International Journal of Parallel Programming, Vol. 17, No. 2, 1988

Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis

Anoop Gupta,[1] Milind Tambe,[2] Dirk Kalp,[2] Charles Forgy,[2] and Allen Newell[2]

Received November 1987; Revised August 1988

Until now, most results reported for parallelism in production systems (rule-based systems) have been simulation results—very few real parallel implementations exist. In this paper, we present results from our parallel implementation of OPS5 on the Encore multiprocessor. The implementation exploits very fine-grained parallelism to achieve significant speed-ups. For one of the applications, we achieve 12.4 fold speed-up using 13 processes. Our implementation is also distinct from other parallel implementations in that we parallelize a highly optimized C-based implementation of OPS5. Running on a uniprocessor, our C-based implementation is 10-20 times faster than the standard lisp implementation distributed by Carnegie Mellon University. In addition to presenting the performance numbers, the paper discusses the details of the parallel implementation—the data structures used, the amount of contention observed for shared data structures, and the techniques used to reduce such contention.

KEY WORDS: Production Systems; Rule-based Systems; OPS5, Parallel Processing; Fine-Grained Parallelism; AI Architectures.

1. INTRODUCTION

As the technology of production systems (rule-based systems) is maturing, larger and more complex expert systems are being built both in industry

[1] Department of Computer Science, Stanford University, Stanford, California 94305
[2] Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213

0885-7458/88/0400-0095$06.00/0 © 1988 Plenum Publishing Corporation

828/17/2-1

and in universities. Often these large and complex systems are very slow in their execution, and this limits their utility. Researchers have been exploring many alternative ways for speeding up the execution of production systems. Some efforts have been focusing on high-performance uniprocessor implementations,[1,2] while others[3–10] have been focusing on high-performance parallel implementations. This paper focuses on parallel implementations.

Until now, most results reported for parallelism in production systems have been simulation results. In fact, very few real parallel implementations exist. In this paper, we present results from our parallel implementation of OPS5 on an Encore Multimax shared-memory multiprocessor with sixteen CPUs. The implementation, called PSM-E (Production System Machine project's Encore implementation), exploits very fine-grained parallelism to achieve up to 12.4 fold speed-up for match using 13 processes. Our implementation is distinct from other parallel implementations in that we parallelize a highly optimized C-based implementation of OPS5. Running on a uniprocessor, our C-based implementation is 10–20 times faster than the lisp implementation of OPS5 distributed by Carnegie Mellon University. A consequence of parallelizing a highly-optimized implementation is that one must be very careful about overheads, else the overheads may nullify the speed-up. One need not be as careful when parallelizing an unoptimized implementation. In this paper, we first discuss the design of an optimized implementation of OPS5, and then discuss the additions that were made for the parallel implementation. For the parallel implementation, we discuss the synchronization mechanisms that were used, the contention observed for various shared data structures, and the techniques used to reduce such contention.

The paper is organized as follows. Section 2 presents some background information about the OPS5 language, the Rete match algorithm, and the Encore Multimax multiprocessor. Section 3 gives an overview of the parallel interpreter and then goes into the implementation details describing how the rules are compiled and how various synchronization and scheduling issues are handled. Section 4 presents the results of the implementation on the Encore multiprocessor. Finally, in Section 5 we summarize the results and conclude.

2. BACKGROUND

This section is divided into three parts. The first subsection describes the basics of the OPS5 production-system language—the language which we have implemented in parallel. The second subsection describes the Rete algorithm—the algorithm that forms the basis for our parallel implemen-

tation. The third subsection describes the Encore Multimax computer system—the multiprocessor on which we have done the parallel implementation.

2.1. OPS5

An OPS5[11] production system is composed of a set of *if-then* rules called *productions* that make up the *production memory*, and a database of temporary assertions called the *working memory*. The assertions in the working memory are called *working memory elements* (wmes). 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 1 shows a production named *find-colored-block* with two condition elements in its left-hand side and one action in its right-hand side.

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 *recognize-act* cycle:

- **Match**: In this first phase, the left-hand sides of all productions are matched against the contents of working memory. As a result a *conflict set* is obtained, which consists of *instantiations* of all satisfied productions. An instantiation of a production is an ordered list of working memory elements that satisfies the left-hand side of the production.
- **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.

```
(p find-colored-block
   (goal  ^type find-block  ^color  <c>)
   (block ^id <i>  ^color <c>)
   (color ^code <c> ^name <s1>)
 -->
   (write "Found Block of Color <s1>"))
```

Fig. 1. A sample production.

A working memory element is a parenthesized list consisting of a constant symbol called the *class* of the element and zero or more *attribute-value* pairs. The attributes are symbols that are preceded by the operator ^. The values are symbolic or numeric constants. For example, the following working memory element has class **C1**, the value **12** for attribute **attr1** and the value **15** for attribute **attr2**.

(C1 ^attr1 12 ^attr2 15)

The condition elements in the left-hand side of a production are parenthesized lists similar to the working memory elements. They may optionally be preceded by the symbol –. Such condition elements are then called *negated* condition elements. Condition elements are interpreted as partial descriptions of working memory elements. When a condition element describes a working memory element, the working memory element is said to *match* the condition element. A production is said to be *satisfied* when: (1) For every nonnegated condition element in the left-hand side of the production, there exists a working memory element that matches it; (2) For every negated condition element in the left-hand side of the production, there does not exist a working memory element that matches it.

Like a working memory element, a condition element contains a class name and a sequence of attribute-value pairs. However, the condition element is less restricted than the working memory element; while the working memory element can contain only constant symbols and numbers, the condition element can contain variables, predicate symbols, and a variety of other operators as well as constants. Variables are identifiers that begin with the character "⟨" and end with "⟩"—for example, ⟨**i**⟩ and ⟨**c**⟩ are variables. A working memory element matches a condition element if they belong to the same class and if the value of every attribute in the condition element matches the value of the corresponding attribute in the working memory element. The rules for determining whether a working memory element value matches a condition element value are: (1) If the condition element value is a constant, it matches only an identical constant. (2) If the condition element value is a variable, it will match any value. However, if a variable occurs more than once in a left-hand side, all occurrences of the variable must match identical values. (3) If the condition element value is preceded by a predicate symbol, the working memory element value must be related to the condition element value in the indicated way.

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 working memory. *Modify* changes one or more values of an existing working memory element. *Remove* deletes an element from the working memory.

2.2. The Rete Match Algorithm

In this subsection, we describe the Rete algorithm used for performing the match-phase in the execution of production systems. The match-phase is critical because it takes 90% of the execution time and as a result it needs to be speeded up most. Rete is a highly efficient algorithm for match that is also suitable for parallel implementations. (A detailed discussion of Rete and other match algorithms can be found in Refs. 4 and 12.) The Rete algorithm gains its efficiency from two optimizations. First, it exploits the fact that only a small fraction of working memory changes each cycle by storing results of match from previous cycles and using them in subsequent cycles. Second, it exploits the similarity between condition elements of productions (both within the same production and between different productions) to reduce the number of tests that it has to perform to do match. It does so by performing common tests only once.

The Rete algorithm uses a special kind of a data-flow network compiled from the left-hand sides of productions to perform match. The network is generated at compile time, before the production system is actually run. Figure 2 shows such a network for productions *p*1 and *p*2, which appear in the top part of this illustration. In Fig. 2 lines have been drawn between nodes to indicate the paths along which information flows. Information flows from the top-node down along these paths. The nodes with a single predecessor (near the top of the figure) are the ones that are concerned with individual condition elements. The nodes with two predecessors are the ones that check for consistency of variable bindings between condition elements. The terminal nodes are at the bottom of the figure. Note that when two left-hand sides require identical nodes, the algorithm shares part of the network rather than building duplicate nodes.

To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as state within the nodes. This way, only changes made to the working memory by the most recent production firing have to be processed every cycle. Thus, the input to the Rete network consists of the changes to the working memory. These changes filter through the network updating the state stored within the network. The output of the network consists of a specification of changes to the conflict set.

The objects that are passed between nodes are called *tokens*, which

```
(p p1 (C1 ^attr1 <x> ^attr2 12)        (p p2 (C2 ^attr1 15 ^attr2 <y>)
      (C2 ^attr1 15  ^attr2 <x>)             (C4 ^attr1 <y>)
    - (C3 ^attr1 <x>)                  -->
  -->                                        (modify 1 ^attr1 12))
      (remove 2))
```

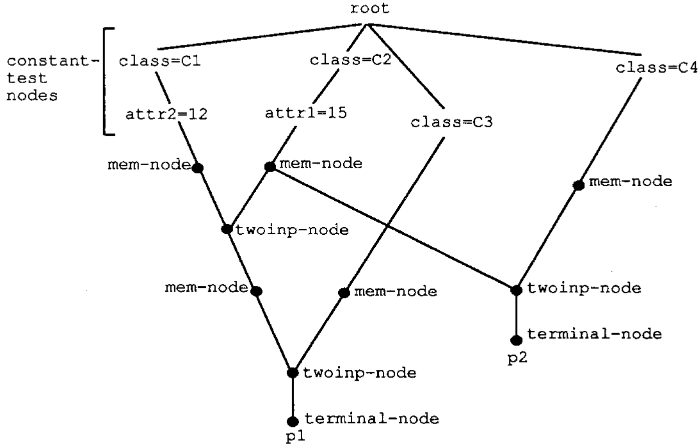

Fig. 2. The Rete network.

consist of a *tag* and an *ordered list of working-memory elements.* The tag can be either a +, indicating that something has been added to the working memory, or a −, indicating that something has been removed from it. No special tag for working-memory element modification is needed because a modify is treated as a delete followed by an add. The list of working-memory elements associated with a token corresponds to a sequence of those elements that the system is trying to match or has already matched against a subsequence of condition elements in the left-hand side.

The data-flow network produced by the Rete algorithm consists of four different types of nodes. 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 always appear in the top part of the network. They have only one input, and as a result, they are sometimes called *one-input* nodes.
2. **Memory nodes:** These nodes store the results of the match phase 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 Fig. 2 stores all tokens matching the second condition-element of production *p*2.

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 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 Fig. 2. Whenever a token flows into a terminal node, the corresponding production is either inserted into or deleted from the conflict set.

The most commonly used interpreter for OPS5 is the Rete-based Franz Lisp interpreter. In this interpreter a significant loss in the speed is due to the interpretation overhead of nodes. In the OPS5 implementation we present in this paper, the interpretation overhead has been eliminated by compiling the network directly into machine code. 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 speed-up of 10–20 fold over the Franz Lisp interpreter (see Table I). In addition to this speed-up, our parallel implementation gets further speed-up by evaluating different node activations in the Rete network in parallel.

2.3. Encore Multimax

In this subsection, we describe the Encore Multimax shared-memory multiprocessor—the computer system on which parallel OPS5 runs. The Multimax consists of 2-20 CPUs, each of which is connected to the shared-memory through a high performance bus. The shared-memory is equally

Table I. Speed-up of *C*-based over Franzlisp-based Implementation

PROGRAM	*VS*-lisp Lisp-based implementation (sec)	*VS*2 Hash-based memories (sec)	Speed-up *VS*-lisp/*VS*2
Weaver	1104.0	85.8	12.9
Rubik	1175.0	96.9	12.1
Tourney	2302.0	93.5	24.6

accessible to all of the processors, in that each processor sees the same latency for memory accesses.

The processors used in our Encore Multimax are National Semiconductor NS32032 chips along with NS32081 floating point coprocessors, each processor capable of approximately 0.75 million instructions per second. There are two processors packaged per board and they share 32 Kbytes of cache memory. The processor boards use a combination of write-through strategy and bus-watching logic to keep the caches on different processor boards consistent. The bus used on the Encore Multimax is called the *Nanobus*. It is a synchronous bus and it can transfer 8 bytes of new information every 80 nanoseconds, thus providing a data transfer bandwidth of 100 Mbytes/second.

The version of Encore Multimax available to us at CMU has 16 processors, 32 Mbytes of memory, and runs the MACH operating system developed at Carnegie Mellon University. The operating system provides a UNIX-like interface to the user, although the internals are different and several extensions have been made to support the underlying parallel hardware. It provides facilities to automatically distribute processes amongst the available processors and it provides facilities for multiple processes to share memory for communication and synchronization purposes. The results reported in this paper correspond to this configuration of the Encore Multimax.

3. ORGANIZATION AND DETAILS OF THE PARALLEL IMPLEMENTATION

3.1. High-Level Structure of the Parallel Implementation

The parallel OPS5 implementation on the Encore (PSM-E) consists of one *control process* and one or more *match processes*. The number of match processes is a user specified parameter, but it is fixed for the duration of any particular run. The system is generally used in a mode where the computer contains at least as many free processors as there are processes in the matcher; this permits each process to be assigned to a distinct processor for the duration of the run (provided the operating system is reasonably clever about assigning processes to processors).

The control process is responsible for performing conflict resolution, evaluating the right-hand side of rules, handling input/output, and all the other functions of the interpreter except for performing match. It is also responsible for starting up the match processes at the beginning of the run and killing them at the end of the run. The match processes do nothing except perform the match. The match processes pipeline their operation

with the control process. Thus when RHS evaluation begins, the match processes are idle. However, as soon as the first working memory change is computed, information about that change is passed to the match processes and they start to work. The control process continues evaluating the RHS, and as more changes are computed, the information is passed immediately to the match processes for them to handle as soon as they are able. When the control process finishes evaluating the RHS, it becomes idle and waits for the match processes to finish. When the last match process finishes, the control process performs conflict resolution and then begins evaluating the next RHS, thus starting the cycle over again.[3]

To perform match, the match processes use the Rete algorithm described in Section 2.2. The match processes exploit the dataflow-like nature of the Rete algorithm to achieve speed-up from parallelism. In particular, a single copy of the Rete network is held in shared memory. The match processes cooperate to pass tokens through the network and update the state stored in the memory nodes as indicated by the tokens. The match is broken into fairly small units of work called *tasks*, where a task is an independently schedulable unit of work that may be executed in parallel with other tasks. In our parallel implementation:

- All of the constant-test node activations constitute a single task. All these constant-test nodes are processed as a group, because individual constant-test node activations take only 2 machine instructions to execute, and that is too fine a granularity.
- The memory nodes in the Rete network are coalesced with the two-input nodes that are below them. Each activation of these coalesced two-input nodes constitutes a single task. The reasons for this coalescing are discussed in Ref. 13. As an example, the task corresponding to the left activation of a two-input node involves: (i) the addition/deletion of the incoming token to the left memory node; (ii) comparison of this token with all tokens in the opposite memory node checking for consistent variable bindings; and (iii) scheduling of matching token pairs for execution as new tasks. Note that multiple activations of the same two-input node constitute different tasks and these can be processed in parallel.
- Each individual terminal node activation constitutes a task.

[3] For simplicity, we are ignoring two kinds of optimizations that are possible. First, it is possible to overlap conflict-resolution with match. Second, if *speculative* parallelism is used (we are willing to be wrong in our prediction sometimes and know how to recover from the error), it is possible to make a guess about the production that will fire next and to evaluate its right-hand side before conflict-resolution is completely finished. We choose to ignore these two optimizations for the present, because conflict-resolution and RHS evaluation are not the bottlenecks in our current implementation.

In our current implementation, each task is represented by a data object called a *token*. The token in the parallel implementation is essentially the same as that used in the sequential Rete matcher (as described in Section 2.2), except that it has two extra items of information: the address of the node to which the token is to be sent, and if that node is a two-input node, an indication of whether to send it to the left or right input. The list of tokens that are awaiting processing is held in a central data structure called a *task queue*. The individual match processes perform match by executing the following loop.

1. Remove a token from the task queue. If the queue is empty, wait until something is added.
2. Process the token. If new tokens are to be sent out, push them onto the task queue.
3. Go to step 1.

3.2. Implementation Details

When studying parallelism in production systems (or in any other application for that matter), it is important to compute the speed-ups with respect to the performance of the most efficient uniprocessor implementations. It is indeed quite easy to obtain large speed-ups with respect to inefficient implementations of the application, but such results have little practical utility. In the case of OPS5, the most efficient uniprocessor implementations are currently based on the Rete algorithm and they compile the Rete network into machine code and use global register allocation. Such compilation into machine code gives approximately 10–20 fold speed-up over Rete-based lisp implementations of OPS5 (see Table I). For this reason, our parallel implementation of OPS5 on the Encore is also Rete-based and compiles the Rete network directly into machine code.[4] Another effect of parallelizing a highly efficient implementation versus an inefficient one is that the number of instructions executed in each parallel subtask (for the same task decomposition) is smaller in the highly efficient implementation. This is equivalent to exploiting parallelism at a finer granularity, and as a result, the issues of synchronization and scheduling are more critical.

As stated in the previous paragraph, the nodes in the Rete network are compiled directly into NS32032 machine code. Some of the operations per-

[4] Note that the argument in the beginning of this paragraph does not say that one has to use the same algorithm (as the most efficient uniprocessor one) for the parallel implementation. It just turns out in our case, that the efficient uniprocessor algorithm is also very good for parallel implementation.[13]

formed by the nodes are too complex to make it reasonable to compile the necessary code in-line. For these operations, subroutine calls are compiled into the network. The subroutines themselves are coded in *C* and assembler. For example, a two-input node is compiled into a combination of subroutine calls for modifying and searching through the node memories plus

```
!!! Rule for which code is presented below
(p p2
   (c1 ^a1 7   ^a2 <x>   ^a3 <y>)
   (c2 ^a1 <x>  ^a2 15   ^a3 <y>)
  -->
   (write  fired successfully ))


_ops_rete_root:
        movd    r6, r4                  ! Register r4 gets pointer to wme
        movd    @_curdir, @_succdir     ! Successor_direc = Current_direc
        cmpd    4(r6),@ops_symbols+4    ! Test if class = c2
        bne     .L11                    ! If test fails try next node
        cmpd    12(r6), $30             ! Test if ^a2 = 15
        bne     .L11                    ! If test fails try next node
        addr    @.L13, r0               ! Push task on to task_queue to
        bsr     _PushTaskQueue          ! begin evaluation at .L13
        br      .L11                    ! Start evaluating next node
.L13:   movd    $0, r3                  ! v----------------------------v
        xord    16(r6), r3              ! Compute hash index for
        xord    8(r6), r3               ! token
        andd    $0xfff, r3              ! ^----------------------------^
        movd    @_curdir, @_succdir     ! v----------------------------v
        movd    $0, tos                 ! Code + procedure call to add/
        bsr     ?_rmem                  ! del token to right memory node
        adjspb  $-4                     ! ^----------------------------^
        cmpqd   $0, r0                  ! Done with node activation if
        bne     @LeaveBetaTask          ! matching conjugate token found.
        cmpqd   $0, 0(_ltokHT)[r3:d]    ! Done with node activation if
        beq     @LeaveBetaTask          ! opposite mem-node empty
.Ltop0: movd    $0, r2                  ! Lev-of-node-actvn as param in r2
        bsr     _ops_lnext              ! Locate next token in opp mem
        cmpqd   $0, r5                  ! If all tokens have been examined
        beq     @LeaveBetaTask          ! then exit
        bsr     .L12                    ! Evaluate two-inp node tests
        br      .Ltop0                  ! Loop back to get next token
.L11:   cmpd    4(r6),@ops_symbols      ! Test if class = c1
        .
        .
        .
.L12:   movd    0(r4), r2               ! v----------------------------v
        movd    8(r5), r1               ! Perform tests to check if vars
        cmpd    16(r2),12(r1)           ! are consistently bound.  If tests
        bne     .L16                    ! fail, then return immediately,
        cmpd    8(r2),8(r1)             ! else push successor nodes on
        bne     .L16                    ! task queue and then return
        br      .L17                    !
.L16:   ret     $0                      ! ^----------------------------^
.L17:   addr    .L18, r0                ! Push address of successor node
        bsr     _PushTaskQueue          ! activation on to task queue
        ret     $0                      ! and return
```

Fig. 3. Code generated for matching a production.

in-line code to perform the node's variable binding tests. The OPS5 compiler uses global register allocation to make the code significantly more efficient. For example, register *r*6 always contains the pointer to the working-memory element currently being matched. The NS32032 assembly code generated to perform match for a simple production is shown in Fig. 3. The code is presented here to provide a feel for the compiler and the level of optimization. For example, it shows that to evaluate a constant-test node it requires only 2 machine instructions, a compare followed by a branch. It is not essential to understand the code to understand the rest of the paper.

All communication between processes (both the match processes and the control process) takes place via shared memory. The virtual address spaces are set up so that the objects in shared memory have the same virtual address in every process. Hence processes can simply pass pointers around in essentially the same way routines within a single process can. For example, the tokens are created in shared memory, and the address of a given token is the same in every virtual address space in the system. Thus when a process places a token onto the central task queue, all it really has to do is to put the address of the token into the task queue. Figure 4 shows how the shared-memory is used to communicate between the various processes.

Synchronization within the program is handled explicitly by executing interlocked test-and-set instructions. The synchronization primitives

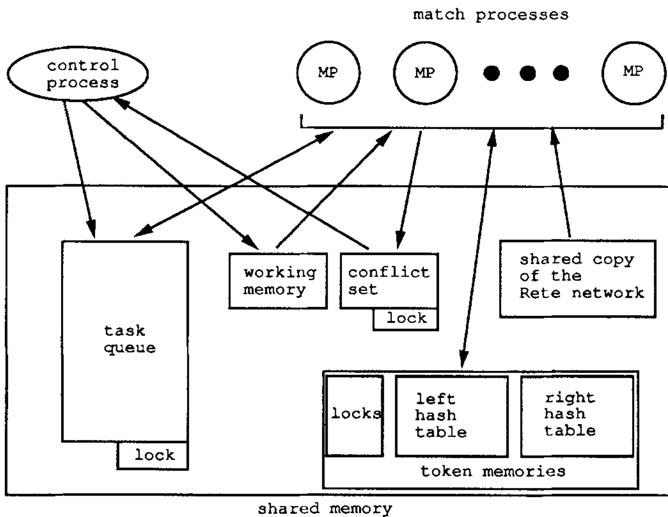

Fig. 4. Use of shared-memory by various processes.

provided by the operating system (for example, semaphores, barriers, signals, etc.,) are not used because of the large overhead associated with them. When a process finds that it is locked out of a critical region it spins on the lock, waiting for a chance to enter the region. In order to minimize the amount of bus traffic generated by the spinning processes, a "test and test-and-set" synchronization mechanism is used. In this scheme, a process uses ordinary memory-read instructions to test the status of a lock until it finds that it is free; then the process uses a test-and-set interlocked instruction to re-read the lock and set it (if it is still free). Note that while the lock is busy, the process spins out of its cache and does not use the bus. This is more efficient than using only the "test-and-set" interlocked instruction for the lock. In this case, the process generates bus traffic to perform the writes while it is busy waiting.

The control process communicates with the match processes primarily through the shared task queue. Whenever the evaluation of an RHS results in a change to working memory, a token is created and marked as being destined for the root node of the network. The control process pushes these tokens onto the task queue in exactly the same way as the match processes push the tokens they create. The tokens are picked up and processed by waiting match processes. When the evaluation of an RHS begins, the match processes are idle. The first token created by the control process causes the match processes to start up. After the first token, the control process proceeds in parallel with the match processes.

Depending on the granularity of tasks (number of instructions executed per task) that are scheduled using the task queue and depending on the number of processors that are trying to access the task queue in parallel, it is quite possible that a single task queue would become a bottleneck. For this reason, Gupta[13] proposed a hardware task scheduler for scheduling the fine-grained tasks. So far we have not implemented the hardware scheduler, and in this paper we present results only for the case when one or more software task queues are used.

After the control process finishes evaluating the RHS, it must wait for the match processes to finish before it can perform the next conflict resolution operation. A global counter, TaskCount, is used to determine when all the match processes have finished. This counter contains the sum of:

- the number of tokens that are currently on the task queue, and
- the number of tokens that are being processed by the match processes.

This count is maintained quite simply. Every time a token is put onto the task queue, the counter is incremented. Every time a match process finishes

working with a token, the counter is decremented. The match phase is finished when the counter goes to zero.

Shifting our focus back to the evaluation of individual two-input node activations, we note that instead of having separate memories for each two-input node, the matcher has two large hash tables which hold all the tokens for the entire network. One hash table holds tokens for left memories of two-input nodes, and the other for right memories of two-input nodes. An alternative scheme is to have separate hash tables for each two input node, but such a scheme was considered to be wasteful of space. The hash function that is applied to the tokens takes into account:

- The values in the token which will have equality tests applied at the two-input node, and
- The unique identifier of the two-input node which stored the tokens. The unique identifier is randomized to minimize the number of hash-table collisions.

This permits the two-input nodes to locate any tokens that are likely to pass the equal-variable tests quickly. It also permits multiple activations of the same two-input node to be processed in parallel.

The processing performed by the individual node activations in the parallel implementation is similar to the processing done in the sequential matcher with two exceptions:

- Code has been added to the two-input nodes to handle conjugate token pairs.
- Sections of code that access shared resources are protected by spin locks to insure that only one process at a time can be accessing each resource.

A *conjugate pair* is a pair of tokens with opposite signs (an add token request and a delete token request), but which refer to the same working memory element or list of working memory elements. Conjugate pairs arise in the match operation for a variety of reasons, which are too complex to go into here (see [Ref. 13]). They occur in both sequential and parallel implementations of Rete, but they present much greater problems in a parallel system. The reason for this is that in a parallel system it is not possible to insure that the tokens will be processed in the order in which they are generated, and consequently in some cases a token with a − (delete) flag will arrive at a two-input node before the corresponding token with the + (add) flag. The parallel matcher code handles this by saving the −tokens that arrive early on an *extra-deletes-list* without otherwise processing the token. When the corresponding +token arrives both tokens are discarded.

Many resources in a parallel system have to be protected with mutual-exclusion locks—the task queues, the count of the number of active tokens, the conflict set, etc. Most of these are relatively straight-forward to protect and a simple variation of standard spin locks is used. The exception is the locks used to control access to the token hash tables. There are several different operations that are performed on the token hash tables, for example, searching for matching tokens, adding and removing tokens, adding and removing conjugate tokens, and we would like many of these operations to proceed in parallel without having any undesirable effects. Because of the importance of the hash tables to the performance of the system, several locking schemes were implemented and tried. Two of these schemes are described here.

The first scheme, the simple one, is easy to describe and it provides a departure point for describing the second more complex one. We define a "line" as a pair of corresponding buckets (buckets with the same hash index) from the left and right hash tables along with their associated extra-deletes lists. In this scheme, each line in the hash table has a flag controlling its use.[5] The flag takes on two values: *Free* and *Taken*. When a process has to work with the hash table, it examines the flag for the line it needs. If the flag is *Free*, it sets the flag to *Taken* and proceeds to perform the necessary operations; when it finishes, it sets the flag back to *Free*. If a process finds the flag set to *Taken*, it waits until the flag is set to *Free*. Of course, the act of testing and setting the flag must be an atomic operation. This synchronization scheme works, but it is a potential bottleneck when several tokens arrive at a node about the same time, and if all of them require access to the same hash table line.

The second scheme is a complex variant of the *multiple-reader-single-writer* locking scheme. It permits several tokens to be processed in the same line at the same time, though even here, some serialization of the processing is necessary when destructive modifications to the lists of tokens are performed. This scheme requires two locks, a flag, and a counter for each line in the hash table. The flag takes on three values: *Unused*, *Left*, and *Right*, to indicate respectively that the line is not currently being processed, that it is being used to process tokens arriving from the left, or that it is being used to process tokens arriving from the right. The counter indicates how many processes are using that line in the hash table; it is needed only so that the last process to finish using the line can set the flag back to *Unused*. The first lock insures that only one process at a time can access the

[5] Note that any given operation on the token hash tables requires access to only a single line of the hash tables. In other words, processing a single node activation never requires access to multiple hash table lines.

flag and the counter. When a process first tries to use a line in the hash table, it gets this lock, and checks the flag. If the flag indicates that tokens from the other side are being processed, the process releases the lock and tries again. If the flag allows the process to continue, it sets the flag if necessary, increments the counter, and releases the lock. For the remaining time that the process uses this line in the hash table, it leaves the flag and the counter untouched; finally, when the process finishes using the line it decrements the counter and if appropriate sets the flag to *Unused* (again, all within a section of code protected by this lock). All this is to insure that tokens from two different sides are not processed at the same time. The second lock is used to insure that only one process at a time can be modifying the token lists. Recall that the first task in processing a two-input node is to update the list of tokens stored in the memory node. To do this, the process gets the modification lock, searches the conjugate or regular token list, and it either adds the token to or deletes it from one of these lists. When it has finished, it releases the modification lock and proceeds with searching the tokens in the opposite hash-table bucket to find those that satisfy the variable binding tests.

More complex locking schemes can be devised and, in fact, were implemented and tested. One other scheme that was tried permitted more than one process to search the token lists to find tokens to delete; in this scheme the only serialization of the tasks occurred when the actual destructive modification of the token list was performed. As in all implementations, the main tradeoff to keep in mind is that in an attempt to speed-up the rare cases, one should not slow down the normal case.

3.3. RHS Evaluation and Conflict Resolution

In our system, the rules' RHSs are compiled into a form of threaded code which is interpreted at run time.[14] Figure 5 shows a small piece of such threaded code. Interpreting the threaded code is slower than executing the compiled code, but since RHS evaluation is not a bottleneck to the performance, threaded code, which is simpler to compile was considered fast enough. Conflict resolution in the system is handled by code written in the *C* language. This code is executed by the control process.

4. RESULTS AND ANALYSIS

In this section, we present results obtained from the execution of three production-system programs. We first present some statistics from our uniprocessor implementation, and we then present the speed-ups obtained

```
p1:  #    -- p1 --                  ! Begin code for RHS of rule p1
          .double _bmake            ! Begin a make-wme action
          .double _symcon           ! v-----------------------------v
          .double ops_symbols       ! Set class of wme to c1
          .double _rval             ! ^-----------------------------^
          .double _tab              ! v-----------------------------v
          .double 4                 !
          .double _fixcon           ! Set 4th field of wme to 5
          .double 5                 !
          .double _rval             ! ^-----------------------------^
          .double _tab              ! v-----------------------------v
          .double 3                 !
          .double _fixcon           ! Set 3rd field of wme to 10
          .double 10                !
          .double _rval             ! ^-----------------------------^
          .double _emake            ! End of make-wme action
          .double _opsret           ! End code for RHS of rule p1
```

Fig. 5. Threaded code used to execute RHS actions.

by our parallel implementation. We also present a detailed analysis of the speed-ups observed. The three programs that we have studied are:

- Weaver,[15] a VLSI routing program by Rostom Joobbani with 637 rules.
- Rubik, a program that solves the Rubik's cube by James Allen with 70 rules.
- Tourney, a program that assigns match schedules for a tournament by Bill Barabash from DEC with 17 rules.

We have chosen Weaver because it represents a fairly large program and it demonstrates that our parallel OPS5 can handle real systems. Rubik is a smaller program that demonstrates some of the strengths of our parallel implementation and the Tourney program demonstrates some of the weaknesses of our parallel implementation.

4.1. Results for the Uniprocessor Implementations of OPS5

Before we did a parallel implementation on the Encore, we initially did several uniprocessor *C*-based implementations of OPS5. In this subsection, we present results for two of these uniprocessor implementations, *vs*1 and *vs*2, for the Microvax-II workstation.[6] The performance results for *vs*1 and *vs*2 implementations are shown in Table II. The base version is *vs*1, and it

[6] The results are presented for Microvax-II and not for Encore, because the uniprocessor implementations were done on the Microvax and only one of these was later taken over to the Encore.

Table II. Uniprocessor Versions on Microvax-II

PROGRAM	*VS*1 List-based memories (sec)	*VS*2 Hash-based memories (sec)	Total number of WM-changes processed	Total number of node activations
Weaver	101.5	85.8	1528	371173
Rubik	235.2	96.9	8350	554051
Tourney	323.7	93.5	987	72040

is characterized by the use of linear lists to store tokens in node memories, just as uniprocessor lisp implementations do.[7]

The second version, *vs*2, uses a global hash table to store all memory-node tokens, as discussed in the previous section. If there are equality tests at the two-input node, the hash-table based scheme (i) reduces the number of tokens that have to be examined in the opposite memory to locate those that have consistent variable bindings, and (ii) for deletes, it reduces the number of tokens that have to be examined in the same memory to locate the token to be deleted. The statistics for the reduction in tokens examined in the opposite memory for the three programs are given in Table III. Note the statistics are computed only for those node activations where the opposite memory is not empty. The statistics for the reduction in tokens examined in the same memory for delete requests are given in Table VI. As can be seen from the Tables III and IV, the savings are substantial,

[7] Note that memory nodes are not shared in either *vs*1 or *vs*2 versions of OPS5, unlike in the Franzlisp version of OPS5. This optimization was not used in *vs*1 or *vs*2 because it is not possible to share memory nodes in the parallel implementations of OPS5 (see [5]), and we did not want to spend the effort just for the uniprocessor implementations.

Table III. Number of Tokens Examined in Opposite Memory

	Tokens in opp mem for left actvns		Tokens in opp mem for right actvns	
PROGRAM	lin mem	hash mem	lin mem	hash mem
Weaver	10.1	7.7	5.2	1.0
Rubik	31.0	3.8	1.6	1.8
Tourney	47.6	5.9	270.1	23.3

Table IV. Number of Tokens Examined in Same Memory for Deletes

	Tokens in same mem for left actvns		Tokens in same mem for right actvns	
PROGRAM	lin mem	hash mem	lin mem	hash mem
Weaver	6.2	3.6	7.0	5.1
Rubik	23.5	2.6	8.1	3.7
Tourney	254.4	40.1	3.8	2.9

especially for the Tourney program. The time-saving effect of hash-based memories can be seen from numbers in Table II.

The second last column in Table II gives the total number of wme-changes processed during the run for which data are presented, and the last column gives the total number of node activations processed during the run (this is also equal to the number of tasks that are pushed/popped from the task queue in the parallel version). Dividing the time in column *vs*2 by the number of tasks, we get the average duration for which a task executes. This has implications for the amount of synchronization and scheduling overhead that may be tolerated in the parallel implementation. Doing this division we get that the average duration of a task for Weaver is 230 microseconds (or approximately 115 machine instructions, as the VAX executes about 500,000 instructions per second), for Rubik is 175 microseconds, and for Tourney is 1300 microseconds.

4.2. Results for the Multiprocessor Implementation of OPS5

While the uniprocessor *C*-based implementations of OPS5 were done on the Microvax-II, the parallel version was done on the Encore multiprocessor. In this subsection, we present speed-up numbers for our implementation on the Encore as we vary (i) the number of task queues that are used and (ii) the locking structures used for token hash-table buckets. The speed-ups reported here are with respect to a parallel version of the program running on a uniprocessor. We also present a detailed analysis of the speed-ups observed.

Figure 6 shows results for the case when a single task queue is used and when *simple* locks (described in Section 3.2) are used with the token hash-table buckets. Figure 6 also shows the uniprocessor times (in seconds) for the three programs. Note that the numbers along the X-axis represent the number of match processes; they do not include the control process.

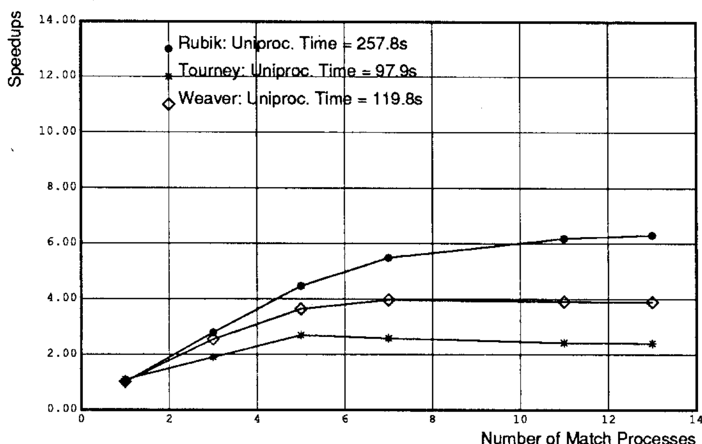

Fig. 6. Speed-up for single task queue and simple hash-table locks.

The speed-ups for all three programs are quite disappointing. This is especially true for Tourney, where the maximum speed-up is 2.6-fold with 5 match processes and it decreases even further as the number of match processes is increased.

There are several possible reasons for the low speed-up: (i) contention for access to the single task queue, (ii) contention for access to the hash-table buckets, (iii) low intrinsic parallelism in the programs, (iv) contention for hardware resources, and so on. We now explore the effects of removing the first two bottlenecks and provide some data on the intrinsic parallelism in the programs.

4.2.1. Reducing Contention for the Centralized Task Queue

The contention for the single task queue can be reduced by the introduction of *multiple task queues*. Every process has its own queue, onto which it pushes and pops tasks. If it runs out of tasks then it cycles through the other processes' task queues, searching for a new task. Figure 7 presents the speed-up obtained when multiple task queues are used, while still using simple hash-table locks. The speed-up increases significantly for both Weaver and Rubik, indicating that the contention for pushing and popping task queues must have been a bottleneck. The speed-up for Weaver for 13 processes goes up from 3.9-fold to 8.2-fold and that for Rubik goes up from 6.3-fold to 11.4-fold. The speed-up for Tourney remains about the same at 2.4-fold.

To get more insight into these results, we instrumented the task queue

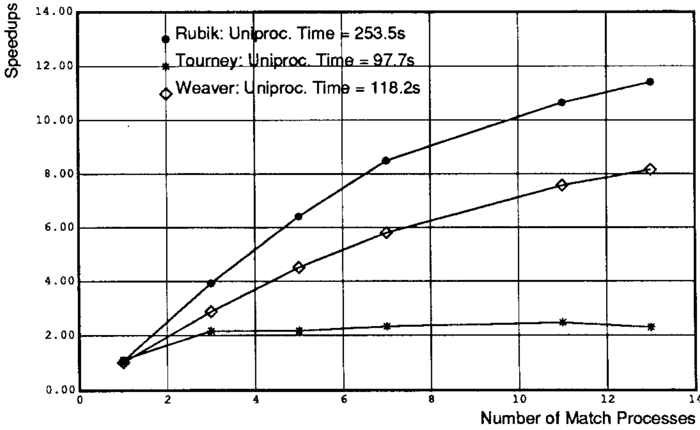

Fig. 7. Speed-up for multiple task queues and simple hash-table locks.

to get data on contention. The results are shown in Fig. 8. Here we plot the number of times a process spins on the task-queue lock as a function of the number of match processes. We see from Fig. 8 that as the number of processes is increased, there is indeed significant contention for the single task queue in case of Weaver and Rubik. For Tourney, there does not seem to be as much contention for the task queue, and that is why the speed-up does not increase when multiple task queues are used. Since the speed-up is still very low for Tourney (only 2.4-fold with 13 processes), in the next subsection we will explore if contention for the hash-table buckets is causing the poor speed-up.

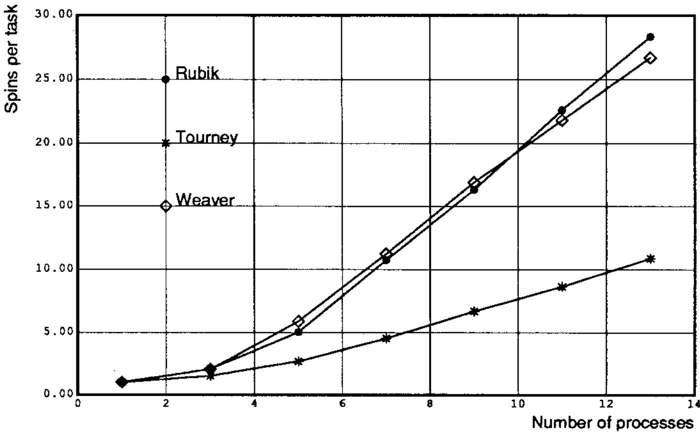

Fig. 8. Contention for the centralized task queue. Measured by the number of times a process spins on the lock before it gets access to the task queue.

Another question that arises when using multiple task queues is: "How many task queues should one use to maximize speed-up?". For example, when using 12 match processes, should we have 2, 4, 8, or 12 task queues. Too few task queues have the disadvantage that the contention for the task queues may still be a bottleneck. An excessive number of task queues has the disadvantage that most of the task queues will be empty, and the processes will waste time scanning several empty task queues before finding one with a task. Figure 9 plots the speed-up obtained for 12 match processes, as the number of task queues is increased. What the graph shows is that the optimal number of task queues varies for different programs. For Rubik, the more the task queues the better the speed-up. For Weaver, the speed-up increases up to 4 task queues, then remains the same up to 8 task queues, and then decreases slowly. For Tourney, the number of task queues really does not seem to matter. However, as a design decision, it seems that erring on the side of too many task queues is better than having too few task queues.

Finally, examining the speed-up for Rubik in Fig. 7, it is interesting to note that we get 3.9-fold speed-up using only 3 match processes. This apparently anomalous behavior of the speed-up being greater than the number of match processes can be explained as follows. When the Rete network is evaluated in parallel, it is quite possible that the total number of node activations evaluated and their complexity is less than that of the sequential implementation. Of course, the final result of the match is still the same as the sequential implementation.

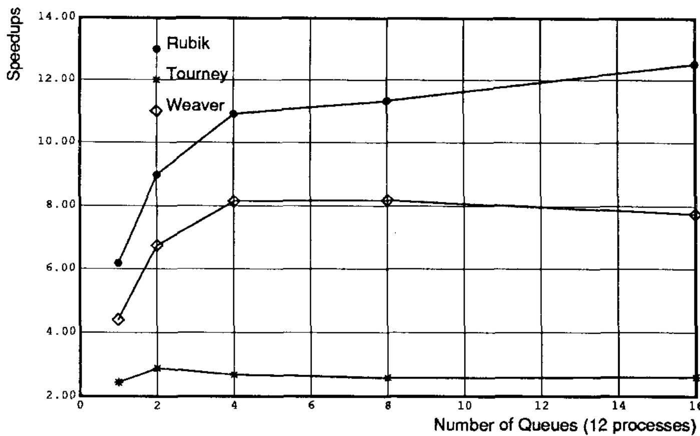

Fig. 9. Speed-up with 12 match processes as the number of task queues is varied.

4.2.2. Reducing Contention for the Hash-Table Buckets

As discussed in Section 3.2, tokens associated with memory nodes in the Rete network are stored in two large hash tables. These hash tables are shared among all the processes. A single lock controls the access to a line, i.e., a pair of corresponding buckets from left and right hash tables. This lock provides a spot of contention for the various processes. The contention for a hash bucket lock can be measured by the number of times a process spins on a lock before it gets access to a line of hash table buckets. For right tokens, that activate the two-input nodes along the right inputs, the contention for the lock is very low—1 or 2 spins per access—for all three programs, and it does not change as the number of processes is increased. This is because the right tokens are distributed evenly and most right tokens typically require very little processing.[13] The right and the left tokens do not typically contend with each other, as the right tokens are evaluated in the beginning of the cycle; while the left tokens are evaluated only later in the cycle.

For the left tokens, which activate the two input nodes from their left inputs, the contention is much higher. Figure 10 shows the contention observed by left tokens when the per-bucket lock used is a simple one (an ordinary spin lock), as discussed in Section 3.2. With 11 match processes, Rubik processes spin 23 times, Tourney processes spin 377 times, and Weaver processes spin 51 times on average before getting access to the hash-table bucket. While the contention for Rubik and Weaver is quite

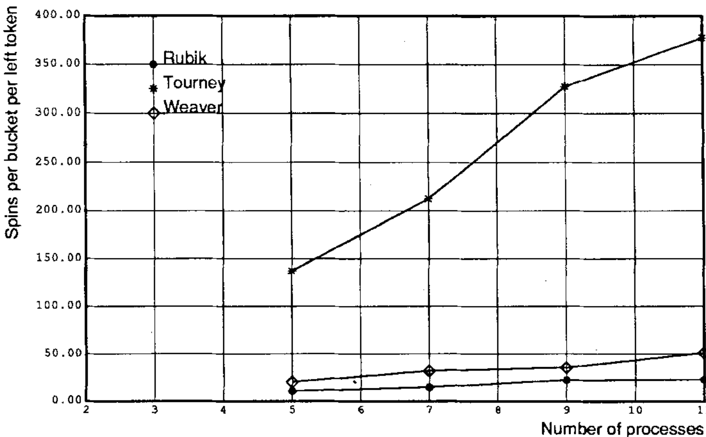

Fig. 10. Contention for hash-table locks for left tokens. Measured by the number of times a process spins on a lock before it gets access to the hash-table bucket.

high, it is enormous for Tourney given that each spin takes several microseconds.

In Fig. 11, we present results for the case when multiple task queues are used and when *complex multiple-reader-single-writer* locks (described in Section 3.2) are used for controlling entry to the token hash tables. We expected the complex locks to benefit those programs that (i) generate cross-products, that is, there are multiple activations of the same two-input node from the same side that need concurrent processing, and (ii) have long lists of tokens in hash-table buckets, where the complex locks help by allowing multiple processes to read the opposite memory at the same time. However, programs for which the previous two conditions are not true may slow down when complex locks are used, because of the extra overhead that they incur due to complex locks. Figure 12 presents some results about contention when complex locks are used. Comparing with Fig. 10, we see that the contention for the hash-table buckets decreases for all three programs, although the contention for Tourney is still very high in alsolute terms. Analyzing the Tourney program in more detail, we found that the large contention for the hash-table locks is resulting from multiple node activations trying to access the same hash-table bucket. This in turn, is the result of a few culprit productions in Tourney that have condition elements with no common variables. By modifying two such productions using domain specific knowledge, we could increase the speed-up achieved using 13 processes from 2.7-fold to 5.1-fold.

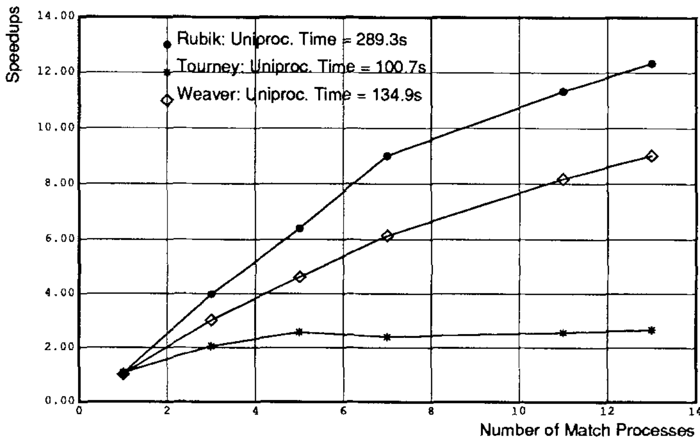

Fig. 11. Speed-up for multiple task queues and multiple-reader-single-writer hash-table locks.

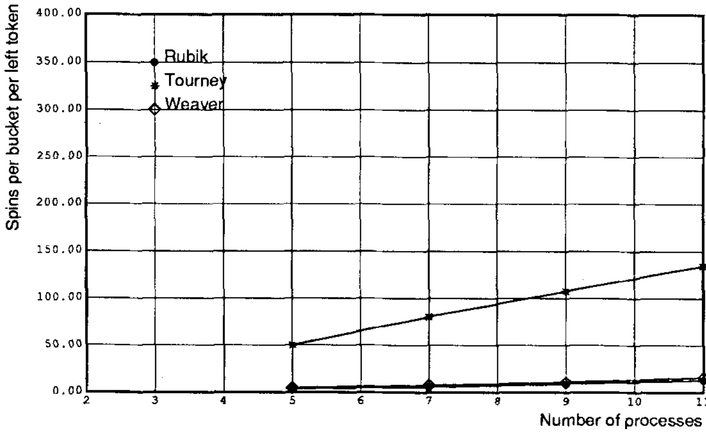

Fig. 12. Contention for hash-table locks when multiple-reader single-writer locks are used.

4.2.3. Other Causes for Low Speed-Up

In this subsection, we explore reasons other than contention for shared-memory objects that limit the speed-up achieved by our implementation. To this end, we examine the speed-ups obtained in individual recognize-act cycles for the programs. (Recall that the computation of an OPS5 program involves a series of recognize-act cycles.) Figure 13 presents the speed-ups obtained in each cycle as a function of the number of tasks (node activations) executed in that cycle.[8] These numbers are presented for Weaver; but they are representative of other OPS5 programs too. The speed-ups were measured with 11 match processes and the implementation used multiple task queues and simple hash-table locks. The 7.5-fold speedup obtained for weaver (shown in Fig. 7), is a weighted average of the speedups for individual cycles shown in Fig. 13.

The data points in Fig. 13 can be divided into two regions:

- Short Cycles Region: Points in the left quarter of the graph (corresponding to cycles with less than 250 tasks), generally achieving a speed-up of 2 to 7-fold.
- Long Cycles Region: Points in the right three quarters of the graph (corresponding to cycles with 250–1000 tasks), generally achieving a speed-up of 6 to 10-fold.

[8] For presentation purposes, the cycles with more than 1000 tasks are shown as containing 1000 tasks.

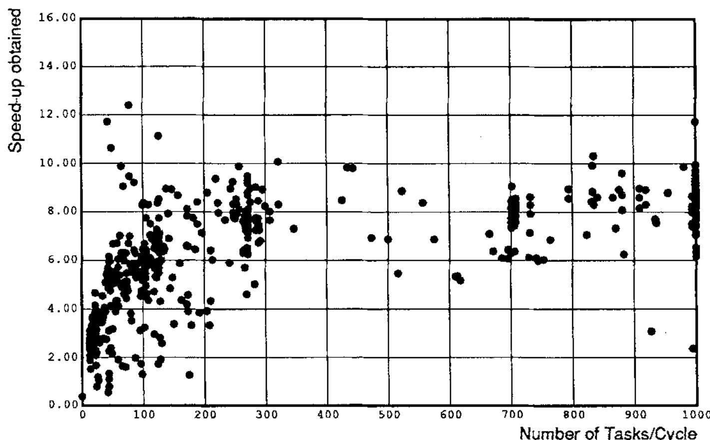

Fig. 13. Weaver: Speed-up as a function of tasks/cycle.

Let us first look at short cycles more closely. To understand the speed-up achieved, we plot the number of tasks in the system (which is the sum of the number of tasks waiting to be processed and these being processed) as a function of time during a cycle. Figure 14 shows such a plot for one of the short cycles in Weaver with about 100 tasks. The graph is plotted for 11 match processes and the time is measured in units of 100 microseconds. The graph may be interpreted as showing the number of processors that

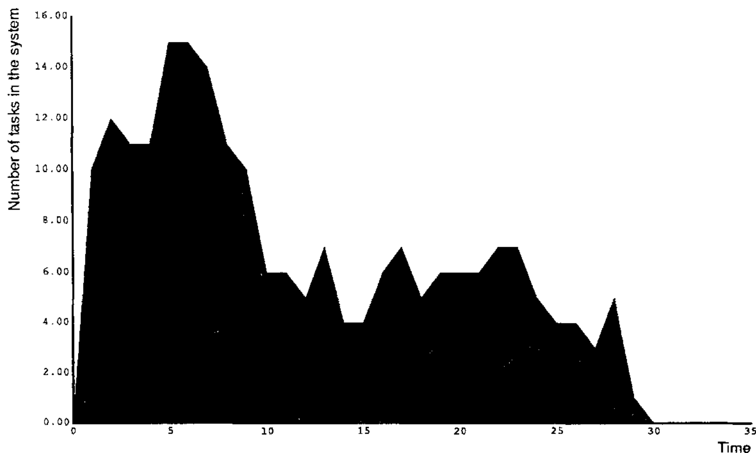

Fig. 14. Weaver: Number of node activations available for parallel processing as a function of time during a *short* cycle. Each unit of time on the X-axis corresponds to $100\mu s$.

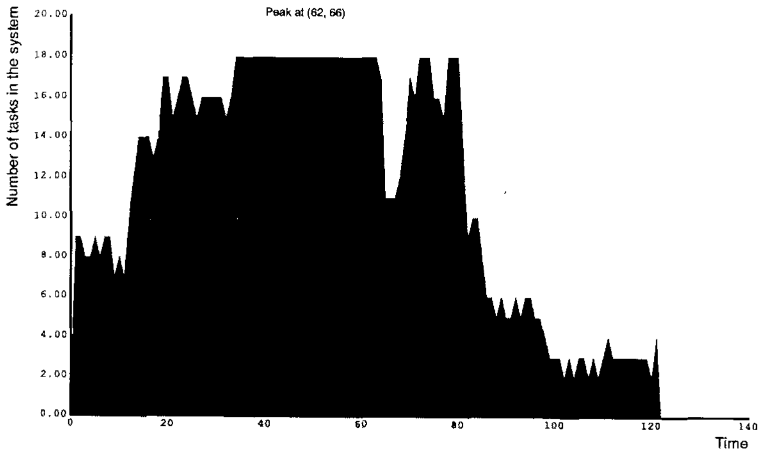

Fig. 15. Weaver: Number of node activations available for parallel processing as a function of time during a *long* cycle.

can be kept busy if infinite processors were present.[9] The average height of this graph (about 6.5 for this cycle) indicates the maximum speed-up that we should expect. In general, the smaller cycles tend to provide such low available parallelism.

Some additional speed-up losses occur due to the fixed overheads associated with match cycles; for example, having to check all the task queues to ensure that the match is actually finished and to inform the control process about the completion of the match. These almost fixed duration overheads affect the speed-up obtained by small cycles much more than that obtained by large cycles, as they form a larger fraction of the small cycle processing cost.

We now explore speed-up limits in long cycles. In Fig. 15, we show a plot similar to that in Fig. 14, except this time for a longer cycle with about 300 tasks. We see that in the early part of the graph (until time 30) the potential parallelism increases slowly, then it rises very steeply peaking at the point (62,66), then it falls rapidly (until time 65), and finally it has a slow spiky decline to the end of cycle (time 120). The portion of the graph that hurts the average speed-up most, however, is the portión from time 90 to time 120, where the system keeps processing a few tasks; each time generating only a few tasks. This behavior is caused by the presence of chains of dependent node activations,[13] which can get especially bad for

[9] This interpretation is not totally correct for portions of the graph where the height of the shaded region is greater than the number of match processes used to produce the graph.

productions that have a large number of condition elements. The impact of long chains on speed-ups increases with increasing number of processes. With more processes, the system can get through the earlier part of the computation (the one marked up to the first 90 time units, in Fig. 15) faster, but it cannot get through the latter part much faster. To counter these long chains, we plan to change the Rete network organization for productions with large number of condition elements. This new network organization is called a *constrained bilinear* organization and it will allow us to reduce the dependencies between tokens (see Refs. 13 and 16 for details).

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented the details of a parallel implementation of OPS5 running on the Encore Multimax. The first observation is that it is important to speed-up an *optimized* sequential implementation, otherwise most of the benefits are lost. For example, speeding-up the Franzlisp implementation by 10–20 fold from parallelism would just bring us to the uniprocessor speed of the *C*-based implementation. Furthermore, the issues in parallelizing an optimized implementation are different from those in an unoptimized implementation, because only very limited oveerheads can be tolerated in an optimized implementation.

The second observation we make is that it is possible to obtain significant speed-ups for OPS5 using fine-grained parallelism on a shared-memory multiprocessor. However, this does not work for all programs. The Tourney program, because of the presence of short cycles and cross-products resisted all our attempts to obtain higher speed-up.

Our third observation is regarding the contention for shared memory objects. The average length of the individual tasks in our parallel implementation varies between 100–700 machine instructions for the three programs that we studied. In trying to exploit this fine-grained parallelism, we found that scheduling tasks using a single task queue formed a major bottleneck for the system. We found it essential to use multiple task queues (instead of a single task queue) to obtain reasonable speed-up. For the Rubik program, going from one task queue to multiple task queues increased the speed-up from 6.3-fold to 11.4-fold.

The other variation that we explored to reduce the contention for shared data structures was in the complexity of locks used for hash-based memory nodes. We used both simple spin-locks and complex multiple-reader-single-writer locks. We observed that special note must be taken of *rare-case* versus *normal-case* execution. Trying to handle rare cases efficiently can slow down the normal case, and can result in overall poorer

performance. For example, the provision of complex hash-table locks reduced the contention for the hash-table buckets, but it slowed down the overall execution speed of the Rubik program.

In the future, we plan to investigate alternative computer architectures for implementing production systems; especially the message-passing architectures. Our analysis indicates that the message-passing architectures are quite suitable for implementing production systems.[17] Currently, simulations of implementing production systems on such machines are in progress.

Our other direction of investigation has been an exploration of the parallelism in Soar,[18] a leaming production system. The parallelism in Soar is expected to be higher than OPS5.[13] Our current implementation of Soar on the Encore Multimax has provided good speedups in the match.[16] The next step there is to parallelize other areas of Soar besides match.

6. ACKNOWLEDGMENTS

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Air Force Avionics Laboratory under Contract N00039-85-C-0134 and by the Encore Computer Corporation. Anoop Gupta is also supported by DARPA contract MDA903-83-C-0335 and an award from the Digital Equipment Corporation.

REFERENCES

1. Charles L. Forgy, *The OPS83 Report*, Technical Report CMU-CS-84-133, Carnegie-Mellon University, Pittsburgh, (May 1984).
2. Theodore F. Lehr, The Implementation of a Production System Machine, In *Hawaii International Conference on System Sciences* (January 1986).
3. P. L. Butler, J. D. Allen, and D. W. Bouldin, Parallel Architecture for OPS5, In *Proceedings of the Fifteenth International Symposium on Computer Architecture*, pp. 452–457 (1988).
4. Anoop Gupta, Charles Forgy, Allen Newell, and Robert Wedig, Parallel Algorithms and Architectures for Production Systems, In *13th International Symposium on Computer Architecture.* (June 1986)
5. Bruce K. Hillyer and David E. Shaw, Execution of OPS5 Production Systems on a Massively Parallel Machine, *Journal of Parallel and Distributed Computing* **3**:236–268 (1986).
6. Daniel P. Miranker, *TREAT: A New and Efficient Algorithm for AI Production Systems*, PhD thesis, Columbia University (1987).
7. Edward J. Krall and Patrick F. McGehearty, A Case Study of Parallel Execution of a Rule-Based Expert System, *International Journal of Parallel Programming* **15**(1):5–32 (1986).

8. Kemal Oflazer, Parallel Execution of Production Systems, In *International Conference on Parallel Processing*. IEEE (August 1984).
9. Raja Ramnarayan, Gerhard Zimmerman, and Stanley Krolikoski, PESA-1: A Parallel Architecture for OPS5 Production Systems, In *Hawaii International Conference on System Sciences* (January 1986).
10. M. F. M. Tenorio and D. I. Moldovan, Mapping Production Systems into Multiprocessors, In *International Conference on Parallel Processing* IEEE (1985).
11. Lee Brownston, Robert Farell, Elaine Kant, and Nancy Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley (1985).
12. Pandurang Nayak, Anoop Gupta, and Paul Rosenbloom, Comparison of the Rete and Treat Production Matchers for SOAR, In *National Conference on Artificial Intelligence*, AAAI-88.
13. Anoop Gupta, *Parallelism in Production Systems*, PhD thesis, Carnegie-Mellon University, (March 1986); also available from Morgan Kaufmann Publishers Inc.
14. Peter M. Kogge, An Architectural Trail to Threaded-Code Systems, *Computer* (March 1982).
15. Rostam Joobbani and Daniel P. Siewiorek, Weaver: A Knowledge-Based Routing Expert, In *Design Automation Conference* (1985).
16. Milind Tambe, Dirk Kalp, Anoop Gupta, Charles Forgy, Brian Milnes, and Allen Newell, Soar/PSM-E: Investigating Match Parallelism in a Learning Production System, In *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pp. 146–161 (July 1988).
17. Anoop Gupta and Milind Tambe, Suitability of Message Passing Computers for Implementing Production Systems, In *National Conference on Artificial Intelligence*, AAAI-88.
18. John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An Architecture for General Intelligence, *Artificial Intelligence* **33**:1–64 (1987).