The SNAP-1 Parallel AI Prototype

Ronald F. DeMara, *Member, IEEE*, and Dan I. Moldovan, *Senior Member, IEEE*

***Abstract*— Semantic Network Array Processor (SNAP) is a parallel architecture for knowledge representation and reasoning using the marker-propagation paradigm. The primary application areas of SNAP are Natural Language Understanding and Speech Processing. A first-generation SNAP-1 system has been designed and constructed using an array of 144 Digital Signal Processors organized as 32 multiprocessing clusters with dedicated communication units, a tiered synchronization scheme, and multiported memory network. Issues in the design, performance, and scalability of a marker-propagation architecture are addressed.**

***Index Terms*— Artificial intelligence, barrier synchronization, marker-propagation, multiport memory, parallel processing, natural language understanding, SIMD/MIMD architectures.**

I. Introduction

PARALLELISM which exists in Natural Language Understanding (NLU), Speech Processing, and other Artificial Intelligence applications can be exploited to increase the execution speed, domain size, and accuracy of the inferencing process. Based on these criteria, the SNAP-1 marker-propagation prototype has been designed and constructed using off-the-shelf components. This paper describes the design decisions and tradeoffs made. Performance of SNAP-1 for linguistic parsing is also shown.

A. Need for Parallel AI Architectures

Despite recent advances in computer technology, machines remain unable to perform realistic, large-scale knowledge processing applications. This is due in part to the complex nature of AI, such as extensive ambiguities in natural languages. However, part of the problem is that current supercomputers are not well-suited for AI applications such as NLU, machine translation, speech and image understanding. These applications require very large knowledge bases and extensive computational power. Yet, present AI systems employ sequential computers almost exclusively.

In this paper, we argue that parallel processing can provide important innovations to knowledge processing. Since the AI field is broad, an effective approach to designing high performance computers for AI is to first narrow down the knowledge processing paradigm for a specific application. Then one is faced with either mapping a clearly defined paradigm into a general-purpose machine or designing a novel architecture for that model of computation. When the paradigm is well understood, bottlenecks with mapping it into general-purpose computers can be identified. It then becomes advantageous to consider special-purpose architectures for the problem. This was the situation with the SNAP project at University of Southern California. We selected a paradigm called *semantic network marker propagation* which proved to be quite viable for written and spoken NLU. After programming certain examples on the CM-2 and iPSC/2 hypercube, we concluded that greater performance would be achieved by designing a specialized architecture for marker-propagation.

The SNAP-1 prototype is capable of real-time NLU using a vocabulary of a few thousand words. Furthermore, it provides a testbed for an architecture which is being designed to handle a one-million concept knowledge base.

B. Semantic Networks for Linguistic Processing

Semantic networks have frequently been used to represent and process structural knowledge. They consist of *nodes* which represent concepts within a domain, and *links* which show relationships between nodes. Each node is also assigned a *color* to indicate the type of concept or class which it belongs to. This provides a flexible representation scheme. There are many ways to encode the same knowledge depending upon the choice of relations, granularity of the nodes used, the network structure, and other criteria.

In the SNAP project, our goal was to construct a knowledge base for linguistic processing [1]. The network was structured hierarchically with more general or abstract concepts at the upper levels and more specific concepts near the bottom. In order to handle the complex and vast linguistic knowledge within a domain, the knowledge base was organized into several layers. The major ones are: 1) the lexical layer at the bottom of the hierarchy, 2) semantic and syntactic constraints in the middle, and 3) concept sequences at the highest layer as shown in Fig. 1.

The lexical layer contains all the words in the vocabulary. Each lexical node connects to one or many other nodes in the layer above. For example, the word `we` connects to the nodes `animate` and `noun phrase` in the syntax module by an `is-a` or subsumption link. For each word, semantic constraints are represented as links between the appropriate nodes. Concept sequences denote basic linguistic patterns which fit many possible utterances. Each concept sequence has a root and elements. For example, the activity of "seeing something" is encoded by the `seeing-event` root with `experiencer`, `see`, and `object` elements. The experiencer must be animate and a noun phrase which are semantic

Manuscript received October 1, 1990; revised March 22, 1993. This work was supported by the National Science Foundation under Grant MIP-90/09109.

R. F. DeMara is with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816.

D. I. Moldovan is with the Department of Electrical Engineering–Systems, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 921056.

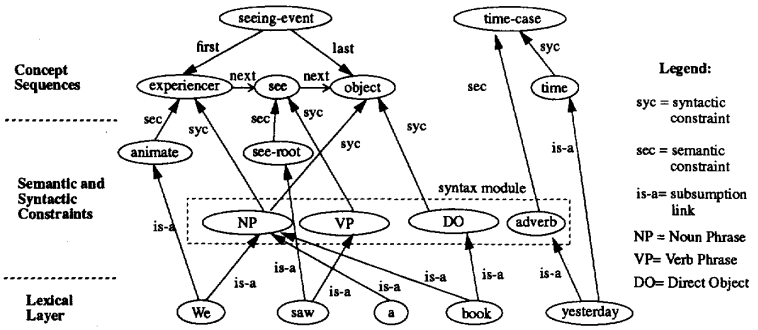

Fig. 1. Semantic network for NLU.

and syntactic constraints, respectively. Optional constituents, such as the time at which an event occurs, are represented as auxiliary concept sequences. For instance, the `time-case` concept sequence is combined with a `seeing-event` basic concept sequence to indicate when it happened, e.g., `yesterday`.

In order to perform natural language understanding for realistic applications, a very large knowledge base is needed. As part of the SNAP project, a knowledge base consisting of a 10 000 word lexicon and over 20 000 nonlexical concepts was developed. Roughly 15K nodes (75%) represent basic concept sequences, 3K (15%) compose the concept-type hierarchy, 1K (5%) form syntactic patterns, and 1K (5%) are used for auxiliary concept storage. A knowledge base of this size is sufficient for encoding information about domains such as "terrorism in Latin America." Within this domain, we have processed tens of pages of newswire text [12] by performing inferencing operations on the semantic network.

C. Marker-Propagation Model

A semantic network knowledge base is only part of the reasoning system. It can be regarded as a static infrastructure which allows the transfer of information between concepts in the domain. The dynamic agents of inference which move knowledge around are implemented as *markers* [4], [5], [7]. Markers are data patterns associated with each node. They represent properties of nodes, membership in different sets, and reflect the state of hypotheses as they travel through the semantic network.

Whenever a marker encounters new nodes, it may change the state of knowledge associated with these nodes. Complex reasoning operations can be achieved by controlling the movement of markers through the semantic network as determined by *propagation rules* which are attached to markers. Each marker individually selects which paths to follow and those to avoid. To quantify properties, markers are given a value which serves as a *measure of belief* during inferencing, such as the cost of accepting a particular concept sequence. They also carry a lightweight arithmetic or logical operation which is performed along each propagation step. This is executed to update values or influence the status of other markers, as will be shown in Section II.

II. System Architecture

SNAP-1 is an implementation of the marker-propagation model. The main difference between marker-propagation and message-passing is the absence of destination addresses for markers, unlike in message-passing systems. To create an architecture for marker-propagation, a set of design goals and several architectural features were developed for the major aspects of the paradigm.

A. System Overview

As shown in Fig. 2, the SNAP-1 system consists of an *array* of 144 processing elements (PE's) to store and process the semantic network, a *controller* which manages the array, and a Sun *host* for the user-interface. Application programs are written and compiled on the host using C language and high-level SNAP instructions for marker-passing. To avoid a bottleneck with the VME bus, the object code for an entire application is downloaded to the controller before execution. The controller manages the execution flow in the application and broadcasts SNAP instructions to the processing array for execution. The array is organized as 32 tightly-coupled clusters of four to five PE's each.[1] Communication occurs over a high-speed backplane which provides separate interconnection networks for instruction broadcast, message transfer, and performance gathering.

The semantic network is stored as a distributed knowledge base. A partitioning function is applied to divide the network into regions. Each region is allocated to a cluster which processes all of its concepts, relations, and markers. The mapping function is variable with up to 1024 nodes per cluster using sequential, round-robin, or semantically-based allocation.

Although operations are initiated by the controller, most of the processing is performed within each cluster. As input words are read from a natural language sentence, the controller broadcasts instructions to set markers on the corresponding lexical nodes. Markers are then propagated upward through the semantic and syntactic layers. As the markers move they perform constraint checks and activate the suitable concept sequences. Markers are propagated in parallel by the PE's

[1] Presently, 16 clusters are implemented in the full five PE configuration while the remaining 16 clusters have four PE's each, totaling 144 PE's.

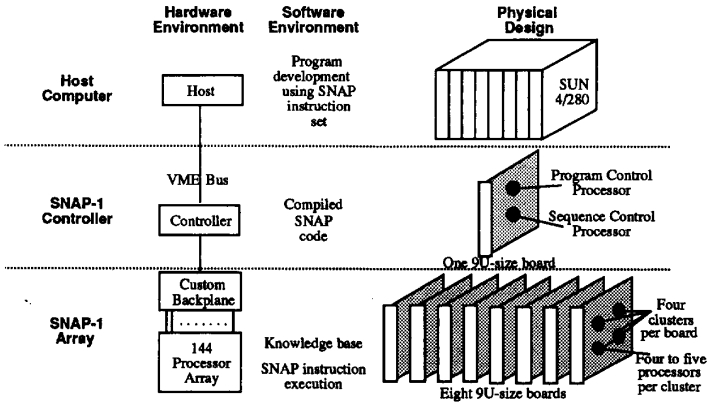

Fig. 2. SNAP-1 system.

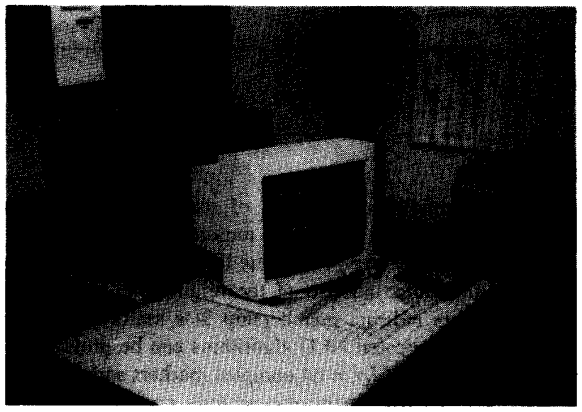

Fig. 3. SNAP-1 array board and Sun host.

TABLE I
DESIGN OBJECTIVES

Requirements	Functionality	provide an AI development platform for coding, executing and analyzing marker-passing applications
		implement the complete set of 20 SNAP instructions
	Capacity	store a 32K node semantic network knowledge base
		provide 10 relations, 128 markers and floating-point registers per semantic network node
	Performance	achieve real-time (subsecond) NLU parsing per sentence
	Instrumentation	support performance experiments to conduct design tradeoffs
Constraints	Hardware	use only off-the-shelf components
	Connectivity	construct an interconnection network within pin limitations of 9U-size backplane
	Complexity	maintain a low part count to reduce development time

within each cluster. When propagation terminates, the controller retrieves the parsing result by collecting the names of all nodes with the required markers and weights.

As shown in Fig. 3, the design physically consists of several large (9U-size) circuit boards. The VME interface and controller reside on a single board. It contains a program control processor (PCP) to manage the execution-flow of the application code and a sequence control processor (SCP) to regulate processing in the array. Each of the eight identical array boards contain the processors, local memory, shared memory, and interconnection interface for four clusters.

B. Design Goals

The design requirements and constraints for SNAP-1 are listed in Table I. The primary objective was to provide real-time NLU while supporting a vocabulary of a few thousand words.

Functionality: The requirements of the architecture were validated through functional simulation. First, a preliminary instruction set for marker-propagation was defined and an instruction-level simulator was constructed [10]. NLU [8], concept classification [6], and property inheritance [13] applications were coded with these instructions. Instruction profiles, propagation patterns, and communication overhead were analyzed to formalize 20 high-level instructions for marker-passing [13].

As shown in Table II, *node maintenance* instructions load the knowledge base by specifying each `source-node`, `relation`, `weight`, and `end-node` comprising the semantic network. Once the knowledge base is created, *search* operations initialize a `marker` with a given `value` at a specified `node`, `relation`, or `color`. Movement of markers is initiated by *propagation* instructions. They send `marker-2` from nodes with `marker-1` along a path defined by the propagation rule. Propagation rules have the format of `rule-type(r1,r2)`. The pre-defined or custom `rule-type` guides the flow of markers. It specifies a traversal strategy for passing through relations `r1` and `r2`. For example, the propagation rule `spread(r1,r2)` sends markers along a chain of `r1` links until a link of type `r2` is encountered at which time they switch to `r2`. Along each relation traversed,

TABLE II
INSTRUCTION SET FOR MARKER-PROPAGATION [13]

Type	Instruction	Operands
Node maintenance	CREATE	`source-node, relation, weight, end-node`
	DELETE	`source-node, relation, end-node`
	SET-COLOR	`node, color`
Search	SEARCH-NODE	`node, marker, value`
	SEARCH-RELATION	`relation, marker, value`
	SEARCH-COLOR	`color, marker, value`
Propagation	PROPAGATE	`marker-1, marker-2, rule-type(r1,r2), function`
Marker node maintenance	MARKER-CREATE	`marker, forward-relation, end-node, reverse-relation`
	MARKER-DELETE	`marker, forward-relation, end-node, reverse-relation`
	MARKER-SET-COLOR	`marker, color`
Boolean	AND-MARKER	`marker-1, marker-2, marker-3, function`
	OR-MARKER	`marker-1, marker-2, marker-3, function`
	NOT-MARKER	`marker-1, marker-2`
	TEST-MARKER	`marker-1, marker-2, value, condition`
Set/clear	SET-MARKER	`marker, value`
	CLEAR-MARKER	`marker`
	FUNC-MARKER	`marker, function`
Retrieval	COLLECT-MARKER	`marker`
	COLLECT-RELATION	`marker, relation`
	COLLECT-COLOR	`marker`

an arithmetic or logical `function` updates the value of `marker-2`.

After propagation, *boolean* operations are performed globally over the semantic network by evaluating `marker-1` and `marker-2` to set or reset each instance of `marker-3`. *Set/clear* operations also update markers at all nodes. However, `marker` status is changed directly without testing its present state. *Marker node maintenance* instructions bind together concepts which have been marked. Nodes with the specified `marker` are linked to an `end-node` by creating a `forward-relation` or `reverse-relation` between them. Results are collected by *retrieval* operations which return to the controller the ID's of nodes with a specific `marker`, `relation`, or `color`. Since the functions are high-level, it is relatively straightforward to initiate parallel operations within the machine. The programmer deals with logical data structures such as markers, relations, and nodes. Their physical allocation remains transparent, regardless of the number of PE's or the size of semantic network used.

Capacity: An extensive semantic network is required for realistic NLU applications. However, 32K semantic network nodes were selected as a compromise between knowledge base size and machine cost. To expedite retrieval while storing information as compactly as possible, we organized this data into three tables. As shown in Fig. 4, the tables are partitioned and stored within each cluster. The size of each field is based on binary encoding and 32-bit floating-point values while the numbers in parentheses indicate the capacity provided.

The *node table* stores the permanent and dynamic properties associated with each of the $N = 32$K nodes. A row in the table is indexed by a physical node-ID number. It contains an arithmetic/logic function for propagation and one of the 256 colors to distinguish the node type. Marker registers contain dynamic information. Two types of markers were designed to reduce the amount of storage required. *Complex markers* provide a 32-bit floating-point value for cost calculation along with a 15-bit source address of the origin node for binding. *Binary markers* indicate membership in a set or hypothesis. Relatively sophisticated NLU algorithms can be programmed with approximately $M_C = 64$ complex markers and $M_B = 64$ binary markers at each semantic network node.

The *marker status table* holds the active/inactive state of the marker. It is packed into rows of *status words* for both complex and binary markers. Each row contains N/W words where W denotes the word length of the CPU in bits. A bit in the status word indicates if the marker is set at the corresponding node. Thus when the table is updated, the status of markers from W nodes are processed simultaneously by each PE.

The *relation table* contains the relation type, destination node ID, and floating-point weight for links at each node. Many link types are needed for the wide variety of relationships between concepts in a domain, so $R = 64$K distinct relation types are supported. Up to 16 outgoing relations can be held in the slots provided for each node. This is adequate for representing most concepts in a linguistic knowledge base. Nodes with fanout greater than 16 are divided into subnodes by a pre-processor when the knowledge base is created. The *destination node ID* provides fields for the local node number and cluster where it is physically stored. Since the relation table is indexed by node-ID, the retrieval time during propagation is small.

Performance: Parsing within a few seconds or less is sufficient for applications such as bulk text understanding or speech-to-speech translation. Thus the performance objective was to be able to understand input sentences in real-time. Execution time grows as the size of the semantic network is

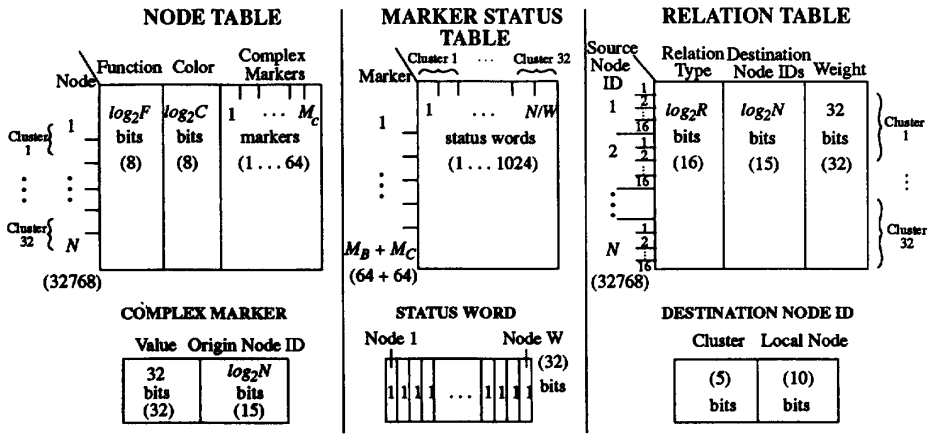

Fig. 4. Memory organization and field sizes.

```
     /* propagate markers to identify concept sequences in the proper context */
        parbegin
            /* set marker m1 on the node NP */
L1:         SEARCH-NODE (NP, m1, 0.0);
            /* set marker m2 on VP and DO */
L2:         SEARCH-NODE (VP, m2, 1.0);
L3:         SEARCH-NODE (DO, m2, 1.0);
        parend
        parbegin
            /* propagate new marker m3 from all nodes which have m2 */
L4:         PROPAGATE (m2, m3, spread(is-a,next), NOP);
            /* propagate m4 from nodes with m1 */
L5:         PROPAGATE (m1, m4, spread(is-a,last), ADD);
        parend
        /* set m5 on nodes with both m3 and m4 */
L6:     AND-MARKER (m3, m4, m5)
        /* collect the names of nodes with m5 set */
L7:     COLLECT-NODE (m5);
```

Fig. 5. Marker-propagation program for Fig. 1.

increased, and inferences requiring a large domain can become intractable. However, numerous markers can be propagated simultaneously to evaluate hypotheses in parallel. To quantify performance, we have developed an integrated measurement system for evaluating marker-propagation algorithms, partitioning functions, communication traffic, and synchronization protocols.

C. Architectural Features

The architecture was developed by analyzing several marker-propagation programs. Some typical program code is shown in Fig. 5 for the knowledge base in Fig. 1. Specific processing requirements are described below.

Levels of Parallelism: Parallelism in the architecture corresponds to three phases in marker-propagation algorithms: configuration, propagation, and accumulation phase. The *configuration phase* sets initial conditions throughout the semantic network. It consists of statements $L1-L3$ in Fig. 5. These statements locate origin nodes to activate the initial markers in parallel using a distributed search capability. PE's with NP, VP, and DO will set m1 or m2 in the marker status table and then initialize values in the node table to activate these nodes.

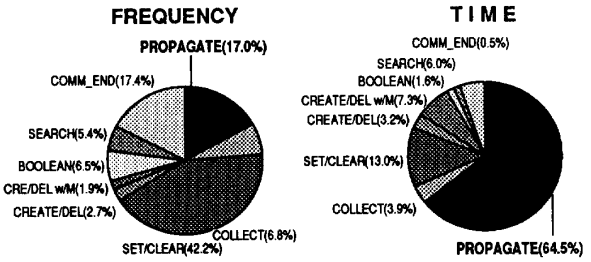

Fig. 6. Relative instruction frequency and execution time.

Markers travel through the semantic network to other nodes during the *propagation phase* in $L4$ and $L5$. While initiated on a global basis, markers are propagated under local control only. For example, when $L5$ is executed, all PE's check if they have any nodes with m1 set. If so then they propagate m4 along a path dictated by the `spread(is-a,last)` propagation rule. Each PE evaluates the rule individually by searching the relation table for `is-a` and `last` links. At every propagation step, the weight of the link is added to the value of m4. The *accumulation phase*, in $L6$ and $L7$, occurs after propagation terminates by comparing markers and then retrieving them. For instance, $L6$ performs a global set intersection. Each PE inspects its portion of the semantic network to update m5 on nodes which have both m3 and m4. When $L7$ is executed, nodes with m5 are collected in parallel by each cluster for use in the primary application thread.

Instruction profiles were measured for NLU applications on a single processor to determine frequency of use and relative execution time. Fig. 6 shows that while the number of PROPAGATE operations is only 17.0% of the total instructions executed, they consume 64.5% of the overall processing time. Thus propagation should be optimized since it dominates execution time.

SNAP-1 reduces propagation time by exploiting two types of parallelism. Intra-propagation parallelism, or *α-parallelism*, is derived from searching relation links and transmitting markers within a single PROPAGATE statement. Let α denote the

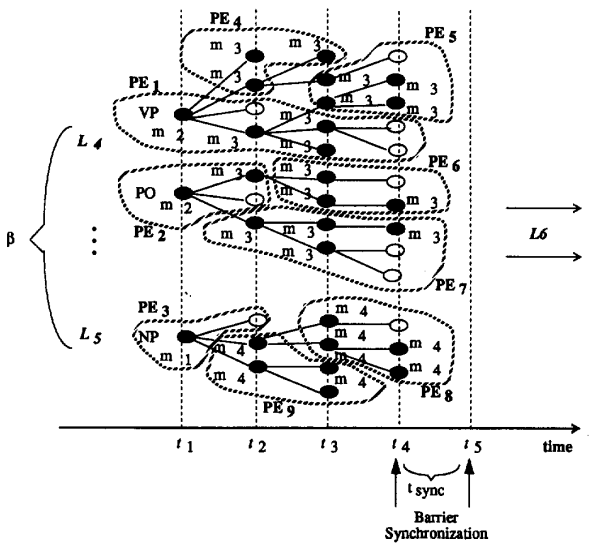

Fig. 7. Barrier synchronization during propagation.

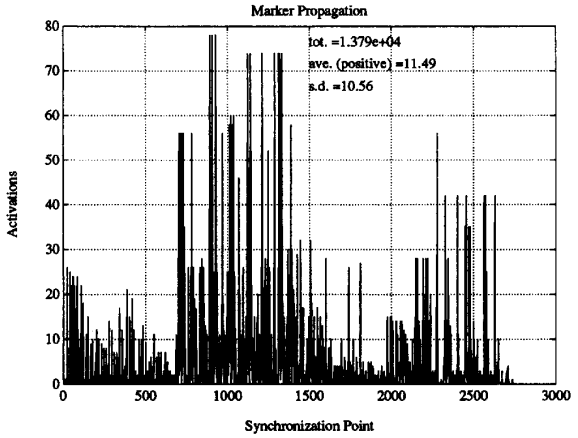

Fig. 8. Marker traffic during execution.

number of nodes activated simultaneously by a propagate instruction. For example, when $L4$ is executed m3 must be propagated from VP and DO because m2 is set on both nodes. They are active in a data-parallel manner so array processing is effective. Concurrently, each PE in the array propagates markers between its active nodes. Inter-propagation parallelism, or *β-parallelism*, exists between $L4$ and $L5$ since there are no data dependencies in the markers used. Let β denote the number of overlapped propagation statements.

Parallelism was analyzed in two marker-propagation algorithms. The PASS speech understanding program had $\beta_{\text{ave}} = 2.8$ and $\beta_{\max} = 6$ while the DMSNAP [8] NLU program had slightly less inter-instruction parallelism with $\beta_{\text{ave}} = 2.3$ and $\beta_{\max} = 5$. For both applications, α-parallelism was highly variable during execution, ranging between 10 and 1000 depending on the length and breadth of the propagation path through the knowledge base. While β is a function of the algorithm, α typically grows as the size of the semantic network is increased.

SIMD/MIMD Processing: To control the execution of α and β parallelism, aspects of both SIMD and MIMD processing are employed. During SIMD mode, a central controller broadcasts one or more SNAP instructions over a global bus to all PE's. This utilizes α-parallelism by providing simultaneous access to many nodes to set initial conditions and collect results. Yet each PE retains some autonomy for deciding how to react to input conditions and which procedures to execute. Thus after broadcast, propagation occurs asynchronously in MIMD mode using only local control. Moreover, utilization is increased by overlapping several PROPAGATE statements to capture β-parallelism.

Barrier Synchronization: As shown in Fig. 7, the data dependencies of propagation must be respected. Before $L6$ can be executed, the PE's which are propagating markers need to be synchronized because of the data dependency with $\{L4, L5\}$. However, it is not known *a priori* how many propagations take place or which PE's are involved. Thus hardware support is desirable to minimize the synchronization overhead time, t_{sync}.

One solution is to implement a wired-logic AND-gate from each processor to detect an idle state. However, one may wish to synchronize on a per marker basis. SNAP-1 uses a tree of AND-gates and counters to report marker creation/termination counts from each processor and maintain a centralized total. The synchronization requirement is that the total number of markers produced equals the total number of markers consumed by all PE's. For example, in Fig. 7, the first level of propagation is executed at time t_1. PE's with VP, DO, and NP propagate m3 and m4 between their local nodes and also send out markers to PE_4, PE_7, and PE_9, respectively. Markers reach terminal nodes at time t_4. Completion of propagation is detected by comparing the total number of messages produced and consumed between PE's.

Interprocessor Communication: On average, roughly 100 to 200 marker propagations occur in parallel over a 32K node knowledge base. Since simulation indicated that marker-propagation was the bottleneck, resources were shared by organizing a commensurate number of PE's into tightly-coupled clusters of two or more marker units. The clustered topology accommodates the 288-pin limitation of 9U-sized backplanes. In particular, network complexity is reduced since only messages between clusters have to be routed through the interconnection network. A total of 32 clusters provide 80 marker units to match the available parallelism while setting granularity at 1K nodes per cluster. Simulation of NLU, speech processing, and knowledge classification programs showed that this provides a good balance between PE utilization and communication overhead [10].

The network capacity is based upon the time distribution of marker traffic. As shown in Fig. 8, parsing generates bursts of marker activation. The vertical axis represents the number of marker activation messages which occurred at each barrier synchronization in the program as indicated on the horizontal axis. While on average 11.49 messages are transmitted per synchronization point, bursts of over 30 messages are typical. When a burst occurs, the interconnection network must be

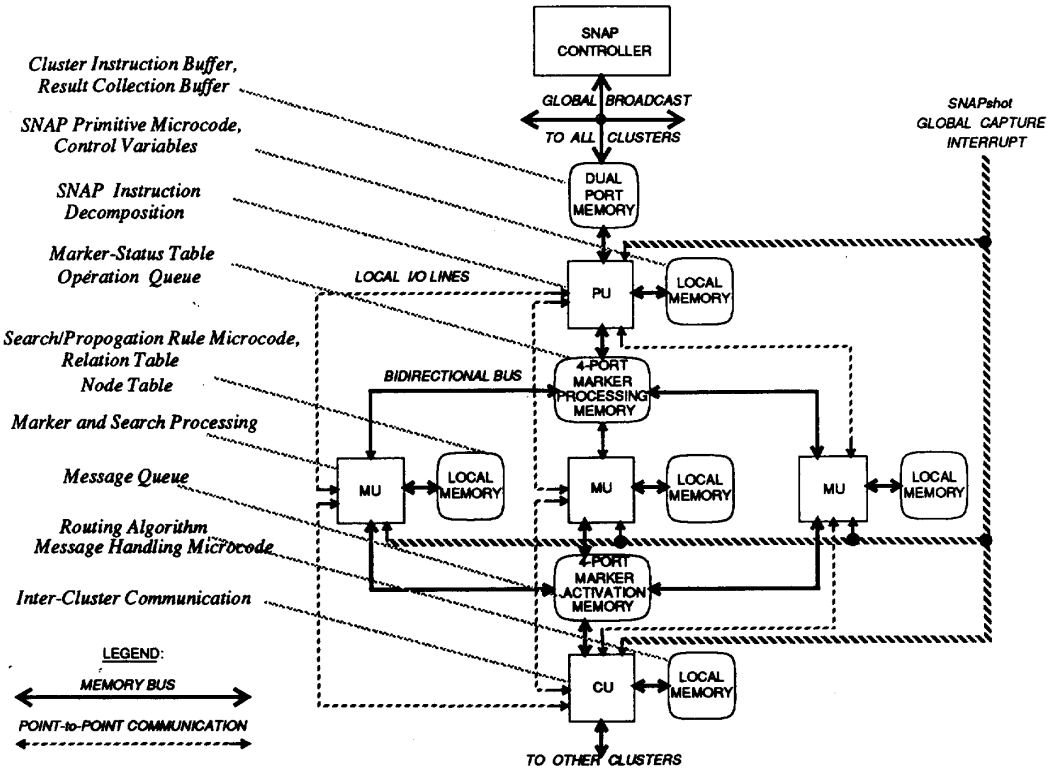

Fig. 9. Cluster design.

able to absorb it, otherwise the sending processor will be blocked. Since the critical path of execution is marker flow, transmission latency needs to be minimized.

A cost-effective approach is to interconnect PE's with *multiport memories*. Within a cluster they implement concurrent-read-exclusive-write (CREW) access between functional units. Externally, multiport memories provide a large buffering capacity. Latency is reduced by using DMA between multiported memory regions.

III. Hardware Design

A microprocessor-based design was selected to reduce development time and risk over custom VLSI while allowing features such as propagation rules to be implemented in software. The design of the processing array, interconnection network, and central controller are described below.

A. Processing Array

The SNAP-1 array consists of 32 clusters of functional units for instruction control, marker processing, and external communication. Each functional unit is implemented as a Digital Signal Processing (DSP) microprocessor chip to provide the required MIMD capability described earlier.

Functional Elements: A block diagram of the SNAP-1 cluster is shown in Fig. 9. The functional elements consist of a *processing unit* (PU), up to three *marker units* (MU's), and a *communication unit* (CU). Each functional unit has local memory and accesses multiported memory regions for input, output, and common data using the hardware design shown in Fig. 10. The functional units execute three stages of SNAP instruction processing.

The PU decodes instructions and acts as the master of the cluster. First, an instruction which was broadcasted by the controller is dequeued from the dual-port memory. As shown in Fig. 10, Busy/Access control signals are exchanged with the 2K x 32 dual-port to control opcode and operand flow. The PU decomposes each instruction based on the opcode type according to the emulation microcode in its 256K local memory. Marker-propagation tasks corresponding to each instruction are placed in the *marker processing memory* which is the primary shared-memory within the cluster. CREW multiport memory control is provided by programmable array logic. The PU continues processing until any of the following occur: a COLLECT-NODE opcode is received, a COMM-END barrier synchronization is requested, or the queue is full. The PU then uses point-to-point control to serialize MU processing. Each PU maintains its own circular instruction queue so up to 64 instructions can be overlapped based on the workload within each cluster. Tasks which the PU has enqueued in the marker processing memory are then executed asynchronously by one of the available MU's.

The role of the MU is to process markers and search the knowledge base. It maintains the knowledge base tables in the marker processing memory and local SRAM as shown in Fig. 10. Each MU performs global operations, such as

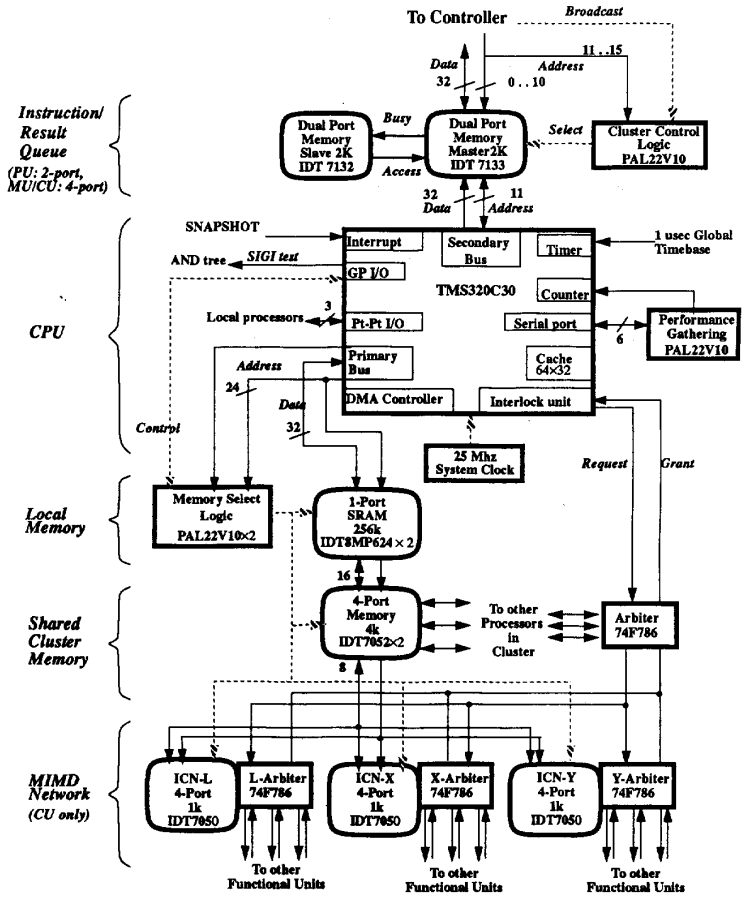

Fig. 10. Functional unit design.

boolean and set/clear instructions, for 32 nodes at a time by manipulating the marker status table. For example, when AND-MARKER(m2,m3,m5) is executed, the rows corresponding to m2 and m3 are fetched. A logical AND of these words updates the status of m5 at the 32 nodes which correspond to each bit. The MU executes PROPAGATE in a breadth-first pattern by first retrieving entries in the marker status table that correspond to the source marker. If a word is nonzero then node ID's are computed for all bits which are set. Each node number provides an index into the relation table which is searched for links as specified in the propagation rule in local memory. If the destination node is within the cluster, the MU proceeds to set the bit in the marker status table. For each arithmetic or thresholding operation, the floating-point value in the node table corresponding to that marker is updated. If a destination node is outside the cluster then an activation message is placed in the *marker activation memory* for transmission by the CU.

The CU provides an interface between clusters and can also task the MU's with jobs. The CU transmits messages from the local marker activation memory to other clusters via a 4-ary hypercube interconnection using independent primary and secondary buses as shown in Fig. 10 to allow DMA between the marker activation memory and interconnection network (ICN) four-port memories: ICN-L, ICN-X, and ICN-Y. Incoming messages are disassembled and relayed to the destination cluster where they are enqueued in the marker activation memory for execution by one of the MU's.

Processor Selection: The choice of microprocessor was based on requirements for emulating the SNAP instruction set. In particular, single-cycle execution was needed because data movement and bitwise logical operations comprised the majority of the instruction count as shown in Fig. 6. Speech and NLU applications also required high-speed floating-point arithmetic to compute strength values of competing hypotheses. As shown in Fig. 10, the Texas Instruments TMS320C30 DSP was selected to provide single-cycle 32-bit logical and multiplication instructions along with on-chip support for performance gathering and arbitration of multiport memories.

Intra-cluster Communication: As shown in Fig. 9, each functional unit has access to multiported memories. They eliminate bus contention problems while minimizing design complexity and cost, but have the drawback of small capacities. Thus two separate memory regions have been provided for marker processing and activation. Integrated Device Technology's four-port memories are used to provide simultaneous

access from four independent ports without read contention [11]. This creates a significant potential for parallelism, but introduces the need for controlled memory write access within the cluster. Three types of memory traffic must be regulated:

- *Type-1* traffic consisting of variables shared within the cluster such as bit-markers and locks within the marker processing memory,
- *Type-2* traffic consisting of microinstructions forwarded by the PU to the MU's and processing results returned by the MU's to the PU via the marker processing memory, and
- *Type-3* traffic consisting of information forwarded between clusters which is off-loaded from the MU's to the CU via the marker activation memory.

Multiport Access Control: A coherent access protocol must be provided for each type of traffic. Since multiport memories allow concurrent reading of the same location, the read portions of read-modify-write instructions may be executed simultaneously. During access to a semaphore, each competing process will claim ownership. Thus traditional operations such as *test-and-set* are insufficient for critical sections required for type-1 traffic.

The cluster arbiter in Fig. 10 resolves the problem by assuring mutually exclusive access to a *semaphore table.* Assume the PE using port #1 holds the semaphore. Before the PE from port #2 can enter the critical section, it asserts the arbitration request line shown in Fig. 10. The interlock unit then delays execution of PE #2 until a grant is returned. The arbiter serves asynchronous requests from each port, assigning one grant at a time on a first-come-first-served basis. If multiple requests occur simultaneously, then priority is randomly assigned. Once processor #2 gains exclusive access to the semaphore table, it tests and updates the in-use flag for the desired critical section, and then relinquishes access to the table. Memory references outside a critical section do not involve the arbiter and are executed in parallel from all ports without delay. This is the case with type-2 and type-3 traffic by dividing the memory into separate queue areas and using a single writer with single reader protocol.

B. Interconnection Network

SNAP-1 uses three independent networks to increase bandwidth while handling synchronization and control requirements. Broadcast of SNAP instructions, transmission of marker messages between PE's, and gathering performance data occur in parallel.

Global Bus: The *global bus* is used for broadcast and collection between the controller and all clusters in the array. As shown in Fig. 10, 32-bit data and 16-bit address lines are provided from the SCP to the dual-port memories on each array card. The upper address bits either select the destination cluster or are masked out using a broadcast-control signal asserted by the SCP. When broadcast is disabled, the memory interface is bidirectional between the SCP and a single cluster to allow both broadcast and retrieval from the array.

4-ary Hypercube Network: Messages between clusters are routed through a *4-ary hypercube* network. The CU's are interconnected by three different four-port memories as shown in Fig. 11. The four clusters on each board communicate using the *L-memory* which is dedicated to messages that are local to that board. The other four-port memories, the *X-memory* and *Y-memory*, are dedicated to off-board communication in the x-dimension (horizontally in Fig. 11) and y-dimension (vertically). High-output buffers are used to drive memory buses over the custom backplane at full speed. Because the ICN memories are mapped into the address space of the CU, their physical location is transparent including memories on remote boards. The topology is similar to a spanning-bus hypercube with spanning buses replaced by multiport memories. Since each memory port is dedicated to a single CU, there is no bus contention.

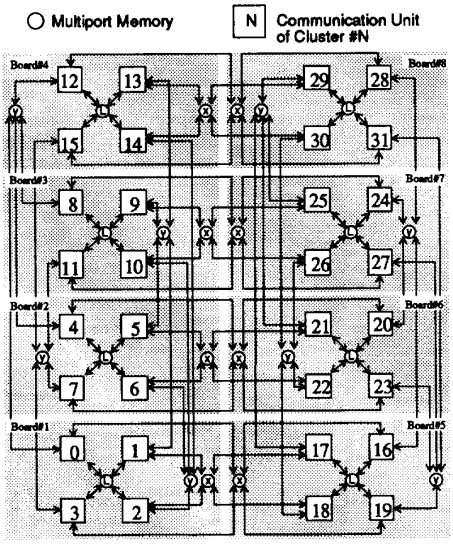

Fig. 11. 4-ary hypercube interconnection network.

Routing is performed using the address of the destination cluster. The 5-b address for each of the 32 clusters is paired to form modulo-4 fields. For example, the CU of cluster number 23 has address $23_{10} = \overbrace{1}^{X}\ \overbrace{01}^{Y}\ \overbrace{11}^{L}{}_2$ and communicates with all CU's which vary by exactly one 2-b field, either X, Y, or L. The number of transfers required is $\lceil \log_4 N \rceil = O(\log N)$. Thus 32 clusters can be accommodated with at most three intermediate hops.

The hypercube network supports 8-b parallel message-passing in 80-ns from port to port. Messages are written into mailboxes for the CU's connected to each ICN memory. The length of the message is 64 b and includes the marker, value, function, destination address, first origin address, and propagation rule. Since the microcode table of propagation rules is downloaded at compile-time, each marker only needs to carry a single-byte token indicating the function to be performed. Thus, fixed-sized messages are used regardless of the complexity of the propagation rule.

Performance Collection Network: A separate network is desirable for gathering performance data at minimal levels of perturbation. Otherwise, transmission over a primary

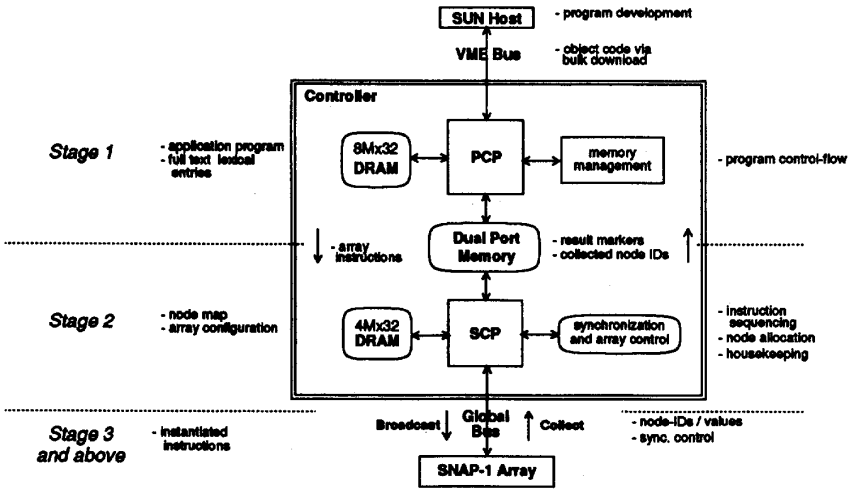

Fig. 12. SNAP-1 instruction pipelining.

ICN will degrade communication bandwidth. An independent *performance collection network* is provided as part of the instrumentation system. Each PE sends performance data to the central collection board via 2-Mb/s serial links. When triggered by a monitoring event, the PE under observation writes an 8-b event code and 24-b status word to its serial-port register. It then resumes execution without delay while the serial-port controller shifts out the data to the network. When the data is received at the central collection board, it is stored in a FIFO queue along with an event timestamp for analysis or transfer to mass storage.

C. Central Controller

Host overhead can significantly impact execution time when semantic networks are processed on parallel machines [2], [3]. On SNAP-1, a dual-processor controller is used to offload control functions from the host.

Program Flow Control: Instruction execution is pipelined as shown in Fig. 12. The first two stages overlap control with marker processing. The *program control processor* (PCP) executes the application code to handle the loop and branch flow in the application program. The SNAP instruction stream is passed to the next stage via a FIFO queue implemented by a dual-port RAM. The *sequence control processor* (SCP) is responsible for instantiating operands in each SNAP instruction. It sequences and broadcasts the instructions for parallel execution in the array. When the pipeline is empty, housekeeping is performed including node management and garbage collection.

Tiered Synchronization Scheme: The problem with synchronization in a MIMD environment is the lack of a global view of processor activity. The controller can provide this perspective by monitoring utilization to detect if the system is idle. However, processing migrates between PE's as the markers propagate. Thus the controller needs to determine whether or not 1) all PE's are idle and 2) any markers are in transit in the ICN. Without requiring an acknowledgment, it is not possible to determine if the receiving MU has processed the propagation. Thus a tiered algorithm is used to report message production and synchronization data to the controller.

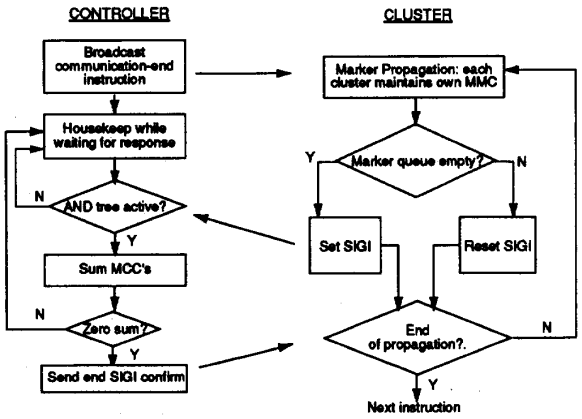

Fig. 13. Synchronization protocol.

Each PE maintains message creation/termination counts at runtime. The flowchart for the protocol is shown in Fig. 13. Tiered process creation information is added to prevent false detection while reducing message overhead by distinguishing the levels of propagation. For example, the global sum of first-level propagations created must be equal to the number of first-level propagations consumed. SNAP-1 employs this tiered protocol using the hardware support shown in Fig. 14. The AND-tree provides a synchronization interlock signal (SIGI) to the SCP when processors are idle using the general purpose I/O (GP I/O) lines from the CPU. The processors maintain a marker message counter for each level to indicate if messages are in transit. It is initialized to zero and is incremented upon each process creation and decremented after each process termination. If the processors are idle and the counters sum to zero, then the propagation has terminated and the barrier is complete.

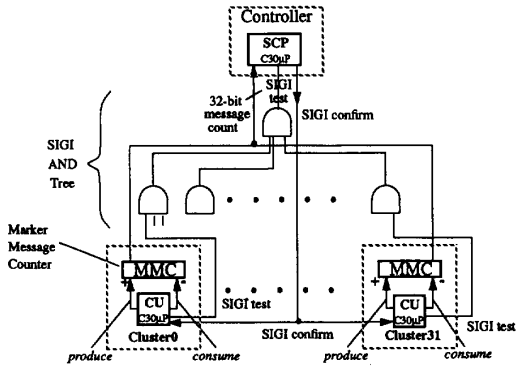

Fig. 14. Tiered synchronization hardware.

TABLE III
INPUT TEXT FROM MUC-4 CORPUS [15]

	Input Text
S1:	POLICE HAVE REPORTED THAT TERRORISTS TONIGHT BOMBED THE EMBASSIES OF THE PRC AND THE SOVIET UNION.
S2:	THE BOMBS CAUSED DAMAGE BUT NO INJURIES.
S3:	A CAR BOMB EXPLODED IN FRONT OF THE PRC EMBASSY, WHICH IS IN THE LIMA RESIDENTIAL DISTRICT OF SAN ISIDRO.
S4:	MEANWHILE, TWO BOMBS WERE THROWN AT A USSR EMBASSY VEHICLE THAT WAS PARKED IN FRONT OF THE EMBASSY LOCATED IN ORRANTIA DISTRICT, NEAR SAN ISIDRO.

IV. PERFORMANCE EVALUATION

Results from several large AI applications indicate that SNAP-1 is able to exploit parallelism to reduce execution time and slope of the scalability curve for the inferencing process. Unless otherwise noted, experiments described below were performed using a 16 cluster (72 processor) array with a 32 MHz controller and 25 MHz array PE clock speed.

Processing Time: A large NLU parsing and information extraction application has been implemented on SNAP-1 [12]. It accepts newswire text as input and generates the meaning of the sentence as output. Parsing is performed by passing markers through a knowledge base about terrorism in Latin America. The knowledge base contains approximately 12 000 semantic network nodes and 48 000 links. Results for parsing time for the sentences in Table III are shown in Table IV. Real-time performance is obtained and sentences can be parsed more quickly than a human can read them.

Most sentences can be processed with around 400–900 SNAP instructions. Each instruction varies in execution time from 50 μs for SET/CLEAR operations to several hundred microseconds for PROPAGATE, depending on the length of the path traversed. The maximum distances of any path of individual propagations ranged from 10 to 15 steps. These occurred during propagations along any of two different relations. Parsing time has been broken down into time for the phrasal parser (P.P. time) and the memory based parser (M.B. time). The phrasal parser is a serial program that executes on the controller and thus its processing time is relatively independent of knowledge base size. The role of the phrasal parser is to break down the input sentence into subparts which can be handled by the memory-based parser. Parsing times for the memory based parser are shown for two knowledge base sizes (5K nodes and 9K nodes). The parsing time increases gradually as more knowledge is added. The overall execution time is roughly proportional to the sentence length in words.

TABLE IV
EXECUTION TIMES FOR MUC-4

Input	SNAP Instr.	d_{max} Prop.	P.P. Time (s)	5K node M.B. Time (s)	9K node M.B. Time (s)
S1	473	14	0.355	0.353	0.574
S2	384	10	0.324	0.325	0.327
S3	527	15	0.524	0.479	0.765
S4	716	15	0.431	0.578	0.671

- SNAP-1: $p = 16$ Clusters
- Topology: 2-D Mesh with $L_x = L_y$

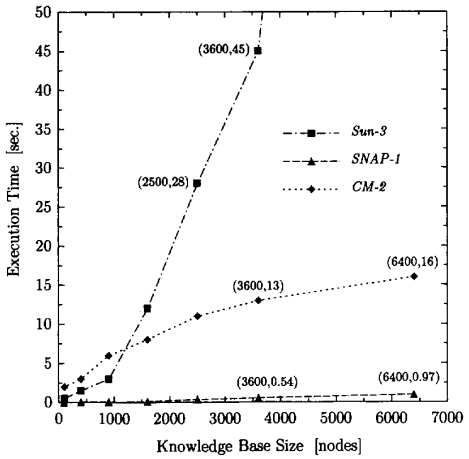

Fig. 15. Results for continutiy 2-D mesh.

Performance was also measured for some basic inferencing operations such as inheritance of attributes from concepts in the knowledge base hierarchy [2]. As shown in Fig. 15, the advantage of parallel propagation becomes more evident as the size of the knowledge base is increased. Execution time for CM-2 is less than 10 s [2] and SNAP-1 less than 1 s for inheritance from root to leaf for up to a 6.4K node knowledge base. The low execution time on SNAP-1 was due to the MIMD capability to perform selective propagation whereas CM-2 had to iterate between the controller and array after each propagation step on the critical path. However, the slope of the increase is higher for SNAP-1 than CM-2 and the lines will cross when larger knowledge bases are used.

Processor Speedup: Speedup was measured under both α-parallelism and β-parallelism during propagation. Fig. 16 shows that to obtain speedup of 20-fold, α-parallelism on the order of 100 source activations was required. For $\alpha = 1000$, nearly linear speedup was obtained up to the full processor

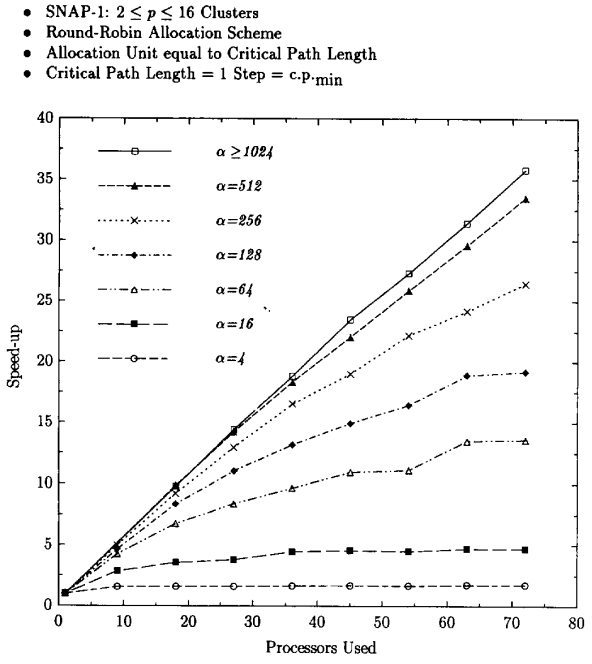

Fig. 16. Speedup obtained for α-parallelism.

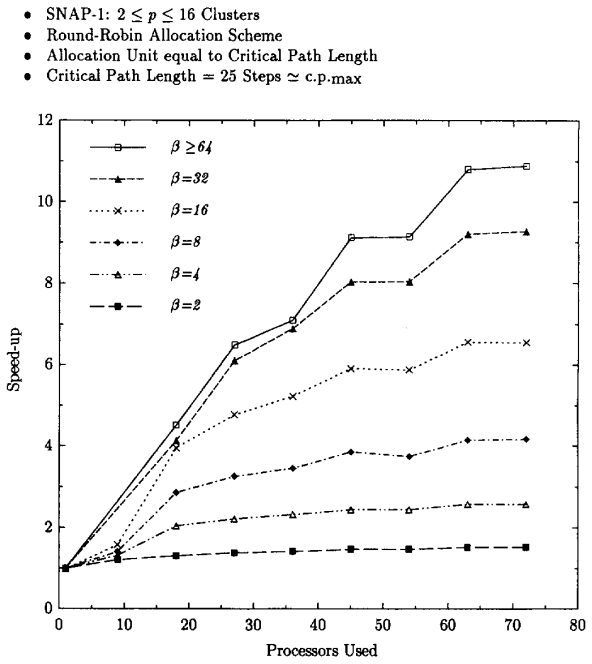

Fig. 17. Speedup obtained for β-parallelism.

configuration. Thus for typical values of α, namely $128 \leq \alpha \leq 512$, speedup ranges from 18-fold to 33-fold in a 72 processor configuration.

As opposed to α-parallelism, increasing the degree of β-parallelism above 16 had little impact on speedup as shown in Fig. 17. These results demonstrate that, in general, acceptable speedup rates can be obtained for marker-propagation programs which have degrees of parallelism $\alpha_{\text{ave}} \simeq 100$ and $\beta_{\text{ave}} \simeq 5$.

Instruction Analysis: The main technique used in SNAP-1 processing is to utilize propagation parallelism to reduce execution time. However, overhead must be controlled so that the benefits of parallelism are not outweighed by an increase in time for communication and synchronization operations. Fig. 18 shows that propagation time was reduced by nearly an order of magnitude by increasing the number of clusters from 1 to 16. Even though some instructions took slightly longer as the number of PE's was increased, they contributed only second-order effects since the amount of time required for other operations was much smaller by comparison.

While Fig. 18 showed changes in the instruction profiles for an increase in the number of processors, and Fig. 19 shows the effect of increasing knowledge base size. It shows that in general propagation dominates. Furthermore, the relative time spent on nonpropagation instruction decreases slightly as the knowledge base grows. Collection is the next most significant operation in terms of running time and more improvement could be made using interleaved memories at the controller.

However, there is some increase in the total number of propagations required as shown in Fig. 20. This occurs because more irrelevant candidates become activated which must be removed by propagating cancel markers during the multiple hypotheses resolution phase. Since large knowledge bases will add candidates which are not directly relevant, the number of propagations is not expected to exceed much more than 5000. Most other operations remained relatively constant with processing dominated by *marker set/clear* (12 000 instructions), *boolean marker operations* (11 000 instructions), and *data collection* (1000 instructions).

Processing Overhead: The components of parallel overheads in a marker-propagation system can be grouped into four categories:

1) *instruction broadcast* time incurred during the configuration phase,
2) *message communication* time between PE's required during the propagation phase,
3) *barrier synchronization* time during the transition from the propagation phase to the accumulation phase, and
4) *result collection* time required during the accumulation phase.

The influence of each component of parallel overhead is shown in Fig. 21. Due to the global bus, the broadcast overhead is small and constant. The overhead for message communication grows slowly, proportional to $\log_4 N$ for an array of N clusters. The barrier synchronization overhead is proportional to the number of processors, but the dependency is small so the degradation is acceptable. The most expensive operation is COLLECT-NODE which is proportional to the number of clusters used. The principle reason for this is overhead incurred in collecting data from separate dual-port memories within each cluster. We are currently providing feedback to the algorithm design to reduce the frequency of collection.

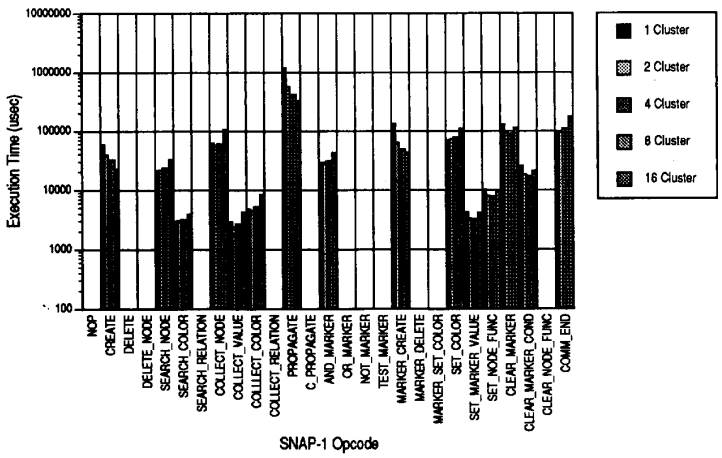

Fig. 18. Components of execution time as PE's increased.

- ATC Domain
- $N = 7988$, $L = 16592$, $Diam = 6$
- Input Sentence: "Tiger-Six Eighteen reduce speed to one-five-zero."

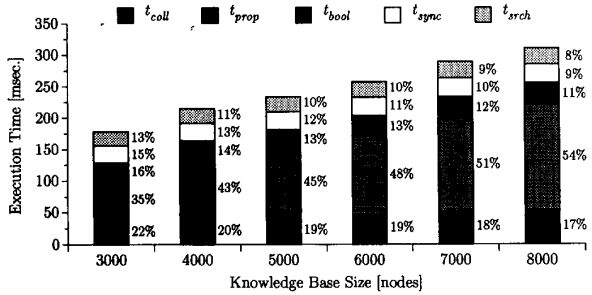

Fig. 19. Components of propagation and accumulation phases.

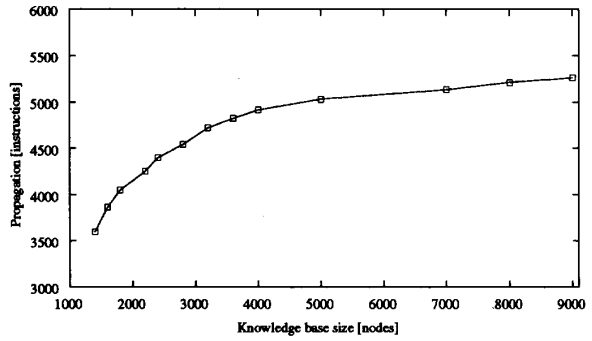

Fig. 20. Increase in number of propagate instructions.

V. Conclusion

Despite the complexity of AI applications, parallel processing can be used to significantly improve the performance of knowledge processing tasks. An effective approach is to begin with a parallel model of reasoning such as marker-propagation. Architectural features are then adapted to suit the processing requirements of the paradigm.

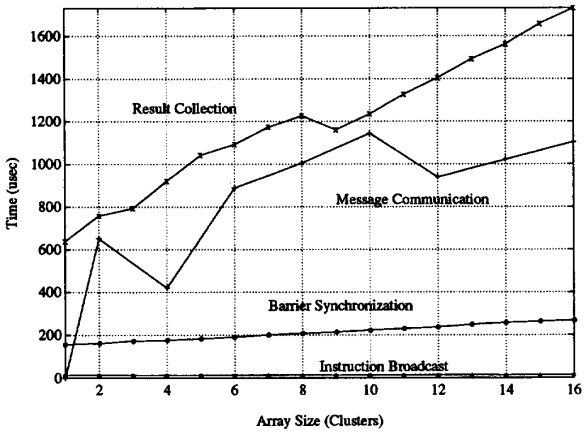

Fig. 21. Overheads of parallelism in a marker-propagation architecture.

While we have obtained encouraging results for speech, NLU and other applications, marker-propagation is not ideal for all AI applications, especially those requiring backtracking. Other limitations of our prototype include a single-user operating system, a restricted knowledge base size, and the need for parallel I/O. These issues will be addressed in the next generation machine which will provide multitasking array controllers and virtual memory using parallel attached disks.

References

[1] M. Chung and D. Moldovan, "Memory-based parsing with integrated syntactic and semantic analysis," Tech. Rep. PKP Lab 91–10, Dep. EE–Systems, Univ. of Southern California, 1991.

[2] S. Chung and D. Moldovan, "Modeling semantic networks on the Connection Machine," *J. Parallel and Distributed Comput.*, vol. 17, no.1 and 2, Jan./Feb. 1993.

[3] M. Evett, J. Hendler, and L. Spector, "PARKA: Parallel knowledge representation on the Connection Machine," Tech. Rep. UMIACS-TR-90–22, Univ. of Maryland, 1990.

[4] S. E. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge.* Cambridge, MA: MIT Press, 1979.

[5] J. A. Hendler, *Integrating Marker-Passing and Problem Solving: A Spreading Activation Approach to Improved Choice in Planning.* Lawrence Erlbaum Associates, 1988.

[6] J. Kim and D. Moldovan, "Parallel classification for knowledge representation on SNAP," in *Proc. 1990 Int. Conf. Parallel Processing,* 1990.
[7] H. Kitano, "DM-Dialog: An experimental speech-to-speech dialog translation system," *IEEE Comput.,* vol. 24, no. 6, pp. 36–50, June 1991.
[8] H. Kitano, D. Moldovan, and S. Cha, "High performance natural language understanding on the semantic network array processor," in *Proc. 1991 Int. Joint Conf. AI,* Aug. 1991.
[9] S. Kowalski, "The SNAP Simulator and Development Environment Version 6.2," Tech. Rep. PKP Lab 92–5, Dep. EE–Syst., Univ. of Southern California, Oct. 1992.
[10] C. Lin and D. Moldovan, "SNAP Simulator Result" Tech. Rep. PKP Lab 90–5, Dep. EE–Syst., Univ. of Southern California, 1990.
[11] J. R. Mick, "Introduction to IDT's FourPort RAM," Application Note AN-45, IDT, 1989.
[12] D. Moldovan, S. Cha, M. Chung, K. Hendrickson, J. Kim, and S. Kowalski, "Description of SNAP system used for MUC-4," in *Proc. Fourth Message Understanding Conf.* San Mateo, CA: Morgan Kaufmann, 1992.
[13] D. Moldovan, W. Lee, and C. Lin, "SNAP: A marker-passing architecture for knowledge processing," Tech. Rep. PKP Lab 90–4, Dep. EE–Syst., Univ. of Southern California, 1990.
[14] M. R. Quillian, "Semantic memory," Ph.D. dissertation, Carnegie Institute of Technology (Carnegie Mellon University), 1966.

Ronald F. DeMara (S'87–M'93) received the B.S. degree in electrical engineering from Lehigh University in 1987, the M.S. degree in electrical engineering from the University of Maryland, College Park, in 1989, and the Ph.D. degree in computer engineering from the University of Southern California in 1992.

From 1987 through 1989, he was an Associate Engineer at IBM Corporation, Manassas, VA, where he was involved in the design of embedded and complex systems. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering, University of Central Florida, Orlando. His research interests are in the areas of parallel processing, artificial intelligence, and computer architecture.

Dan I. Moldovan (S'76–M'78–SM'91) was born in Sibiu, Romania. He received the M.S. and Ph.D. degrees in electrical engineering and computer science from Columbia University, New York, NY, in 1974 and 1978, respectively.

He was a member of the Technical Staff at Bell Laboratories from 1976 to 1979, after which, he took on a position as an Assistant Professor of Electrical Engineering at Colorado State University from 1979 to 1981. He is currently at the University of Southern California teaching and performing research as an Associate Professor of Computer Engineering. He took a one year sabbatical leave from 1987 through 1988 to work at the National Science Foundation, Washington, DC, as Program Director for Experimental Systems in the Division of Microelectronics and Information Processing Systems. His primary research interests are in the fields of parallel processing and artificial intelligence, in which he has published over 75 papers.