CIS:
A MASSIVELY CONCURRENT RULE-BASED SYSTEM

Guy E. Blelloch
AI Lab, Massachusetts Institute of Technology
Rm. 739, 545 Technology Square
Cambridge, MA 02139
Net Mail: guyb@mit-ai

Abstract

Recently researchers have suggested several computational models in which one programs by specifying large networks of simple devices. Such models are interesting because they go to the roots of concurrency – the circuit level. A problem with the models is that it is unclear how to program large systems and expensive to implement many features that are taken for granted in symbolic programming languages.

This paper describes the Concurrent Inference System (CIS), and its implementation on a massively concurrent network model of computation. It shows how much of the functionality of current rule-based systems can be implemented in a straightforward manner within such models.

Unlike conventional implementations of rule-based systems in which the inference engine and rule sets are clearly divided at run time, CIS compiles the rules into a large static concurrent network of very simple devices. In this network the rules and inference engine are no longer distinct. The Thinking Machines Corporation, Connection Machine – a 65,536 processor SIMD computer – is then used to run the network. On the current implementation, real time user system interaction is possible with up to 100,000 rules.

1 Introduction

The Concurrent Inference System (CIS) is a interactive rule-based system similar to Mycin [Davis77]. It asks the user questions and makes inferences according to the answers. The current version is capable of forward and backward chaining, which run concurrently; using meta-rules of the sort described by Davis [1980]; and reasoning with uncertainty, using a variation of Zadeh's [1965] rules. With 100,000 rules on the current implementation of CIS, a global inference step takes less than two seconds. A global inference step is the time needed for a single change to percolate through all the rules.

CIS was implemented to show that much of the functionality of a rule-based system can be implemented with a simple and implementationally cheap concurrent model of computation, and furthermore that programming the system in the model is relatively straightforward. The model used is the activity flow network (AFN) model [Blelloch86]. Activity flow networks are similar to the connectionist networks of Hinton [1981], Feldman [1982] and Rumelhart [1986].

CIS does much of its work at compile time, leaving at run time a static network of computational devices not significantly more complex than logic gates. It is easy and efficient to run such networks on massively concurrent SIMD computers such as the Connection Machine.

Because the networks are completely static and use very simple devices, it is hard and expensive to implement the general power of logic programming languages such as Prolog. For example, with CIS it is expensive to use high-precision numbers, hard to dynamically bind arbitrary values to a parameter, and not possible to execute general-purpose unification or create an arbitrary number of instances of an object. This paper argues that many practical rule sets do not require these features. For example, the rule sets of Mycin [Davis 77], R1 [McDermott 80] and Prospector [Gaschnig 82] can be implemented cleanly without them.

Section 2 discusses the AFN model. Section 3 gives a brief outline of AFL-1, the language CIS is programmed with. Sections 4 and 5 discuss CIS. Section 6 discusses the implementation of AFNs on the Connection Machine. Section 7 discusses some issues of concurrency.

2 Activity Flow Networks

In the past decade, researchers have proposed many models of concurrent processing many of which may be described as networks of nodes and links. As Fahlman [1982] noted, a useful way to categorize these models is by the complexity and content of the messages sent among the nodes. Figure 1 shows a taxonomy of models categorized in this way.

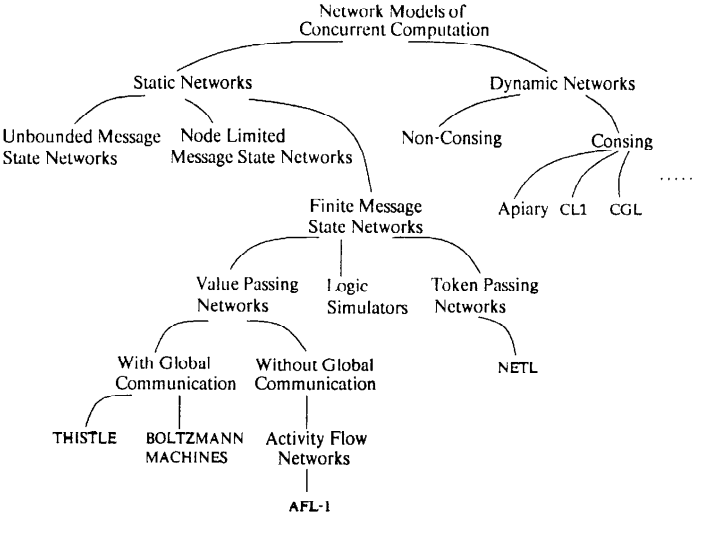


Figure 1: Hierarchy of Network Models of Concurrent Computation.

The taxonomy divides network models of concurrent computation into two sub-classes: static and dynamic networks. In a static network each node communicates with a fixed set of other nodes, while in a dynamic network each node can dynamically choose which other nodes it wants to talk to. Static networks are further categorized by the complexity of their messages. Finite message state (FMS) networks are static networks that only send messages containing one of a fixed finite set of states. Because the messages of an FMS are typically short and simple, FMS networks usually consist of a large number of simple nodes. FMS networks are categorized by the content of their messages. Value passing networks (VPNs) are FMS networks that only send messages containing a finite approximation of the real numbers.

VPNs are categorized by whether they allow global communications. Any path that allows a central controller to inspect the nodes and make a decision according to the results is considered global communications. In Thistle, a VPN suggested by Fahlman [1983], global communications are used heavily. For example, the host might put a value on all the **human** nodes and then have those nodes send the value over their **hair-color** links. The system might then take a global OR of the **brown** nodes to see if the network knows of any brown-haired humans. In contrast, activity flow networks (AFNs) do not include any global communications. The only global control in AFNs is a clock signal that allows the network to take a step. This clock does not provide a path from the nodes back to the controller.

Some connectionist models are AFNs. Many fall in other categories either because they use global control, such as those discussed by Ackley [1985] or Touretzky [1985], or because not all their messages are finite representations of real numbers, such as in the model of Feldman [1982].

In summary, an AFN is a static network of simple nodes and links, that:

- only passes finite approximations of real numbers over its links;
- is controlled by a global clock;
- can only communicate with the external world through a set of nodes designated as its inputs and outputs.

The particular AFN model used by AFL-1 consists of five types of nodes – **input, output, max, min** and **sum**; and two types of links – **inhibitory** and **excitatory**. Nodes can have an arbitrary number of inputs and outputs. Each node has four parameters – a **threshold**, a **slope**, a **rise** and a **saturation** which are set at compile time. A node combines its inputs using maximum, minimum or sum (according to the nodes type) and applies the function shown in Figure 2 to compute its output. In AFL-1, in contrast to the distributed view used by Hinton [1981] and Touretzky [1985], the nodes are each assigned a single name.

A link makes a directional connection between two nodes. Each link has a weight. As an activity is sent through a link it gets multiplied by this weight. The clock is used to synchronize all the nodes after each node sends its activity through its output and receives a new activity.

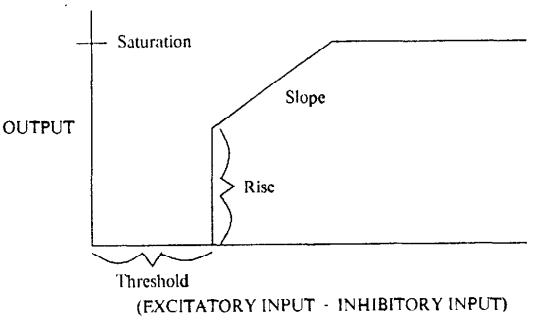


Figure 2: The Output Function of an AFL-1 Node.

3 AFL-1

AFL-1 is an extension of the Symbolics 3600 programming environment. The purpose of the language is to translate high level description of tasks into AFNs which can then be simulated. The language consists of functions that add nodes and links to the network and the **defgroup** form which allows the user to hierarchically define structures (node groups) out of nodes and links. To add nodes and links to the AFN, one instantiates a NG by appending "make-" to the group name and passing the instance name and any arguments. This structure defining mechanism is similar to the mechanisms found in constraint languages [Steele 80, Sussman 80] and circuit design languages [Batali 80].

4 CIS

This section introduces the Concurrent Inference System. In this section we will use parameter-value pairs to represent propositions. The next section describes how to assign parameters to specific objects so as to use object-parameter-value triplets to represent propositions.

The specification of a rule set in CIS consists of a list of **parameter** definitions followed by a list of **rule** definitions. Each parameter definition contains a parameter name and a set of possible values. Each rule contains a list of **if-parts**, each a parameter-value pair, and a list of **then-parts**, each a parameter-value pair with a certainty factor.

The following set of parameter and rule definitions will be used as an example throughout the description of CIS.

```
(make-parameter 'covering '(feathers hair))
(make-parameter 'animal-class
                '(mammal bird reptile))
(make-parameter 'eating-class
                '(ungulate carnivore))
(make-parameter 'ped-type '(claws hoofs))

(make-rule 'animal-rule-1
  '(if (covering hair))
  '(then (animal-class mammal .95)))

(make-rule 'animal-rule-2
  '(if (animal-class mammal) (ped-type hoofs))
  '(then (eating-class ungulate .9)))
```

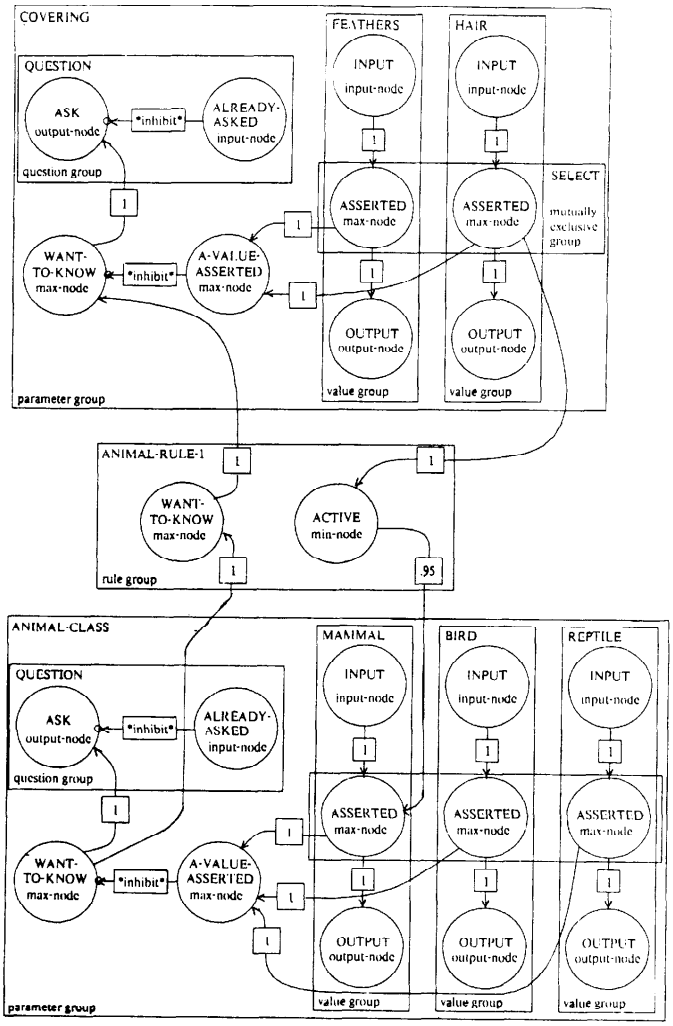


Figure 3: Part of the Network Created by the Animal Example.

In the above example, **rule** and **parameter** are both names of node groups (NGs) defined by CIS using the **defgroup** form. Each of the **make-rule** or **make-parameter** statements creates an instance of one of these NGs. The reader should keep in mind that each top level call in AFL-1 adds nodes and links to the AFN at compile time. At run time, the only computation that occurs is the flow of activity values along the precompiled network.

Figure 3 shows part of the network that results from compiling the above definitions. In this figure, the nodes of the AFN are shown as circles and the links as small squares. Within a node, the type of node is specified below the node instance name. The weight of each link is shown inside the link's square. Node group instances are circumscribed by rectangles. The node group name is given in the lower left corner and the instance name is given in the upper left corner.

4.1 Forward Chaining

At run time, the activity value of an **asserted** node gives the certainty that a parameter-value proposition is true. Inferences in CIS are made by forward flow of this activity through the **active** nodes of the **rule** NG instances.

The **active** node of each **rule** NG instance takes the AND of that rule's **if-parts**. Since the if parts are activity values, not binary, they are combined with MIN and thresholded. This method for taking the AND of uncertain propositions is the same as the methods used by Mycin's rules and Prospector's *logical relations*. The **asserted** node of the **value** NG instances takes the OR of its inputs. Maximum is used for the OR of uncertain propositions.

The rule implementer, using the weight in each **then-part** of a rule, sets the weights on the links between the **active** and **asserted** nodes.

4.2 Backward Chaining

A backward chaining inference systems (also called goal directed or consequent reasoning systems) looks for parameters that can satisfy a goal parameter. By backward chaining, a rule-based system will only ask the user questions relevant to what the system is looking for. CIS backward chains by using the **want-to-know** nodes. The "want-to-know" activity flows in the opposite direction than the "truth" activity – i.e. from the **then-parts** to the **if-parts**.

The backward chaining done by CIS has two advantages over many current rule-based systems. Firstly, since the "want-to-know" and "true" activity flow across different links, the antecedent and consequent reasoning happen concurrently. Consider a medical diagnosis program which is searching for disease X, and meanwhile stumbles across all the symptoms for a perhaps much more serious disease Y. Most current systems would ignore this, and perhaps not come to disease Y for a long time. CIS would find Y as it searched for X.

Secondly, unlike Prolog and Mycin, the inferences are not restricted by recursion to follow the same path as the control. The backward chaining done by CIS is therefore easier to extend. Three examples of such extension are:

- Backward chaining links can easily be excluded from some rules; this allows a mixing of antecedent and consequent rules.
- With few changes, meta-rules of the sort discussed by Davis [1980] can be implemented; such rules are discussed in section 4.5.
- It is easy to make "fuzzy" backward chaining links and use these links as one of many heuristics that guide the search rather than force it.

A potential problem with concurrent backward chaining is that the questions the system asks the user might be unfocused. This problem can be solved in CIS by making the **want-to-know** and **ask** nodes have analog values and using heuristic rules to judge the current "interest" of a parameter and activate the **want-to-know** nodes accordingly. These rules are implemented using more network structure. The host will pick the parameter with the highest value on its **ask** node when it selects a question to ask the user. The heuristic rules for activating the **want-to-know** nodes might include: a) asking related questions together (the rule set implementor can specify which questions are related), b) ask about the **if-parts** of almost active rules, and c) ask about parameters that lead to more goals. Methods for implementing these heuristics are discussed in [Blelloch 86].

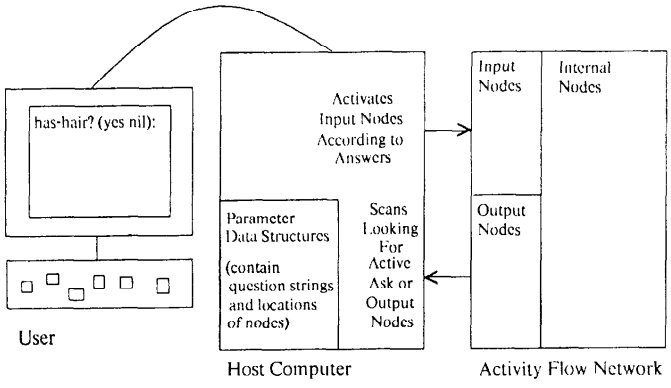


Figure 4: The Interface Between the User and the AFN of CIS.

4.3 Input and Output

The only way an AFN can communicate with the outside world is through the **input** and **output** nodes. At run time, the CIS system uses a serial host computer to communicate with these nodes. Figure 4 pictures how the communications work. For sensor based rule-based systems such as PDS [Fox 83] it is possible to have the I/O nodes connected directly to sensors and activators. Such direct connections will not be discussed in this paper.

4.4 Values

Each parameter can have several values, each of which creates an instance of the **value** NG (see Figure 3). The **a-value-asserted** node of a **parameter** is used to recognize if any of its values are asserted above some threshold. A **mutually-exclusive** group is placed around the **asserted** nodes of the values so that only one of the values (the one with the highest input) is asserted at a time. The **mutually-exclusive** group can be left out if desired.

Because there is a separate NG for each value, there can only be a moderate number of values specified at compile time. Most rule sets do not require many values and in many current rule-based systems including Mycin, KEE and Prospector, one defines the possible values of each parameter at compile time. This helps prevent errors.

The need for a NG instance for each value also precludes the use of high-precision integers and floating point numbers. Although most other rule-based systems allow such values, many applications don't need them. For example Mycin uses integer values only for age, body temperature and dates of last examination or immunization. For these parameters, 32 bit integers are not needed; 100 or so values will suffice. The rule sets of Prospector and R1 also require no high-precision numbers.

4.5 Meta Rules

In practice, it is important to have task specific rules that control the invocation of other rules [Davis 80, Gaschnig 82]. An example of such a rule is: *"if the patient has stepped on a rusty nail, then ask questions about tetanus* (activate the tetanus rules)." Davis [1980] names such rules, "meta-rules", and Prospector [Gaschnig 82] names them "contextual relations".

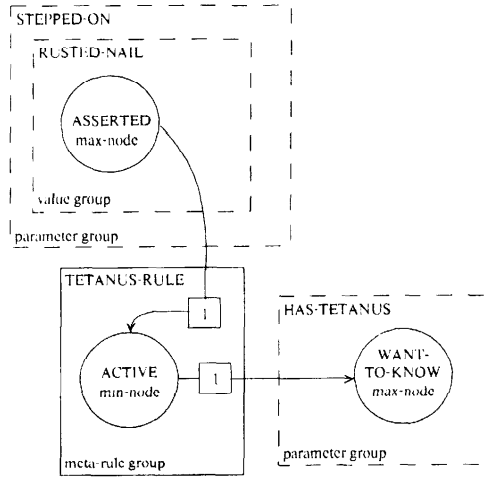


Figure 5: The Network Created by the Tetanus Meta-Rule. In the diagram the dashed groups signify that not all the nodes in those groups are shown.

It is easy to add this type of rule to CIS. Figure 5 shows the network required for the tetanus rule. This network causes the system to ask all the questions relevant to *tetanus* when the parameter *"stepped on rusted nail"* is asserted.

5 Objects

When a rule set includes several instances of an object that all obey the same rules, it is convenient to create a single set of rules which are valid for all instances. To allow for this, systems such as Mycin, OPS5 and KEE have generic objects (often part of the object, attribute, value triplet) which are used in the rules. As well as allowing multiple instances, objects allow a clean way to separate sets of rules into modules.

The same sort of object abstraction can be used in CIS by creating a separate network for each instance of an object. This is done by defining a Node Group for an object and instantiating it, at compile time, for each instance. At run time, the **host** computer assigns names of new instances to the precompiled instance sub-networks. For example, if we wanted CIS to reason about two animals we could use the following definition.

```
(defgroup animal ()
  (make-parameter 'covering '(feathers hair))
  . .
  . .
  (make-rule 'animal-rule-1
    '(if (covering hair))
    '(then (animal-class mammal .95)))
  . .)

(make-animal 'first-animal)
(make-animal 'second-animal)
```

To create the needed network at compile time, CIS must know the maximum number of instances that might be needed. In most applications this is not a problem since this number does not vary greatly from one use of a system to the next: it is easy enough to put an upper bound on the number. For example, in

Mycin we know there will only be one patient and the patient will usually have at most three cultures taken, each with possibly three organisms.

The above method of creating multiple instances of an object does not address the problem of creating rules that cross between different instances. Such rules might include:

```
IF (and (father x y) (zebra x))
THEN (zebra y))

IF (= (number-of zebras) 3)
THEN (herd-of zebras)
```

There are relatively simple ways to include such rules in CIS; see [Blelloch86].

6 Implementation

The **network-processor** is used to run a compiled activity flow network. A single cycle of the **network processor** is called an **afl-step**. It consists of all the nodes sending their activities, receiving new ones and computing the node function.

The Thinking Machines Corporation Connection Machine is currently used as the **network-processor**. At compile time, each node is placed on a separate processor. All the output links of a node are placed in processors immediately following the node processor. Processors are also used for each input of a node and are placed immediately preceding the node processor. This means that for l links and n nodes, $2l + n$ processors are required (by overlapping input and output links, $l + n$ processors can be used).

At run time, a copy-prefix operation is used [Kruskal85] to distribute the output value of a node to all the output links. After the copy-prefix, each link multiplies its input value by its **weight** and sends the result, using the router, to the other end of the link – a processor preceding a node processor. **Max, min** and **sum** prefixes are used to compute the **max, min** and **sum** of the inputs for each node. The node processors then compute the node function. Because of the prefix operations, the time taken by this method is independent of the largest fan-in or fan-out. The routing cycle is the most expensive step taken by this method.

To include more links than processors, each physical processor can simulate several virtual processors (VPs). Such simulation causes a slightly greater than linear slow-down with the number of virtual processors. Figure 6 shows the time required by an **afl-step** as a function of the number of links when implemented on a 64K-processor Connection Machine.

By making some assumptions about the rules and parameters and imposing a limit on the time the user is willing to wait between questions, an upper bound on the number of rules that CIS may have can be given. With the following assumptions, it is possible to include 100,000 rules in CIS.

- The maximum time a user is willing to wait is 2 seconds.
- The maximum depth of inferences in the system is 20 rules.
- The average rule has three antecedent and two consequent parts.
- The average parameter has five values.

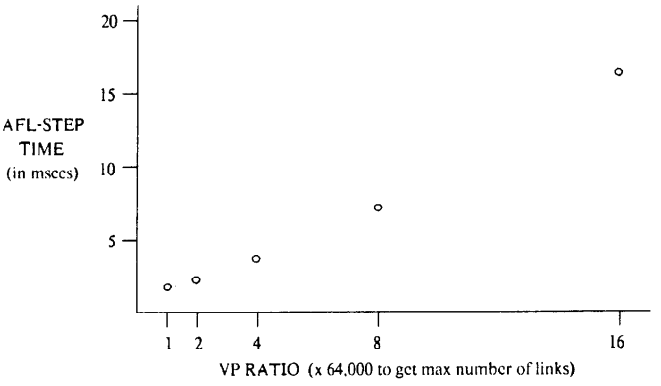


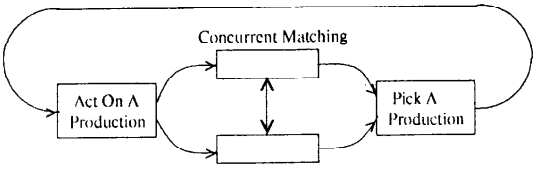
Figure 6: The Time Taken by an **Afl-step** as a Function of the Number of Links in the Activity Flow Network (On 64K Processor CM).

- There are five times as many rules as parameters.

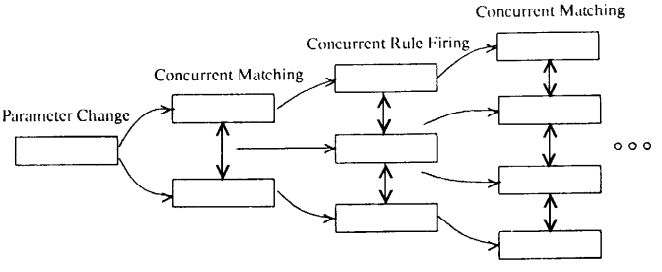
For 100,000 rules, the above assumptions require a network of 2 million links. With 2 million links an **afl-step** requires .05 seconds which allows 40 of them to be executed between questions. This is enough time for the answer to propagate the whole depth of the inferences.

7 Concurrency

Researchers have argued that one can achieve at best a constant speedup by implementing a rule-based system on a parallel rather than serial machine [Forgy 84, Oflazer 84]. They make these arguments based on rule-based systems developed for single processor machines and only consider a limited interpretation of a rule-based system. In particular, they consider a model that forces the selection of rules through a single channel so only one rule can fire at a time (the "conflict resolution" stage). This type of concurrency is pictured in Figure 7a.



a) Type discussed by [Oflazer84] and [Forgy84].



b) Type used by CIS.

Figure 7: Two Types of Forward Chaining Concurrency.

The selection of a limited set of rules is necessary for some applications, in particular problem-solving systems, but even for these applications the restriction to a **single** rule is unnecessary. Usually only certain sets of rules must be prevented from firing simultaneously. In CIS, one can prevent rules from firing simultaneously by placing a mutually exclusive group around the rules of concern.

Because much of the work is done at compile time, and because of the concurrency of AFNs, CIS can take advantage of several sources of concurrency at run time. Among these are:

- Subrule and subparameter concurrency: Within the rules and parameters all the parts act concurrently. For example, at the same time that a value activates the rules it is connected to, it deactivates all the other values of its parameter, activates the **parameter-known** node, and activates its output node.
- Concurrent matching: All the antecedent parts of a rule are matched concurrently. In fact it only takes a single **afl-step** to match every rule in the system. This is possible because the variable references are compiled out so the variable slots do not become bottlenecks.
- Concurrent forward propagation: All the rules can propagate their inferences concurrently. There can potentially be a large fan-out so that a single change could propagate to make thousands of changes in just a few **afl-steps**. Note that this offers much more concurrency than the model considered by Forgy [1984]; Figure 7 shows the difference between the two types.
- Concurrent backward propagation: A completely concurrent AND/OR search is executed from the "goal" parameter to the parameters which can affect it. Unlike the concurrent implementations of logical inference systems discussed by Douglass [1985] and Murakami [1984], concurrent "AND" searching is does not present problems. Again, this is because the variables are compiled out.
- Forward and Backward propagation happen together: As mentioned in section 4.2.
- Concurrent question selection: With the addition of the question focusing mechanism, the system can do a concurrent search of a single question to ask the user.

Although CIS allows all these sorts of concurrency, how well the system takes advantage of them depends on the rule set being used. Since no large rule sets have been implemented in CIS, no data is available.

8 Conclusions

The initial study of CIS suggests that it is possible to implement practical systems using activity flow networks, and more generally, with connectionist or other circuit level models. The AFL-1 code needed to implement CIS is quite simple, no more complex than the code used in other rule-based systems, and the running time of CIS is very good. With expected advances in massively concurrent hardware, the times will greatly improve.

As mentioned in the paper, the AFN model imposes some restrictions on CIS. Some of these limitations can be avoided by expanding the model slightly. For example, to manipulate high-precision numbers, one could use a model that intermixes data flow and activity flow networks. To dynamically create instances of an object, one could use a system that creates extra network structure as it is needed.

Acknowledgements

I thank Phil Agre and David Waltz for their helpful comments on various parts of this work.

This research was supported by the Defense Advanced Research Projects Agency (DOD), ARPA Contract N0014-85-K-0124.

References

Ackley, D.H., Hinton, G.E., Sejnowski, T.J., "A Learning Algorithm for Boltzmann Machines", Cognitive Science, 1985, 9, 147-169.

Batali, J., Hartheimer, A., "The Design Procedure Language Manual", Memo 598, MIT AI Laboratory, September 1980.

Blelloch, G.E., "AFL-1: A Programming Language for Massively Concurrent Computers", MS Thesis, Dept. of Electrical Engineering and Computer Science, MIT, June 1986.

Davis, R., Buchanan, B., Shortliffe, E., "Production Rules as a Representation for Knowledge-Based Consultation Program", Artificial Intelligence, 1977, 8, 15-45.

Davis, R., "Meta-Rules: Reasoning about Control", Artificial Intelligence, 1980, 15, 179-222.

Douglass, R.J., "A Qualitative Assessment of Parallelism in Expert Systems", IEEE Software, May 1985, 70-81.

Fahlman, S.E., "Three Flavors of Parallelism", Proc. National Conference of the Canadian Society for Computational Studies of Intelligence, May 1982, Saskatoon, Saskatchewan, 230-235.

Fahlman, S.E., Hinton G.E., Sejnowski, T.J., "Massively Parallel Architectures for AI: Netl, Thistle and Boltzmann Machines", Proc. AAAI, August 1983, Washington D.C, 109-113.

Feldman, J.A., Ballard, D.H., "Connectionist Models and Their Properties", Cognitive Science, 1982, 6, 205-254.

Forgy, C.L., "The OPS5 User's Manual", TR, Carnegie-Mellon University, Department of Computer Science, 1981.

Forgy, C., A. Gupta, A. Newell, R. Wedig, "Initial Assessment of Architectures for Production Systems", Proc. AAAI, August 1984, Austin, TX., 116-120.

Fox, M.S., Lowenfeld, S., Kleinosky, P., "Techniques for Sensor-Based Diagnostics" Proc. IJCAI, August 1983, Karlsruhe W. Germany, 158-163.

Gaschnig, J., "Prospector: An Expert System for Mineral Exploration", in Michie (Ed.), *Introductory Readings in Expert Systems,* New York, Gordon and Breach, 1982.

Hinton, G.E., "Implementing Semantic Networks in Parallel", in G.E. Hinton and J.A. Anderson (Ed.), *Parallel Models of Associative Memory*, Hillsdale, NJ: Erlbaum, 1981.

Kruskal, C.P., Rudolph, L., Snir, M., "The Power of Parallel Prefix", Proc. Int'l. Conference on Parallel Processing, August 1985, 180-185.

McDermott, J., "R1: an Expert in the Computer Systems Domain", Proc. AAAI, August 1980, Stanford University, 269-271.

Murakami, K., Kakuta, T., Onai, R., "Architectures and Hardware Systems: Parallel Inference Machine and Knowledge Base Machine", Proc. Int'l Conf. Fifth Generation Computer Systems, 1984, Tokyo, 18-36.

Oflazer, K., "Partitioning in Parallel Processing of Production Systems", Proc. Int'l Conf. Parallel Processing, August 1984, 92-100.

Rumelhart, D.E., McClelland, J.L., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, MIT Press, Cambridge Mass., 1986.

Steele, G.L. Jr., "The Definition and Implementation of a Computer Programming Language Based on Constraints", AI-TR 595, MIT AI Laboratory, August 1980.

Sussman, G.J., Steele, G.L. Jr., "Constraints - A Language for Expressing Almost-Hierarchical Descriptions", Artificial Intelligence, 1980, 14, 1-39.

Touretzky, D.S., "Symbols Among the Neurons: Details of a Connectionist Inference Architecture", Proc. IJCAI, August 1985, Los Angeles, 238-243.

Zadeh, L.A., "Fuzzy Sets", Information and Control, 1965, 8, 338-353.