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.

Parallelism in Production Systems: The Sources and the Expected Speed-up

Anoop Gupta
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

December 1984

Abstract

Production systems (or rule-based systems) are widely used in Artificial Intelligence for modeling intelligent behavior and building expert systems. On the surface production systems appear to be capable of using large amounts of parallelism—it is possible to perform match for each production in parallel. Initial measurements and simulations, however, show that the speed-up available from such use of parallelism is quite small. The limited speed-up available from the obvious sources has led us to explore other sources of parallelism. This paper represents an initial attempt to identify the various sources of parallelism in production system programs and to characterize them, that is, to determine the potential speed-up offered by each source and the overheads associated with it. The paper also addresses some implementation issues related to using the various sources of parallelism.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. 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 Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction
2. Production Systems
    2.1. OPS5
    2.2. SOAR
3. The Rete Algorithm
4. Parallelism in Production Systems
    4.1. Parallelism in Match
        4.1.1. Production-level Parallelism
        4.1.2. Node-level Parallelism
        4.1.3. Action Parallelism
    4.2. Parallelism in the Conflict-Resolution Phase
    4.3. Parallelism in the Act Phase
    4.4. Parallelism from the Run-time Addition of Productions
    4.5. Application Parallelism in Production Systems
5. Detailed Results of Simulations
    5.1. The Simulation Model
    5.2. Production Systems Measured
    5.3. Production-level Parallelism
    5.4. Node-level Parallelism
    5.5. Action Parallelism
    5.6. Bottlenecks in Obtaining Speed-up from Parallelism
    5.7. Comparison of the Sources of Parallelism
    5.8. Limitations of the Simulation Results
6. Summary
7. Acknowledgments
References

1. Introduction

Production systems form an important part of the basic research and applied research going on in Artificial Intelligence. As a part of basic research they are being used in the study of learning systems and problem-solving systems [13, 14, 21, 27]. As a part of applied research they are being used to develop expert systems spanning a large variety of applications in several areas including medicine, computer-aided design, and oil exploration [1, 2, 11, 12, 17, 23]. Production system programs, however, are very computation intensive and run quite slowly. For example, production system programs written in the OPS5 language [3] run at a speed of only 1-10 production firings per second on a VAX-11/780. Although sufficient for many interesting applications, this slow speed of execution precludes the use of production systems in many domains requiring high performance and real-time response. The limited performance also impacts the research that is done with production systems, since researchers naturally avoid programming styles and applications which run too slowly. For the above reasons a significant increase in the execution speed of production systems would be very valuable to both the practitioners and the researchers.

There are several different methods for speeding up the execution of production systems: (1) the use of faster technology, (2) the use of better algorithms and architecture, and (3) the use of parallelism. This paper focuses on the third of the above three methods. It identifies the various sources of parallelism in production systems and discusses the feasibility of exploiting them. The paper concentrates on the parallelism available in OPS5 [3] and SOAR [15] production systems. OPS5 was chosen because several large, diverse, and real production system programs have been written in it. These programs form an excellent base for measurements and analysis. SOAR was chosen because it represents an interesting new approach in the use of production systems for problem solving. SOAR programs are also capable of improving their performance by learning new productions at run-time. Since only OPS5 and SOAR programs are considered, the analysis of parallelism presented in this paper is biased by the characteristics of these languages and may not be safely generalized to programs written in languages with substantially different characteristics, for example, to programs written in EMYCIN [19], EXPERT [28], or KAS [2]. Furthermore, the analysis is based on programming styles currently prevalent, and as the styles evolve over time the results may have to be updated.

The paper consists of the following sections. Section 2 briefly describes the OPS5 and SOAR production systems. Section 3 gives a detailed description of the Rete algorithm, which forms the basis for much of the analysis presented later in the paper. Section 4 analyzes the sources of parallelism in production systems. For each source it provides an estimate of the expected speed-up and the associated overheads. Section 5 presents the simulation model and the detailed results of simulations that were done to evaluate parallelism in production systems (some of these results are also presented in Section 4). It also presents an analysis of the bottlenecks encountered in exploiting parallelism and the limitations of the simulations results. Section 6

presents a summary of the results.

2. Production Systems

This section describes the OPS5 and SOAR production systems. The basic structure of OPS5 programs is introduced first and the distinguishing features of SOAR [13] are described next. The description of SOAR focuses on those features that affect the amount of parallelism available in its programs.

2.1. OPS5

A production system is composed of a set of *if-then* rules called *productions* that make up the *production memory* and a database of assertions called the *working memory*. The assertions in the working memory are called *working memory elements*. Each production consists of a conjunction of *condition elements* corresponding to the *if* part of the rule (also called the *left-hand side* of the production) and a set of *actions* corresponding to the *then* part of the rule (also called the *right-hand side* of the production). The actions associated with a production can add, remove, or modify working memory elements, or perform input-output. Figure 2-1 shows an OPS5 production named p1, which has three condition elements in its left-hand side and one action in its right-hand side.

```
(p  p1  (C1  ↑attr1 <x>     ↑attr2 12)
        (C2  ↑attr1 15      ↑attr2 <x>)
        (C3  ↑attr1 <x>)
   -->
        (remove 2))
```

Figure 2-1: A Sample Production

The production system *interpreter* is the underlying mechanism that determines the set of satisfied productions and controls the execution of the production system program. The interpreter executes a production system program by performing the following cycle of operations:

- Match: In this first phase, the left-hand sides of all productions are matched against the contents of the working memory. As a result we obtain a *conflict set*, which consists of instantiations of all satisfied productions.

- Conflict Resolution: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.

- Act: In this third phase, the actions of the production selected in the conflict resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, the first phase is executed again.

A working memory element in OPS5 is a parenthesized list consisting of one or more *attribute-value* pairs. The attributes are symbols that are preceded by ↑. The values are symbolic or numeric constants. The

condition elements in the left-hand side of a production are parenthesized lists similar to working memory elements. However, the condition elements are less restricted than the working memory elements; while a working memory element can contain only constant symbols and numbers, a condition element can contain variables[1], predicate symbols, and a variety of other operators as well as constants.

A working memory element matches a condition element if the object types of the two match and if the value of every attribute in the condition element matches the value of the corresponding attribute in the working memory element. If the condition element value is a variable, it will match any value in the working memory element. However, if a variable occurs more than once in a left-hand side, all occurrences of the variable must match identical values. Thus the working memory element

```
(C1      ↑attr1  12      ↑attr2  15)
```

will match the condition element CE1, but it will not match the condition element CE2 below.

```
CE1:  (C1      ↑attr1  12      ↑attr2  <x>)
CE2:  (C1      ↑attr1  <x>     ↑attr2  <x>)
```

A production is said to be *satisfied* when, for every condition element in the left-hand side of the production, there exists a working memory element that matches it.[2]

The right-hand side of a production consists of an unconditional sequence of actions which can cause input-output, and which are responsible for changes to the working memory. Three kinds of actions are provided to effect working memory changes. *Make* creates a new working memory element and adds it to the working memory. *Modify* changes one or more values of an existing working memory element. *Remove* deletes an element from the working memory.

2.2. SOAR

SOAR [13, 16] is a new production system architecture developed at Carnegie-Mellon University to perform research in problem solving and learning. The current version of SOAR is realized as a modified OPS5 system. Thus, except for the features mentioned below, programs in SOAR behave like other OPS5 programs: (1) SOAR production system programs can improve their performance over time by learning new productions at run-time. This feature of SOAR poses interesting problems for the implementor, since it is necessary that the new productions be incorporated into existing data structures at run-time. (2) While OPS5 programs are restricted to firing only one production on each execution cycle, SOAR allows for multiple firings on each execution cycle. Multiple firings result in an increased number of working memory changes

---

[1] A variable is an identifier that begins with the character "<" and ends with ">"—for example, <x> and <status> are variables.

[2] In addition to regular condition elements, a production may also contain *negated condition elements*, which are satisfied only when there exists no working memory element that matches them.

on each cycle, which in turn influences the amount of parallelism that is available to implement SOAR production systems. (3) SOAR programs have the unifying feature that they all function by performing a heuristic search in a problem space. It is possible to exploit this uniform structure of programs to obtain increased parallelism; it is possible to evaluate many paths in the search space in parallel.

3. The Rete Algorithm

The most time consuming step in the execution of OPS-like production systems is the match step. Even with specialized algorithms, it constitutes around 90% of the interpretation time. The match algorithm used by uniprocessor implementations of OPS5 is called Rete [4]. The Rete algorithm is described below in some detail, as it forms the basis for much of the analysis presented later in the paper.

The Rete algorithm exploits (1) the fact that only a small fraction of working memory changes every cycle by storing results of match from previous cycles and using them in subsequent cycles, and (2) the similarity between condition elements of productions by performing common tests only once. These two features combined together make Rete a very efficient algorithm for match.

To perform match, the Rete algorithm uses an augmented discrimination network constructed from the left-hand sides of productions. Figure 3-1 shows such a network for productions p1 and p2 which appear in the top part of the figure. The nodes in the network represent abstract operations to be performed during match and are interpreted at run-time by the OPS5 interpreter. The objects that are passed between nodes in the network are called *tokens*. Each token consists of a pointer to a list of working memory elements that matches a subsequence of condition elements in a left-hand side. The network consists of four different kinds of nodes.[3] These are:

1. **Constant-test nodes:** These nodes are used to test if the attributes in the condition element which have a constant value are satisfied. These nodes appear in the top part of the Rete network. Note that when two left-hand sides require identical nodes, the compiler shares part of the network rather than building duplicate nodes.

2. **Memory nodes:** These nodes store the result of matching from previous cycles as state within them. The state stored in a memory node consists of a list of the tokens that match a part of the left-hand side of the associated production. For example, the right-most memory node in Figure 3-1 stores all tokens corresponding to working memory elements with "Class = C4".

3. **Two-input nodes:** These nodes test for joint satisfaction of condition elements in the left-hand side of a production. Both inputs of a two-input node come from memory nodes. When a token arrives on the left input of a two-input node, it is compared to each token stored in the memory node connected to the right input. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action is taken when a token arrives on the right

[3] For reasons of brevity, some nodes that occur only rarely have been ignored.

input of a two-input node.

4. **Terminal nodes:** There is one such node associated with each production in the program, as can be seen at bottom of Figure 3-1. Whenever a token flows into a terminal node, the corresponding production is either inserted into or deleted from the conflict set.

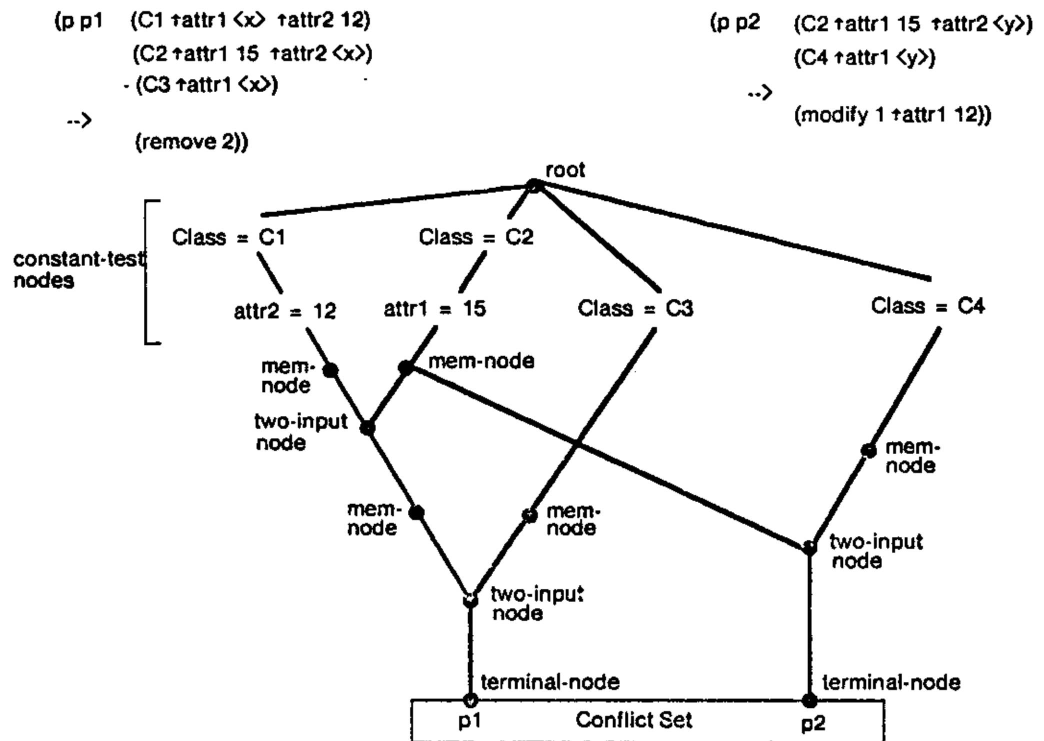

Figure 3-1: The Rete Network

From a global viewpoint, the input to the Rete network consists of changes to working memory. The changes filter through the network, updating the state stored within the network. The output of the network consists of changes to the conflict set. The OPS5 interpreter executes production systems at a rate of about 10 production firings per second.

In OPS5 [3] a significant loss in the speed is due to the interpretation overhead of nodes. In OPS83 [5] this overhead has been eliminated by compiling the Rete network directly into the machine code for the VAX-11/780. While it is possible to escape to the interpreter for complex operations during match or for setting up the initial conditions for the match, the majority of the match is done without an intervening interpretation level. This has led to a large speed-up and the OPS83 interpreter executes about 50 production firings per second, which is 5 times faster than OPS5. The results of simulations reported in the subsequent sections of this paper are based on the computational requirements of the OPS83 interpreter.

4. Parallelism in Production Systems

As described in Section 2.1, there are three steps that are repeatedly performed to execute a production system program: the match, the conflict-resolution, and the act. Figure 4-1 shows the flow of information between the three stages of the interpreter. It is possible to use parallelism while performing each of these three steps. It is further possible to overlap the processing in these three stages to achieve more speed-up. The run-time addition of productions described in Section 2.2 affords still more parallelism, in that it is possible to perform the task of compiling and updating the state of newly added productions in parallel with the execution of the rest of the production system.

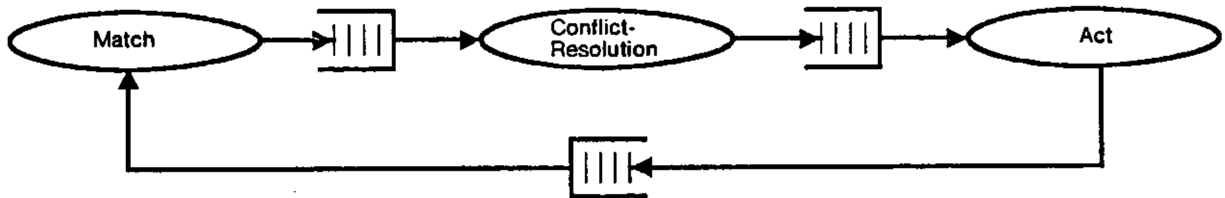

Figure 4-1: The Basic Interpreter Cycle

When studying the parallel implementation of an algorithm, it is necessary to address the following two issues: (1) The decomposition of the algorithm into a set of parallel processes along with their communication graph. Usually, the granularity (the size) of the processes can be traded off against the communication required between the processes. (2) The mapping of the suggested decomposition onto a given hardware/software architecture. The following discussion on parallelism addresses both these aspects. It also presents estimates of the speed-up expected from the various sources of parallelism and the overheads associated with exploiting those sources.

Note that the data presented in the following sections is based on the results of measurements and simulations performed on a number of production system programs. The results of measurements are presented in [7] and the results of simulations are presented in detail in Section 5 (only a summary is presented here). The set of production system programs that was used in the analysis is given in Section 5.2.

4.1. Parallelism in Match

Current OPS interpreters spend almost 90% of their time in the match phase, and only 10% of their time in the conflict-resolution and the act phases. For this reason it is most important to be able to speed up the match phase as much as possible. The following discussion presents three important ways in which

parallelism may be used to speed up match.[4]

4.1.1. Production-level Parallelism

To use production-level parallelism, the productions in a program are divided into several partitions and the match for each of the partitions is performed in parallel. In the extreme case, the number of partitions equals the number of productions in the program, so that the match for each production in the program is performed in parallel. A graphical representation of use of production-level parallelism is shown in Figure 4-2. The production system program is shown to be divided into *N* partitions, named *P*1, . . . ,*PN*. A major advantage of using production-level parallelism is that no communication is required between the processes performing the match.

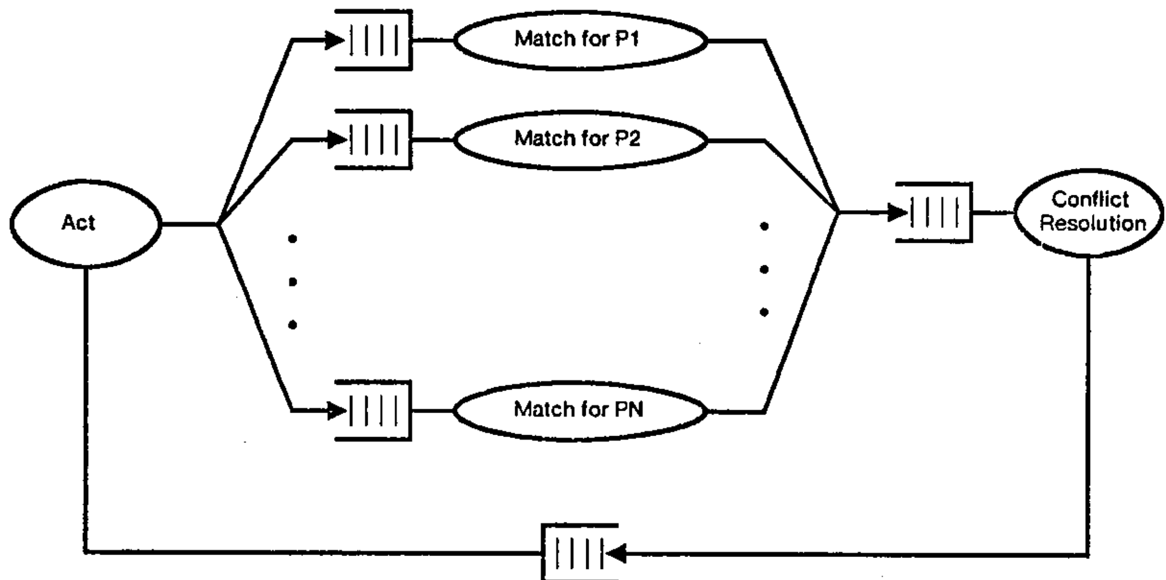

Figure 4-2: Production-level Parallelism in Production Systems

A natural question to ask of any parallelism scheme is: "How much speed-up can we expect from such a scheme?" For example, if a production system is divided into 1000 partitions, there will be 1000 processes performing the match in parallel. Do we expect a 1000-fold reduction in the total time required for match? The answer to the previous question is "no," at least for all the OPS5 and SOAR production system programs we have studied so far.

Consider the match for a production following a change to the working memory. The production is said to

---

[4] A discussion on parallelism in match can also be found in our paper [6]. The discussion presented in the following paragraphs is more detailed than that in our earlier paper and uses slightly different terminology. The following paragraphs also lay the groundwork for the material presented in Section 5 of this paper.

be *affected* by the change, if the new working memory element satisfies at least one of its condition elements. The significance being, that a production which is not affected requires negligible processing compared to a production which is affected. It is now possible to state the reasons why we do not expect a 1000-fold reduction in the time for match:

- Measurements [7] show that in both OPS5 and SOAR production system programs the average size of the *affect-set*[5] is quite small, about 32 productions. Furthermore, measurements indicate that the average size of the affect-set is quite independent of the number of productions in the program. Since most of the match time is taken by the productions in the affect-set, the maximum speed-up that can be expected from production-level parallelism is about 32. This implies that if there is a separate processor performing match for each production in the program, only 32 processors will be performing useful work and the rest will have no work to do. However, as stated below, there are other reasons which make the speed-up even smaller.

- The second factor that affects the speed-up obtainable from production-level parallelism is the variance in the processing time required by the affected productions. The speed-up that can be obtained is proportional to the ratio t_{avg}/t_{max}, where t_{avg} is the average processing time taken by an affected production to finish match and t_{max} is the maximum time taken by any affected production to finish match. The parallelism is inversely proportional to t_{max} because the next production execution cycle cannot begin until all productions have finished match. Taking this factor into account, recent simulations for OPS5 and SOAR production system programs indicate that the maximum speed-up that can be obtained from production-level parallelism is around 6, a factor of 5 less than the average size of the affect-sets.

- The third factor that influences the speed-up is the loss of sharing in the Rete network when production-level parallelism is used.[6] The loss of sharing happens because tests which would have been performed only once for similar productions on a uniprocessor are now performed independently for such productions. Recent measurements show that the processing cost increases by a factor of around 1.4 due to loss of sharing.

Some implementation issues associated with using production-level parallelism are now discussed. The first point that emerges from the previous discussion is that it is not advisable to allocate one processor per production for performing match. If this is done most of the processors will be idle most of the time and the hardware utilization will be poor [6, 8].[7] If only a small number of processors are to be used, there are two important alternative strategies. The first strategy is to divide the production system program into several partitions so that the processing required by productions in each partition is almost the same, and then allocate one processor for each partition. The second strategy is to have a task queue shared by all processors

---

[5] The set of productions affected by a change to the working memory.

[6] Recall from Section 3 that the Rete algorithm saves processing time on uniprocessors by performing operations common to several productions only once.

[7] Low utilization is not justifiable here, even if hardware is inexpensive, for it indicates that some alternative design can be found that can attain more performance at the same cost or the same performance at less cost.

in which entries for all productions requiring processing are placed. Whenever a processor finishes processing one production, it gets the next production that needs processing from the task queue. Some advantages and disadvantages of the two strategies are given below.

The first strategy is suitable for both shared-memory multiprocessors and non-shared memory multicomputers. It is possible for each processor to work from its local memory and little or no communication between processors is required. The main difficulty, however, is to find partitions of the production system that require the same amount of processing. The task of partitioning is difficult because good models are not available for estimating the processing required by any given production. Furthermore, the processing required by a production varies over time, which makes the partitioning task even more difficult [20].

The second strategy is suitable only for shared memory architectures, because it requires that each processor have access to the code and state of all productions in the program.[8] This strategy has the advantage that no load-distribution problems are present because the tasks are allocated dynamically to the processors. Another advantage of this strategy is that it extends very well to lower granularities of parallelism. However, this strategy encounters loss of performance due to synchronization, scheduling, and memory contention overheads.

In conclusion, the maximum speed-up that can be obtained from production-level parallelism is equal to the average number of productions affected per change to working memory. However, in practice, the speed-up is expected to be much less. This is due to (1) the variance in the processing time required by affected productions, (2) the loss of sharing in parallel decompositions, and (3) the overheads of mapping the decompositions onto hardware architectures. Measurements [7] show that average size of the affect-set for production system programs is around 32. Furthermore, the size of the affect-set appears to be independent of the total number of productions present in the program, which means that bigger production systems do not have more inherent parallelism. Deducting for the various factors reducing the maximum speed-up, recent simulations predict that the actual speed-up will be around 6. The detailed results of the simulations can be found in Section 5.

[8] While it is possible to replicate the code (the Rete network) in the local memories of all the processors, it is not possible to do so for the dynamically changing data.

4.1.2. Node-level Parallelism

When node-level parallelism is used, activations of different two-input nodes in the Rete network of a production system are evaluated in parallel.[9] It is important to note that node-level parallelism subsumes production level parallelism, in that node-level parallelism has a finer grain than production-level parallelism. Thus, using node-level parallelism, both activations of two-input nodes belonging to different productions (corresponding to production-level parallelism), and activations of two-input nodes belonging to the same production (resulting in the extra parallelism) are processed in parallel.

The main reason for going to this finer granularity of parallelism is to reduce t_{max}, the maximum time taken by any affected production to finish match. The desirability of reducing this value was shown in the previous section. This decreased granularity of parallelism, however, leads to increased communication requirements between processes evaluating the nodes in parallel. In node-level parallelism a process must communicate the results of a successful match to its successor two-input nodes. No communication is necessary if the match fails. To evaluate the usefulness of exploiting node-level parallelism it is necessary to weigh the advantages of reducing t_{max} against the cost of increased communication and the associated limitations on feasible architectures.

The extra speed-up available from node-level parallelism over that obtained from production-level parallelism is bounded by the number of two-input nodes present in a production. (Note that the number of two-input nodes in a production is one less than the number of condition elements present in that production.) The reason for this is that the extra parallelism comes only from the parallel evaluation of nodes belonging to the same production. Since the average number of condition elements in production systems is around 4, the maximum extra speed-up expected from node-level parallelism is around 3.[10] The results of simulations for OPS5 and SOAR programs indicate that using node-level parallelism it is possible to get speed-ups of about 8, which is 1.3 times more than the speed-up that could be obtained if production-level parallelism alone was used.

The implementation considerations for node-level parallelism are very similar to those for production-level parallelism described in Section 4.1.1. However, since the communication required between the parallel processes is more, shared-memory architectures are preferable. The size of the tasks when node-level parallelism is used is smaller than when production-level parallelism is used. Simulations indicate that the

---

[9] An activation of a two-input node corresponds to the processing required when a token flows into the left or right input of a two-input node (see Section 3 for details).

[10] This statement is not completely true. This is because of the t_{max}/t_{avg} factor described in the previous section. It is possible in some cases for the extra speed-up to be close to the number of two-input nodes in the largest production of the production system.

average task length is around 50-100 computer instructions. This number is significant in that it indicates the amount of synchronization and scheduling overhead that may be tolerated in a shared-memory implementation.

4.1.3. Action Parallelism

Usually, when a production fires, it makes several changes to the working memory. Measurements show that the average number of changes made to the working memory per execution cycle is 5.3. Processing these changes in parallel, instead of sequentially, can lead to increased speed-up from both production-level and node-level parallelism.

The reasons for the increased speed-up from production-level parallelism when used with action parallelism are the following. In Section 4.1.1, it was shown that the speed-up available from production-level parallelism is proportional to the average number of affected productions. The set of productions which is affected as a result of processing many changes simultaneously is the union of the affect-sets of the individual changes to the working memory. Since this combined affect-set is larger than the individual affect-sets, more speed-up can be obtained. For example, consider the case where a production firing results in two changes to working memory, such that change-1 affects productions p1, p2, and p3, and change-2 affects productions p4, p5, and p6. If change-1 and change-2 are to be processed sequentially, it is best to use three processors. Each change takes one cycle and the total cost is two cycles.[11] However, if change-1 and change-2 are processed concurrently, they can be processed in one instead of two cycles using six processors. Simulations indicate that the use of action parallelism increases the speed-up obtainable from production-level parallelism alone by a factor of about 1.3. The extra speed-up is less than the average number of working memory changes per cycle, because the affect-sets of the changes are not distinct but have considerable overlap.

Analysis of production-system programs shows that often two successive changes to working memory affect two distinct condition elements of the same production, as a result causing two distinct two-input node activations. It is then possible, using node-level parallelism, to process these node activations in parallel, thus increasing the available parallelism. For example, consider the case where both change-1 and change-2 affect productions p1, p2, and p3. If the activations correspond to distinct two-input nodes, it is possible to process both the changes in parallel, in one instead of two cycles. Simulations indicate that the use of action parallelism increases the speed-up obtainable from node-level parallelism alone by a factor of around 1.7.

So far the cost of computing the affect-set for a change to the working memory has not been discussed. When many changes are to be processed, it is possible to compute the affect-set for one change, while

[11] Assuming that each affected production takes the same amount of processing time.

simultaneously processing productions affected by the previous change. Alternatively, the affect-sets of all changes to working memory can be computed simultaneously. This overlapped computation is another source of parallelism when processing many changes in parallel.

4.2. Parallelism in the Conflict-Resolution Phase

Measurements [7] show that on each cycle of production system execution around 5 changes are made to the conflict-set. Thus at best it would be possible to achieve a 5-fold speed-up in conflict-resolution from parallelism. This speed-up appears to be sufficient, in the sense that conflict-resolution is not expected to become a bottleneck in the near future. The reasons are:

- In current production-system interpreters conflict-resolution takes about 5% of the execution time. Speeding it up by a factor of 5 implies that only 1% of the time will be devoted to it. This will then not be a bottleneck until we speed up match by about 50-fold, and parallelism does not seem to provide that much speed-up. Note that any speed-up in match due to better or faster processors applies uniformly to match and conflict-resolution and thus does not change the argument.

- In both production-level and node-level parallelism discussed earlier, the match for the affected productions finishes at different times because of the variance in the processing required by the affected productions. Thus many changes to the conflict set are available to the conflict-resolution process, while some productions are still performing match. Thus much of the conflict-resolution time can be overlapped with the match time, reducing the chances of conflict-resolution becoming a bottleneck.

- If the conflict-resolution ever becomes a bottleneck, there are simple strategies for avoiding it. For example, to begin the next execution cycle, it is not necessary to perform conflict-resolution for the current changes to completion. It is only necessary to compare each current change to the highest priority production instantiation so far. Once the highest priority instantiation is selected the next execution cycle can begin. The complete sorting of the production instantiations can be overlapped with the match phase for the next cycle. Hardware priority queues provide another strategy.

4.3. Parallelism in the Act Phase

The act step like the conflict-resolution step only takes about 5% of the total time for the current production systems. When many productions are allowed to fire in parallel, as in SOAR, it is quite straight forward to execute them in parallel.[12] Even when the right-hand side of only a single production is to be evaluated, it is possible to overlap some of the input/output with the match for the next execution cycle. For the above reasons the act step is not expected to be a bottleneck in speeding up the execution of production systems.

---

[12] This assumes that the execution of one right-hand side does not affect the result of executing another right-hand side, which is true of SOAR.

4.4. Parallelism from the Run-time Addition of Productions

Section 2.2 described the SOAR production-system architecture in which productions are added at run-time. Addition of productions at run-time poses two new computational requirements: (1) the integration of the new productions into the existing data structures for performing match, and (2) updating the state[13] associated with the new productions with respect to some subset of the contents of the working memory. Note that this subset of working memory can be significantly larger than the average number of changes made to the working memory per execution cycle.

An important characteristic of the newly added productions in SOAR programs is that the new productions only enhance the performance. The final outcome of a program is not changed even if these productions are not incorporated into the program.[14] However, the inclusion of the new productions greatly reduces the number of execution cycles necessary to compute the final result. This feature of SOAR programs permits the update of the data structures and the state associated with new productions to be extended over several match-execute cycles and can be done in parallel with the execution of the rest of the production system. Currently, no data is available on the extra speed-up that is obtainable from this source of parallelism.

4.5. Application Parallelism in Production Systems

Lastly, there is substantial speed-up to be gained from application parallelism, where a number of cooperating but loosely coupled production system tasks execute in parallel. The cooperating tasks could arise in the context of search, where there are a number of paths to be explored and it is possible to explore each of the paths in parallel (similar to OR parallelism in logic programs [26]). The cooperating tasks could also arise in the context where there are a number of semi-independent tasks, all of which have to be performed, and they can be performed in parallel (similar to AND parallelism in logic programs). The maximum speed-up that can be obtained from application parallelism is equal to the number of cooperating tasks, which can be significant. Unfortunately, most current production systems do not exploit such parallelism, because (1) the production system programs were expected to run on a uniprocessor, where no advantage is to be had from having several parallel tasks, and (2) current production system languages do not provide the features to write multiple cooperating production tasks easily. Although not exploited currently, the SOAR production system architecture provides a uniform problem-solving framework[15] that makes it easy to exploit OR parallelism.

---

[13] Recall that the Rete algorithm stored the result of match from previous cycles as state with the productions to avoid performing match with the same working memory elements over and over.

[14] This is because SOAR systems can fall back on more basic problem-solving mechanisms when specialized knowledge is not available.

[15] All problem solving in SOAR systems is done as heuristic search within a problem space.

5. Detailed Results of Simulations

This section describes the detailed results of simulations performed to determine the speed-up available from parallelism in the match step. It describes the simulation model, the programs that were measured, the results of the simulations, and the limitations of the simulation results. This section may be skipped by those readers who are not interested in these details. Many of the important results have already been summarized in the previous section.

5.1. The Simulation Model

Previous analysis of parallelism in production systems [7, 8] was done using very simple models. Only production-level parallelism was explored and even there the variation in the cost of processing the production activations was not taken into account. The measurements were still important because they provided some robust upper-bounds on the speed-up that could be obtained using production-level parallelism. To explore the parallelism in more detail the current simulator was constructed. The goals were the following: (1) To determine the amount of speed-up that could be achieved from each source of parallelism individually, so that it is possible to trade off the extra speed-up from a source with the overheads of using that source. (2) To study the bottlenecks in obtaining speed up from parallelism. Once the bottlenecks are understood, it should then be possible to devise means to eliminate them. (3) To study the effect of different cost models for node activations on the amount of speed-up that could be obtained from parallelism. (4) To study the effect of architecture (shared memory versus non-shared memory) on the speed-up. The conflict-resolution and the act steps were not considered, since they are not expected to contribute significantly to the overall speed-up and because exploiting parallelism in those steps does not appear to be as complex.

The simulator that has been constructed for determining the parallelism in production systems is event-driven. The input to the simulator consists of: (1) a detailed trace of node activations in the Rete network corresponding to a production-system run; (2) a cost model that can be used to determine the cost of any given node activation; and (3) a specification of the parallel computational model on which the trace is to be executed. The output of the simulator consists of statistics for the overall run and the individual cycles in the run.

Figure 5-1 shows a small fragment of a trace that is fed to the simulator. The trace contains information about the dependencies between the node activations, and the simulator understands which node activations can be processed in parallel and which cannot be processed in parallel. The trace also contains other information which is necessary to determine the cost of a given node activation.

The simulator uses a cost model to determine the processing cost of the node activations found in the trace.

```
(wme-change)
((prev 5) (cur 10007) (type and) (node-id 6) (prods (p1)) (side right) (flag new) (numl 2) (numr 2) (lev 2) (ntests 2) (tests (teqb 10002 1 teqb 1 2)) (nsent 1))
((prev 10007) (cur 10008) (type p) (node-id 7) (prods (p1)) (flag new) (lev 3))

(wme-change)
((prev 6) (cur 10009) (type and) (node-id 6) (prods (p1)) (side right) (flag new) (numl 2) (numr 3) (lev 2) (ntests 2) (tests (teqb 10002 1 teqb 1 2)) (nsent 1))
((prev 10009) (cur 10010) (type p) (node-id 7) (prods (p1)) (flag new) (lev 3))

(wme-change)
((prev 7) (cur 10011) (type and) (node-id 3) (prods (p1)) (side right) (flag new) (numl 2) (numr 2) (lev 1) (ntests 0) (tests nil) (nsent 2))
((prev 10011) (cur 10012) (type and) (node-id 6) (prods (p1)) (side left) (flag new) (numl 2) (numr 3) (lev 2) (ntests 2) (tests (teqb 10002 1 teqb 1 2)) (nsent 1))
((prev 10011) (cur 10014) (type and) (node-id 6) (prods (p1)) (side left) (flag new) (numl 3) (numr 3) (lev 2) (ntests 2) (tests (teqb 10002 1 teqb 1 2)) (nsent 1))
((prev 10012) (cur 10013) (type p) (node-id 7) (prods (p1)) (flag new) (lev 3))
((prev 10014) (cur 10015) (type p) (node-id 7) (prods (p1)) (flag new) (lev 3))
```

Figure 5-1: A Sample Trace Fragment

The cost depends on the type of the node activation, the amount of state associated with the node, whether the state is stored as a linear list or in a hash table, the number of tests that have to be performed to check if two tokens are consistent, etc. For the majority of the simulations the cost model used is based on measurements made on the OPS83 interpreter.[16] The model differs from the OPS83 interpreter in that the interpreter uses linear memories to store the tokens associated with a node, while the model assumes that the tokens are stored in a hash table.

The computation model input to the simulator specifies how the trace is to be executed. It specifies:

- The kinds of parallelism that may be used while executing the trace (some combination of production-level, node-level, and action parallelism). For example, when only production-level parallelism is allowed, the simulator does not allow activations of nodes belonging to the same production to be evaluated in parallel. It further disallows sharing of Rete nodes between different productions.

- Whether the processors have shared memory or not. The main implication of not having shared memory is that productions must be statically assigned (at the beginning of the run) to specific processors, and node activations corresponding to a given production can only be processed on the associated processor. The restriction of processing a node activation on a specific processor exists because the state associated with the given node is present in the local memory of a specific processor and its communication to another processor is very expensive. This restriction is not present in shared-memory architectures, where a node activation can be processed on any available free processor. Presence of shared memory, however, entails overheads associated with memory contention and synchronization. The affects of memory contention and synchronization are not modeled in the current simulator. This results in better performance for shared memory architectures than will be actually observed. Despite this limitation, the simulation results have brought to light a number of facts about the parallel execution of production systems.

---

[16] The OPS83 interpreter was chosen as the basis for the cost model because it represents a state of the art software implementation of the Rete algorithm. The measurements on the OPS83 interpreter were made by Charles Forgy.

- The number of processors that are available. This obviously determines the maximum number of node activations that can be evaluated in parallel.
- Whether scheduler optimizations are present, that is, if it is allowed to reorder the evaluation of node activations to optimize overall performance.

The statistics output by the simulator consist of both per-cycle information and overall-run information. The statistics that are output for each match-execute cycle of the production system are:

- $S_{max-i} = k_i \cdot t_{avg-i} / t_{max-i}$, where S_{max-i} is the maximum speed-up that can be achieved in the ith cycle irrespective of the number of processors used, k_i is the number of tasks[17] in the ith cycle, t_{avg-i} is the average cost for tasks in the ith cycle, and t_{max-i} is the maximum cost of any task in the ith cycle as determined by the simulator. Note that $k_i \cdot t_{avg-i}$ represents the cost of executing the ith cycle on a uniprocessor.
- $S_{act-i} = k_i \cdot t_{avg-i} / t_{cyc-i}$, where S_{act-i} is the actual speed-up that is achieved in the ith cycle using the number of processors specified in the computational model, and t_{cyc-i} is the cost of the ith cycle as computed by the simulator. Note that it follows from the definition of t_{max-i} and t_{cyc-i} that $t_{cyc-i} \geq t_{max-i}$.
- $PU_i = S_{act-i} / NumProcessors$, where PU_i is the processor utilization in the ith cycle and $NumProcessors$ is the number of processors specified in the computation model.

The same set of statistics can also be computed at the level of the complete run. The overall statistics are:

- $S_{max} = \sum_{i=1}^{N} k_i \cdot t_{avg-i} / \sum_{i=1}^{N} t_{max-i}$, where S_{max} is the maximum speed-up that can be achieved over the complete program run irrespective of the number of processors used.
- $S_{act} = \sum_{i=1}^{N} k_i \cdot t_{avg-i} / \sum_{i=1}^{N} t_{cyc-i}$, where S_{act} is the actual speed-up over the complete run using the number of processors specified in the computation model.
- $PU = S_{act} / NumProcessors$, where PU is the processor utilization over the complete run.

The following sections mainly refer to the overall statistics. The following equations show the relationship between the overall statistics and the per-cycle statistics:

$$S_{max} = \sum_{i=1}^{N} S_{max-i} \cdot (t_{max-i} / \textstyle\sum_{j=1}^{N} t_{max-j}).$$

$$S_{act} = \sum_{i=1}^{N} S_{act-i} \cdot (t_{cyc-i} / \textstyle\sum_{j=1}^{N} t_{cyc-j}).$$

The above equations state that the overall speed-up is not a simple average of the per-cycle speed-ups but a

[17] A task here corresponds to an independently schedulable piece of work that can be executed in parallel. Thus when using production-level parallelism, a task corresponds to all node activations belonging to a production. When using node-level parallelism a task becomes more complex, corresponding approximately to a sequence of dependent-node activations, i.e., a set of node activations no two of which could have been processed in parallel.

weighted average of the per-cycle speed-ups. The weight for the ith cycle is $t_{max-i}/\sum t_{max}$ in the first equation and $t_{cyc-i}/\sum t_{cyc}$ in the second equation. Thus the per-cycle statistic is weighted by its fraction of the total cost in the parallel implementation (not the total cost in the uniprocessor implementation). As a result, a few long cycles with low speed-ups can destroy the overall speed-up for a run.

5.2. Production Systems Measured

Traces from eight different production system programs were used to analyze the parallelism in the match phase. These production systems are:[18]

- XSEL [18], a program acting as a sales assistant for VAX computer systems. It is written in OPS5 and consists of 1443 productions. For the XSEL system two traces have been included in the analysis. These are referred to as xsel-tr1 and xsel-tr2 in the subsequent sections. The second trace of XSEL is included because it involves interaction with an external database (working memory changes are fetched from outside), and such working memory changes were found to have much larger affect-sets than the normal changes.

- PTRANS [9], a program for factory management. It is written in OPS5 and consists of 1016 productions.

- MUD [10], an OPS5 program which does analysis of mud used in oil drilling. It consists of 872 productions.

- DAA [12], an OPS5 program which automatically designs computers from a high level specification of the system. It consists of 314 productions.

- R1-SOAR [22], a program for configuring VAX computer systems. It is written in the SOAR language and implements only a small part of the functionality of the corresponding OPS5 program. It consists of 235 productions.

- R1LRN-SOAR, is the same as R1-SOAR except that it consists of nine new productions automatically learned by the program. It consists of 244 productions.

- EIGHT-SOAR [16], a SOAR program which solves the eight puzzle. It consists of 108 productions.

- ELRN-SOAR, is the same as EIGHT-SOAR except that it includes fifteen new learned productions. It consists of 123 productions.

The above programs represent a variety of applications and programming styles. For example, XSEL and PTRANS programs are knowledge-intensive expert systems and are currently being used in the industry. Both are forward-chaining systems and are primarily data driven. The MUD system is a backward-chaining

---

[18] Note many of the production systems mentioned below are still growing. The number of productions listed with the programs correspond to the number actually present in the programs when the traces were obtained.

production system (though still written in OPS5) and is primarily goal driven. The DAA program represents a computation-intensive task compared to the knowledge-intensive tasks performed by XSEL, PTRANS, and MUD programs. The last four programs represent programming styles in SOAR, both when the productions are totally written by humans and when some fraction are automatically learned by the program.

5.3. Production-level Parallelism

As described in Section 4.1.1, when using production-level parallelism it is possible to process node activations corresponding to different productions in parallel. Whether they are all actually processed in parallel depends on the number of processors available, whether the processors have shared memory or not, etc. Figures 5-2 and 5-3 show the speed-ups (for overall runs) that can be obtained by using production-level parallelism over a uniprocessor implementation. Figure 5-2 shows the speed-up for a multiprocessor with shared memory and Figure 5-3 shows the speed-up for a multicomputer that does not have shared memory.

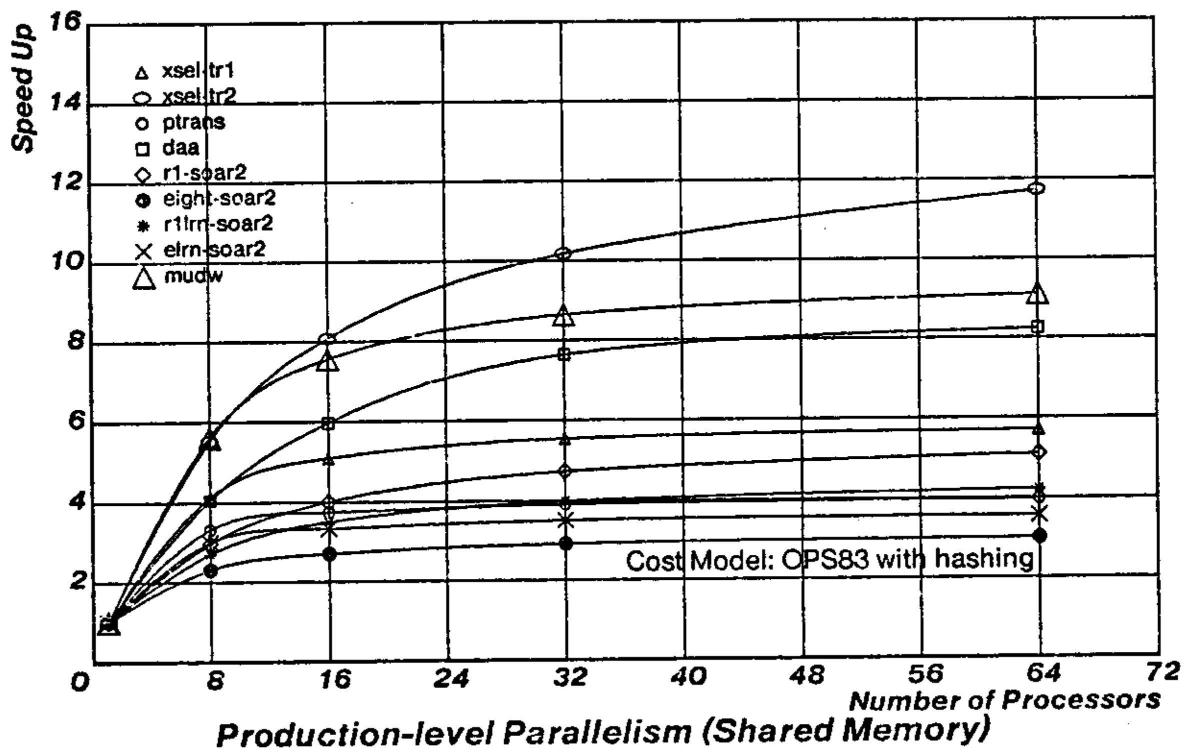

Figure 5-2:

Figures 5-2 and 5-3 show that the speed-up available from production-level parallelism tapers off quite sharply. To explain the nature of the graphs it is convenient to divide the curves into two regions. The first region, the *active* region, of the curve is where the overall speed-up is increasing significantly with an increase in the number of processors. The second region, the *saturated* region, corresponds to the portion where the

curve is almost flat.[19]

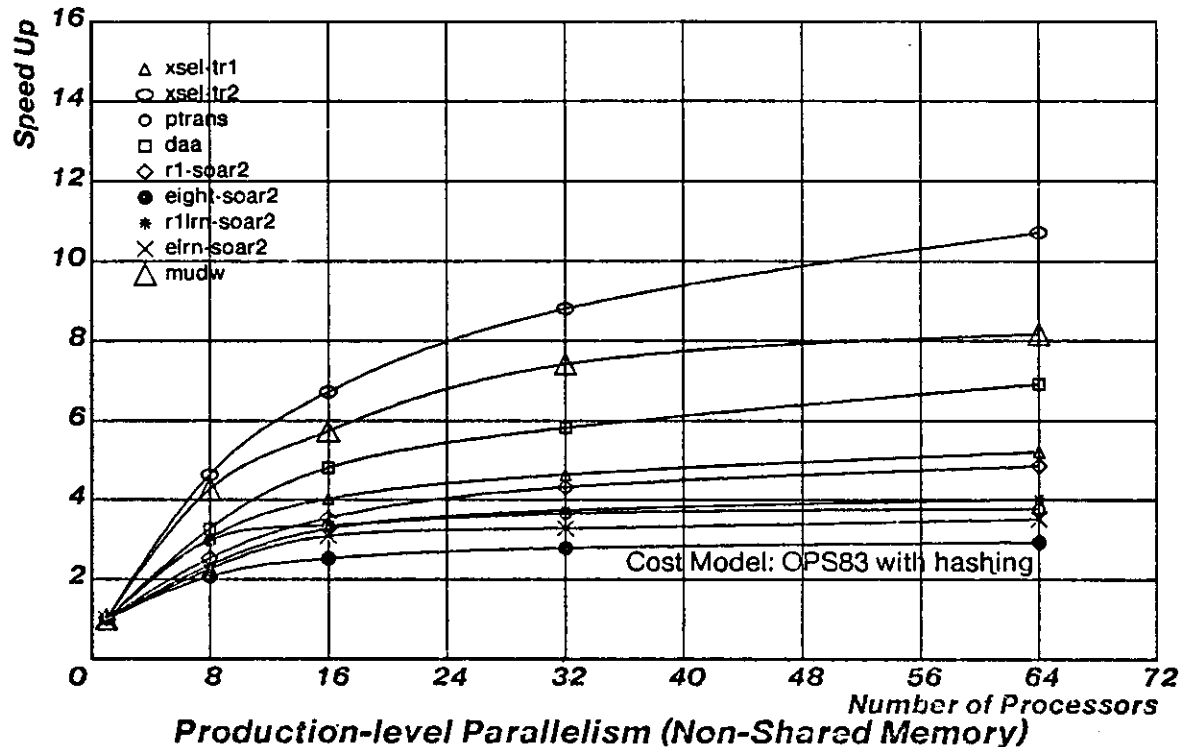

Figure 5-3:

The saturation speed-up, or the maximum speed-up, available from production-level parallelism is primarily determined by two factors. (1) It is limited by the number of affected productions, that is, the number of productions whose state changes as a result of a working memory change. For the traces under consideration the average size of the affect-set is 32. For the xsel-tr2 and mudw traces, which show a large saturation speed-up, the average affect-set size is 50 and 39 respectively. (2) The saturation speed-up is proportional to the ratio t_{avg}/t_{max}. For the curves shown in the figures the average saturation speed-up is 6, which is much smaller than the average affect-set size of 32. Thus a large factor of almost 5 is lost due to the variance in the costs of the different production activations. Consequently one of the main objectives of our research has been to develop a computational model to reduce this variance, a model where the cost of

[19] Note that since the scales for the x-axis and the y-axis are different, the line representing linear speed-up (with a slope of 1) will have an angle much more than 45 degrees in the graphs.

processing activations of different productions takes approximately the same time.[20]

The speed-up in the active region of the curves, in addition to being limited by and affected by the factors affecting the saturation speed-up, is dependent on the following factors. (1) The speed-up is obviously bounded by the number of processors in the system. (2) The speed-up is reduced by the loss of sharing in the nodes of the Rete network.[21] The effect of lost sharing is quite significant and results in a loss of a factor of around 1.4. For example, in the trace corresponding to the DAA program when 8 processors are present, although the processor utilization for the 8 processors is 80% (resulting in a virtual speed-up of 6.4), the actual speed-up is only 4.1, due to the loss of sharing. (3) The speed-up is also reduced by the variance in the size of the affect-sets. The variance results in a loss of processor utilization because within the same run for some cycles there are too many processors (the excess processors remaining idle) and for some cycles there are too few processors (some processors have to process more than one production activation, while other processors are waiting for these to finish). This is the reason why even if the average affect-set size is the same as the number of processors and all activations cost exactly the same, 100% processor utilization is not achieved. (4) In the case of non-shared memory multicomputers, the speed-up is greatly dependent on the quality of the partitioning, that is, the uniformity with which the work is distributed amongst the various processors. The round-robin[22] partitioning strategy was used to obtain the results shown in Figure 5-3.

In summary, it is observed from the graphs that the speed-up that may be obtained from production-level parallelism is limited to 4-8 fold. This is significantly below our initial expectations and the expectations of other researchers [7, 8, 24, 25]. The major blow comes from the limited number of affected productions (limiting the number of independent tasks) and the large variance in the processing requirements of these productions. To obtain more speed-up it is essential to either increase the number of independent tasks, or to decrease the variance, or to do both.

---

[20] This is the reason why the cost model for the simulations is derived from OPS83 with hash table based node memories. If node memories based on linked lists are used the variance increases significantly and simulations show that the saturation speed-up drops by a factor of almost 1.3.

[21] Recall that in a uniprocessor implementation similarities between the left hand sides of productions are exploited to share tests and operations. In order to gain the independence of being able to perform the tests and operations in parallel for various productions activations, the sharing has to be given up.

[22] In this strategy the kth production in the source file is allocated to the $(k \bmod \mathit{NumProcessors})^{th}$ processor. This strategy works reasonably well for production system programs written by humans, where textually close productions usually respond to the same working memory elements.

5.4. Node-level Parallelism

One way to increase the number of independent tasks and to decrease the variance in their processing requirements is to decompose the larger tasks into smaller tasks each of which can be processed in parallel. This is exactly what node-level parallelism attempts to do. When using node-level parallelism, in addition to processing the activations of different productions in parallel, multiple node activations corresponding to the same production are processed in parallel. Of course, it is not always possible to process all node activations corresponding to a production in parallel because of dependencies between them (one node activation causing another node activation). Furthermore, when using simple node-level parallelism, several activations of the same node in the Rete network are not allowed to be processed in parallel because of the excessive synchronization required by several processes working on the same data. Figure 5-4 shows the speed-up that can be obtained using node-level parallelism.

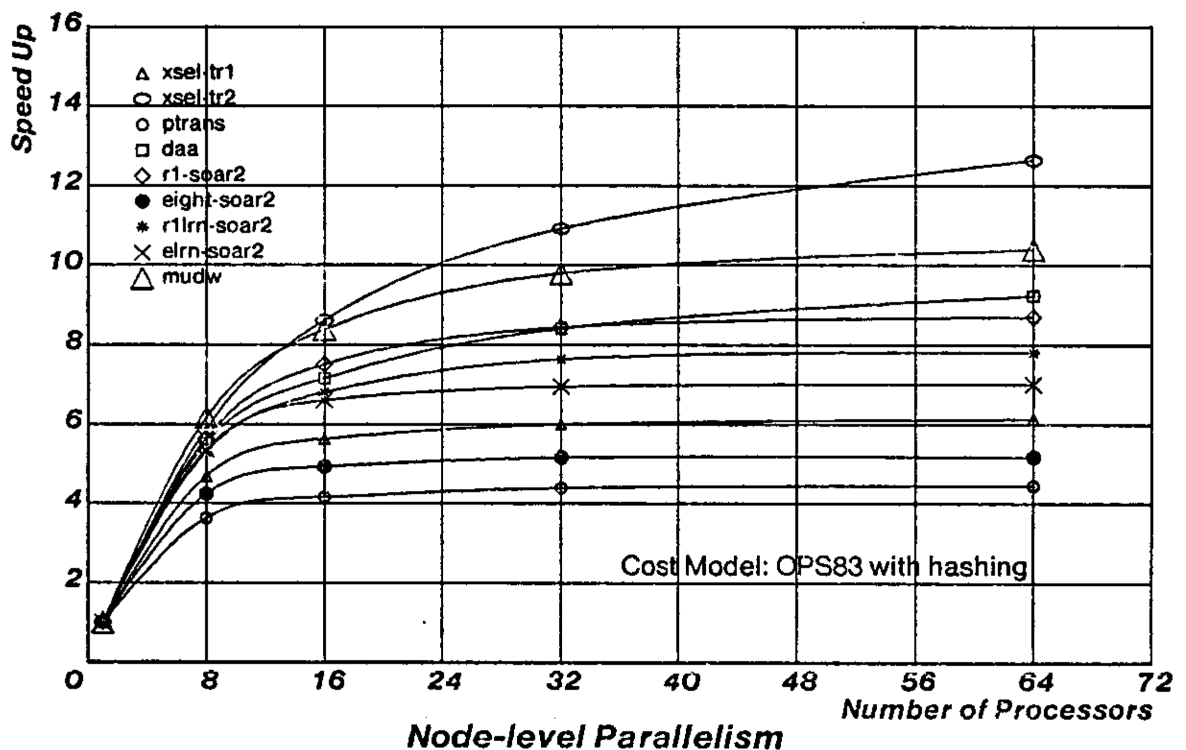

Figure 5-4:

An advantage of exploiting node-level parallelism over production-level parallelism is that it is possible to share nodes in the Rete network, as is done in the uniprocessor implementations. Node sharing is not possible when using production-level parallelism because each production is processed separately and consequently a separate Rete network is required for each production.[23] Because of the finer granularity of

[23] Network sharing is possible when using production-level parallelism only at the intra-production level, that is, within the network for a single production.

node-level parallelism, it is no longer necessary to keep the separate identity of productions, and whenever two productions have similar condition elements it is possible to share the corresponding two-input nodes.

Computing the extra speed-up from node-level parallelism over production-level parallelism, it is seen that for the case of 8 processors we gain a factor of 1.38; for 16 processors a factor of 1.36; for 32 processors a factor of 1.33; and for 64 processors a factor of 1.31.[24] This factor although significant is not very large. The reason for the small gain is that the parallelism is still being restricted by small affect-sets, long chains of dependent node activations, and multiple activations of the same node which have to be processed sequentially. These bottlenecks are discussed in detail in Section 5.6.

5.5. Action Parallelism

A production firing usually results in several changes to the working memory. For example, each cycle of an OPS5 program results in 2.4 changes to the working memory, and the corresponding number for SOAR is 8.8 changes.[25] This subsection discusses the results of processing these changes in parallel. Figures 5-5 and 5-6 show the results of using combinations of production and action parallelism, and node and action parallelism respectively.

As discussed in Section 4.1.3, the primary effect of using action parallelism is that it increases the number of independent tasks that may be processed in parallel. The result of using action parallelism with production-level parallelism enhances the maximum obtainable speed-up by a factor of 1.27, and using it with node-level parallelism enhances the maximum obtainable speed-up by a factor of 1.72. There are two reasons why the increase in speed-up is much less than 5.3 (the average number of changes processed in parallel): (1) The changes processed in parallel have overlapping affect-sets, that is, the multiple changes affect the same productions or they result in multiple activations of the same nodes which have to be processed sequentially. The simulations show that while on average 5.3 changes are processed in parallel, the average size of the affect-set increases only by a factor of 2. In one extreme case (xsel-tr2), there was one production firing that resulted in 109 changes to the working memory. However, the affect-sets of all the changes were exactly the same, so no speed-up was gained from processing them in parallel. (2) When the affect-sets of several working memory changes are combined together, the value of t_{avg} does not increase, but the value of t_{max} increases. This makes the value of t_{avg}/t_{max} small and decreases the obtainable speed-up.

---

[24] The reason for the decreasing gain as the number of processors increases is that the extra gain from the sharing of nodes in the network decreases as the number of processors increases.

[25] The number for SOAR corresponds to the combined number of changes made to the working memory by the productions firing in parallel.

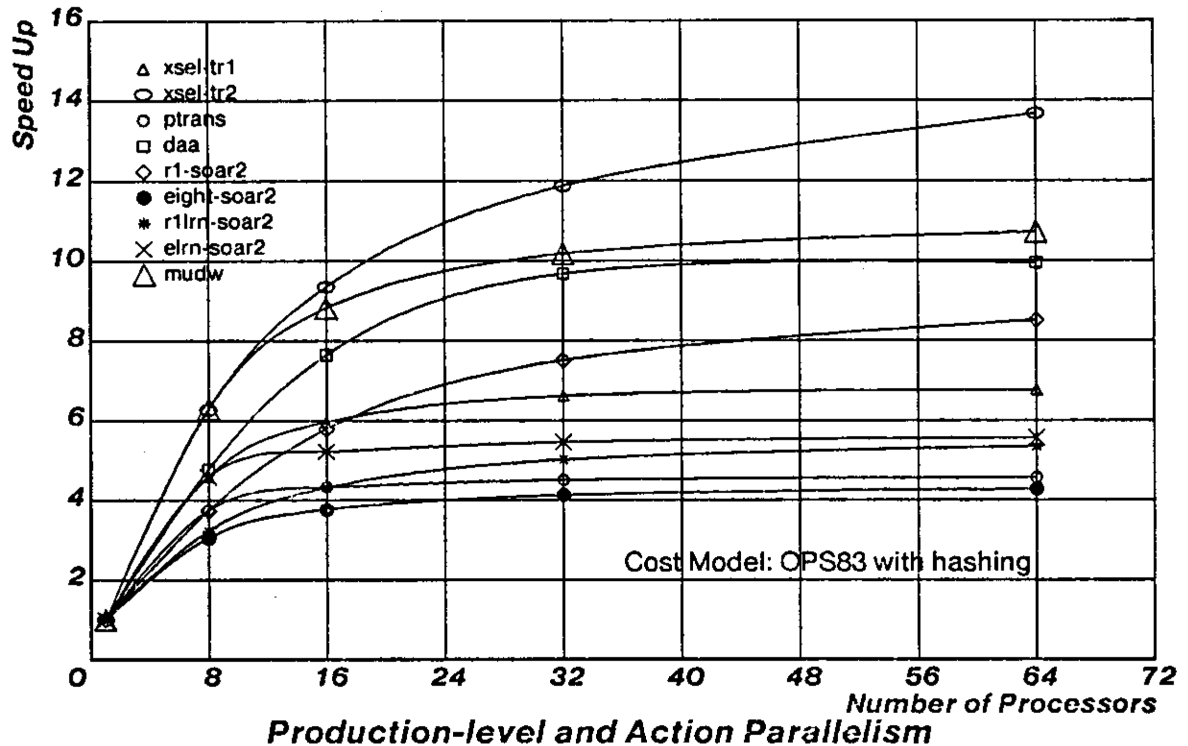

Figure 5-5:

Note in Figure 5-6 that the performance of r1lrn-soar is much worse than the performance of r1-soar even though the production systems are almost identical. This is a consequence of the fact that the productions which are learned by r1-soar have many more condition elements (one of the learned productions has more than 100 condition elements) than the number of condition elements in productions written by humans. The large number of condition elements leads to long chains of dependent node activations for r1lrn-soar, which greatly decreases the factor t_{avg}/t_{max}, and correspondingly decreases the speed-up.

Figure 5-7 shows the maximum speed-up obtainable from the various sources of parallelism (shown in the corners) and the multiplicative increase when an additional source of parallelism is exploited (shown in the middle of the connecting line). An observation that requires some explanation is that the enhancement of speed-up is more when action parallelism is used with node-level parallelism (factor of 1.72) than when it is used with production-level parallelism (factor of 1.27). The reason is that, when action parallelism is used with production-level parallelism, if two changes affect the same production they have to be processed sequentially, and no extra speed-up is gained. When action parallelism is used with node-level parallelism, then even if the two changes affect the same production, it is often the case that they affect two different nodes belonging to that production. Since the two node activations can be processed in parallel extra speed-up is obtained.

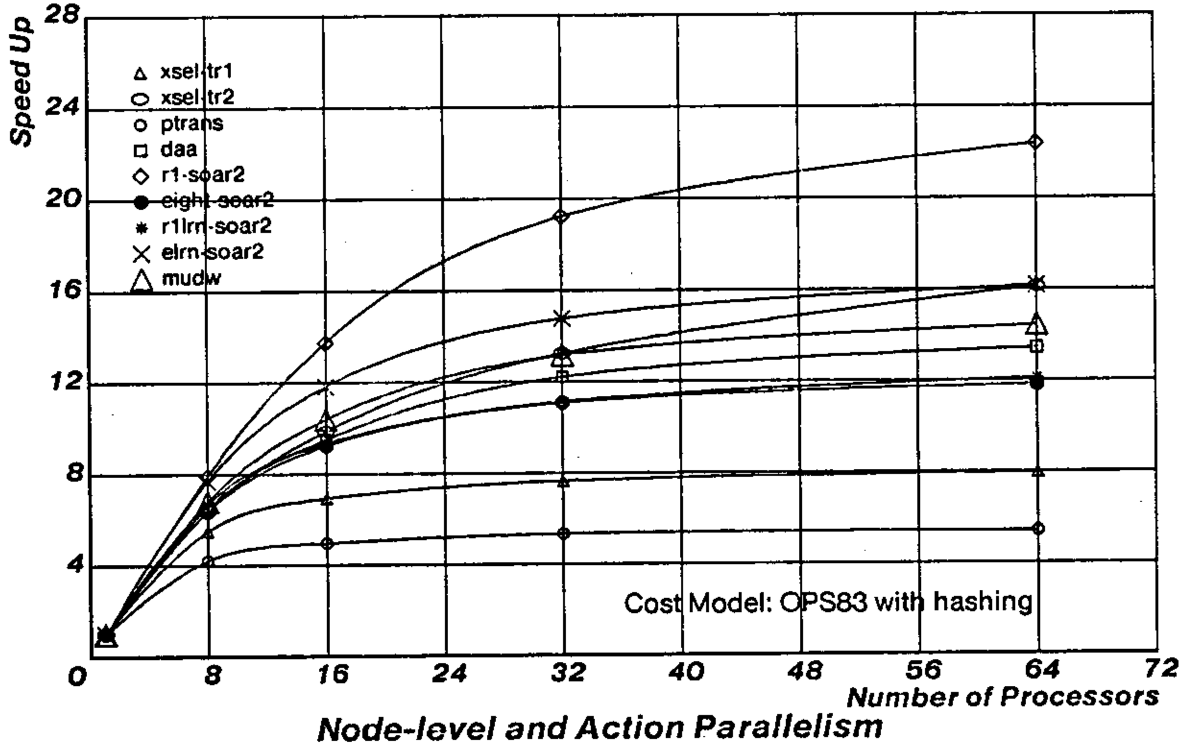

Figure 5-6:

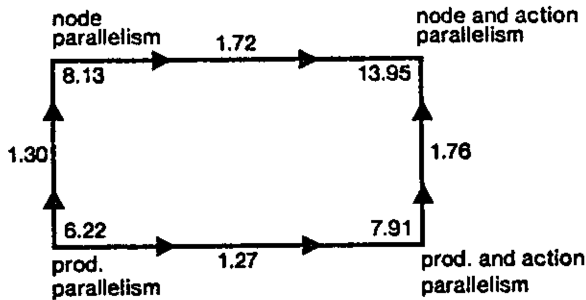

Figure 5-7: Maximum Speed-up Obtainable from Various Sources

In summary, it is essential to use action parallelism if large speed-up is to be obtained from parallelism. This kind of parallelism is especially important in systems where a large number of changes are made to the working memory on every cycle. The large number of changes could result from (1) interaction with the external environment or database (as in XSEL), (2) parallel firing of productions (as in SOAR systems), or (3) use of application parallelism, where multiple threads of computation are being followed simultaneously. Of course, a large number of changes by itself is not sufficient to ensure large speed-ups. It is essential that these changes have differing affect-sets, so that the resulting node activations can be processed in parallel.

5.6. Bottlenecks in Obtaining Speed-up from Parallelism

As stated in previous subsections, the two primary factors limiting speed-up are the small number of affected productions and the small value of t_{avg}/t_{max}. Since the number of productions that are affected on each cycle is not controlled by the implementor of production system interpreter (it is governed mainly by the author of the program and the nature of the problem), this subsection concentrates on what can be done to increase the value of t_{avg}/t_{max}. Looking at the execution of the traces in the simulator, two major causes for the large value of t_{max} were found. The first cause is the *cross-product* effect and the second cause is the *long-chain* effect. These causes are pictorially depicted in Figure 5-8.[26]

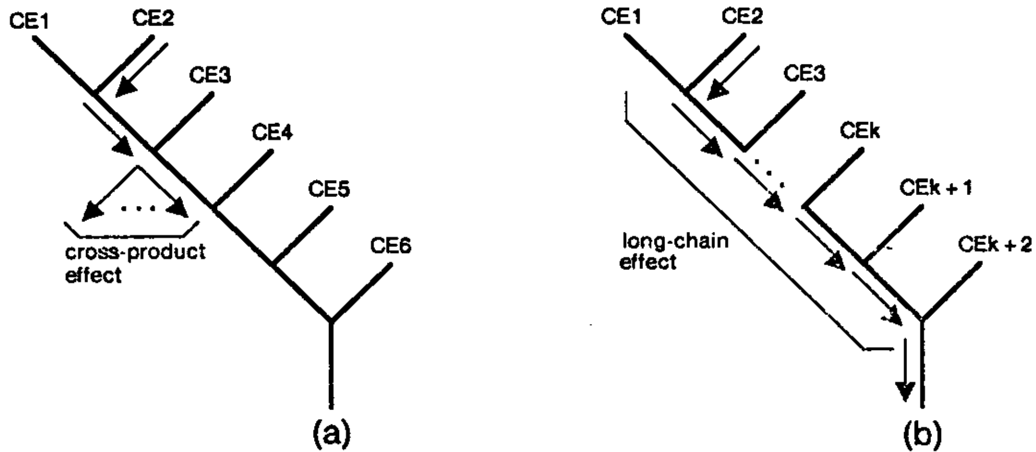

Figure 5-8: Bottlenecks

In the cross-product effect shown in Figure 5-8-a, a token arriving at a two-input node finds a large number of tokens with consistent bindings in the opposite memory. As a result a large number of new tokens are sent to its successor nodes. The successor nodes are now subject to this large number of activations, which have to be processed sequentially,[27] causing a large value of t_{max} and resulting in low speed-up. If the multiple activations of the same node are processed in parallel, simulations show that the maximum speed-up obtainable from node-level and action parallelism goes up from 13.95 to 24.06, thus providing an extra factor of 1.72.[28]

During the simulations it was often observed that the large value of t_{max} resulted from a long chain of

[26] The arrows represent the flow of tokens in the Rete network, while the thick lines represent the network for the production.

[27] The reasons for processing multiple activations of the same node sequentially are similar to the reasons for locking a shared-stack data structure in a multiprocessor.

[28] As a result of recent research, we now have a solution to the problem of processing multiple activations of the same node in parallel without significant overhead.

dependent node activations (see Figure 5-8-b), that is, one node activation causing an activation in a successor node, which in turn caused an activation in its successor node, and so on. These long chains occur in productions having a very large number of condition elements.[29] We are currently working on methods for reducing these long sequences of node activations. While the sequences of dependent node activations cannot be totally eliminated, we believe that they can be shortened significantly. Simulations show that in the hypothetical case where the dependent chains are eliminated totally an extra factor of 2 in speed up is obtained.

5.7. Comparison of the Sources of Parallelism

So far the sources of parallelism have been considered in relative isolation. To enable the comparison of speed-up from the various sources, Figure 5-9 shows the speed-up from combinations of various sources averaged over all the traces. As expected the combination of node-level and action parallelism does much better than any of the other combinations. Node-level parallelism comes second and the combination of production and action parallelism a close third. The rest are all clustered below.

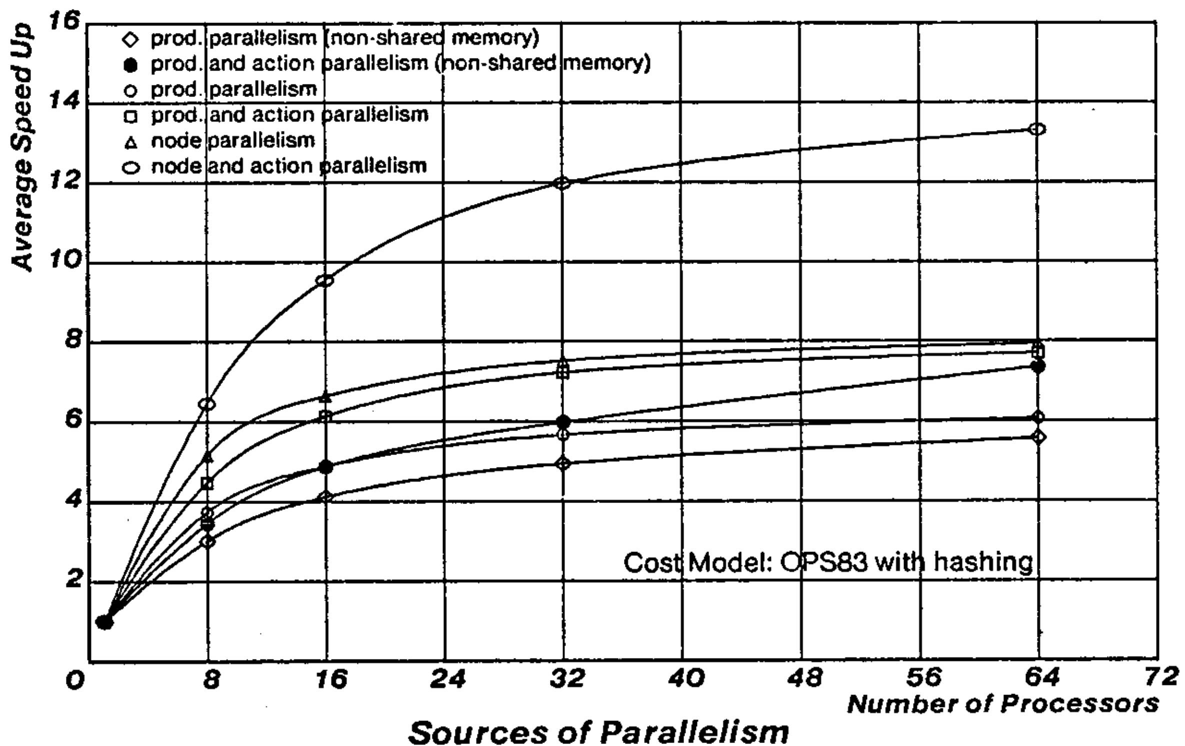

Figure 5-9:

[29] The Rete network compiler constructs a linear network for productions, that is, the two-input nodes for the production are strung in a linear chain. It has been empirically observed in uniprocessor implementations that this reduces the amount of state that has to be stored in the network.

5.8. Limitations of the Simulation Results

An important fact that was mentioned in Section 5.1, but omitted in the subsequent discussion is that most curves shown in graphs so far are results of a simulation model that does not take scheduling, synchronization, and memory contention overheads into account.[30] Several observations can be made:

- All the curves corresponding to shared-memory multiprocessors in Figure 5-9 will be pushed down, while the curves corresponding to non-shared memory multicomputers will retain their positions, as they do not encounter the above overheads. In fact, one might speculate as to what the curves will look like if the overheads are taken into account. One possible outcome is shown in Figure 5-10. It is expected that the systems which will offer the best cost-performance in the small speed-up range will be non-shared memory or shared-memory architectures using simple sources of parallelism (p-par, pa-par), since the associated overheads are small. However, if a very high performance system is desired, then it will be necessary to exploit all the possible sources of parallelism (na-par) even if it means high overheads [6].

- For small number of processors (10-20) the memory contention and synchronization overheads are not expected to be very significant. The two main reasons are: (1) because of the constraints imposed on the processing model (for example, no two activations of the same node are to be processed in parallel) most processors will be working on different data, resulting in very few conflicts; and (2) there is a large amount of read-only data which can either be cached or be replicated in the local memories of the processors to reduce contention.

- The predicted speed-ups without taking memory contention and synchronization overheads into account represent upper-bounds[31] on the speed-ups that can actually be achieved. While the analysis presented in [7] gave some upper-bounds on the basis of the size of the affect-sets, the upper bounds determined in this paper are much tighter.

As stated in Section 5.2, the simulation results presented in this paper are based on traces derived from eight programs written in OPS5 and SOAR. There are some issues related to the representativeness of the traces and thus the results. The production systems considered in the simulations are limited in two ways: (1) they are limited in that they represent only a fraction of all existing OPS5 and SOAR programs, and (2) they are limited in that they say little about the characteristics of production systems several years hence. The following observations can be made:

- Although the production system programs included in the simulations form only a fraction of the programs written using OPS5 and SOAR, they include some of the largest programs written in these languages [9, 10, 18, 22]. Also, many programs not included in the simulations but studied in other papers [7, 20] show characteristics similar to the programs included in the simulations. Since the characteristics are similar, there is good reason to believe that the speed-up obtainable from parallelism will be similar.

---

[30] We are currently working on models that will take such overheads into account.

[31] The notion of upper-bound is used in a loose sense here and refers to the speed-up obtainable within the specified model of parallelism. It is quite possible that if the style of programming production systems changes in the future or with the discovery of new parallel algorithms the predicted speed-up is exceeded.

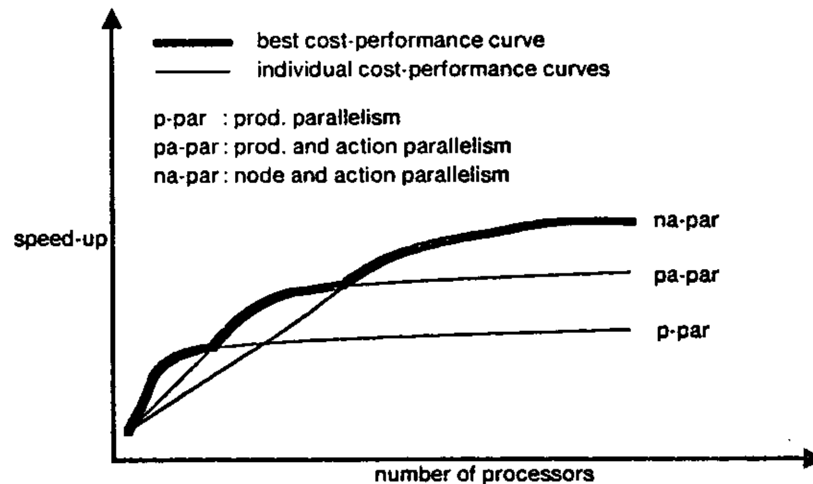

Figure 5-10: Cost-Performance Tradeoffs

- An important factor determining the speed-up available from parallelism in production systems is the number of productions affected by a change to working memory. This number has been observed to be quite small and of similar magnitude in all the programs studied (irrespective of the total number of productions in the programs). An intuitive explanation for this observation is that programmers recursively divide problems into subproblems when writing the programs. Then, at any given time, the program execution corresponds to solving only one of these subproblems. The size of the subproblems (which is correlated to the number of productions associated with the subproblems) is independent of the size of the original problem and primarily depends on the complexity of the subproblem and the complexity that that the programmer can deal with at the same time. Thus, if the above explanation is true, then many programs written by programmers in the future will exhibit small affect-sets, and consequently will have only limited benefits from parallelism.[32]

A few words on the dependence of the results presented in this paper on the use of the Rete algorithm as the base for the parallel implementation. Although the exact numbers about the speed-up available from paralellism have certainly been influenced by the use of the Rete algorithm, we believe that the overall nature of the results has not been influenced greatly. For example, we do not think that the use of a different algorithm will increase the speed-up obtainable from parallelism (of course, compared to the best uniprocessor implementation of production systems) to a 100-fold instead of the 10-fold to 20-fold that we have found. The reasons are:

- The Rete algorithm embodies two general principles that do not restrict the use of parallelism. The first principle is that of precompiling the productions into a form so as to gain run-time

[32] The above discussion only addresses the case where application parallelism (see Section 4.5) is not exploited. In case application parallelism is used, it is possible for a program to be working on several subproblems simultaneously, thus having a large set of affected productions.

efficiency. This is not incompatible with the use of parallelism.[33] The second principle is that of storing the results of match from previous cycles as state, so that only incremental processing has to be done on each cycle. Although it is possible to keep a large number processors busy if no state is stored, our experience has been that this does not result in an increased absolute speed of program execution [8].

- Any attempt to explore parallelism in production systems must begin with some base algorithm. After considering many alternative algorithms, we have not found any algorithm, significantly different from Rete, that is more suitable for a parallel implementation (at least for the production systems resembling the ones considered in this paper). It appears that the data-flow like representation used by the Rete algorithm captures most of the inherent parallelism present in production system programs. Finally, it is important to note that the Rete algorithm is being used only as the starting point and where necessary it will be and has been modified to suit the needs of a parallel implementation.

6. Summary

This paper describes the various sources of parallelism that can be used to speed up the execution of production systems. The sources of parallelism considered are: (1) production-level parallelism, (2) node-level parallelism, (3) action parallelism, (4) parallelism in conflict-resolution, (5) parallelism in act, (6) parallelism from the run-time addition of productions, and (7) application parallelism. Out of these seven sources, the first three are examined in greater detail than the others. Results of simulations show that it is possible to speed up the match phase by up to 6-fold using production-level parallelism, up to 8-fold using node-level parallelism, and up to 14-fold using a combination of node-level and action parallelism. While the speed-ups obtained from parallelism are significant, they are much below our initial expectations (order of 100-1000 fold). The main reasons for the limited speed-up are (1) the small number of affected productions for each change to the working memory, (2) the large variance in the processing requirements of the production activations, and (3) the fact that successive changes to working memory affect almost the same set of productions. While the first and the third bottlenecks listed above are beyond the direct control of the person implementing the production system interpreter, it is possible to do something about the second bottleneck. Our recent efforts to reduce the variance in the cost of production activations show that it may now be possible to achieve as much as 24-fold speed-up over state of art uniprocessor implementations.

Finally, the following assumptions of the analysis should be kept in mind. The analysis is based on measurements and simulations performed on programs in OPS5 and SOAR. The analysis is dependent on

[33] In actuality the situation is not as simple as this point makes it out to be. Since the traces for the simulations are obtained from a particular uniprocessor implementation of Rete, in some instances specific choices made in the implementation have influenced the results. For example, the existing Rete compiler compiles productions into linear networks (linear list of two-input nodes). Although it would be interesting to evaluate the performance when the productions are compiled into binary networks, it was not possible to do it from the existing traces corresponding to the linear networks. However, for most OPS5 systems analyzed in this paper, the average number of condition elements per production is so small that binary networks would cause only minor changes in the results.

prevailing programming styles and will lose some of its validity as programming styles evolve. The simulation results presented do not include the scheduling, synchronization, and memory contention overheads that will be experienced in a shared-memory multiprocessor, and to that extent the results represent an upper-bound on the amount of speed-up that is obtainable. Our current research on multiprocessor architectures for production systems, however, shows that it is possible to use custom hardware to significantly reduce some of the above overheads.

7. Acknowledgments

I wish to thank Charles Forgy, John McDermott, and Allen Newell for many insightful comments and suggestions. I also wish to thank H.T. Kung for his enthusiastic support of the research reported in this paper.

References

[1] B.G. Buchanan and E.A. Feigenbaum.
DENDRAL and Meta-DENDRAL: their applications dimensions.
Artificial Intelligence 11(1,2), 1978.

[2] R.O. Duda, J.G. Gaschnig, and P.E. Hart.
Model Design in the PROSPECTOR Consultant System for Mineral Exploration.
In D. Michie (editor), *Expert Systems in the Micro-Electronic Age.* Edinburgh University Press, Edinburgh, 1979.

[3] Charles L. Forgy.
OPS5 User's Manual.
Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, 1981.

[4] Charles L. Forgy.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence 19, September, 1982.

[5] Charles L. Forgy.
The OPS83 Report.
Technical Report CMU-CS-84-133, Carnegie-Mellon University, Pittsburgh, May, 1984.

[6] Charles Forgy, Anoop Gupta, Allen Newell, and Robert Wedig.
Initial Assessment of Architectures for Production Systems.
In *National Conference for Artificial Intelligence.* AAAI-1984.

[7] Anoop Gupta and Charles L. Forgy.
Measurements on Production Systems.
Technical Report CMU-CS-83-167, Carnegie-Mellon University, Pittsburgh, 1983.

[8] Anoop Gupta.
Implementing OPS5 Production Systems on DADO.
In *International Conference on Parallel Processing.* IEEE, 1984.

[9] P. Haley, J. Kowalski, J. McDermott, and R. McWhorter.
PTRANS: A Rule-Based Management Assistant.
Technical Report, Carnegie-Mellon University, Pittsburgh, 1983, (in preparation).

[10] Gary Kahn and John McDermott.
The MUD System.
In *The First Conference on Artificial Intelligence Applications.* IEEE Computer Society and AAAI, December, 1984.

[11] Jin Kim, John McDermott, and Daniel Siewiorek.
TALIB: A Knowledge-Based System for IC Layout Design.
In *National Conference on Artificial Intelligence.* AAAI-1983.

[12] Ted Kowalski and Don Thomas.
The VLSI Design Automation Assistant: Prototype System.
In *20th Design Automation Conference.* ACM and IEEE, June, 1983.

[13] John E. Laird.
Universal Subgoaling.
PhD thesis, Carnegie-Mellon University, Pittsburgh, December, 1983.

[14] John E. Laird and Allen Newell.
A Universal Weak Method: Summary of Results.
In *International Joint Conference on Artificial Intelligence.* 1983.

[15] John E. Laird and Allen Newell.
A Universal Weak Method.
Technical Report CMU-CS-83-141, Carnegie-Mellon University, Pittsburgh, June, 1983.

[16] John E. Laird, Paul S. Rosenbloom, and Allen Newell.
Towards Chunking as a General Learning Mechanism.
In *National Conference on Artificial Intelligence.* AAAI-1984.

[17] John McDermott.
R1: A Rule-based Configurer of Computer Systems.
Technical Report CMU-CS-80-119, Carnegie-Mellon University, Pittsburgh, April, 1980.

[18] John McDermott.
XSEL: A Computer Salesperson's Assistant.
In J.E. Hayes, D. Michie, and Y.H. Pao (editor), *Machine Intelligence.* Horwood, 1982.

[19] W. Van Melle, A.C. Scott, J.S. Bennett, and M. Peairs.
The Emycin Manual.
Technical Report STAN-CS-81-885, Stanford University, October, 1981.

[20] Kemal Oflazer.
Parallel Execution of Production Systems.
In *International Conference on Parallel Processing.* IEEE, August, 1984.

[21] Paul S. Rosenbloom.
The Chunking of Goal Hierarchies: A Model of Stimulus-Response Compatibility.
PhD thesis, Carnegie-Mellon University, Pittsburgh, August, 1983.

[22] Paul S. Rosenbloom, John E. Laird, John McDermott, and Allen Newell.
R1-Soar: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture.
In *IEEE Workshop on Principles of Knowledge Based Systems.* 1984.

[23] E. H. Shortliffe.
Computer-Based Medical Consultations: MYCIN.
North-Holland, 1976.

[24] Salvatore J. Stolfo and David E. Shaw.
DADO: A Tree-Structured Machine Architecture for Production Systems.
In *National Conference on Artificial Intelligence.* AAAI-1982.

[25] Salvatore J. Stolfo, Daniel Miranker, and David E. Shaw.
Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence.
In *International Joint Conference on Artificial Intelligence.* 1983.

[26] Shinji Umeyama and Koichiro Tamura.
A Parallel Execution Model of Logic Programs.
In *The 10th Annual International Symposium on Computer Architecture.* IEEE and ACM, June, 1983.

[27] D.A. Waterman and Frederick Hayes-Roth.
Pattern-Directed Inference Systems.
Academic Press, 1978.

[28] S.M. Weiss and C.A. Kulikowski.
EXPERT: A System for Developing Consultation Models.
In *International Joint Conference on Artificial Intelligence.* 1979.