Soar/PSM-E: Investigating Match Parallelism in a Learning Production System

Milind Tambe, Dirk Kalp, Anoop Gupta[1], Charles Forgy, Brian Milnes, Allen Newell

Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa 15213

Abstract

Soar is an attempt to realize a set of hypotheses on the nature of general intelligence within a single system. Soar uses a production system (rule based system) to encode its knowledge base. Its learning mechanism, chunking, adds productions continuously to the production system. The process of searching for relevant knowledge, matching, is known to be a performance bottleneck in production systems. PSM-E is a C-based implementation of the OPS5 production system on the Encore Multimax that has achieved significant speedups in matching. In this paper we describe our implementation, Soar/PSM-E, of Soar on the Encore Multimax that is built on top of PSM-E. We first describe the extensions and modifications required to PSM-E in order to support Soar, especially the capability of adding productions at run time as required by chunking. We present the speedups obtained on Soar/PSM-E and discuss some effects of chunking on parallelism. We also analyze the performance of the system and identify the bottlenecks limiting parallelism. Finally, we discuss the work in progress to deal with some of them.

1. Introduction

Soar is an architecture for a system that is to be capable of general intelligence [8]. Its development started as an AI system in 1981 and it is now also under exploration as a model of human cognition [14]. Soar has been exercised on a large variety of tasks: many of the classic AI mini tasks such as the blocks world and the Towers of Hanoi, as well as large tasks such as the R1 computer configuration task [16], the Neomycin medical diagnosis task [19] and the Cypress algorithm design task [18]. Soar exhibits a wide range of problem-solving, learning and human-like performance.

Soar uses a production system that provides a single representation for its knowledge base. Each knowledge base item is represented by a condition-action rule, or production, that *fires* whenever its conditions *match* elements in working memory. Soar uses *chunking* [9] as its sole learning mechanism; chunking creates new productions, *chunks*, that summarize the results of problem solving and adds them to the existing set of productions. These chunks then fire in appropriate later situations, providing a learning-transfer mechanism.

Large and complex systems built in Soar are slow in their execution; this limits their utility. The dominating factor in this slowdown is the production matching procedure. As chunking adds new productions, the demands of matching increase, so it is important to optimize the match as much as possible. Researchers have been exploring many alternative ways of speeding up the execution of the matching procedure in production systems: efforts have focused on high-performance uniprocessor implementations [4, 10], as well as parallel implementations [5, 6, 17, 12, 15, 21]. Most results for parallel processing have been simulation results or the results of parallelization of slow Lisp-based implementations. The PSM-E implementation [6] of the OPS5 production system [2] on the Encore Multimax multiprocessor, differs from other efforts: its C-based compiler writes highly optimized machine code to achieve significant speedups on actual production systems[2].

In this paper, we describe the Soar/PSM-E implementation of Soar, that uses PSM-E for match. We investigate the available parallelism of the matching procedure for Soar systems. Soar's chunking mechanism provides a new dimension in the study of parallel production system match that is absent in other investigations concerned with parallelism in non-learning production systems. Chunking continuously creates

[1] Department of Computer Science, Stanford University, Stanford, CA. 94305

[2] Certain production systems showed between ten to twenty times speedup over the original CMU Lisp implementation of OPS5 [6].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

new productions; Soar's production system must be able to incrementally compile chunks without large overheads, as all the gains of a highly optimized system such as PSM-E could be lost by such overheads. As chunks arise automatically as a result of the problem-solving, they present various computational phenomena that do not appear in non-learning production systems [20]. Soar/PSM-E provides a useful tool for studying the previously unexplored effects of chunking on parallelism.

This paper is focused on the parallel processing aspects of the Soar matcher, using the C-based PSM-E implementation. Hence, all measurements presented are of the match time. Other mechanisms of Soar, that have been left in their original Lisp form, are unoptimized and dominate the system's total run time; they will be the subject of future work.

The paper is organized as follows: Section 2 presents background information on the OPS5 production system, the matching procedure used in OPS5 and the PSM-E implementation. Section 3 presents a brief overview of Soar. Section 4 describes the structure of the Soar/PSM-E system. Section 5 explains the extensions to PSM-E for Soar and discusses the various tradeoffs involved. Section 6 presents the results of our measurements of Soar programs and analyzes the factors limiting our speedups. Section 7 proposes future work to improve the match speedups for Soar programs.

2. Background

Soar uses a variation of OPS5 as its production system. It also uses OPS5's RETE matching algorithm. In this section we briefly describe the OPS5 production system language and the RETE matching algorithm. We then present a brief overview of the PSM-E implementation of OPS5. Readers familiar with OPS5, RETE or our previous publications on the PSM-E system may wish to skip one or more of these subsections.

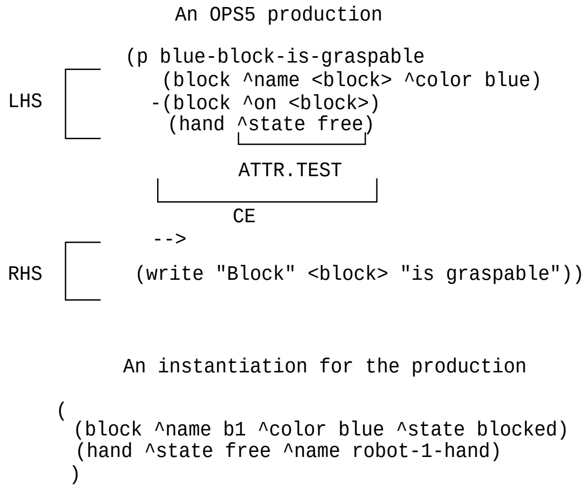
2.1. OPS5

An OPS5 production system is composed of a database of temporary assertions, called the *working memory* (*WM*), and a set of *if-then* rules, or productions, that constitute the *production memory* (*PM*). The assertions in WM, called *working memory elements* (wmes), are record structures with a fixed set of named access functions, called attributes, much like Pascal records. Each production is a list of condition elements (CEs), corresponding to the *if* part of the rule (the left-hand side or LHS), and a set of actions corresponding to the *then* part of the rule (the right-hand side or RHS).

A CE is a pattern that tests for the existence, or absence, of a wme in WM. A CE may be optionally negated, i.e., preceded with a dash ("–"), signifying that it tests for the absence of any matching wme. Each of the non-negated CEs of a production must match a wme, and none of the negated CEs may match, before the production is ready to fire, or is *satisfied*.

Each CE is composed of a set of tests for a wme's attributes' values. All of the CE's attribute value tests must be matched for that wme to match the CE. There are two types of attribute tests: constant and equality. Constant tests check that an attribute of a wme holds some constant, usually a symbol or number. Most attribute tests are constant tests. An equality test binds a variable (syntactically any symbol enclosed in "<" and ">") to an attribute's value in the scope of a single production. The variables in these tests will, on first occurrence, match any value, but if the variable appears again in a later test, then the attribute tested must hold the same value. When the CEs of a production are all matched in conjunction by some list of wmes, an *instantiation* of that production, which is the list of the matching wmes, is created and added to the *conflict set* (*CS*). OPS5 uses a selection procedure called *conflict resolution* to choose a single production's instantiation from the CS, which is then *fired*. When a production fires, the RHS *actions* associated with that production are executed, in the context of the LHS's variable bindings. Actions add or remove wmes and perform input/output.

Production systems repeatedly cycle through three phases: match, select and fire. The matcher first updates the CS with all of the current matches for the productions. Conflict resolution selects one of these instantiations, removes it, and then fires it. Figure 2-1 displays an OPS5 production and an instantiation for the production.



```
(p blue-block-is-graspable
   (block ^name <block> ^color blue)
  -(block ^on <block>)
   (hand ^state free)
  -->
  (write "Block" <block> "is graspable"))
```

```
(
 (block ^name b1 ^color blue ^state blocked)
 (hand ^state free ^name robot-1-hand)
)
```

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

2.2. The RETE Matching Algorithm

The RETE matching algorithm [3] is a highly efficient algorithm for match that is also suitable for parallel implementation. The algorithm gains its efficiency from two sources. First, it stores the partial results of match from previous cycles for use in subsequent cycles, exploiting the fact that only a small fraction of WM changes on each cycle. Second, it attempts to perform tests common to CEs of the same and different productions, only once by *sharing* them in a directed acyclic graph structure, or network.

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

```
(p blue-block-is-graspable
   (block ^name <block> ^color blue)
  -(block ^on <block>)
   (hand ^state free)
   --> (modify 1 ^state graspable))
```

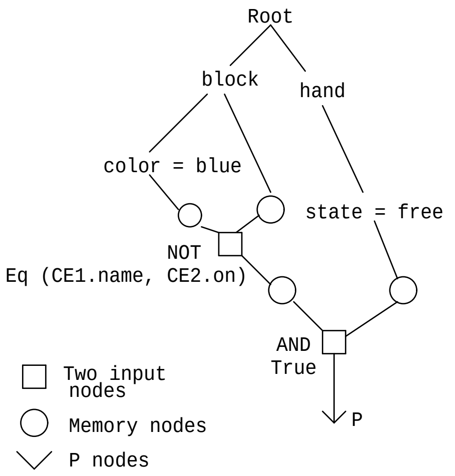


Figure 2-2: An example production and its network.

This data-flow network passes items called *tokens* across its arcs between its nodes. Each node has one or two input arcs and zero or one output arc. There are three types of one input nodes: *constant*, *production (P)* and *memory*; and two types of two-input nodes: *and* and *not*. Each token contains an add or delete flag and a *partial instantiation* (*PI*), i.e., a list of wmes, matching CEs.

1. *Constant* test nodes implement the constant attribute tests. The top of the network is composed only of these and forms a network that discriminates wmes based on the constants they contain. Each constant node receives a token, on its one input, that contains only one wme in its PI. If its wme has a certain constant for some attribute's value, then it sends it out its single output arc.

2. *P* nodes are the terminal nodes of the network. They add and delete instantiations of productions from the CS, and so need no output link. When a token arrives on its input, if it has an add flag, its PI is added to the CS as an instantiation for its production. If the token has a delete flag and it has not yet been removed by being fired, the production node removes the PI from the CS.

3. *Memory* nodes hold the partial state of the match by storing the set of the PIs whose tokens have appeared at its single input link. If the token, on its one input arc, has the add flag, its PI is added to the set; with the delete flag it is removed. Memories always place their input token on their output arc. These nodes make RETE a state saving algorithm: it need only see the changes to WM on each cycle to calculate the CS's contents.

4. *And* nodes combine PIs into larger PIs. These nodes have two input arcs, called left and right; each must be preceded by a memory node. When a token arrives at an arc, its PI is compared with all of the PIs in the opposite arc's preceding memory. Any pair which preserve the variable bindings of the production that compiled into the node are joined into a new PI. These new PIs are sent out the output link for further matching. The and node ensures that the value bound to every variable in a CE is consistent with the value bound to the same variable in the preceding CEs of that production. And nodes join the wmes that match CEs in a left to right *linear fashion*.

5. *Not* nodes also have two input arcs: left and right. Not nodes keep a memory of the items that have passed in their left input and count the number of items in the their right arc's preceding memory, which match them. A not puts its input item from its left link onto its output only if there are no PIs in the right memory that block it by matching. Not nodes implement negated conditions by only passing partial instantiations that do not have a match for a negated CE.

The majority of node activations are for constant test nodes. However with suitable indexing techniques, these may be reduced by almost half [5]. Since one-input nodes are much simpler to execute than than two input nodes, close to 90% of the processing time in an optimized implementation is spent in the two-input nodes.

2.3. PSM-E: Parallel Implementation of OPS5 on the Encore Multimax

PSM-E [6] is a highly optimized C-based parallel implementation of the OPS5 production system on the Encore Multimax[3]. It produces a machine coded version of the RETE data-flow network. Before starting a run, the PSM

[3] The Encore Multimax is a 16-CPU shared memory multiprocessor. The machine employed for this research used the NS32032 processor, which runs at approximately 0.75 MIPS.

compiler generates a tree structured representation of the RETE network for the current set of productions. This data structure is used to generate OPS83 [4] style assembly code for the network. The system then uses the assembling, linking and loading facilities provided by the operating system to create the executable image.

PSM-E consists of one *control process* that selects and then fires an instantiation and one or more *match processes* that actually perform the RETE match. PSM-E exploits parallelism at the granularity of *node activations*. Previous work has demonstrated that to achieve significant speedups via parallelism in production systems, it is necessary to exploit parallelism at a very fine granularity [5]. A node activation consists of the address of the code for a node in the RETE network and an input token for that node. These node activations are called *tasks* and are held in one or more shared *task queues*. Each individual match process performs match by picking up a task from one of these queues, processing the task and, if any new tasks are generated, pushing them onto one of the queues. When the task queues becomes empty, one production system cycle ends; the control process applies conflict resolution to select and fire an instantiation from the CS. Exploiting parallelism at the level of node activations allows PSM-E to achieve significant speedup for match using up to 13 processes [6].

3. Soar

The goal of the Soar project is to build a system capable of general intelligent action. Soar is based on the problem space hypothesis [13], which states that all goal-oriented behavior is search in problem spaces. The problem space determines the set of legal states and operators that can be used during the processing to attain the goal. The states represent situations. There is an initial state representing the initial situation, and a set of desired final states. An operator is applied to a state in the problem space to yield a new state for that problem space. When an operator application generates a desired state, the goal is achieved.

Soar is composed of three modules: Decide, chunking and a production system. Decide is a universal subgoaling mechanism [7], and is responsible for the creation and deletion of all of the system's goals, as well as the selection of problem spaces, states and operators. We also refer to Decide as the performance system of Soar. Decide works in a two phase loop: elaborate and decide. In the elaboration phase, all the productions in the system are matched (by the production system) to determine the CS. However, unlike OPS5, all of the instantiations in the CS are then fired in parallel. This constitutes one *elaboration cycle* within the *elaboration phase*. If this results in new instantiations, then the elaboration phase continues by entering another elaboration cycle. This process continues until quiescence is reached, i.e., when no more instantiations are generated. At the end of the elaboration phase Soar enters the decision phase. If a decision can be reached about the problem space, state or operator (the *context* element) to be used, then the wmes related to the new context element are added to the system and the older wmes are removed. If a decision cannot be reached, then an impasse results and the system creates a subgoal to solve the impasse. This subgoal allows Soar to bring to bear the full power of its problem solving capability on the impasse. All of Soar's goals are sewn together in a stack called the *context stack*. Each goal entry in the context stack is represented using three wmes: one for the problem space chosen to solve this goal, one for the state from which problem solving is progressing and one for the current operator.

Chunking is Soar's sole learning mechanism. It generalizes and caches the results of problem-solving as new productions. These productions may then fire in similar situations, preventing impasses and reducing the problem-solving effort. Chunking works by recording the wmes of each instantiation and the wmes created by firing that instantiation. When a wme is created that is accessible from any context, other than the most recent context, chunking builds a new chunk to summarize the creation of this *result* wme. Chunking performs a dependency analysis by searching backward through the instantiation records to find the wmes that existed before the result context that were used to generate this result. It then constructs a new production whose LHS is based on these wmes and whose RHS reconstructs the result. This chunk can then recreate the result wmes without hitting the impasse; this prevents the impasse from reoccurring and speeds problem solving.

Soar systems can be run without chunking, i.e., with chunking turned off. We will refer to this as the *without chunking* run. We will refer to the runs with chunking turned on, as *during-chunking* runs. In these runs, the performance system learns while it is solving a problem. After chunking on an input, sometimes Soar systems are run on the same input, to test the efficiency of the learned rules. We will refer to these as the *after-chunking* runs.

It is already well established that the addition of chunks improves the performance in Soar a great deal, when viewed in terms of subproblems required and the number of decisions within the subproblem [19]. However, in one of the examples presented in this paper, chunking causes an increase in total match time and hence total run time. In a high level view, which measures gains in terms of the number of decisions, certain *computational effects* [20] are ignored. These effects may cause the time per decision to increase drastically, completely offsetting match time gains. In this paper, however, we will not address such cost/benefit issues of chunking, and concentrate instead on the total match time.

The production system in Soar is similar to OPS5 with modifications, some of which are listed below:

- RETE must support the run-time addition of productions, unlike OPS5, and must update the memory nodes of the production with the current contents of WM, and the CS with the instantiations for this production.

- OPS5's negated condition elements cannot test for the absence of a conjunction of matching wmes. Soar adds LHS syntax and modifies the RETE to support these *conjunctive negations*.
- The Soar CS differs from OPS5 in that all productions may be fired in parallel.
- Soar systems use collections of smaller wmes to represent data that an OPS5 program would typically represent in a single wme.
- Soar productions only add wmes. The decision module keeps track of which wmes are accessible from the context stack, and automatically garbage collects inaccessible wmes.

We used three Soar programs to examine the various aspects of the implementation and the results of parallelism:

1. Cypress-Soar [18], an algorithm design system with 196 productions. We chose a run that derives the quick-sort algorithm.
2. Eight-puzzle-Soar, a system that solves the eight-puzzle mini task with 71 productions [9].
3. Strips-Soar, a system that plans in the domain of Robot control [1] with 105 productions.

4. Organization of the Soar/PSM-E System

The Soar/PSM-E implementation of Soar on the Encore consists of one Soar process that maintains all the functionality except the capability of matching; a PSM-E control process, and one or more PSM-E match processes. The number of match processes remains fixed for the duration of a particular run.

We are planning a full port of Soar to C, but our current structure allows us to concentrate on our primary goal of investigating parallelism in the match. Unfortunately, as Lisp and C processes cannot share memory on the Encore, this arrangement causes some data structures to be duplicated in Soar and PSM-E. Further, Soar and PSM-E can communicate only through Unix[4] pipes.

Soar/PSM-E operates in a mode where Soar uses PSM-E as a matching engine. Both Soar and PSM-E keep a copy of WM. As the Decide module adds or deletes wmes, it sends messages to PSM-E to repeat those operations on its wmes. If this adds new instantiations to the PSM-E CS, then as PSM-E fires these instantiations it also sends copies of them over to Soar, so that Soar may also fire them.

Both Soar and PSM-E then fire these instantiations, updating their copies of WM and repeating match. If new chunks are created, Soar passes them over to PSM-E at the end of the elaboration cycle. This organization is depicted in Figure 4-1.

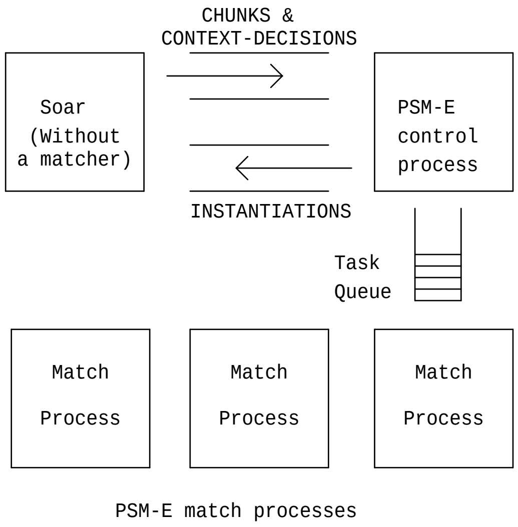


Figure 4-1: The organization of the Soar/PSM-E system.

In the above description of Soar/PSM-E, we have used PSM-E to imply a modified version of the PSM-E implementation of OPS5. In the next section, we describe the modification required to PSM-E in order to support Soar.

5. Extensions Required to Support Soar

The implementation of Soar on the PSM-E required many changes to be made to PSM-E. In this section, we describe the most significant change made to PSM-E: the capability of adding productions at run-time in order to support chunking in Soar. The first subsection describes the run-time code generation for new productions. Soar also requires that the memory nodes of the newly added production be updated with the current contents of WM, so that the chunk can be made immediately available for use. The second subsection describes our solution to this updating problem.

5.1. Run Time Addition of Productions

The problem of adding a new production at run-time on the PSM-E is significant because it requires that the production be compiled into machine code, like the rest of the system. Interpretive techniques, though simple, are not suitable because execution speed is very important. To exploit the benefits of *node sharing*, the new code must also be integrated into the existing machine code for the rest of the network. Recall that the RETE network shares common tests and nodes between different productions to save work at run-time. Sharing is especially important in Soar, since chunks are generated from the existing set of productions. Therefore, schemes for compiling the chunk as a separate piece of code are ruled out, because they are too inefficient.

Soar adds chunks only at the end of an elaboration cycle, i.e., when the match is quiescent. This eliminates the com-

[4] Unix is a registered trademark of Bell Laboratories.

plexity of synchronizing and modifying the code while the match processes are executing. However, the code generation for the newly added production must be efficient, so that this processing itself does not become a serial bottleneck. This efficiency requirement rules out using the assembler provided by the operating system, since forking off a process to assemble and then link and load generates an unacceptably large amount of overhead. To speed up the compilation of the chunks, the production system compiler used to generate the assembly code on the PSM-E was modified to generate machine code directly and was included as part of the run-time system. Two mechanisms are used to make the compilation faster and provide sharing. These mechanisms are :

- A tree data structure that provides a high level description of the RETE network. This is similar to the structure shown in Figure 2-2.
- A jumptable to integrate the new code with the existing code. Figure 5-1 shows a jumptable. It shows an indirect jump to the *label-12* through the second index in the jumptable.

The run-time addition of productions is illustrated below with a simplified example.

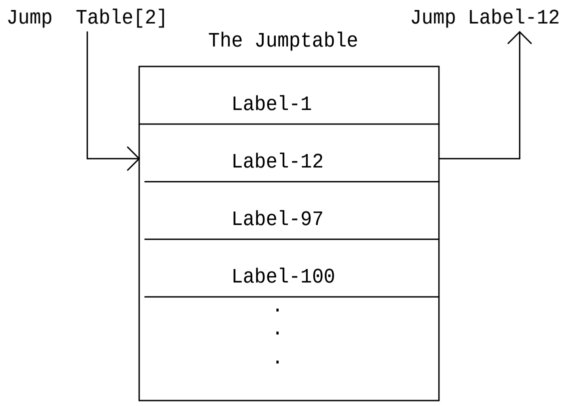


Figure 5-1: The jumptable mechanism.

The Figure 5-2 shows a production *P-new* (depicted by bold lines) being added to an existing production *P-old* in the system. P-new shares some nodes, corresponding to its first two CEs, with existing nodes in P-old. To locate these shared nodes, a high level data structure, similar to the one shown in the diagram is maintained by the system. On the Soar/PSM-E, when an activation for the *parent* node in P-old succeeds, it queues an activation for the successor node *c1*, into the shared task queue. The jumptable contains an entry (with the index of 50 in this case) for the node *c1*. This jumptable entry maintains a pointer to *next-code*, the section of the code to be executed by a process, after it places the c1-activation into the task queue. When the node *cnew* is added as the successor of the *parent* node, it requires the *parent* node activation to queue a cnew-activation along with the c1-activation. Therefore, a successful *parent* node activation has to execute the following sequence:

1. Place the c1-activation into the task queue.
2. Place the cnew-activation into the task queue.
3. Execute *next-code*.

```
      Production: P-old
(p blue-block-is-graspable
      (block ^name <block> ^color blue)
     -(block ^on <block>)
      (hand ^state free)
      --> (modify 1 ^state graspable))

       Production: P-new

 (p blue-block-can-be-placed-on-table
      (block ^name <block> ^color blue)
     -(block ^on <block>)
      (place ^table free)
      --> (modify 1 ^state on-table))
```

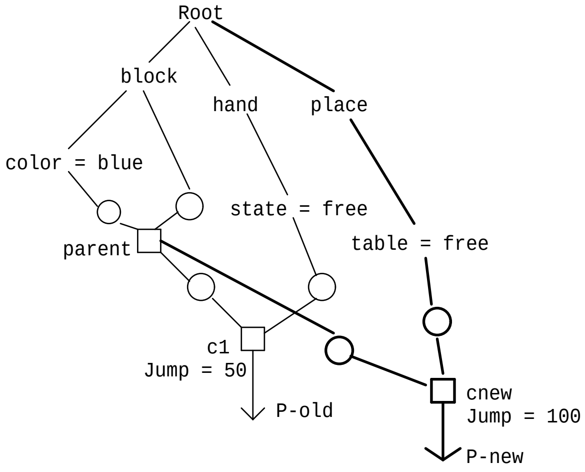


Figure 5-2: Adding a production at run-time.

An area of shared memory is reserved for generating *cnew-code*, the code for the node *cnew*. The node *cnew* is given an entry into the jumptable (with the index of 100 in this case). The following sequnce of assignments is then used:

Jumptable [100]:= Jumptable [50];

Jumptable [50]:= Code for queuing the node *cnew*;

This causes the successful parent-activation to place both the c1-activation and the cnew-activation into the task queue before it goes on to execute *nextcode*, as it did before. A process which picks up the cnew-activation then executes the *cnew-code*. Thus the jumptable maintains a link between any two sections of code, between which the code for a new node could in principle be inserted. An entry for a node in the jumptable points to the section of code to be executed after queueing that node in the the task queue. The process of integration of the new code then reduces to changing entries in the jumptable. In reality, the procedure of adding new nodes is complicated by a large number of special cases. Two issues can be raised about the jumptable:

1. The overhead of the jumptable during match in the three programs has been measured to be about 1-3%, much less than the 20-30% loss due to an unshared network (see below).
2. When there are two or more successors to a node, then only one jumptable entry is maintained for all of the successors together. Thus the size of the jumptable has not been an issue.

Tables 5-1 and presents some data about the chunks added in the three systems. The first column names the tasks. The second column lists the average number of CEs in the task's Soar productions. The third column gives the average number of CEs in the chunks. The fourth column gives the amount of code generated per chunk. The fifth column gives the amount of memory used per a two-input node.

Task	Avg. nbr. of CEs in the Task Ps	Avg. nbr. of CEs in the chunks	Avg. nbr. of bytes/ chunk	Avg. nbr. of bytes/ 2-input node
Eight-puzzle	18	36	7,900	219
Strips	13	34	8,500	250
Cypress	26	51	15,500	304

Table 5-1: Number of CEs per chunk.

Two points can be noted from the data provided here. First, the chunks produced have about two to three times more CEs than the original hand-coded Soar productions. Second, even with sharing, the current PSM technique of coding chunks requires 250 bytes/two-input node. This large size is due to the inline expansion of procedures. However, if these calls are closed coded, then with some inefficiencies, the size of this code can be reduced to about 15-20 bytes per two-input node.

Task	Number of chunks added	Time spent in adding the chunks (sec)	Time spent in adding the chunks -unshared (sec)
Eight-puzzle	20	23.7	25.5
Strips	26	31.5	34.7
Cypress	26	56.7	60.2

Table 5-2: Time for compiling chunks at run-time.

Table 5-2 shows the time spent in compiling chunks at run-time. The first column gives the name of the task. The second column gives the number of chunks added to the system. The time to compile the chunks appears in the third column. The fourth column gives the time to compile the chunks when sharing in the two-input nodes is turned off. Sharing requires that the RETE data-structure be searched for points to share. The numbers in Table 5-2 show that even with that overhead, the code generation time for the version with sharing is less than the time to generate code for the unshared version. This is because sharing reduces the amount of code generated. We are currently investigating the use of parallelism to reduce this time.

5.2. Run Time Update of State

As mentioned in Section 2.2, RETE is a state-saving algorithm, i.e., it saves the partial results of the match in the various memory nodes in the network. When chunks are added at run-time, the unshared memory nodes of the chunks are empty. The empty memories must be updated with PIs representing the partial matches of the WM contents to the new production. The updating procedure of the newly added chunk has to ensure that no duplicate state is added to pre-existing memory nodes, that already contain the required tokens. The update procedure must not become a serial bottleneck by being very complex.

A simple method of updating the node memories for the new production would be to pass the contents of WM back through the network and permit only those node tasks associated with the new production to execute. However, some of the nodes associated with the new production are shared with existing network, and therefore such a scheme would add duplicate state to those nodes resulting in incorrect behaviour in the execution of the program.

The updating must therefore be confined to only the new nodes. The identification of the new, and unshared nodes, is facilitated by the fact that the RETE network is *linear*, i.e., once one node in the production loses sharing, all its descendents remain unshared. Therefore, a simple node ID scheme allows the identification of the nodes to be updated. All nodes in the network have incrementally assigned unique ID numbers and a newly added node is always assigned an ID greater than any other existing node in the network. Identifying the IDs for the last shared node and the first new node allows the determination of all nodes that are required to be updated. The algorithm runs the entire WM through the normal network with only two local modifications. First, the task queues are changed to ignore tasks with IDs less than the first new node. Second, the last shared node must be specially executed in order to pass down all the of the PIs that it has stored as state. In Figure 5-2, the node labeled *parent* is the last shared node. The node labeled *cnew* is the first new node. The algorithm will update the memories corresponding to *cnew*, without adding duplicate state to *parent*. As the existing task queue and network structure is used, with only minimal modification, the full parallelism of the match is available to speed up the state updating process.

Sharing in the network reduces the number of node-activations and provides good speedups. Sharing the two-input nodes provides a gain of about 20% in the Eight-puzzle task and 25% in Strips during the update phase. The speedup provided by sharing two-input nodes in the runs after chunking, is about 30% in the Eight-puzzle task and 20% in Strips[5]. It can be seen that with the systems adding about 30 chunks, substantial gains have been made. With systems that add large numbers of chunks, more gains could be expected. The next section discusses how the update algorithm benefits from parallelism.

6. Results and Analysis

In this section we present and compare the speedups for our three tasks with a varying number of match processes and discuss the effects of chunking on parallelism. We also present a detailed analysis of the speedups observed.

We make one assumption about the match to correct for a feature of the current Lisp-C implementation. The match starts only after all the wme changes corresponding to an elaboration cycle are finished, rather than after the first wme change. The PSM-E control process is responsible for all the wme changes in an elaboration cycle. However, since the process simultaneously communicates with Soar, its rate of execution of wme changes is much reduced, and this slows down the entire system. This is a temporary state of affairs. When the Decide and chunking modules are ported to C, this communication bottleneck will disappear. Thus, we have currently compensated for this factor by letting all wme changes in a cycle complete before starting the match.

6.1. Speedups Without Chunking

Figure 6-1 presents the speedups for the three tasks run without chunking. The figure also shows the uniprocessor times (in seconds) for the three tasks. The numbers along the X-axis represent the number of match processes, i.e., this number does not include the PSM-E control process. The speedups are measured for runs with a single shared task queue. The speedups in all three tasks are fairly low: the maximum speedup is about 4.2 fold. In fact, the speedup decreases with more than 9 match processes.

The low speedups and the decrease in the speedup with an increasing number of processes indicate some form of contention for shared memory objects. In our system, there are two shared memory objects: the memory nodes of the RETE algorithm and the single shared task queue.

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

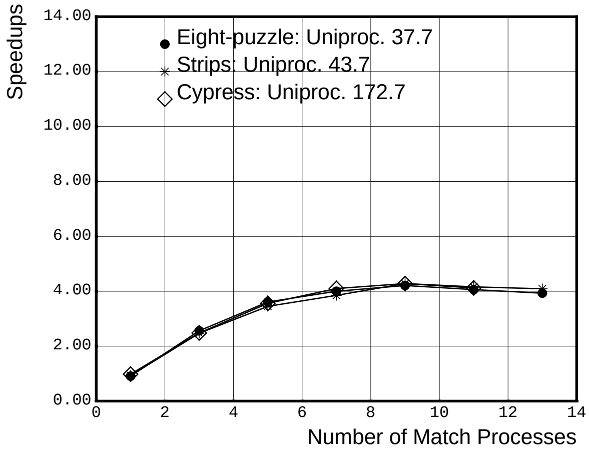


Figure 6-1: Speedups without chunking, single task queue.

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 the left tokens, which activate the two input nodes from their left inputs, the contention is low in Cypress and Eight-puzzle — 1 or 2 spins/access — with up to 13 match processes. But, the contention in Strips is higher: about 15 spins/access when run with 13 processes. The graph in Figure 6-2 displays this difference. The graph presents the total number of accesses for one bucket in one elaboration cycle; the accesses may not really be concurrent. However, the total number of accesses is an upper bound on the number of concurrent accesses and gives an indication of the number of concurrent accesses. The graph is to be interpreted as follows: in Eight-puzzle and Cypress, 70% of the time, there is only one left token accessing a bucket in a cycle. Thus 70% of the time, the left tokens will not contend with any other left-token. In Strips, this is true only 40% of the time. Furthermore, in Eight-puzzle, there are never more than four left-tokens trying to access the same bucket in one cycle. While in Strips, about 18% of the time, there are more than four tokens accessing the same bucket in one cycle.

For right tokens, that activate the two-input nodes along the

[5] The initial set of productions in Cypress task are unusually big, with an average size of 26 condition elements. Due to some problems with the assembling of these productions, the network for the task had to be broken up, and so no reliable figures for sharing are available for Cypress.

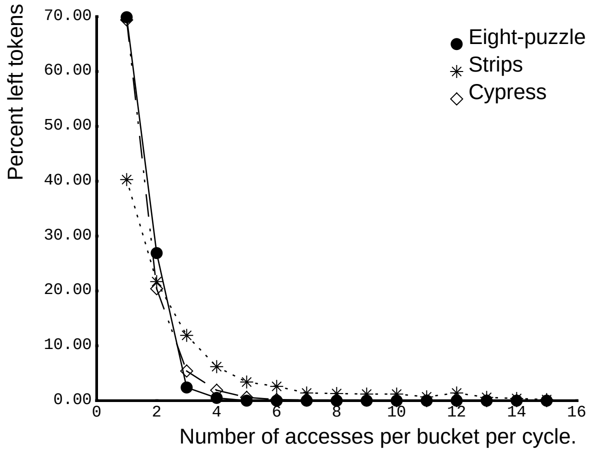


Figure 6-2: Contention for the Hash buckets.

right inputs, the contention for the lock is very low — 1 or 2 spins/access — in all three tasks, and it does not change by increasing the number of processes. This is because the right tokens are distributed evenly and most right tokens typically require very little processing [5]. 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[6].

However, even the higher contention for left-tokens in Strips is comparable with the average hash bucket contention for the OPS5 programs [6]. The reason for this low contention is the special nature of the Soar productions. By necessity, each CE in a Soar production is linked to a previous CE in that production via an equal variable test. Since hashing is dependent on the equal variable test, the tokens get distributed evenly. OPS5 tasks do not have such restrictions, which can sometimes cause an uneven distribution of the tokens in the hash-table. In these OPS5 programs, more aggressive locking schemes were seen to improve performance, but at the cost of some overheads [6]. This indicates that adapting aggressive locking schemes in Soar is not appropriate, even in Strips, since the gains are too low to be justified by the overheads. Furthermore, since the hash table is not the source of the contention, it must be for the other shared memory resource in the system: the single task queue.

The contention for the task queue can also be measured in terms of spins on the queue lock before a process gets hold of a task. Figure 6-3 shows this contention as a function of the number of processes. The graph shows the increase in contention for the tasks (spins/task) with the increasing number of processes. This increase implies a larger waiting time for the processes before they can push or pop tasks from the queue: explaining the saturation with about 8-10 processes. It is interesting to note that the contention for all the three tasks rises at approximately the same rate. This can be explained by:

- The code for locking the queue is the same in all three tasks, thus locking requires similar amounts of processing.
- The individual tasks in each of the three tasks require similar amounts of processing (described later in this section).

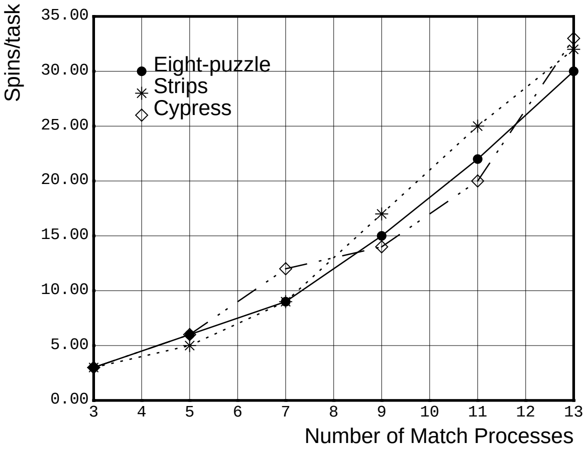


Figure 6-3: Task-queue contention with increasing number of processes.

The dip in the speedup curves in the three tasks (when the number of processes reaches 13) in Figure 6-1, can be explained by the phenomenon of *failed pop operations*. When a task is pushed into a queue, all the idle processes try to access that task. However, only one of them can get that task. The efficient way of informing other processes about the empty queue is to let them lock the queue and find the empty queue for themselves. These failed pop operations increase with an increasing number of processors, and interfere with the operation of the system. This leads to a slowdown.

This contention for the 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 6-4 presents the speedups for the three tasks with the introduction of multiple task queues. The graph shows that, as expected, parallelism has increased in all three tasks. The maximum speedup is seen in both Strips and Cypress: about 7 fold. Measurements of the task-queue contention show that with 13 match processes, the number of spins/task has reduced to about 2-3.

It is interesting to look at the granularity of the tasks ex-

[6] This is further emphasized by the big reduction in contention for the hash-table locks by the introduction of complex locks in [5]. The complex locks allow only similar (left or right) tokens to be processed in a pipelined manner, yet the reduction is seen.

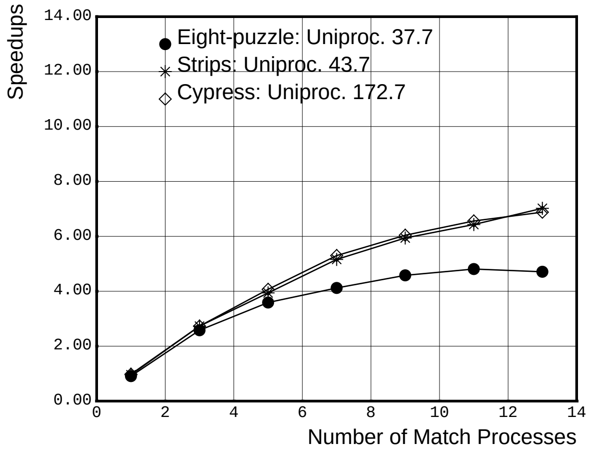


Figure 6-4: Speedup without chunking, multiple task queues.

ecuted on the PSM. Table 6-1, gives the time per task and total number of tasks generated in a run for the three systems. These figures show that the tasks are executing at approximately 400 microseconds on the 32032 processor[7].

Program	Uniproc. time (sec)	Total number of tasks executed	Avg. time per task (µs)
Eight-puzzle	37.7	87974	428
Strips	43.7	99611	438
Cypress	172.7	432390	400

Table 6-1: The granularity of the tasks on the PSM.

We have seen in this section that the only source of contention in the system was the single task queue. However, even after removing this source of contention by the introduction of multiple task queues, the speedups are much less than the ideal (linear) speedups. This problem can be seen to be particularly acute in the Eight-puzzle system.

6.2. Causes of the Low Speedups

As described in the earlier section, it is clear that some phenomena other than contention for shared memory objects are responsible for limiting the achievement of ideal speedups. To understand these phenomena, it is necessary to look at the speedups obtained in individual elaboration cycles. As explained before, computation in Soar proceeds in synchronous elaboration cycles. Figure 6-5 presents the speedups obtained in each cycle as a function of the number of tasks (tokens) executed in that cycle. The speedups observed in the eight puzzle with 11 match processes in Figure 6-4 are a weighted average of the speedups for the individual elaboration cycles of Figure 6-5.

These numbers are presented for Eight-puzzle; but they are representative of other Soar systems. These speedups were measured on a system with 11 match processes, so the maximum speedup obtainable in any one cycle should be 11.

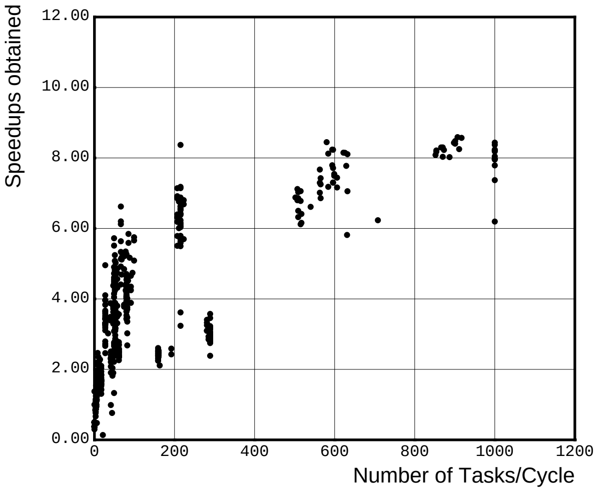


Figure 6-5: Eight-puzzle: Speedups as a function of Tasks/cycle.

Two separate phenomena are observable in this graph:

1. There are some cycles with a large number of tasks that show low speedups, e.g., there is a group of cycles with about 300 tasks/cycle that shows around 3 fold speedup.

2. The cycles with fewer tasks in general show low speedups compared to cycles with higher numbers of tasks. In fact in some of the smaller cycles, speedup is less than 1.

The occurrence of large cycles with low speedups can be understood if the number of tasks in the system (which is the sum of the number of tasks waiting to be processed and those being processed) are plotted as a function of time in those cycles. Figure 6-6 shows such a plot for one of the cycles in Eight-puzzle with about 300 tasks. This trace was taken for an execution of the task with 11 match processes. For presentation purposes, the graph was truncated at 25 tasks. The time is measured in units of 100 microseconds.

This trace shows that in the earlier part of the cycle, there is enough work for every process, and there exists a high potential for parallelism. However, after 200 time units, the system keeps processing a few tasks; each time generating only a few new tasks. This behaviour is caused by the presence of long chains of dependent node activations, *long chains*, — caused by a production with large numbers of CEs [5]. Long chains are commonly seen in the chunks built in many Soar

[7] The variation in the time for executing a task is from about 200 µs to 800 µs. The run without chunking has a large number of the smaller tasks.

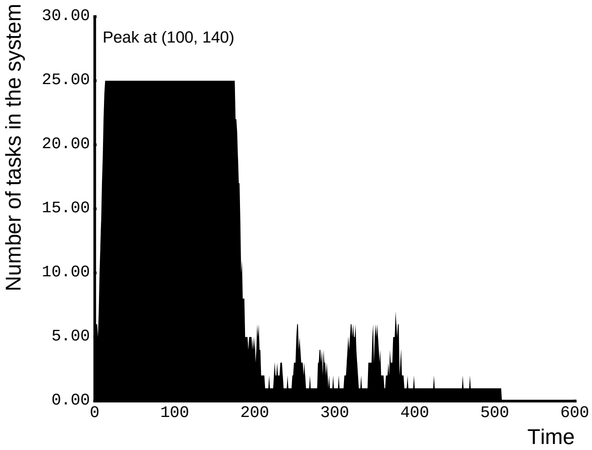


Figure 6-6: Eight-puzzle: Cycles with large numbers of tasks and low speedups

systems [20]. A part of a long chain from the Strips system is shown in Figure 6-7 — only a part is shown, since the chain has 43 CEs in it.

The impact of long chains on speedups 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 200 time units, in Figure 6-6) faster, but it cannot get through the long chain any faster, it will still require about 300 time units of computing.

To counter such long chains, we plan to introduce a *constrained bilinear network* organization. This organization is shown in Figure 6-8. It reduces the length of the chain to 15 CEs. The matching in all of the CEs in the production is constrained by the matches for the first few CEs. Without the constraint, a combinatorial explosion of state would be generated [5]. Currently, our compiler is not equipped to handle such network organization. We plan to develop the compiler technology to deal with these kinds of bilinear network[8].

The other problem for parallelism is the small elaboration cycles, at the far left of Figure 6-5. Three factors contribute to this phenomenon:

1. *Overhead:* There is a certain overhead associated with each cycle. This is caused by multiple processes having to check all of the task queues and to inform the control processor about the completion of the match. If very few tasks are generated in a cycle (0-10), the overhead causes a slowdown.

```
(Production Monitor-Strips-State
        (goal <g> ^problem-space <p>)
        (problem-space <p> ^name Strips)
        (goal <g> ^state <s>)
        (state <s> ^object <robby>)
        (object <robby> ^name robby-the-robot)
        (object <robby> ^type robot)
        (state <s> ^door-status <rclk-rram>)
        (door-status <rclk-rram> ^status closed)
        (object <rclk-rram> ^name rclk-rram-door)
        (object <rclk-rram> ^type door)
        (state <s> ^door-status <rclk-rpdp>)
        (door-status <rclk-rpdp> ^status open)
        (object <rclk-rpdp> ^name rclk-rpdp-door)
        (object <rclk-rpdp> ^type door)
                  .
                  .
                  .
```

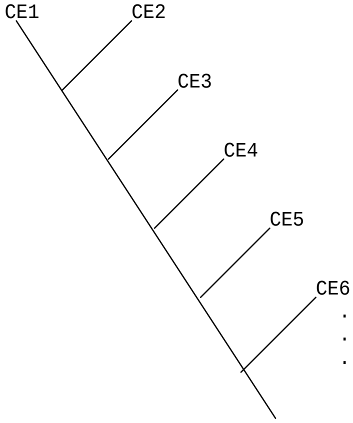


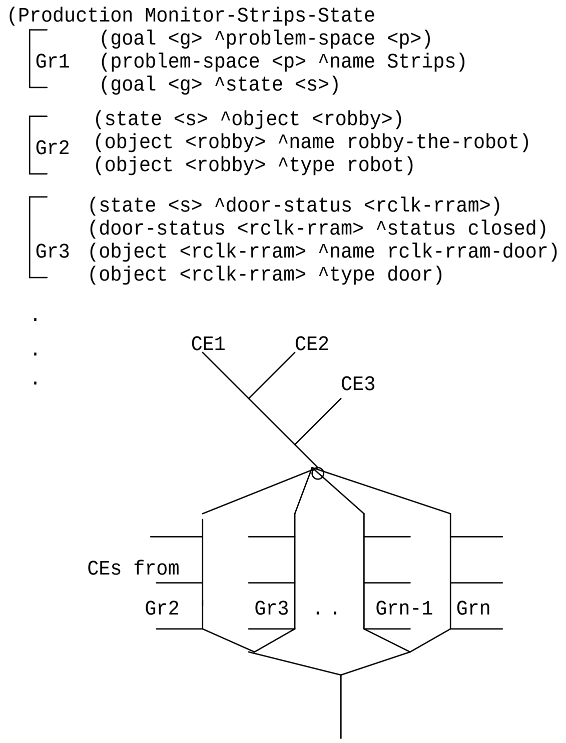
Figure 6-7: A long chain production.

2. *Long Chains:* In cycles with a small number of tasks, relatively short productions may have too much implicit sequentiality in their node execution. Even though the cycle may have as many as 100 tasks, at no time will more than four or five tasks be concurrently available. This produces a very flat graph of the number of tasks across time.

3. *Tail-end effect:* Although many of these cycles do have more concurrently available tasks than available processes, they may not be evenly distributed. The first half of these cycles exhibits good parallelism, but the second half is marked with a very uneven availability of tasks. This produces a task graph that has a large hump in its first half and then bounces between 1 and 10 available tasks in its second half.

These effects underscore some of the difficulties in scheduling via task queues. Those parts of an elaboration cycle with large numbers of tasks in them require multiple task queues to avoid contention. But, near the end of the cycle, fewer task-queues (about one-two) are required, so that the tasks in the queues can be located easily. However, detect-

[8] Constrained bilinear networks are also useful for the *conjunctive negations* — a construct used in Soar for testing the absence of a set of wmes. Currently we use a weak version of the constrained bilinear networks presented here to match them.



```
(Production Monitor-Strips-State
┌     (goal <g> ^problem-space <p>)
│Gr1  (problem-space <p> ^name Strips)
└     (goal <g> ^state <s>)

┌     (state <s> ^object <robby>)
│Gr2  (object <robby> ^name robby-the-robot)
└     (object <robby> ^type robot)

┌     (state <s> ^door-status <rclk-rram>)
│     (door-status <rclk-rram> ^status closed)
│Gr3  (object <rclk-rram> ^name rclk-rram-door)
└     (object <rclk-rram> ^type door)
 .
 .
 .
```

Figure 6-8: The constrained bilinear network.

ing the end of a cycle is very difficult; which implies that switching from multiple task queues to a single task queue is also difficult.

A major cause of the small cycles in Soar is the initial decisions for filling up the context slots (problem-space, state, operator) in a subgoal. Recall that when the problem-solver in Soar reaches an impasse, a subgoal is generated automatically. Initial decisions about filling up the roles in this subgoal are made serially; and very little matching is required to make these decisions. We expect that the future versions of Soar will make the initial context decisions in parallel. Thus the intensity of the small-cycle problem will be reduced by a large amount. Another factor for the small cycles is synchronous elaboration cycles. This is dicussed in Section 7.

6.3. Effect of Chunking on Parallelism

There are two aspects to the study of the effect of chunking on parallelism. As described earlier, chunking requires updating the state in the productions at run-time. Updating is thus an important part of a during chunking run. It is therefore interesting to see the speedups obtained while updating the productions. The second, more important aspect is the impact of chunking on parallelism. We will first present the speedups obtained in updating productions. We will then discuss the impact of chunking on parallelism.

Figure 6-9 presents the speedups obtained in the update phase of the during-chunking run. The speedup is calculated over the time spent in updating the entire set of chunks. The graph shows a high speedup obtained in the update phase. There are two reasons for the high speedup: (1) The entire set of wmes is matched, providing a high opportunity for parallelism (2) Matching the chunks while updating them allows a high degree of parallelism.

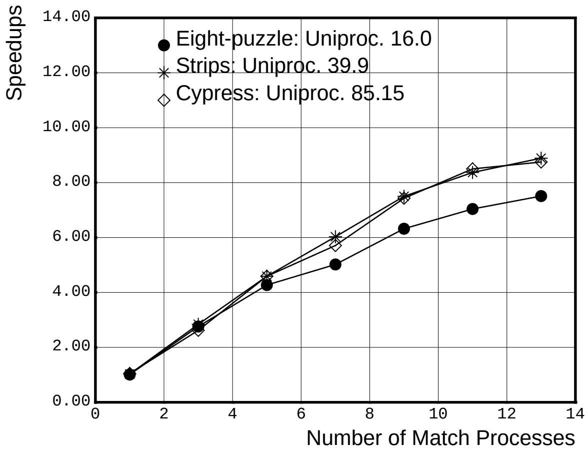


Figure 6-9: Speedups in the update phase, multiple task queues.

The speedups obtained in the during-chunking runs are a combination of the speedups obtained in the without chunking runs, the update phase and the after-chunking runs (described below). We will therefore not present those speedups here.

Figure 6-10 presents the results for the three tasks after chunking. The measurements were done on a system with multiple task queues. We see an increase in parallelism with chunking in Eight-puzzle and Strips. The Cypress run after chunking is very short and therefore inconclusive with respect to the impact of chunking on parallelism. The biggest increase is in the Eight-puzzle. This after chunking run in the Eight-puzzle is the case where maximum speedup is seen in the system — about 10 fold with 13 match processes.

Figure 6-11 shows a histogram of the percentage of tasks (node-activations) per cycle without chunking in the eight puzzle. Each interval in the histogram corresponds to 25 tasks/cycle. We see that 60% or more of the cycles have less than 100 tasks per cycle. Very few (about 3%) of the cycles have 1000 or more tasks per cycle. Figure 6-12 shows a similar histogram of tasks per cycle for the eight puzzle run after chunking. We see that now, over 30% of the cycles have 1000 or more tasks in them. We have already seen that small cycles typically provide low parallelism, while the larger cycles provide more parallelism. This partly explains the increase in parallelism. The decrease in small cycles is to be expected, since after chunking, the problem solver does not use the subgoaling mechanism to reach a solution. (Recall

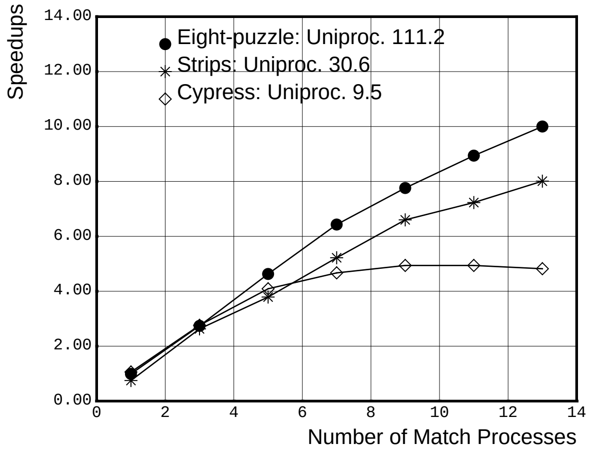


Figure 6-10: Speedups after chunking, multiple task queues.

that the small cycles were caused by initialization in subgoals).

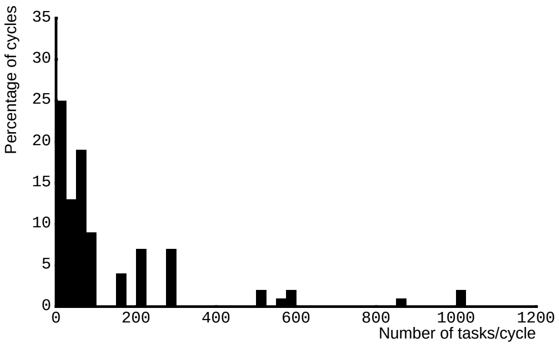


Figure 6-11: Eight-puzzle without chunking: tasks/cycle vs. percentage cycles.

The other reason for the increase in parallelism can also be understood by observing that there is an increase in cycles with 1000 or more tasks in them. These cycles are a result of the increase in the *affect-set size*, i.e., increase in the number of productions processed in a cycle. This increase has come about due to the addition of the chunks: in some cycles, these chunks have to be processed along with the original set of productions. This adds to the parallelism in those cycles. In the particular eight-puzzle example, the increase in tasks/cycle is particularly steep, because the eight-puzzle chunks are *expensive* [20], i.e., they require a large number of tasks for processing. In Strips the gains are smaller, because the chunks generate correspondingly fewer node-activations.

Previous research on parallelism in production systems establishes that a limited amount (10-20 fold) of parallelism is available in production systems [5, 6]. The production systems considered in this research were *non-learning* production systems. In these systems the limited amount of computation done per cycle limits the amount of available parallelism. However, in *learning* production systems such as Soar, the match computation may not remain limited. The machine-learning literature contains some analysis of systems which add such large numbers of productions [11]. That analysis indicates that there is an increase in the computation per cycle done in such systems. We have already seen such an increase, although on a small scale in this section. Thus the 10-20 fold empirical bound does not apply to learning production systems.

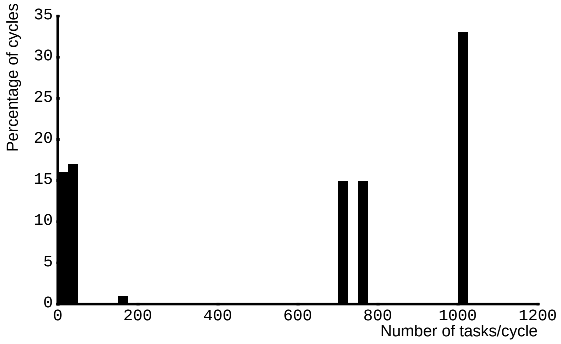


Figure 6-12: Eight-puzzle after chunking: tasks/cycle vs. percentage cycles.

7. Future Work

Soar is evolving and acquiring new domains of application. Some of the future modifications and new applications will impact the matching procedure. The effect of these changes on parallelism needs to be understood. We list two planned modifications, which would seem to lead to an increase in parallelism:

1. The elaboration cycles in Soar will be changed to fire asynchronously. Synchronization will only be enforced at the level of decision cycles, not at the level of elaboration cycles.

2. A Soar input/output module is under construction. This, in conjunction with new applications in fields such as Robotics, is expected to cause a significant increase in the rate of change of working memory, and hence increase the parallelism.

All the tasks presented in this paper perform a serial heuristic search. However, Soar tasks can also perform parallel heuristic search. One of the items that needs to be investigated is the impact of such parallel heuristic search on parallelism. Though such search will increase the amount of parallelism, it will also generate a lot of additional work. Thus increased parallelism could get lost in offsetting the additional work, providing no real speedup.

As was seen in this paper, good speedups were not achieved in some Soar tasks. A possible avenue of investigation is to equip the system with diagnostic tools to automatically deduce the causes of the low speedups. For example, to identify long chains, the system can look at the last few node activations on the cycles with low parallelism. The system can then make adaptive changes, such as introducing bilinear networks, to increase the speedups. Other areas for future study include the effects of chunking over long periods of time on parallelism. A longer term goal includes the optimization of the Lisp-based portion of the system, its conversion to C and parallelizing other areas of the system besides match.

8. Summary

In this paper, we have explored techniques for efficient parallel implementation of Soar, a significant AI system. This provided a unique opportunity to study the match parallelism of a learning production system. We presented techniques for adding productions and updating their state at run-time. We presented the speedups obtained in the match on our system and the effects of chunking on the speedups. We showed that Soar/PSM-E is capable of achieving significant speedups. The discussion of the impact of chunking on parallelism indicates that the opportunities for exploiting parallelism should increase a great deal in Soar systems that add a large number of chunks. We also analyzed speedups in detail and showed that two effects limit the parallelism in the system: short cycles and long chains. Some solutions to these problems were proposed, which would increase the speedups achievable. However, other modules in Soar still need to be optimized for this system to be useful as a real engine for Soar users.

Acknowledgement

We thank John Laird, Paul Rosenbloom and Peter Steenkiste for helpful comments on earlier drafts of this paper. We also thank Kathy Swedlow for her technical editing.

This research was sponsored by Encore Computer Corporation, Digital Equipment Corporation and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the:

Avionics Laboratory
Air Force Wright Aeronautical Laboratories
Aeronautical Systems Division (AFSC)
Wright-Patterson AFB, OHIO 45433-6543

Anoop Gupta is supported by DARPA contract MDA903-83-C-0335 and an award from the Digital Equipment Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Encore Computer Corporation, Digital Equipment Corporation and the Defense Advanced Research Projects Agency or the US Government.

References

1. Fikes, R., Hart, P., and Nilsson, N. "Learning and Executing Generalized Robot Plans". *Artificial Intelligence 3*, 1 (1972), 251-288.

2. Forgy, C. L. OPS5 User’s Manual. Tech. Rept. CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July, 1981.

3. Forgy, C. L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem". *Artificial Intelligence 19*, 1 (1982), 17-37.

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

5. Gupta, A.. *Parallelism in Production Systems.* Morgan-Kaufman, Los Altos, California, 1987.

6. Gupta, A., Forgy, C. L., Kalp, D., Newell, A., & Tambe. M. Results of Parallel Implementations of OPS5 on the Encore Multiprocessor. Proceedings of the International Conference on Parallel Processing, August, 1988. To appear.

7. Laird, J. E. Universal Subgoaling. In Laird, J.E., Rosenbloom, P.S., and Newell, A., Ed., *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, Kluwer Academic Publishers, Boston, Massachusetts, 1986, pp. 1-131.

8. Laird, J. E., Newell, A., and Rosenbloom, P. S. "Soar: An Architecture for General Intelligence". *Artifical Intelligence 33*, 1 (1987), 1-64.

9. Laird, J. E., Rosenbloom, P. S., & Newell, A. "Chunking in Soar: The Anatomy of a General Learning Mechanism". *Machine Learning 1*, 1 (1986), 11-46.

10. Lehr, T. F. The Implementation of a Production System Machine. Proceedings of the Hawaii International Conference on Systems Sciences, January, 1986, pp. 177-205.

11. Minton, S. Selectively Generalizing Plans for Problem-solving. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August, 1985, pp. 596-599.

12. Miranker, D. P. Treat: A Better Match Algorithm for AI Production Systems. Proceedings of AAAI-87, August, 1987, pp. 42-47.

13. Newell, A. Reasoning, Problem Solving and Decision Processes: The Problem Space as a Fundamental Category. In Nickerson, N., Ed., *Attention and Performance VIII*, Lawrence Erlbaum and Associates, Hillsdale, New Jersey, 1981, pp. 693-718.

14. Newell, A. Unified Theories of Cognition. The William James Lectures. Harvard University. Available in videocassette from Harvard Psychology Department.

15. Oflazer, K. Partitioning in Parallel Processing of Production Systems. Tech. Rept. CMU-CS-87-114, Computer Science Department, Carnegie Mellon University, March, 1987.

16. Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., and Orciuch, E. "R1-Soar: An Experiment in Knowledge-intensive Programming in a Problem-solving Architecture". *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 5 (1985), 561-569.

17. Schreiner, F. , Zimmerman, G. Pesa-1- A Parallel Architecture for Production Systems. Proceedings of the International Conference on Parallel Processing, August, 1987, pp. 166-169.

18. Steier, D. M. CYPRESS-Soar: A Case Study in Search and Learning in Algorithm Design. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, August, 1987, pp. 327-330.

19. Steier, D. E., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R. A., Golding, A., Polk, T. A., Shivers, O. G., Unruh, A., & Yost, G. R. Varieties of Learning in Soar: 1987. Proceedings of the Fourth International Workshop on Machine Learning, June, 1987, pp. 300-311.

20. Tambe, M. & Newell, A. Why Some Chunks Are Expensive. Proceedings of the Fifth International Workshop on Machine Learning, June, 1988. To appear.

21. Tenorio, M. F. M. and Moldovan, D. E. Mapping Production Systems Into Multi-processors. Proceedings of the International Conference on Parallel Processing, August, 1985, pp. 56-62.

Table of Contents

List of Figures

List of Tables