Compiling Dataflow into Control Driven Languages: Preliminary Results on Performance and Parallel Scalability on an IBM SP2 System

Ling-Hua Chang
University of Houston, Department of Computer Science
Houston, TX 77204-3475
Phone:(713)743-3353
Fax:(713)743-3335
E-mail: changli@cs.uh.edu
and
Ernst L. Leiss
University of Houston, Department of Computer Science
Houston, TX 77204-3475
Phone:(713)743-3353
Fax:(713)743-3335
E-mail: coscel@cs.uh.edu

October 14, 2002

Abstract

Until now, most results reported for parallelism in production systems have been simulation results – very few parallel implementations exists. In this paper, we present the initial implementation of our parallel OPS5 compiler on the IBM SP2. It is evident that the Rete match algorithm is suitable for parallel processing on distributed memory systems, as supported by the fact that the best performance of our parallel OPS5 compiler is achieved on the benchmark program *make-teams* with 100 employees using 16 processors and runs 7.76 faster than parallel OPS5 using a single processor. From our observations and experimental results, we believe that enhancements can make our parallel OPS5 compiler more effective. We also analyze the approach to parallelization taken in our prototype compiler and discuss various improvements.

Keywords: *Parallel OPS5, Scalability, Parallel Rete match algorithm, Message-passing machine, IBM SP2, Data distribution nodes, State transition graph*

1 Introduction

Production systems occupy a prominent place in the field of Artificial Intelligence (AI), including cognitive modeling, problem solving systems, and expert systems. However, production system programs are highly computation intensive. For a long time, this has limited the applicability of production systems. Over the last three decades, considerable research has been devoted to efficient implementations of production system languages. Some efforts have focused on high-performance uniprocessor implementations, while others concentrated on high-performance parallel implementations. Although the combination of better algorithms, efficient compilation techniques and faster hardware platforms has yielded increases in speed of several orders of magnitude, further research and development in production systems will require enlarging the production-memory (knowledge bases) in these systems (as in [6] and [7]) which further exacerbates the problem of long execution times. The common characteristic of these applications is that they process orders of magnitude more data than traditional applications. Production system implementations, however, have been notorious for their inability to handle large amounts of data due to uncontrollably expanding working space. This leads us to introduce the *state transition graph* and the *partition working memory elements* methods. These two methods not only avoid unnecessary comparisons in each match phase but also reduce the huge work space. Thus we believe that these approaches help in resolving the problem of excessive space requirements.

Sophisticated compilation techniques and parallelization have been the two main approaches taken by researchers in their efforts to solve the problem. In 1988, Anoop Gupta et al. pointed out that their parallel implementation of OPS5 on the Encore Multiprocessor achieved a 12.4 fold speedup using 13 processes [2]. Anurag Acharya has addressed the issue of efficiently implementing production systems on the generation of message-passing computers [3]. As one can see from these examples, in this past, research on parallel implementations of production systems has focused on shared memory multiprocessors. Recent advances in interconnection network technology have provided high communication bandwidth and low latency which makes us more interested in implementing production systems on distributed memory computers.

In this paper, we discuss two approaches to tackling the growth in execution time due to growing data sets – scalable parallelism and scalable match algorithms. From our preliminary experimental results, it is evident that the Rete match algorithm is very suitable for parallel processing on distributed memory systems. Another motivation for studying message-passing machines is their easy scalability to a large number of processors as opposed to shared-memory systems [5]. This makes it interesting to consider message-passing computers for the implementation of production systems.

The remainder of this paper is organized as follows. In Section 2, we sketch earlier attempts to deal with the problem. Section 3 outlines production systems issues, using OPS5 as a specific example. The Rete match algorithm is described in Section 4. Our prototype implementation of a parallel OPS5 compiler is discussed is Section 5, as are design and implementation issues. Preliminary performance results are presented in Section 6, which also contains a discussion of enhancements to our parallel compiler. The last section concludes the paper.

2 Related Work

In early production system programs, matching was by far the most expensive phase. As a result, initial research on parallelizing production system programs focused on parallelizing the match phase. Anoop Gupta implemented an OPS5 compiler on the Encore Multimax [10]. The Encore Multimax has 16 processors, a large shared memory, a fast bus and snooping caches, but the scheduler was implemented in software. Another shared memory architecture proposed for production system execution was the MANJI machine [11]. MANJI consisted of tens of 32-bit processors connected to a shared memory via a single bus. In addition to the bus, MANJI provided a multicast mechanism. A parallel version of Rete (see Section 4) considers the nodes of the Rete network as a set of interconnected objects passing partial and complete matches as messages. These nodes are partitioned among the processors. Gupta and Tambe [9] proposed a fine-grain mapping of Rete onto a group of message-passing processors. A small number of processors are assigned for the tests of one-input nodes and the select and act phases; the majority of the processors is used to implement the memory nodes and to perform the tests of two-input nodes. Examples of fine-grained machines are Mosaic [14] and J-machine [15]. Acharya [3] examined low latency medium-grain message-passing machines. In this mapping, there are no dedicated processors for the tests of the one-input nodes.

Instead, all the match processors perform the tests of the one-input nodes prior to performing memory node operations or the tests of the two-input nodes. Examples of medium-grained machines are Nectar [16] and Intel iPSC/2 [13]. They conducted their simulations of these two mappings of production systems on message-passing computers, respectively. The simulation for the medium-grained was based on Nectar [16] and the results indicated reasonable speedups.

3 OPS5

An OPS5 [1] production system consists of three major components, namely the working memory (WM), the production memory (PM), and the inference engine.

The working memory serves as a global database. Each entry in the working memory, called a working memory element (WME), represents a fact or assertion of the application domain.

The production memory is composed of condition elements 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). There are two types of tests: constant tests and equality tests. The set of WMEs that conjunctively match a production is referred to as an instantiation of the production. The set of all instantiations, active at any given time, is called the conflict set. Actions do things such as create WMEs, change values in WMEs, remove WMEs from working memory, print output, and stop the program.

The inference engine executes a production system by performing the so-called recognize-act cycle, until no rule can be instantiated. The match phase uses the Rete algorithm [4] to find all of the instantiations and stores them in the conflict set. The select phase uses a conflict resolution strategy [1] to choose a single instantiation from the conflict set. The fire phase fires the chosen instantiation of a production after which the right-hand side actions associated with the production are executed; then this instantiation is removed from the conflict set.

4 Rete

The Rete match algorithm [4] trades space for time by saving the match state between recognize-act cycles. When a WME is created, it is compared with all condition elements in the program; it is stored with each condition element to which it matches. Therefore, only incremental changes to working memory are matched in each cycle. There are three types of nodes in a Rete network; one-input nodes, memory nodes, and two-input nodes. One-input nodes perform the constant tests of the condition elements. Memory nodes store the results of the match phase from previous cycles as a state. This state consists of a list of the tokens that match a part of the LHS of the associated production. Only changes made to the working memory by the most recent production firing must be processed in each cycle. Two-input nodes test for joint satisfaction of condition elements in the LHS of a production. New tokens are created for token pairs that have been matched to the condition elements of the production; these flow down to the successor memory.

5 Parallel OPS5

Parallel OPS5 transforms a sequential production system into an equivalent parallel form which a multicomputer system can execute.

5.1 Match phase

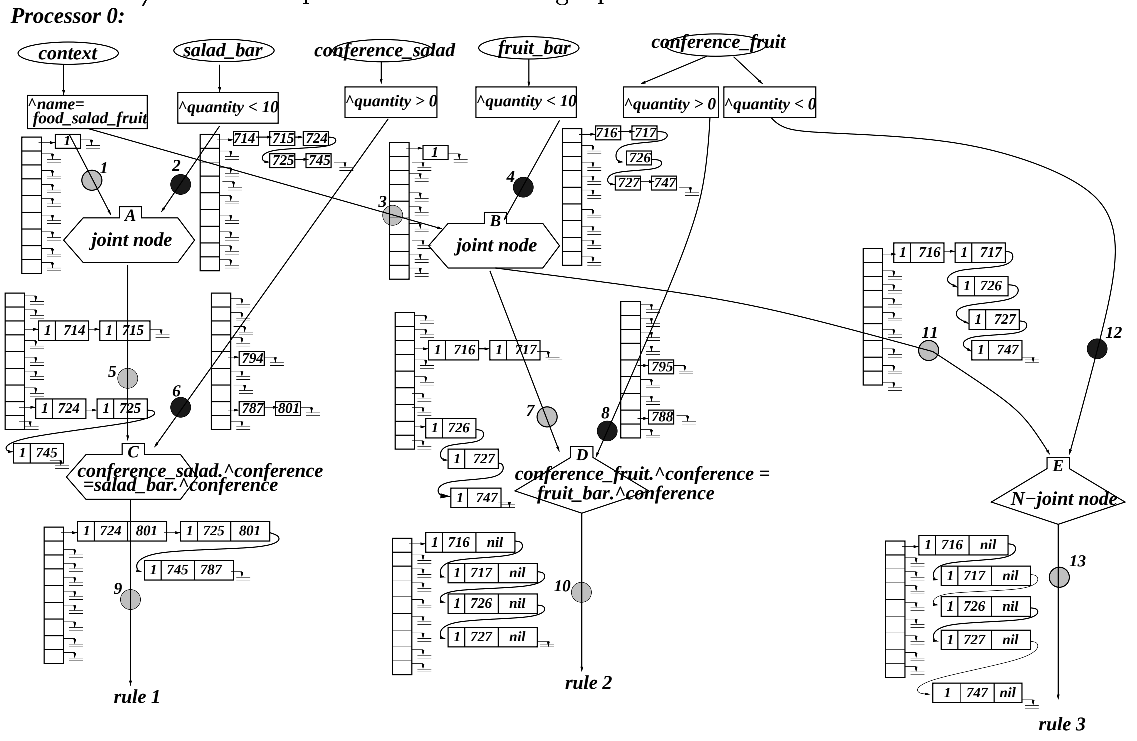
We parallelized the Rete match algorithm in the match phase. There are two subphases in the match phase, namely *initiated Rete-network* and *revisited Rete-network*. Each time a control WME is created [1], the associated WMEs in this control state will be compared and the matched pairs will be stored in the successor memory node; we call this subphase *initiated Rete-network*. The changes to working memory will be matched in each cycle until this control WME is removed; we call this subphase *revisited Rete-network*. Searching every two-input node is done by depth first search.

5.1.1 Initiated Rete-network

The initiated Rete match algorithm performs the following operations.

1. One-input test nodes: The master-slave paradigm [12] is used to assign a one-input test node to a slave processor. When the test is done, the slave processor informs the master processor how many elements have passed the test at that node and the master processor assigns an unexecuted one-input

test node to this slave processor. The process is complete when all the tests of one-input nodes have been executed by the slave processors. Then the master processor determines which one-input node has the maximal amount of WMEs passing the test and informs the slave processor who owns these WMEs to scatter them. Then the processors which conduct the tests of the other one-input nodes broadcast their WMEs that successfully matched a condition element. Since the WM data which pass the tests at the one-input nodes have been distributed, each processor can start working on the tests of the two-input nodes and with n processors, the computation time of two-input nodes for each processor should be *1/n* of the computation time of a single processor.



1	*(context ^name food_salad_fruit)*	*801*	*(conference_salad ^conference blue ^quantity 5)*
714	*(salad_bar ^conference red ^number 1 ^quantity 8)*	*808*	*(conference_salad ^conference red ^quantity 0)*
715	*(salad_bar ^conference red ^number 2 ^quantity 0)*	*716*	*(fruit_bar ^conference red ^number 1 ^quantity 8)*
724	*(salad_bar ^conference blue ^number 1 ^quantity 6)*	*717*	*(fruit_bar ^conference red ^number 2 ^quantity 7)*
725	*(salad_bar ^conference blue ^number 2 ^quantity 5)*	*726*	*(fruit_bar ^conference blue ^number 1 ^quantity 5)*
734	*(salad_bar ^conference green ^number 1 ^quantity 20)*	*727*	*(fruit_bar ^conference blue ^number 2 ^quantity 3)*
735	*(salad_bar ^conference green ^number 2 ^quantity 30)*	*736*	*(fruit_bar ^conference green ^number 1 ^quantity 10)*
744	*(salad_bar ^conference yellow ^number 1 ^quantity 10)*	*737*	*(fruit_bar ^conference green ^number 2 ^quantity 20)*
745	*(salad_bar ^conference yellow ^number 2 ^quantity 9)*	*746*	*(fruit_bar ^conference yellow ^number 1 ^quantity 30)*
787	*(conference_salad ^conference yellow ^quantity 6)*	*747*	*(fruit_bar ^conference yellow ^number 2 ^quantity 1)*
794	*(conference_salad ^conference green ^quantity 3)*	*788*	*(conference_fruit ^conference yellow ^quantity 4)*
		795	*(conference_fruit ^conference green ^quantity 2)*

Figure 1: An example of a Rete network

2. Memory nodes: Each two-input node has two memories – one each for storing tokens from its two inputs. A right memory holds tokens received from its one-input node. Each token contains a point to a WME that matched the condition element. A left memory holds sets of tokens that were the result of the work of a previous two-input node. These tokens contain addresses of WMEs which have matched a group of condition elements earlier in the rule. Each memory node has a hash table to store tokens. The hash function that is applied to tokens uses the variable bindings for equality. The tokens which hash to the same slot will be put in a linked list in that slot. Figure 1 presents a complete example used to explain how tokens are stored in the hash tables.

Example 5.1 *In Figure 1, assume that there are 23 WMEs which will be executed by processor 0. The right most side of each WM data is the address. In memory nodes 5, 6, 7 and 8, the hashing key is $\uparrow$conference and the WMEs with value $\uparrow$conference red are stored in slot 3, blue and yellow in slot 8 and green in slot 5 of their corresponding hash tables. For example, the token stored in the left hash table of the memory node 5 combines two WMEs whose addresses are 1 and 714. It will be mapped to the 3rd slot of the left hash table, because the second WME whose address is 714 has value $\uparrow$conference* *red and $\uparrow$conference is a hashing key for the two-input node C. Memory nodes 9, 10 and 13 are leaf nodes and the resulting sets of tokens become instantiations and are placed in the conflict set.*

Here, we employ hash tables on memory nodes; the right and the left memory nodes because the hash value *h(k)* with key *k* can be computed in *O(1)* time and the time required to search for an element with this key *k* depends linearly on the length of the list *T[h(k)]*. In order to avoid many keys hashing to the same slot, we use the division method [8] to create hash functions and then we introduce the tree structure to build memory nodes of a Rete network, so memory nodes can be accessed at once.

3. Two-input test nodes: There are four types of two-input test nodes called joint nodes, regular nodes, N-joint nodes, and N-regular nodes. Notice that each processor conducts all the tests of two-input nodes individually and no communication among processors is needed.

   - Joint nodes: Each joint node accepts the tokens from its two memory nodes and combines the WME pointers in both tokens to form a new one. This new token will be stored in the proper slot of the successor left hash table. The two-input nodes A and B in Figure 1 are joint nodes.
   - Regular nodes: A regular node joins a positive condition element which matches elements of the previous condition element. Each consistent match that occurs causes the node to combine the WME pointers in both tokens to form a new one which will be stored in the proper slot of the successor left hash table. The two-input node C in Figure 1 is a regular node.
   - N-joint nodes: Tokens in the left hash table pass through an N-joint node and are stored in the proper slot of the successor left hash table as long as no tokens are in the right memory. The two-input node E in Figure 1 is an N-joint node.
   - N-regular nodes: N-regular nodes join a negated condition element which refers to variables bound in previous condition elements. Each token in the left hash table must be matched against tokens in the right hash table. The two-input node D in Figure 1 is an N-regular node.

5.1.2 Revisited Rete-network

The revisited Rete-network is used only when the same Rete-network is visited again. Each processor conducts the test of each revisited two-input node independently; no communication among processors is needed.

1. One-input test nodes: Each processor conducts the test of each one-input node whenever new WMEs are created. The WMEs which pass the test are copied into a linked list called the added element list.

2. Two-input test nodes: We illustrate the following test operations of these four types of revisited two-input nodes. Each revisited two-input node has four types of lists, namely the removed token list, the added token list, the removed element list, and the added element list. The removed token list and the added token list are generated from tests of the previous two-input node. The removed element list stores removed WMEs which are the result of previous firings. The added element list holds new WMEs which have passed tests of one-input nodes.

   (a) N-joint nodes (see Figure 2):

      i. If the right memory is not empty and

         A. if the right memory was empty last time, then all the tokens in the left hash table are copied to the new removed token list, tokens in the removed token list are removed from the left hash table and tokens in the added token list are added to the left hash table.

         B. otherwise

            . each token in the removed token list is removed from left hash table, and

            . each token in the added token list is copied to the left hash table.

      ii. otherwise

         A. if the right memory was empty last time,

            . tokens in the removed token list are copied to the new removed token list and removed from the left hash table, and

            . tokens in the added token list are copied to the new added token list and the left hash table.

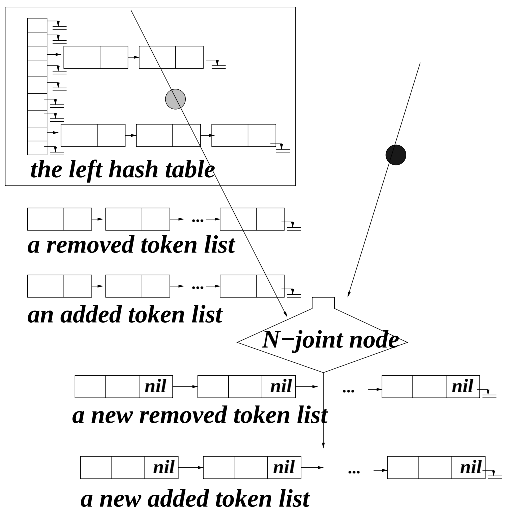


Figure 2: A revisited N-joint node

B. otherwise

- each token in the removed token list is removed from the left hash table,
- each token in the added token list is copied to the left hash table, and
- all the tokens stored in the left hash table are copied to the new added token list.

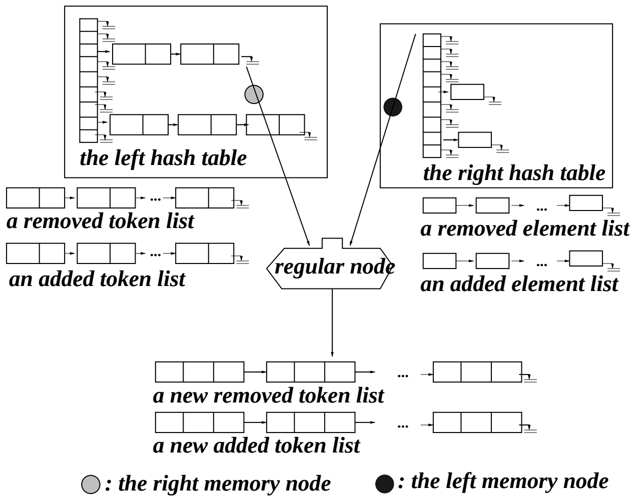


Figure 3: The remove function and the add function

(b) Joint nodes or regular nodes: Joint and regular nodes have the same test operations except that the regular nodes conduct equality tests. Two functions are used, namely the remove function and the add function. Note that the remove function is performed first.

i. The remove function (see Figure 3):

A. Each token in the removed token list is compared with tokens in the right hash table and the new tokens which pass the test at this node are added to the new removed token list.

B. Each token in the removed token list is removed from the left hash table.

C. The elements in the removed element list are compared with tokens in the left hash table and the new tokens are added to the new removed token list.

D. Elements in the removed element list are removed from the right hash table.

ii. The add function is analogous (see Figure 3).

(c) N-regular nodes (see Figure 4): An array *not-matched* is used to store the state of each token in the left hash table to see if the token has a match when it is compared against tokens in the right hash table.

i. Set the array *not-matched* to 0 at each element.

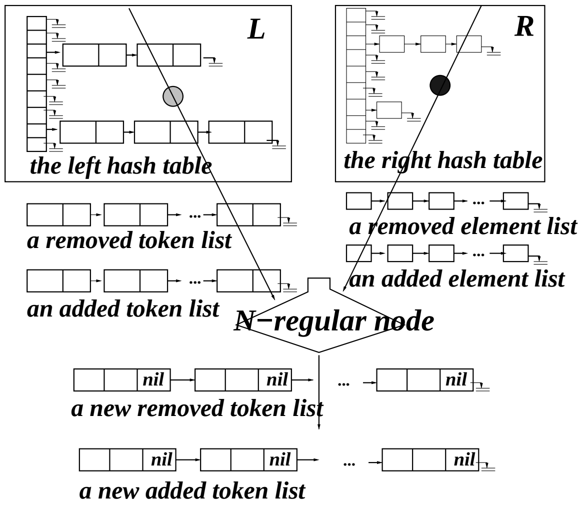


Figure 4: A revisited N-regular node

ii. Compare each token in the removed token list with the tokens in the right hash table. If the token does not match any token in the right hash table, it is copied to the new removed token list.

iii. Each token in the removed token list is removed from the left hash table.

iv. Compare each token in the left hash table with the tokens in the right hash table. If the token does not match any token in the right hash table then a slot in the array *not-match* is set to 1; this slot represents the given token.

v. Copy each element in the added element list to the right hash table.

vi. Remove each element in the removed element list from the right hash table.

vii. Compare each token in the left hash table with the tokens in the right hash table.

   A. If the token has a match and the array *not-matched* shows that it did not have any match last time, copy it to the new removed token list.

   B. If the token has no match and the array *not-matched* shows that it had a match, copy it to the new added token list.

viii. Compare each token in the added token list with the tokens in the right hash table. If the token does not have any match, copy it to the new added token list.

ix. Copy each token in the added token list to the left hash table.

5.2 Select phase

The master-slave paradigm is also employed in the select phase. In this phase, each processor, including the master processor, performs conflict resolution, using LEX (lexicographic-sort) or MEA (means-ends-analysis), to choose an instantiation of a rule from the conflict set for firing. The slave processors send the chosen instantiation to the master processor. Next, the master processor determines which instantiation should be chosen to fire by conflict resolution. Finally, the processor who owns the chosen instantiation broadcasts it to all the processors.

5.3 Fire phase

All the processors fire the same instantiation of the production rule. Working memory is updated immediately as each action is performed. Whenever there are *make* or *modify* actions, new WMEs are copied to their associated added element lists and removed WMEs are copied to their associated removed element lists.

6 Performance Analysis

We used an IBM SP for our preliminary tests. The IBM SP system used at PDC (Center for Parallel Computers, KTH) at Sweden has 170 separate processor nodes. Most of the nodes are standard POWER 2 architecture RS6000 processors. We have run three benchmark OPS5 programs. The benchmark program *hotel* simulates the operation of a large hotel for one-day-reservations, check in, maid service, laundry, and banquet functions, etc. The benchmark program *clusters* operates on image regions that are characterized by position and type (i.e., road, hangar, tarmac, etc.). The benchmark program *make-teams* operates on a database of employees which contains information about their area of expertise and previous experience. It also contains an overall numerical evaluation of each employee's past performance. We summarize the characteristics of each OPS5 benchmark program in Table 1.

programs	# of rules	# of classes	# of one-input nodes	# of two-input nodes
hotel	79	64	112	180
clusters	13	6	14	16
make-teams	9	4	11	13

Table 1: The characteristics of each benchmark

6.1 Experimental results

We have implemented two compilers, a parallel OPS5 compiler (called *Parallel OPS5*) and a sequential version using the same approach as *Parallel OPS5* but using a single processor. They were implemented on the IBM SP2 and we evaluated the performance in terms of speed of the resulting code. We define speedup as $\frac{Time\ for\ an\ algorithm\ on\ a\ single\ processor}{Time\ for\ an\ algorithm\ on\ N\ processors}$, where N is the number of processors (2, 4, 8, 16 or 32). The experiments measure and compare the performance of *Parallel OPS5* with Parallel OPS5 using a single processor on the IBM SP2 over these three benchmarks. Since previous studies exploring parallelism in production systems were based on simulators, we did not compare our experimental results with their simulation results. We used the best compiler options: *mpcc -O3 -qarch=pwr2 -qstrict -qtune=pwr2 program1.c, program2.c ....* We used the *MPI_Wtime()* primitive to measure the wall clock time. We compiled the OPS5 programs *hotel*, *make-teams* and *clusters* using the *Parallel OPS5* compiler, and then submitted the jobs with the number of nodes ranging from 1, 2, 4, 8, 16 to 32 nodes. Table 2 shows how many initial WMEs of each benchmark are

programs		# of initial WM	# of firings
hotel		2257	1903
make-teams with	50 employees	50	2364
	100 employees	100	18858
clusters with	100 image regions	100	3015
	200 image regions	200	10365
	400 image regions	400	37975
	500 image regions	500	168882

Table 2: # of initial WMEs and firings

invoked. The program *hotel* has a large data set and the programs *make-teams* and *clusters* have a scalable data set. Table 2 also shows how many instantiations of each benchmark have been fired. We present our experimental results in Tables 3 and 4.

Table 3 shows the run time of each benchmark program compiled by *Parallel OPS5.* Table 4 shows the *speedup* of each benchmark program. The benchmark *hotel* consist of 79 rules, 64 classes, 2257 initial WMEs and 1903 firings. The benchmark program *clusters* consists of 13 rules and 6 classes. The benchmark program *make-teams* consists of 9 rules and 4 classes. Although the benchmark programs *hotel* and *clusters* don't scale well, *make-teams* with 100 employees, when using 8 and 16 processors, runs 6.61 and 7.76 faster

programs		1 proc	2 procs	4 procs	8 procs	16 procs	32 procs
hotel		71.41	67.26	64.42	61.50	62.67	65.79
make-teams	50	28.44	17.62	10.60	8.81	8.18	10.68
(employees)	100	1384.83	776.23	338.91	209.56	178.49	190.18
clusters	100	10.28	10.03	9.95	9.84	10.28	12.89
(image	200	91.97	79.89	76.56	76.03	76.06	74.72
regions)	400	1036.21	850.20	778.82	759.26	767.59	744.43
	500			12527.27	12045.92	11784.73	11760.16

Table 3: The wall clock times (*secs*) of benchmark programs compiled by *Parallel OPS5*

programs		2 procs	4 procs	8 procs	16 procs	32 procs
hotel		1.06	1.11	1.16	1.14	1.09
make-teams with	50	1.61	2.683	3.23	3.48	2.66
(employees)	100	1.78	4.09	6.61	7.76	7.28
clusters with	100	1.02	1.03	1.05	1.00	0.80
(image	200	1.15	1.20	1.21	1.21	1.23
regions)	400	1.22	1.33	1.37	1.35	1.39

Table 4: The *speedups* of benchmark programs

than using a single processor (see Table 4). From our observations, we think that the performance of our *Parallel OPS5* still can be improved. These experimental results also show that scalability of *Parallel OPS5* is achieved primarily for large data sets. Below we explain why *Parallel OPS5* at this moment does not scale well in all instances.

6.2 Analyze parallel OPS5

We list strengths and weaknesses of our parallel OPS5 algorithms. This provides the starting point for improvements.

6.2.1 Advantages

1. State transition graph: We use the rule-cluster control technique [1] to manipulate a large rule base.

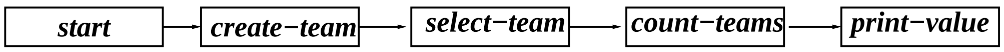


Figure 5: State transition graph for program *make-teams*

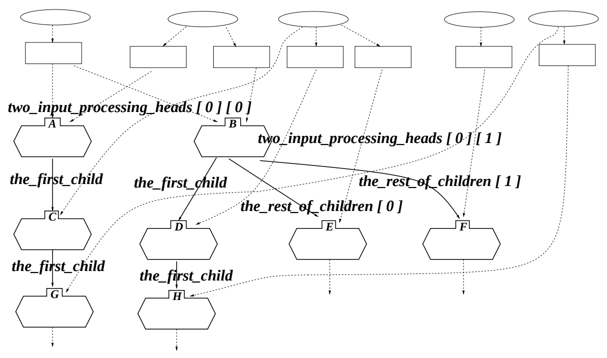
A state transition graph generated by the partition of the control elements helps to reduce the work space used by the left memory and the right memory. The parallel Rete match algorithm uses a lot of work space; for example, the benchmark program *clusters* with 500 image regions and *make-teams* with 200 employees consumes more than 4 Mbytes work space using multiprocessors. So, the *state transition graph* introduced here addresses this problem. The *state transition graph* is based on the idea of using control states to partition the OPS5 program into several control states. For example, the program *make-teams* consists of five control states, namely *start*, *create-team*, *select-team*, *count-teams* and *print-value*; its state transition graph is shown in Figure 5. The transitive relations of these control states are obtained based on the timing of the control states created and deleted. We can use the state transition graph to determine whether a current state will be revisited; otherwise all the hash tables used in this state will be freed.

2. Partition WMEs approach: Because each processor has the whole piece of initial WMEs, so when the WMEs grow, the work space grows too much and which crashes the production system. For example, we are unable to finish running programs *clusters* with 500 image regions using 1 or 2 processors and *make-teams* with 200 employees. We introduce the *partition WMEs approach* to reduce the huge size of WMEs. This method partitions the WMEs among the processors for those chosen classes which own a huge size of data set. This method should resolve problems related to the excessive WM space.

3. The tree structure: We introduce a tree structure to build two-input nodes, the left memory nodes and the right memory nodes of each Rete network, separately. The tree data structure of two-input nodes described in C code is shown in Figure 6. Figures 7 and 8 present the tree structure of the left memory nodes and the tree structure of the right memory nodes. It can be seen in Figures 6, 7, and 8, that these three tree structures are the same. This is why whenever a two-input node conducts tests, its memory nodes can be accessed immediately. This approach also differs from that of previous researchers.

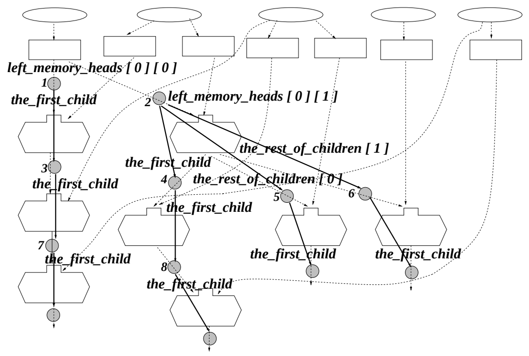
6.2.2 Disadvantages

1. Parallel Rete is not well-parallelized: This is due to redundant work on each processor. We have discovered this problem in our parallel Rete algorithm which makes parallel Rete unlikely to be parallelized fully.


two_input_processing_heads [0] [0]
A
B
two_input_processing_heads [0] [1]
the_first_child
the_first_child
the_rest_of_children [1]
the_rest_of_children [0]
C
D
E
F
the_first_child
the_first_child
G
H

```
struct two_input_processing {
        :
    struct two_input_processing *the_first_child;
    struct two_input_processing
        *the_rest_of_children [ max_number_of_the_rest_of_children ] ;
            } *two_input_processing_heads [ number_of_states ] [ max_number_of_branches ] ;
```

Figure 6: The tree structure of two-input nodes of a Rete network



```
structure left_memory_node {
        :
    structure left_memory_node *the_first_child;
    structure left_memory_node
            *the_rest_of_children[max_number_of_the_rest_of_children];
  } *left_memory_heads[number_of_states][max_number_of_branches];
```

Figure 7: The tree structure of the left memory nodes of a Rete network

Example 6.1 *In Figure 9, assume there are 1 goal, 80 objects and 10 groups which have passed their test respectively and 100 group-counts. If a Rete network is performed by a single processor, there are 80 comparisons in node A and 10 comparisons in node B. If the same Rete network is executed by a multicomputer system (using 4 processors and applying our parallel OPS5), there will be 20 comparisons in node A and 10 comparisons in node B. Consider production rules 0 and 1; since WM data of the class object have been scattered, rules 0 and 1 can be done in 1/4 of the wall clock time which a single processor takes. However, in production rule 2, we expect 250 comparisons instead of 1000 comparisons (if every WME matches every condition element), therefore there is no timing saved by conducting tests at any two-input node of production rule 2.*

In order to make parallel Rete more effective, we propose the *find_data_distribution_nodes* method. We illustrate the idea underlying *find_data_distribution_nodes* as follows (see Figure 9). Note that each rule can have only one data distribution node.

(a) Sort the numbers of WMEs of each class (*1 goal, 80 objects, 10 groups* and *100 group_counts*) in decreasing order; we obtain 100 *group-counts*, 80 *objects*, 10 *groups* and 1 *goal* in sequence.

(b) Choose the first class in that order and choose the class *group-count.*

(c) Because 100 *group-counts* flow to nodes D and E, we mark node D for rule 1 and node E for rule 2, respectively.

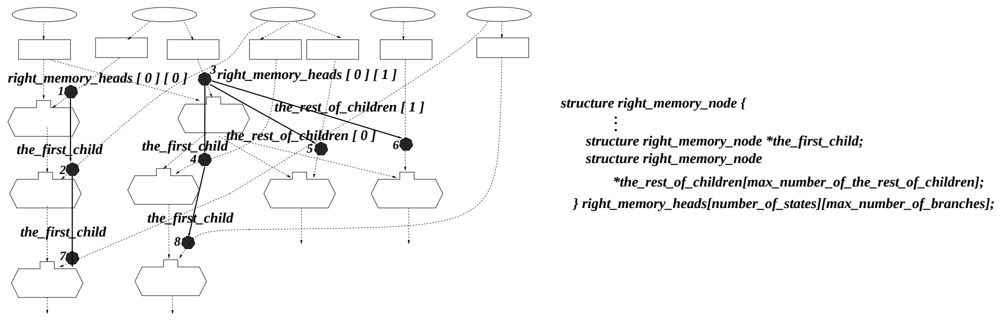
(d) Next, the class *object* is chosen, 80 *objects* flow to node A; we mark node A for rule 0 and 1. Because rule 1 has been assigned to node D, we drop node A and instead we pick the class *groups* and mark node C for rule 0 and node B for rule 2. Since rule 2 has been assigned to node E, we unmark node B.

(e) Therefore we obtain node C which is chosen for rule 0, node D for rule 1 and node E for rule 2,

(f) and the data distribution nodes are nodes C, D and E.

(g) Then 10 groups and 100 group-counts are scattered when two-input nodes C, D and E are executed. Therefore this parallel Rete algorithm can be more effective than its conventional parallel implementation.

2. The *Parallel OPS5* compiler consumes too much memory space: As we have mentioned above, each processor has the same piece of initial WMEs and some production systems crashed because of the uncontrolled growth of the WMEs. We use the *partition WMEs* approach to solve this problem.


right_memory_heads [0] [0]
3
right_memory_heads [0] [1]
1
the_rest_of_children [1]
the_rest_of_children [0]
the_first_child
2
the_first_child
4
5
6
the_first_child
the_first_child
8
7

```
structure right_memory_node {
          :
    structure right_memory_node *the_first_child;
    structure right_memory_node
          *the_rest_of_children[max_number_of_the_rest_of_children];
  } right_memory_heads[number_of_states][max_number_of_branches];
```

Figure 8: The tree structure of the right memory nodes of a Rete network

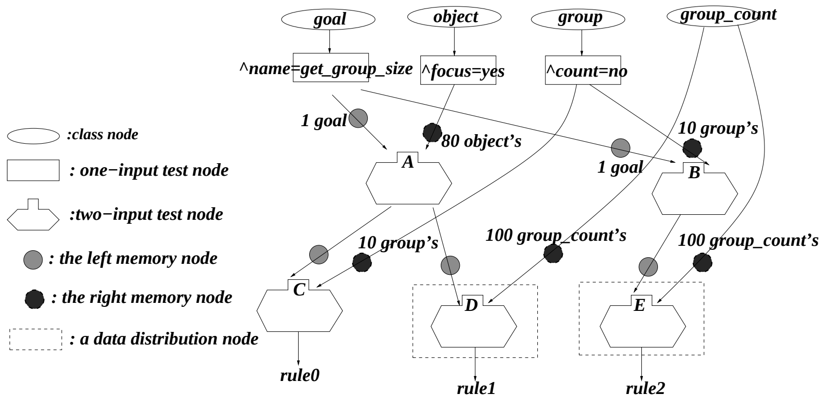


Figure 9: Find data distribution nodes

The chosen classes of WMEs will be partitioned among the processors. But this is not as easy as it looks. For example, in Figure 9, the WMEs of classes *group* and *group-count* may be chosen to be scattered, but if the WMEs of the class *group* are partitioned, when the two-input node B is executed, the partitioned WMEs of the class *group* will be compared and this will lead to erroraneous results, because the two-input node B is not a data distribution node. So, in this case, only the WMEs of the class *group_count* can be partitioned among the processors and the whole set of WMEs of the class *group* will still be in each processor. Before the two-input node C is executed, the whole set of WMEs of the class *group* will conduct a one-input node test and then the passed matches will be scattered to the processors. If the WMEs of the new class are created by using the associated partitioned WMEs of say class *group-count*, then this new class will also be a partitioned class that corresponds to that partition of the *group-count* class.

3. Fire phase: In the fire phase, we have mentioned that all the processors fire the same instantion of the production rule in each recognize-act cycle; this makes the *Parallel OPS5* not fully effective. We are certain that the methods *find_data_distribution_nodes* and *partition WMEs* approach introduced previously will solve the problem in fire phase automatically. Each processor can fire a different instantiation which makes firing in fire phase be parallelized as well.

7 Conclusion

Our *parallel* OPS5 compiler transforms a sequential production system into an equivalent parallel form which a multicomputer system can execute. The concept of data disribution is employed to partition workload among processors. Data distribution processes will be executed on the one-input test nodes of the *initiated Rete-network*; as a result, the computation time of two-input nodes for each processor should be *1/n* of the computation time of a single processor. The Rete match algorithm trades space for time by saving the match tokens in memory nodes between recognize-act cycles. Since both the tokens and the WMEs are stored in the memory, as the size of data set increases, WMEs and work space always grow too fast and the explosion of the memory leads to an OPS5 program whose enormous data set may make it unable to finish its work. We introduced the *state transition graph* and *partition WMEs* to solve this problem; as the results indicate, this did not alleviate this problem substantially. We will introduce the approach *find_data_distribution_nodes*. After WMEs are partitioned, each processor only owns *1/n* WMEs which saves a lot of WMEs space. The method *find_data_distribution_nodes* determines which two-input test node will be chosen as a data distribution node. While a data distribution two-input node conducts testing, the tokens arriving at the right memory will be partitioned evenly and then the workload and workspace of each processor will be reduced to be approxmatically *1/n*.

Currently, from our experimental results, the benchmark program *make-teams* with 100 employees using 16 processors runs 7.76 faster than parallel OPS5 using a single processor; thus it is evident that the Rete match algorithm is suitable for parallel processing on distributed memory systems and we also believe that our revised *parallel* OPS5 compiler will work better in general. We will investigate the application of our parallel pattern match algorithm to CLIPS. CLIPS is a rule-based programming language providing interferencing and representation capabilities which are similar to those of OPS5. We expect the application of our parallel Rete network algorithm to CLIPS to be feasible.

Acknowledgements

We are very thankful to Anurag Acharya for providing the benchmark programs and initial WM data generators and also to PDC in Sweden for providing the IBM SP system.

References

[1] Thomas Cooper and Nancy Wogrin *: Rule-based Programming with OPS5*. Digital Equipment Corporation, Morgan Kaufmann Publishers, Inc. 1988.

[2] Anoop Gupta, Milind Tambe, Dirk Kalp, Charles Forgy, and Allen Newell: Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis. *In 13th International Symposium on Computer Architecture*. Nov. 1987.

[3] Anurag Acharya, M. Tambe, and A. Gupta: Implementations of productions systems on message-passing computers. *IEEE Transactions on Parallel and Distributed Computing*. 3(4):477-487, July 1992.

[4] C. L. Forgy: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*. Vol. 19, pp. 17 - 37, 1982.

[5] Kai Hwang and Zhiwei Xu: Scalable Parallel Computers for Real-Time Signal Processing. In *IEEE Signal Processing Magazine*. 50-66, July 1996.

[6] J. Bachant and J. McDermott: R1 revisited: Four Years in the Trenches. *AI Mag.* Vol 5, No 3, pp. 21-32, 1984.

[7] A. Newell: *Unified Theories of Cognition*. Cambridge, MA: Harvard Univ. Press, 1990.

[8] Thomas Cormen, Charles E. Leiserson and Ronald L. Rivest: *Introduction to Algorithms*. The MIT Press, McGraw-Hill Book Company 1990.

[9] A. Gupta and M. Tambe: Suitability of Message Passing Computers for Implementing Production Systems. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 687-92, Aug. 1988.

[10] A. Gupta, C.L. Forgy, D. Kalp, A. Newell and M. Tambe: Parallel OPS5 on the Encore Multimax. In *Proceedings of the International Conference on Parallel Processing*, pp. 271-80, Aug. 1988.

[11] J. Miyazaki, K. Takeda, H. Amanom and H. Aiso: A New Version of a Parallel Production System Machine MANJI-II. In *Proceedings of the Sixth International Workshop on Database Machines*, pp. 317-30, June 1989.

[12] Kai Hwang and Zhiwei Xu: Advanced Computer Architecture: Parallelism, Scalability, and Programmability. *McGraw-Hill*. New York, 1993.

[13] Intel Scientific Computers, Beaverton, OR: The iPSC/2 brochures and application software reference material. Order number 28 110-001.

[14] W. C. Athas and C. L. Sietz: Multicomputers: Message-Passing Concurrent Computers. *IEEE Comput. Mag.,* Vol. 46, pp. 9-24, Aug. 1988.

[15] W.J. Dally et al.: The J-machine: A Fine-Grain Concurrent Computer. in *Proc. IFIPS Congress,* G. X. Ritter, Ed., 1989, pp. 1147-1153.

[16] E. Arnould, F. Bitz, E. Cooper, H.T. Kung, R.D. Sansom, and P. Steenkiste, "The Design of Netar: A network backplane for heterogeneous multicomputers," *Proc. Third Int. Conf. Architectural Support for Programming Languages Oper. Syst.,* 1988, pp. 205-216.