the Defense Advanced Research Projects Agency or the US Government.

References

1. Anderson, J. R.. *The Architecture of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1983.

2. Arnould, E., Bitz, F., Cooper, E., Kung, H. T., Steenkiste, P. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. Proceedings of the Third International Conference on Architectural Support for Programming languages and Operating Systems, 1988. (to appear).

3. Bachant, J. and McDermott, J. "R1 Revisited: Four years in the trenches". *AI Magazine 5*, 3 (1984), 21-32.

4. Bitz, F. Simulator for the Nectar Multi-processor. Personal Communication.

5. Brownston, L., Farrell, R., Kant, E., and Martin, N.. *Programming Expert Systems in OPS5: An introduction to rule-based programming*. Addison-Wesley, Reading, Massachusetts, 1985.

6. Butler, P. L., Allen, J. D., and Bouldin, D. W. Parallel architecture for OPS5. Proceedings of the Fifteenth International Symposium on Computer Architecture, 1988, pp. 452-457.

7. Dally, W. J. *A VLSI Architecture for Concurrent Data Structures*. Ph.D. Th., Department of Computer Science, California Institute of Technology, January 1986.

8. Dally, W. J. Directions in concurrent computing. Proceedings of the International Conference on Computer Design: VLSI in Computers, October, 1986, pp. 102-106.

9. 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. International Symposium on Computer Architecture, 1987.

10. Forgy, C. L. "Rete: A fast algorithm for the many pattern/many object pattern match problem". *Artificial Intelligence 19*, 1 (1982), 17-37.

11. Forgy, C. L. The OPS83 Report. Tech. Rept. CMU-CS-84-133, Computer Science Department, Carnegie Mellon University, May, 1984.

12. Garey, M. R. and Johnson, D. S.. *Computers and Intractability: A guide to the theory of NP-completeness.* W. H. Freeman and Company, San Francisco, California, 1978.

13. Gupta, A. *Parallelism in Production Systems.* Ph.D. Th., Computer Science Department, Carnegie Mellon University, March 1986.

14. Gupta, A. and Tambe, M. Suitability of message passing computers for implementing production systems. Proceedings of the National Conference on Artificial Intelligence, August, 1988, pp. 687-692.

15. Gupta, A., and Tucker, A. Exploiting Variable Grain Parallelism at Runtime. Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems, July, 1988.

16. Gupta, A., Forgy, C. L., Kalp, D., Newell, A., and Tambe, M. Parallel OPS5 on the Encore Multimax. Proceedings of the International Conference on Parallel Processing, August, 1988, pp. 271-280.

17. Gupta, A., Tambe, M., Kalp, D., Forgy, C. L., and Newell, A. "Parallel implementation of OPS5 on the Encore Multiprocessor: Results and analysis". *International Journal of Parallel Programming 17*, 2 (1989).

18. Harvey, W., Kalp, D., Tambe, M., Acharya, A., McKeown, D., and Newell, A. Measuring the Effectiveness of Task-Level Parallelism for High-Level Vision. Proceedings of the DARPA Image Understanding Workshop.

19. Ishida, T. and Stolfo, S. Towards the parallel execution of rules in production system programs. Proceedings of the International Conference on Parallel Programming, August, 1988, pp. 568-574.

20. Jobbani, R. and Siewiorek, D. Weaver: A knowledge-based routing expert. Proceedings of the International Conference on Design Automation, June, 1985.

21. Kalp, D., Tambe, M., Gupta, A., Forgy, C., Newell, A., Acharya, A., Milnes, B., and Swedlow, K. Parallel OPS5 User's Manual. Tech. Rept. CMU-CS-88-187, Computer Science Department, Carnegie Mellon University, November, 1988.

22. Kruskal, C. and Smith, C. "On the notion of granularity". *Journal of Supercomputing 1*, 1 (1988), 395-408.

23. Laird, J. E. *Soar User's Manual: Version 4.0.* Intelligent Systems Laboratory, Palo Alto Research Center, Xerox Corporation, 1986. Reprinted and available from Carnegie Mellon University.

24. Laird, J. Final Discussion from the the Soar V Workshop. Computer Science Department, Carnegie Mellon University, September, 1988.

25. Laird, J. E., Newell, A., and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33*, 1 (1987), 1-64.

26. Laird, J.E., Yager, E.S., Tuck, C.M., Hucka, M. Learning in tele-autonomous systems using Soar. To appear in the proceedings of the NASA Conference on Space Telerobics, January 30-February 2, 1989.

27. Mattern, F. "Algorithms for distributed termination detection". *Journal of Distributed Computing 2*, 2 (1987), 161-175.

28. McDermott, J. "R1: A rule-based configurer of computer systems". *Artificial Intelligence 19*, 2 (1982), 39-88.

29. McKeown, D.M., Harvey, W.A. and McDermott, J. "Rule Based Interpretation of Aerial Imagery". *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-7*, 5 (September 1985), 570-585.

30. Miranker, D. P. Treat: A better match algorithm for AI production systems. Proceedings of the National Conference on Artificial Intelligence, August, 1987, pp. 42-47.

31. Newell, A. Unified Theories of Cognition. The William James Lectures. Harvard University. Available in video cassette from Harvard Psychology Department.

32. Oflazer, K. *Partitioning in Parallel Processing of Production Systems.* Ph.D. Th., Computer Science Department, Carnegie Mellon University, March 1987.

33. Schreiner, F. and Zimmerman, G. Pesa-1: A parallel architecture for production systems. Proceedings of the International Conference on Parallel Processing, August, 1987, pp. 166-169.

34. Sietz, C. L. "The cosmic cube". *Communications of the ACM 28*, 1 (1984), 22-28.

35. Stolfo, S. "Initial performance of the DAD02 prototype". *IEEE Computer 20*, 1 (1987), 75-83.

36. Tambe, M., Kalp, D., Gupta, A., Forgy, C.L., Milnes, B.G., and Newell, A. Soar/PSM-E: Investigating match parallelism in a learning production system. Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems, July, 1988, pp. 146-160.