Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

SIMULATION OF HETEROGENEOUS NEURAL NETWORKS ON SERIAL AND PARALLEL MACHINES

Trent E. Lange

November 1990
CSD-900041

Simulation of Heterogeneous Neural Networks on Serial and Parallel Machines

Trent E. Lange

November 1990

Technical Report UCLA-AI-90-08

Simulation of Heterogeneous Neural Networks on Serial and Parallel Machines†*

Trent E. Lange

Artificial Intelligence Laboratory
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024
lange@cs.ucla.edu

ABSTRACT

There has recently been a tremendous rebirth of interest in neural networks, ranging from distributed and localist spreading-activation networks to semantic networks with symbolic marker-passing. Ideally these networks would be encoded in dedicated massively-parallel hardware that directly implements their functionality. Cost and flexibility concerns, however, necessitate the use of general-purpose machines to simulate neural networks, especially in the research stages in which various models are being explored and tested. Issues of a simulation's timing and control become more critical when models are made up of heterogeneous networks in which nodes have different processing characteristics and cycling rates or which are made up of modular, interacting subnetworks. We have developed a simulation environment to create, operate, and control these types of connectionist networks. This paper describes how massively-parallel heterogeneous networks are simulated on serial machines as efficiently as possible, how large-scale simulations could be handled on current SIMD parallel machines, and outlines how the simulator could be implemented on its ideal hardware, a large-scale MIMD parallel machine.

1. INTRODUCTION

Connectionist networks, often known as neural networks or spreading-activation networks, have recently been the subject of a tremendous rebirth of interest, as researchers have begun to explore their advantages for cognitive models ranging from low-level sensory abilities to high-level reasoning. Connectionist models employ *massively parallel* networks of relatively simple processing elements that draw their inspiration from neurons and neurobiology, as opposed to traditional symbolic artificial intelligence (AI) models, which are generally based on serial Von Neumann architectures.

---

†Appears in *Parallel Computing 14, 287-303,* 1990 (special issue on neural networks).

* This research has been supported in part by a contract with the JTF program of the DOD and grants from the W.M. Keck and ITA Foundations. DESCARTES has been implemented on equipment donated to UCLA by Hewlett-Packard, Inc., and Apollo Computer, Inc. I would like to thank the other members of the DESCARTES group for their help in developing the simulator: Jack Hodges, Maria Fuenmayor, and Leo Belyeav. I would also like to thank Eric Melz for his helpful comments on previous drafts of this paper.

A few designers have implemented neural networks directly in special-purpose VLSI hardware (e.g. [Akers & Walker, 1988] and [Mead, 1989]) that achieves the networks' true parallelism and offers tremendous speed-up over simulation on serial machines. Unfortunately, such special-purpose hardware is too inflexible and expensive for the exploration and testing of different types of models. Because of this, there is a need for software tools to allow researchers to design and simulate the many kinds of connectionist network models.

An effective simulation environment is particularly critical for development of ***heterogeneous*** neural networks, which may contain elements with different processing characteristics or effective cycling rates, and which may be made up of modular or more complicated interacting sub-networks. Another area of research requiring simulation is that of *hybrid* networks that integrate elements from more than one paradigm of connectionist modelling, ranging from distributed and localist connectionist networks to symbolic marker-passing networks.

We have developed DESCARTES (Development Environment for Simulating Connectionist ARchiTEctureS) [Lange *et al*, 1989a, 1989b] to create, operate, and control heterogeneous and hybrid connectionist networks. This paper describes how DESCARTES simulates massively-parallel networks on serial machines as efficiently as possible, how large-scale simulations could be handled on current SIMD parallel machines, and outlines how the simulator could be implemented on its ideal hardware, a large-scale MIMD parallel machine.

2. CONNECTIONIST MODELS

Connectionist networks are made up of a large number of relatively simple computing elements, called *nodes*, which are connected by *links* [Feldman & Ballard, 1982]. Nodes are simple, neuron-like processing elements that each have an internal numeric *activation* level and a numeric *output* level that is spread to other nodes in the network. A node's activation, $a_i(t+1)$, is calculated by its *activation function* from the outputs coming from its incoming links (calculated by a *net input function*) and possibly its previous activation:

$$a_i(t+1) = f_a(a_i(t), net_i(t+1))$$

Usually the net input function to a node, $net_i(t+1)$, is the sum of its *weighted* inputs, or:

$$net_i(t+1) = \sum_j w_{ij} o_j(t)$$

where w_{ij} is the weight on the incoming link from node j to node i, and $o_j(t)$ is the output of node j at cycle t. Weights are either positive (*"excitatory"*) or negative (*"inhibitory"*) real values. Other net input functions are possible, with one common variant adding a *self-biasing* activation term to the sum of its weighted inputs.

A node's output, $o_j(t+1)$, is generally a function of its activation:

$$o_i(t+1) = f_o(a_i(t+1))$$

Both the activation and output functions for nodes vary from model to model, and are sometimes collapsed into one (i.e. the activation and output are the same). Some variants are the output function of the *percep-*

tron [Rosenblatt, 1962], which is equal to a linear threshold of the net input; the activation function of simple PDP nodes [Rumelhart *et al*, 1986], which is equal to a sigmoidal function of the net input; and the activation function of nodes in *Boltzmann Machines* [Hinton & Sejnowski, 1986], which is a stochastic rule.

2.1. Paradigms of Connectionist Processing

Within the connectionist approach there are three paradigms, each having its own advantages and disadvantages: *Distributed Connectionist Networks* (DCNs), *Localist Connectionist Networks* (LCNs), and *Marker-Passing Networks* (MPNs).

DCNs, sometimes known as *Parallel Distributed Processing* or *Subsymbolic* models, are networks which represent knowledge as *distributed* patterns of activation across their nodes. Most DCN models have learning rules, such as backpropagation [Rumelhart *et al.*, 1986], to *train* their links' weights to generate desired input/output behavior. With such training rules, DCNs are able to perform statistical category generalization, perform noise-resistant associative retrieval, and exhibit robustness to damage. They have been successfully employed for tasks such as visual pattern recognition [Fukushima *et al.*, 1983], speech consonant recognition [Waibel, 1989], and assigning roles to constituents of sentences [McClelland & Kawamoto, 1986]. On the other hand, DCNs have (so far) had difficulty with both dynamic variable bindings and the representation of structure needed to handle complex conceptual relationships, and so are not currently well-suited for high-level cognitive tasks such as natural language understanding and planning.

LCNs also use nodes with simple numeric activation and output functions, but instead represent knowledge using *semantic networks* in which concepts are represented by individual nodes and their interconnections. Unlike DCNs, localist networks are parallel at the knowledge level and have structural relationships between concepts built into the connectivity of the network. Because of this, LCNs are especially well-suited for cognitive tasks such as word-sense disambiguation [Waltz & Pollack, 1985] and limited inference [Shastri, 1988]. Unfortunately, LCNs lack the powerful learning and generalization capabilities of DCNs and also have had difficulty with dynamic variable bindings and other capabilities of symbolic models.

MPNs are unlike DCNs and LCNs in that their nodes do not use numeric activation functions, but instead use built-in symbolic capabilities. Like LCNs, they also represent knowledge in semantic networks and retain parallelism at the knowledge level. Instead of spreading numeric activation values, MPNs propagate symbolic markers, and so support the variable binding necessary for rule application while preserving the full power of symbolic systems. Because of this, they have been able to approach high-level areas such as planning [Hendler, 1988] and natural language understanding [Charniak, 1986]. On the downside, MPNs' nodes are more complex than those of DCNs and LCNs, they do not possess the learning capabilities of DCNs, and they do not exhibit the constraint-satisfaction capabilities of LCNs.

2.2. Heterogeneous and Hybrid Connectionist Models

While the majority of connectionist models are made up of a single network of homogeneous elements, a number of researchers are exploring *heterogeneous* networks made up of elements with different processing characteristics (e.g. [Shastri, 1988], [Tomabechi & Katino, 1989], [Lange & Dyer, 1989a,b,c]). Another type of heterogeneous model uses multiple interacting sub-networks that communicate via shared elements, such as *modular networks* — DCNs composed of networks that are trained separately (e.g. [Waibel, 1989], [Miikkulainen & Dyer, 1989]). Yet another area of connectionist research lies in *hybrid* models which combine elements from DCNs, LCNs, and/or MPNs in order to approach problems that would be difficult or impossible using a single paradigm (e.g. [Hendler, 1989], [Lange *et al.*, 1990b]).

Figure 1 illustrates many of the concepts of heterogeneous and hybrid networks. The figure shows a simplified network built to understand the sentence, *"John put the pot inside the dishwasher because the police were coming"* (**Hiding Pot**). Network-A in Figure 1 is a combined LCN and MPN whose nodes and connections represent part of the real-world knowledge needed to understand the sentence. The MPN supports variable-binding and rule-firing, while the LCN activates and combines evidence for individual schemas. These then combine their functionality to support predictions and perform inferencing and disambiguation[1].

The figure also shows how different connectionist approaches may be combined by having separate networks that interact with each other, where each one performs a different cognitive task. Network-B in Figure 1 is a DCN trained by backpropagation to recognize words from line segments. It feeds into Network-A to provide the input for its semantic processing. Network-B itself could be built up of two modular subnetworks — one trained to recognize letters from the line segments, and another trained to recognize words from those letters.

A final feature of heterogeneous models is the ability to have different effective cycling rates of node updating. For instance, the constraint-satisfaction disambiguation process of the LCN in **Hiding Pot** is a slower process than marker-passing. It would therefore be desirable for the LCN nodes of Network-A to cycle more quickly than the MPN nodes. One might also want to have the nodes of Network-B cycle more quickly than those of Network-A, since Network-B must first recognize the words from all of the line segments before any semantic processing can occur.

3. THE DESCARTES CONNECTIONIST SIMULATOR

While there are several existing connectionist simulators (e.g. [D'Autrechy *et al.*, 1988], [Goddard *et al.*, 1989], [Mesrobian *et al.*, 1989], and [Wilson *et al.*, 1989]), none allows the simulation of multiple interacting heterogeneous or hybrid networks, as in **Hiding Pot**, that integrate elements from more than one paradigm of connectionist modelling. We have developed the DESCARTES simulation environment specifically to address this kind of integration. DESCARTES enables researchers to design, simulate, and debug heterogeneous and hybrid connectionist architectures that combine elements of distributed, localist, and/or marker-passing networks.

DESCARTES is implemented in COMMONLISP, the ANSI Lisp standard, and the COMMONLISP Object System, CLOS, which provides hierarchical inheritance for DESCARTES classes and ensures flexibility by allowing the user to use pre-defined functional classes or create his own to customize network semantics. The DESCARTES system consists of two interactive components: network *elements*, such as nodes and links, and *processing controllers*, which organize network elements and coordinate their processing. All elements and controllers are instances of DESCARTES classes.

3.1. Processing Controllers

When DESCARTES is loaded and running, the required processing controllers are a *meta-controller* (a supervisor for all elements and sub-controllers present in the run-time system) and at least one *network controller*

---

[1]The inferencing and frame selection needed to understand sentences such as **Hiding Pot** is explained more thoroughly in [Lange & Dyer, 1989b,c], which describe ROBIN, a LCN model that performs high-level inferencing without marker-passing.

Hiding Pot

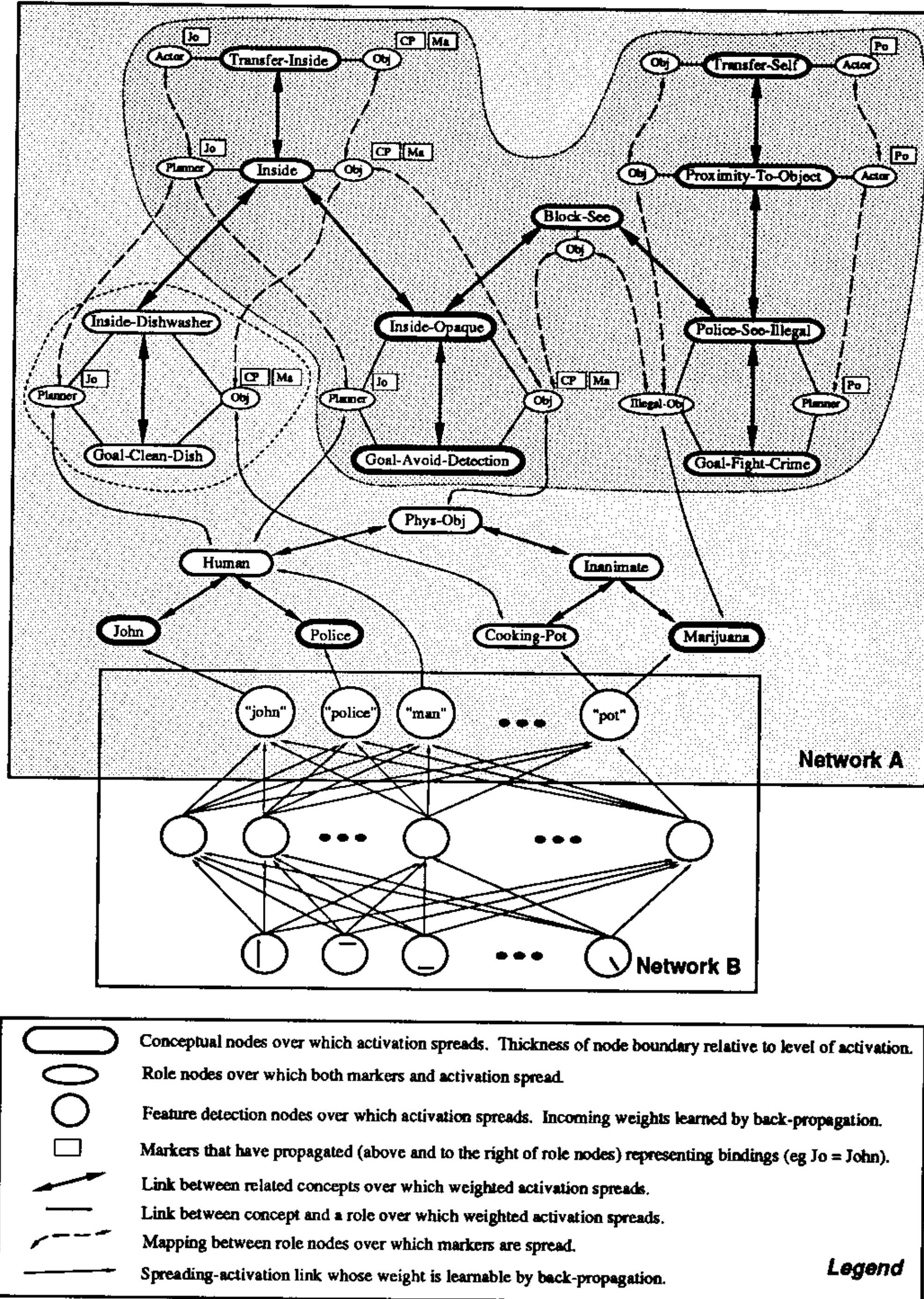

Figure 1: The sentence *"John put the pot in the dishwasher because the police were coming."* illustrates the utility of integrating semantic networks (Network-A) and distributed networks (Network-B). The darkest area represents the most highly-activated set of nodes representing the network's plan/goal analysis of the sentence. Not all markers are shown. Location role nodes and other parts of the network are also not displayed.

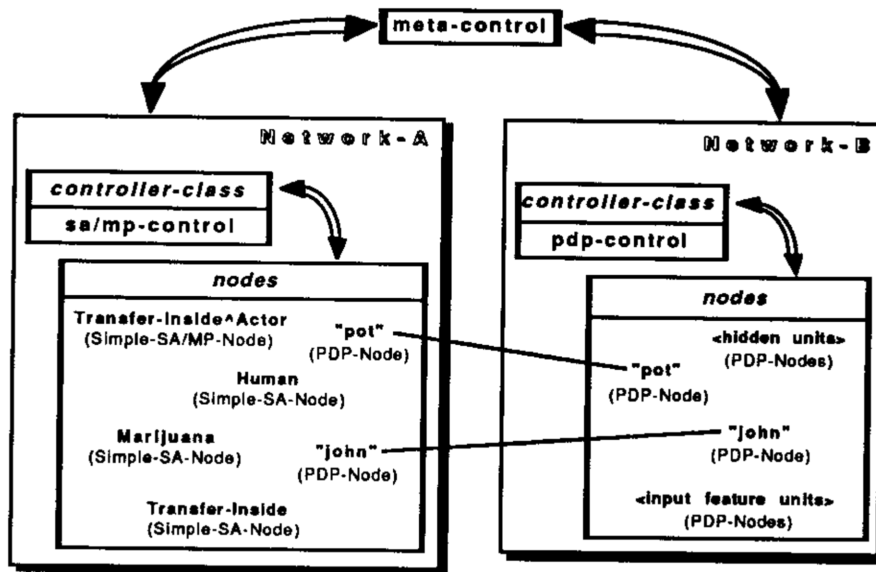

Figure 2: DESCARTES Processing Architecture applied to **Hiding Pot**. Shown in each network are a few of their nodes, with the class of each node being declared in parentheses below their names. PDP-Nodes "pot" and "john" are shared by both networks.

(a supervisor for an individual network and its elements). Figure 2 shows the architecture for **Hiding Pot**, which is controlled by a meta-controller (Meta-Control) that coordinates the two network controllers (Network-A and Network-B). In this case the controller for Network-A is of class SA/MP-Control, which combines both spreading-activation and marker-passing functionality, while Network-B is of class PDP-Control, which enables its spreading-activation nodes to be trained by backpropagation.

3.2. Node Classes and Processing Functionalities

As shown in Figure 2, each node is an instance of a specific class. Each spreading-activation node class has its own net-input, activation, and output functions, and may or may not have learning capabilities. Because many connectionist nodes are made up of combinations of similar processing characteristics, classes in DESCARTES can be built by combining different processing *functionalities* that define those characteristics. For example, see Figure 3, which shows the definitions of two of the spreading-activation node classes used in **Hiding Pot**.

Both Simple-SA-Node and PDP-Node in Figure 3 are defined to include System-Building-Block-Node, the class with the CLOS slots and methods necessary for basic node processing. The *primary* net-input function of both classes is the same — the sum of the activations on their incoming links (Net-Input-Sum-Functionality). However, PDP-Nodes include an *:around* net-input functionality (Net-Input-Around-Self-Bias-Functionality) that adds a self-biasing activation term to the primary net-input function, so that:

$$net_i(t+1)_{Simple\text{-}SA\text{-}Node} = \sum_j o_j(t)$$

$$net_i(t+1)_{PDP\text{-}Node} = \sum_j o_j(t) + bias_i(t)$$

```
(Node-Class Simple-SA-Node (Output-Local-Threshold-Functionality
                            Activation-Around-Linear-Decay-Functionality
                            Net-Input-Sum-Functionality
                            System-Building-Block-Node))

(Node-Class PDP-Node       (Learning-PDP-Functionality
                            Activation-PDP-Sigmoid-Functionality
                            Net-Input-Around-Self-Bias-Functionality
                            Net-Input-Sum-Functionality
                            System-Building-Block-Node))
```

Figure 3: Definition of two spreading-activation node classes and their processing characteristics.

Where $o_j(t)$ in each case is the output stored on incoming link j to the node, and $bias_i(t)$ is the self-biasing activation of the node. Another example is that Simple-SA-Node has an activation :around functionality (Activation-Around-Linear-Decay-Functionality) that adds a linear decay of its previous activation to its primary activation function (the default, which is equal to its net-input). Any number of such :around functionality classes can be used to modify the processing characteristics of each node class. Other node spreading-activation functionalities allow alternative *input sites* for links to a node that effect the node's processing differently than the normal incoming-links, such as the breakdown between input sites for excitatory and inhibitory links in some of the activation rules of [Grossberg, 1980]. Modification and definition of DESCARTES' other element classes is handled with similar combinations of functionality classes.

Having :around functionalities that modify each node function's primary characteristics (as opposed to simply defining a new primary characteristic for each node class) is important for two reasons. First, it allows for simple definition of new node classes that combine commonly-used processing characteristics. Equally important is that it allows for potentially substantial speed-ups for simulations on SIMD parallel machines (described in Section 5).

3.3. The Simulation Cycle

Once the networks have been designed and built, the user runs the simulation by (1) optionally defining the cycling, termination, and display sequence for each network, (2) initializing the meta-controller to clear out all activation and markers, (3) activating or marking the desired nodes, and (4) starting the cycling sequence and specifying the number of global cycles to run. The meta-controller provides for timing coordination between the networks to control different cycling rates and functionalities. Networks cycled in parallel behave as if they were a single net, even though they need not operate at the same cycling rates or with the same functionality. With serial cycling, one network may wait until another network completes a specified number of cycles or reaches stability before starting to cycle itself.

Each global network cycle is comprised of four steps: (1) determination of which networks need to be cycled, (2) update of active nodes in the cycling networks, (3) spread from active nodes in the cycling networks to their outgoing-links, and (4) report any requested output.

Determining Active Networks: The meta-controller determines which of the networks in the system need to be cycled in parallel on the given cycle, according to defaults and any user specifications. In **Hiding Pot**, for instance, spreading-activation nodes in Network-A might be cycled on every

global cycle, while marker-passing nodes might be cycled only on global cycles 1, 4, 7, and so on, until stability.

Update: Each active node in the cycling networks queries its incoming links for new activation and/or markers. Spreading-activation nodes calculate their new activation by applying their net input and activation functions, while marker-passing nodes store any new markers they have received.

Spread-To-Out-Links: Each active node in the cycling networks calculates its output (either activation or markers) and sends it to its outgoing links. The output of spreading-activation nodes is calculated by applying their output function, while the output of marker-passing nodes is generally their new markers. The links receive input from their source nodes and store it, multiplied by their weights if appropriate.

Report Output: The final step of a cycle entails querying the cycling networks for results. Each controller can optionally display the status of nodes at specified cycles or trace new activation and/or markers.

4. DESCARTES' SERIAL IMPLEMENTATION

Like most general-purpose connectionist simulators, DESCARTES currently runs solely on serial computers. The execution time for simulation of the synchronous update cycle is therefore linear with the number of nodes and links in the networks, which constrains the size of networks that can be reasonably simulated. Fortunately, the efficiency of serial simulations can often be improved by exploiting the dynamics of the spreading-activation process to prune the number of nodes that actually need to be updated on any given cycle.

Once the networks have been created and their cycling procedure defined (a process described in [Lange *et al.*, 1990a]), simulating the actual cycling of the networks is relatively straightforward. As described in the previous section, on each discrete global cycle the meta-controller needs only to (1) determine the active networks (and hence nodes) that will be updated, (2) serially walk through the active networks' nodes to calculate and store their new activations by calling their net-input and activation function methods (Update), and then (3) serially walk through their nodes again to spread their new output (calculated by applying their output-function to their new activation) to each of their nodes' outgoing links (Spread-To-Out-Links). Due to DESCARTES' object-oriented implementation, the simulation cycle works exactly the same for heterogeneous networks as for homogeneous networks — differing node classes simply have different net-input, activation, and output function methods.

The timing described above is critical for the serial simulation of synchronous update cycles. Because each of the nodes' activation updates depends on the *previous* cycle's outputs of their incoming links, all updates must be performed before any of the newly updated activations are spread to the nodes' outgoing links. Whereas in networks with homogeneous cycling rates it is unimportant in which order the update and spread-to-out-links steps are performed (so long as they are done separately), heterogeneous cycling requires that the spread-to-out-links step be done immediately before the next global cycle so that the nodes' new output values become available to the rest of the network.

4.1. Speeding Up the Simulation

Connectionist models using synchronous updating assume that *all* nodes are updated on each cycle, taking into account effective cycling rates. However, there are often cases in which the simulation process can be sped up when the activations of individual nodes remain constant over a number of cycles.

Feed-forward backpropagation networks (such as **Network-B** in **Hiding Pot**) are one example of this. When the nodes of the first layer are clamped to the activations of a new input pattern, the activations of the second layer's nodes will change on the next cycle, since their incoming links' outputs have changed. Similarly, on the third cycle, the third layer's nodes need to be updated as the activation is "fed-forward". However, the activations of the second layer remain constant, since their inputs (the first layer) remain clamped to their original values. There is therefore no need to update their activations, since they are guaranteed to remain unchanged. In general, only the layer that activation has just reached needs to be updated, thus decreasing the time needed for a serial simulation to feed activation forward from the inputs to the outputs by an average factor of n, where n is the number of layers in the network.

Feed-forward networks are not the only case in which large subsets of nodes do not need to be updated on every cycle. In large LCNs, for instance, activation may have only reached a small portion of the network at any given cycle, with the rest of the network remaining inactive. In fact, with most deterministic activation functions, any node whose level of activation stays constant over an update cycle will remain unchanged until one of its incoming links receives a different output value.

DESCARTES provides a functionality that exploits this by recognizing unchanging nodes and updating only those nodes whose activation (or markers) have the potential to change on a given cycle. With this functionality, nodes whose activation is unchanged after an update are declared to be *asleep* and removed from their networks' list of actively processed nodes. They remain asleep (and unprocessed) until one of their incoming links *wakes* them when they receive a new output value.

Figure 4 illustrates how activation spreads through a simple network after one of the nodes (A) has been given an initial activation of 0.5. The figure shows the activations on the nodes and on their outgoing-links (as modified by their link weights) after each of the first four cycles. Notice that some of the nodes are asleep on each cycle, limiting the amount of node updates needed to be performed. A more complete description of the simulator's serial implementation can be found in [Lange *et al.*, 1989b, 1990a].

5. SIMULATION ON SIMD MACHINES

Although serial simulations can reasonably handle networks of up to medium size, the massively-parallel nature of neural networks dictates that large models be simulated on similarly massively-parallel machines. The only current machines with enough processors to allocate a processor per node for large networks are SIMD ("Single Instruction Multiple Data") parallel processors. Unfortunately, because all processors in SIMD machines must execute the same instruction at a time, they are not ideally suited for simulation of heterogeneous networks. Large numbers of processors must remain idle as the processing functionalities of each different node class are simulated separately. Even so, the simulation time on a SIMD machine will be only linear with the number of different processing functionalities. Dramatic increases in simulation speed are thus possible over serial simulations, since there are at most a handful of different classes of nodes and links in a given connectionist model, whereas there can be thousands or millions of separate nodes and links.

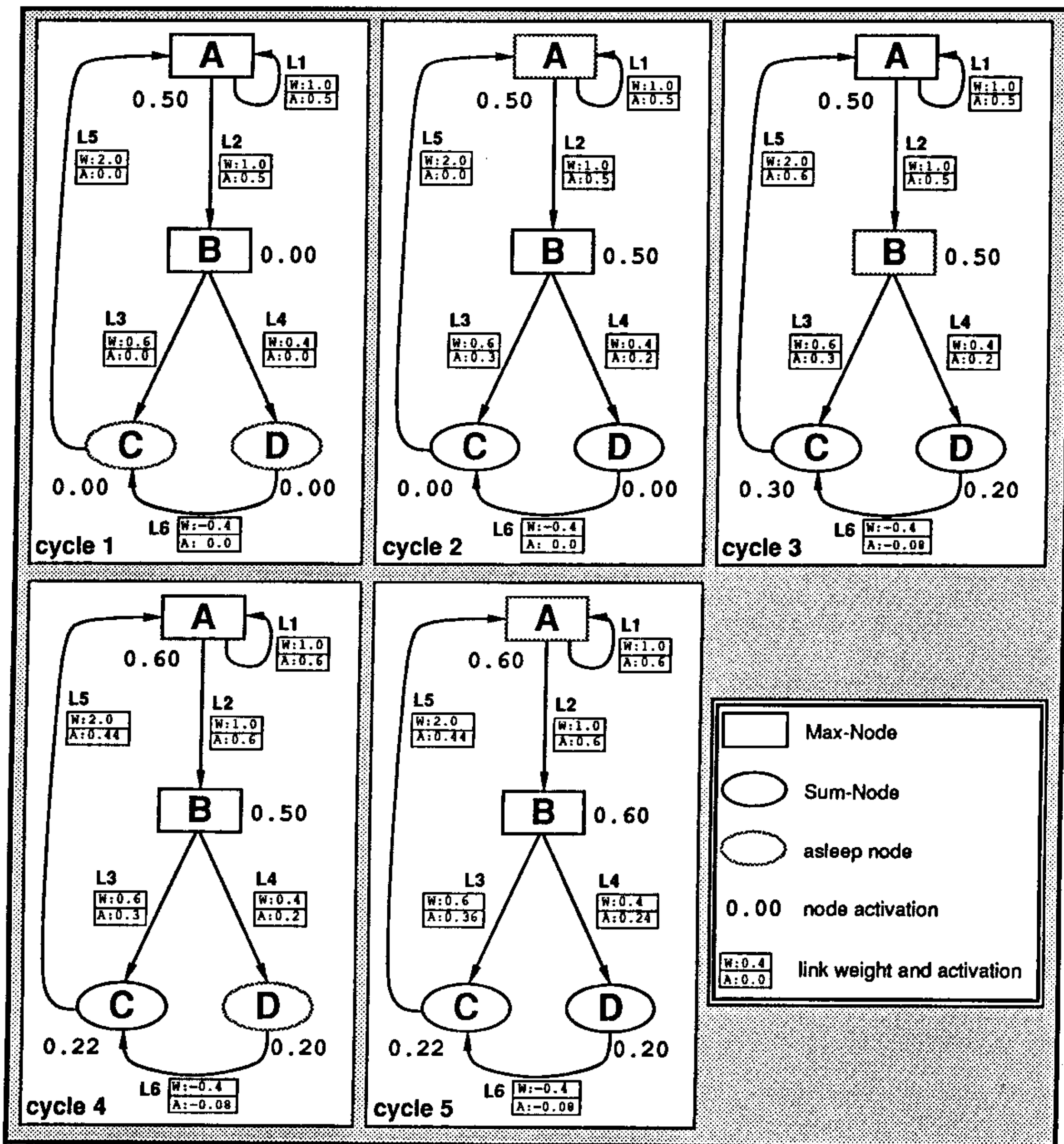

Figure 4. An example of a simple spreading-activation network. Sum-Nodes calculate their activation as the sum of their input, while Max-Nodes calculate their activation as the maximum of their inputs.

5.1. The Connection Machine

For simulations of connectionist networks with arbitrary connectivity, it is essential that the SIMD machine allow each processor to communicate efficiently with any other processor. The current large-scale SIMD machine probably best meeting this requirement is the *Connection Machine* (CM), which consists of

up to 64 *K* one-bit processors arranged in a 12-dimensional hypercube [Hillis, 1985]. Each processor has a local memory of 256 *K* bits which can be partitioned by CM microcode to form over a million virtual processors. Most importantly, the CM has sophisticated routing hardware for parallel communication between processors. All processors are controlled by instructions broadcast from a (serial) workstation host.

Using these capabilities, simulations of general backpropagation networks on the CM have attained over 10 million weight updates per second [Blelloch & Rosenberg, 1987], a nearly two thousand-fold increase over the speed of DESCARTES' serial implementation[2]. However, connectionist simulations on the CM and other SIMD machines have so far been limited to implementations of specific, generally homogeneous, models.

We are currently planning on implementing DESCARTES on the Connection Machine to allow massively-parallel simulation of general heterogeneous connectionist networks. To retain the network creation, handling, debugging, and flexibility features of DESCARTES, its Connection Machine implementation would continue to use CLOS object-oriented instantiations of network nodes and links on the CM's host. To realize massively-parallel simulation, however, all actual network processing would be performed by a mirrored copy of the network allocated across the processors of the CM.

5.2. Layout of Processors

As long as enough processors are available, the most efficient use of processors for general networks on fine-grained parallel machines such as the Connection Machine is to allocate one processor for each node and at least one processor for each link[3]. The actual processor layout needed to achieve optimal performance on a given machine depends on the costs of different processor communication operations on that machine.

On the Connection Machine, there are two primary communication techniques — *router* and *scan* operations. The *router* operations allow any processor to read from or write into the memory of any other processor. The *scan* operations allow the summation of the values of many adjacent processors into one processor, or the copying of the value of one processor into many adjacent ones. *Scan* operations are generally faster than *router* operations, though less flexible.

Using the above information on CM communications' costs, Blelloch and Rosenberg [1987] devised a processor layout for homogeneous backpropagation networks of arbitrary connectivity that allocates one processor per node and two processors per link. Their layout is designed to allow most communication in the spread-to-out-links and update phases to be performed with fast *scan* operations to minimize communication delays.

Blelloch and Rosenberg's basic processor layout will also work for general heterogeneous networks, though the simulation process itself needs to be extended. The processor for each node is immediately followed by the processors for all of its outgoing links, and is immediately preceded by the processors for all of its in-

---

[2][Zhang *et al.*, 1990] and [Singer, 1990] describe implementations of backpropagation on the CM that are even faster for certain networks and training sets, but which are not as easily generalizable to arbitrary heterogeneous networks.

[3]One alternative is to only allocate processors for nodes and update their activations by looping through lists of links stored on each processor. A *node-based* implementation such as this will force node processors with small numbers of links to remain idle while processors with larger number of links continue looping through their remaining links.

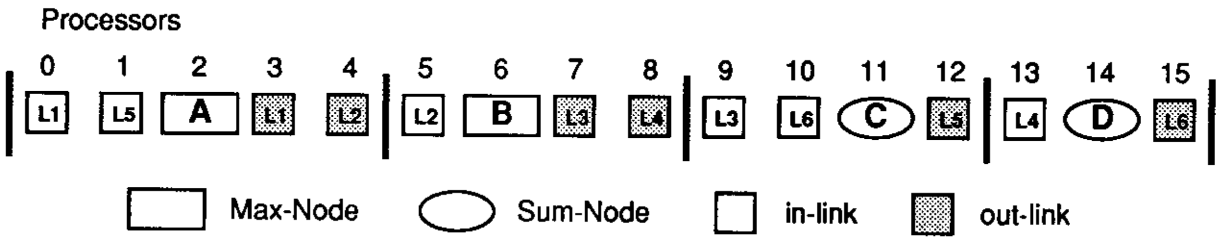

Figure 5. Layout of nodes and links of the network of Figure 4 as would be implemented on the Connection Machine.

coming links. The beginning and ending of these continuous segments of processors is marked by flags. Figure 5 shows the example network of Figure 4 as it would be allocated using this layout.

5.3. The SIMD Simulation Cycle

The overview of what occurs in each global simulation cycle will remain exactly the same in the CM implementation as in the current serial implementation (Section 3.3). The networks to be cycled need to be determined, all nodes in those networks must be updated, the new output values must be spread to the nodes' output links, and any desired output must be reported.

The simulation's meta-controller will reside on the host machine. Because the number of sub-network controllers is always relatively small, the actual network controllers will also reside on the host, where each cycle's selection of active network controllers will be computed. If the list of active controllers is different than on the previous cycle, then the host will broadcast the new list to the CM's processors. All processors for nodes and links that are in one of those controllers will be selected; those that are not will be turned off and will stay off throughout the remainder of the cycle.

5.4. The SIMD Cycle's Update Stage

The next part of the global cycle is the *update* stage in which all active nodes update their activations by calculating their net-inputs, their activations, and their outputs. All incoming link processors (i.e. 0, 1, 5, 9, 10, and 13 in Figure 5) will already hold the link-modified output (e.g. weighted output) of their source nodes from the spread-to-out-links of the previous cycle. The first part of the update stage is for all nodes to calculate their net-inputs. Each net-input processing functionality used in the network must be done separately. This will be done by the following algorithm:

```
1) For each primary net-input functionality in the active networks DO
    a) Select the processors of the nodes and links that use it
       from the subset of processors already active.
    b) Perform a forward scan into the nodes with the reduction function
       specified by the net-input functionality (e.g. "+" for net-input-
       sum-functionality, "max" for net-input-max-functionality), storing
       the result in the net-input variable of the active node processors.
    c) Return to previously active processors.
```

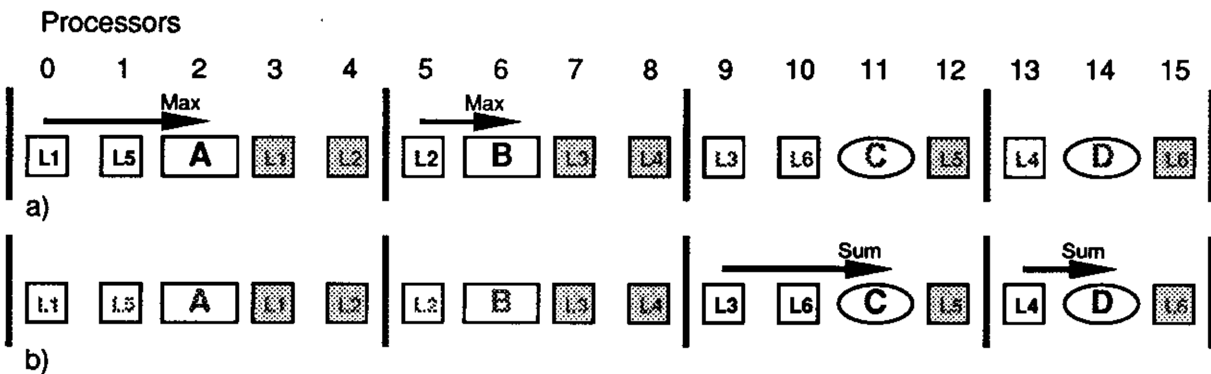

Figure 6. Using *scans* to update: first (a) the nodes with primary net-input function Net-Input-Max-Functionality (for Max-Nodes), and then (b) the nodes with primary net-input function Net-Input-Sum-Functionality (for Sum-Nodes). Processors with shaded node or link names are idle.

```
2) For each :around net-input functionality in the active networks DO
   a) Select the processors of the nodes and links that use it
      from the subset of processors already active.
   b) Modify the value in the processors' net-input variable, as specified
      by the net-input :around functionality method (e.g. adding a
      self-bias for net-input-around-self-bias-functionality).
   c) Return to previously active processors.
```

As an example, the network of Figure 4 has two primary net-input functionalities: Net-Input-Max-Functionality for its Max-Nodes (A and B) and Net-Input-Sum-Functionality for its Sum-Nodes (C and D). They have no modifying :around net-input functionalities. Figure 6 shows the two steps that will be needed to calculate their net-inputs. In Figure 6a, the processors which use Net-Input-Max-Functionality (0, 1, 2, 5, and 6) are selected, and a forward scan is used to place the maximum output value of processors 0 and 1 (in-links L1 and L5) into the net-input variable of processor 1 (node A), and the maximum output value of processor 5 (in-link L2) into processor 6 (node B). Afterwards, the processors which use Net-Input-Sum-Functionality are selected, and a scan is used to sum up the values of the incoming link processors and place them in the net-input variables of the Sum-Node processors (Figure 6b).

After the net-inputs to all of the active node processors have been calculated and stored, their activations must be calculated. As with the net-input functions, all of the primary and :around activation functionalities used by the nodes in the active networks must be calculated separately, with all of the primaries being calculated first. Finally, all of the active node processors must calculate their outputs by similarly applying their primary and :around output functionalities. Unlike the net-input functions, neither the activation nor output functions will generally require any inter-processor communication.

Although the breakdown between primary and :around node functionalities makes the simulator's update algorithm slightly more complicated, it permits large speed-ups for SIMD simulation of heterogeneous networks. This is true because it allows common processing characteristics of different node classes to be simulated concurrently. For example, if the net-input functions of the Simple-SA-Node and PDP-Nodes defined in Figure 3 were simulated separately, then the sum of the net-inputs would have to be performed twice: once for the net-input function of the Simple-SA-Nodes and once for the net-input function of the PDP-Nodes. However, with the algorithm described above, the sum of the net-inputs would only have to be per-

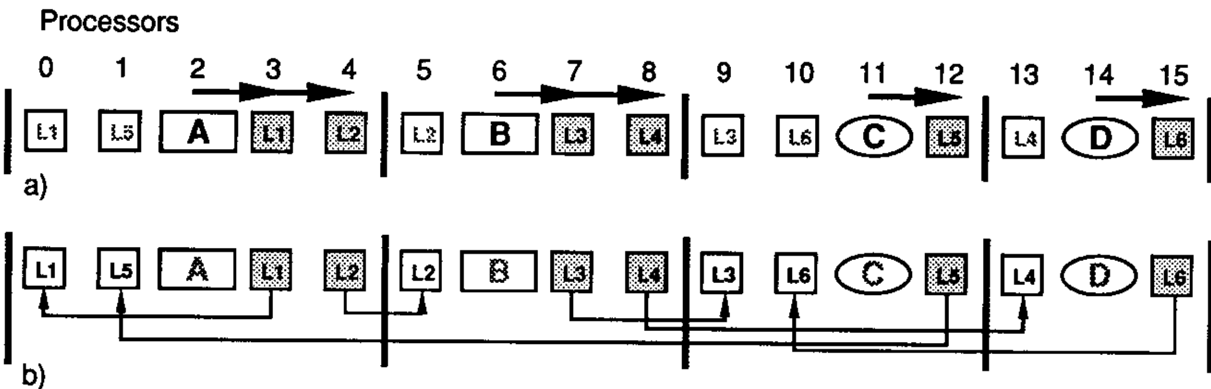

Figure 7. The CM Spread-To-Out-Links process: (a) First a *scan* is used to copy the output of the nodes to their outgoing links. (b) After the link functions have been applied, the *router* is used to copy links' output values from their outgoing link processors to their incoming link processors. Processors with shaded node or link names are idle.

formed once, when the primary Net-Input-Sum-Functionality is applied. An important aspect of this algorithm is that it has the same efficiency as dedicated homogeneous network simulators when there is only class of nodes in the network.

5.5. The SIMD Cycle's Spread-To-Out-Links Stage

After all of the nodes in the active controllers have been updated, their new outputs must be spread to their outgoing links. First a *copy-scan* will be used to quickly copy the outputs of all active nodes to their outgoing links (Figure 7a). Then the functions for each different link class used in the active controllers will be applied (separately) to convert the outputs the links received from their source nodes to the value for their sink nodes (e.g. weighted links multiply the value received by their weight). Finally, the link output values are sent using the router to their equivalent incoming links so that they will be available for the next cycle's update (Figure 7b).

5.6. Efficient Backpropagation on SIMD Machines

The same layout of processors described for the spreading-activation process can be used for training networks' weights with backpropagation. The forward-propagation process of backpropagation is simply the global spreading-activation cycle carried out for as many cycles as there are layers in the network. The backwards error-propagation stage will use the same nodes and processors, with the scans and link routing being applied backwards to send back the error deltas calculated by the PDP logistic error function.

Because of the nature of feed-forward networks, one problem with simply using the above algorithm is that it would leave most of the processors idle, since only the layer that activation has just fed forward to (or error fed backward to) will ever change. Thus the characteristic that serial simulations can take advantage of for more efficient simulation (Section 4.1) wastes resources on massively-parallel SIMD machines.

A solution to this problem is to *pipeline* the training set through both the feed-forward and feed-backward stages [Blelloch & Rosenberg, 1987]. Using this method, the nodes in the input layer are clamped to a new input pattern on every cycle. Thus, if the network has *m* layers, a total of *2m* patterns will be processed at any given time, with all processors remaining active. An added requirement is that each node processor

keep a history of its output values for the last $2m$ cycles so that the feed-backward error function can use the corresponding output from its feed-forward stage[4]. This output history is also useful for debugging, and allows recurrent backpropagation networks to be trained without the separate layers normally used to copy the values of recurrent layers.

An added benefit of using the pipelining method of training is that it allows training of separate modular networks to proceed in parallel, regardless of whether sub-networks have different numbers of layers. Each modular sub-network simply needs to keep track of its own training set.

6. SIMULATION ON HYPOTHETICAL MIMD MACHINES

The ideal machine for simulation of large heterogeneous connectionist networks is a large-scale MIMD ("Multiple Instruction Multiple Data") machine. Because each processor would concurrently execute its own functions for simulating element processing characteristics, there could be as many different processing functionalities as there are elements without any degradation in performance. This is unlike SIMD machines, in which large numbers of processors must often remain idle as different node functionalities are simulated sequentially.

Unfortunately, there are currently no MIMD machines with enough processors to allocate a processor per individual node and link on large networks. Current MIMD machines are therefore limited to providing (somewhat less than) linear speedup with the number of processors by simulating either different segments of the network or different portions of the training sets (e.g. [Pomerleau *et al.*, 1988]). Simulations on parallel machine with small numbers of processors are certainly valuable tools, as are simulations on supercomputers such as the Cray Y-MP, but neither truly take advantage of the massive parallelism that is the very nature of neural networks.

When technology becomes available to affordably build massively-parallel MIMD machines, their simulation of connectionist networks will be much like simulation on current SIMD machines. As in SIMD simulations, arbitrary connectivity will make it desirable to have at least one processor allocated per node and link. Each processor, however, need not have huge amounts of local memory — only enough to hold required processing functionalities and local parameters (such as activations and weights). As with SIMD machines, communication costs will determine the optimal processor layout of nodes and links.

The meta-controller used to control the overall simulation and interact with the user would still need to be on a "host" processor or machine. The meta-controller would take on one added duty for the simulation: that of making sure that all processors are synchronized. Whether synchronization would need to take place only before the start of every cycle, or also need to be performed before the spread-to-out-links stage of every cycle would be determined by the optimal processor communication method. If a global communication method such as the scan operation on the Connection Machine is best, then the scan portion of the spread-to-out-links stage would have to wait until all active nodes have updated their outputs. On the other hand, if individual processor-to-processor communications were relatively inexpensive, then each node processor could "inform" its own outgoing link processors when its output was updated, and therefore immediately do its own spread-to-out-links. This method would have the advantage of better processor utilization when subsets of the nodes have relatively quick update but slow spread-to-out-link times.

---

[4] In normal backpropagation, a history of output values is not necessary, since the output of nodes remains constant after activation has been fed-forward through its layer.

7. CONCLUSIONS

This paper describes a development tool, DESCARTES, which provides researchers with the capability to simulate heterogeneous connectionist (neural) networks in which nodes and links may have different processing characteristics and effective cycling rates, or which are made up of modular, interacting sub-networks. DESCARTES also allows researchers to build hybrid networks which combine elements from distributed, localist, and symbolic marker-passing networks.

DESCARTES is currently implemented on serial machines. It is able to simulate networks of up to medium size with reasonable performance by taking advantage of the dynamics of the spreading-activation process to prune unchanging nodes from the update and spreading cycles.

The massively-parallel nature of neural networks, however, dictates that large networks be simulated on similarly massively-parallel machines. SIMD machines are currently the only machines with enough processors to approach this ideal. We have described how DESCARTES could be implemented on a SIMD machine (the Connection Machine). Every node in the network would be allocated one processor and every link two processors: one as the outgoing link from their source node, and the other as the incoming link to their sink node. Because every active processor in a SIMD machine must execute the same instruction, each separate processing characteristic of active nodes and links (e.g. different activation or learning functions) must be simulated sequentially, with processors not using that characteristic remaining idle. Compared to serial simulations, however, SIMD simulations offer great potential speed-up in run times, since there are generally only a few different node and link types in a given model, while there may be thousands or millions of separate nodes and links.

Ideally, heterogeneous networks would be simulated on a massively-parallel MIMD machine with at least one processor per node and link. Because each processor would execute its own functions for simulating element processing characteristics, there could be as many different processing functionalities are there are elements, with the only degradation in performance being due to processor synchronization.

REFERENCES

Akers, L. A., & Walker, M. R (1988): A Limited-Interconnect Synthetic Neural IC. *Proceedings of the IEEE Second Annual International Conference on Neural Networks (ICNN-88)*, San Diego, CA, July 1988.

D'Autrechy, C. L., Reggia, J. A., Sutton, G. G., & Goodall, S. M. (1988): A General-Purpose Simulation Environment For Developing Connectionist Models. *Simulation, 51(1)*, p. 5-19.

Belloch, G. & Rosenberg, C. (1987): Network Learning on the Connection Machine. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August, 1987.

Charniak, E. (1986): A Neat Theory of Marker Passing. *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, August, 1986.

Feldman, J. A. & Ballard, D. H. (1982): Connectionist Models and Their Properties. *Cognitive Science, 6 (3)*, p. 205-254.

Fukushima, K., Miyake, S., & Ito, T. (1983): Neocognitron: A Neural Network Model for a Mechanism of Visual Pattern Recognition. *IEEE Transactions on Systems, Man, & Cybernetics, SMC-13, 5*, p. 826-834.

Goddard, N., Lynne, K. J., Mintz, T., & Bukys, L. (1989): *Rochester Connectionist Simulator.* Technical Report TR-233 (revised), Department of Computer Science, University of Rochestor.

Grossberg, S. (1980): How Does the Brain Build a Cognitive Code? *Psychological Review, 87,* p. 1-51.

Hendler, J. (1988): *Integrating Marker-Passing and Problem Solving: A Spreading Activation Approach to Improved Choice in Planning.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Hendler, J. (1989): Marker-Passing Over Microfeatures: Towards a Hybrid Symbolic/Connectionist Model. *Cognitive Science, 13 (1),* p. 79-106.

Hillis, W. D. (1985): *The Connection Machine.* Cambridge, MA: The MIT Press.

Hinton, G. E. & Sejnowski, T. J. (1986): Learning and Relearning in Boltzmann Machines. In Rumelhart & McClelland (eds.), *Parallel Distributed Processing: Vol 1,* p. 282-317. Cambridge, MA: The MIT Press.

Lange, T. & Dyer, M. G. (1989a): Dynamic, Non-Local Role-Bindings and Inferencing in a Localist Network for Natural Language Understanding. In David S. Touretzky (ed.), *Advances in Neural Information Processing Systems I,* p. 545-552. San Mateo, CA: Morgan Kaufmann.

Lange, T. & Dyer, M. G. (1989b): Frame Selection in a Connectionist Model of High-Level Inferencing. *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society (CogSci-89),* Ann Arbor, MI, August 1989.

Lange, T. & Dyer, M. G. (1989c): High-Level Inferencing in a Connectionist Network. *Connection Science, 1 (2),* p. 181-217.

Lange, T., Hodges, J. B., Fuenmayor, M., & Belyaev, L. (1989a): DESCARTES: Development Environment For Simulating Hybrid Connectionist Architectures. *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society (CogSci-89),* Ann Arbor, MI, August 1989.

Lange, T., Hodges, J. B., Fuenmayor, M., & Belyaev, L. (1989b): Simulating Hybrid Connectionist Architectures. *Proceedings of the 1989 Winter Simulation Conference,* Washington, DC, December 1989.

Lange, T., Hodges J. B., Fuenmayor, M., & Belyaev, L. (1990a): *The DESCARTES User's Manual.* Research Report, Computer Science Department, University of California, Los Angeles.

Lange, T., Holyoak, K., Wharton, C., & Melz, E. (1990b): *Disambiguation and Analogical Retrieval in a Symbolic-Connectionist Network,* Research Report, Computer Science Department, University of California, Los Angeles.

McClelland, J. L. & Kawamoto, A. H. (1986): Mechanisms of Sentence Processing: Assigning Roles to Constituents of Sentences. In McClelland & Rumelhart (eds.), *Parallel Distributed Processing: Vol 2,* p. 272-325. Cambridge, MA: The MIT Press.

Mead, C.A. (1989): *Analog VLSI and Neural Systems.* Addison-Wesley.

Mesrobian, E., Stiber, M., & Skrzypek, J. (1989): *UCLA SFINX — Structure and Function in Neural Connections.* Technical Report MPL-TR 89-8, Computer Science Department, University of California, Los Angeles.

Miikkulainen, R. & Dyer, M. G. (1989): A Modular Neural Network Architecture for Sequential Paraphrasing of Script-Based Stories. *Proceedings of the International Joint Conference on Neural Networks,* Washington, DC, June, 1989.

Pomerleau, D., Gusciora, G., Touretzky, D., & Kung, H. (1988): Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. *Proceedings of the IEEE Second Annual International Conference on Neural Networks (ICNN-88),* San Diego, CA, July 1988.

Rosenblatt, F. (1962): *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Washington, DC: Spartan Books.

Rumelhart, D. E., Hinton, G. E., & McClelland, J. L. (1986): Learning Internal Representations by Error Propagation. In Rumelhart & McClelland (eds.), *Parallel Distributed Processing: Vol 1*, p. 318-364. Cambridge, MA: The MIT Press.

Shastri, L. (1988): A Connectionist Approach to Knowledge Representation and Limited Inference. *Cognitive Science, 12 (3)*, p. 331-392.

Singer, A. (1990): Implementations of Artificial Neural Networks on the Connection Machine. *Parallel Computing*. In press.

Tomabechi, H. & Kitano, H. (1989): Beyond PDP: The Frequency Modulation Neural Network Architecture. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, August 1989.

Waibel, A. (1989): Consonant Recognition by Modular Construction of Large Phonemic Time-Delay Neural Networks. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems I*, p. 215-223. San Mateo, CA: Morgan Kaufmann.

Waltz, D. & Pollack, J. (1985): Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation. *Cognitive Science, 9 (1)*, p. 51-74.

Wilson, M. A., Upinder, S. B., Uhley, J.D., & Bower, J. M. (1989): GENESIS: A System for Simulating Neural Networks. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems I*, p. 485-492. San Mateo, CA: Morgan Kaufmann.

Zhang, X., McKenna, M., Mesirov, J., & Waltz, D. (1989): An Efficient Implementation of the Backpropagation Algorithm on the Connection Machine CM-2. *Proceedings of the IEEE Conference on Neural Information Processing Systems — Natural and Synthetic (NIPS-89)*, Denver, CO, December 1989.