A General Framework for Concurrent Simulation of Neural Network Models

Gregory L. Heileman, *Member, IEEE*, Michael Georgiopoulos, *Member, IEEE*, and William D. Roome, *Member, IEEE*

Abstract—**The analysis of complex neural network models via analytical techniques is often quite difficult due to the large numbers of components involved, and the nonlinearities associated with these components. For this reason, simulation is seen as an important tool in neural network research. In this paper we present a framework for simulating neural networks as discrete event nonlinear dynamical systems. This includes neural network models whose components are described by continuous-time differential equations, or by discrete-time difference equations. Specifically, we consider the design and construction of a concurrent object-oriented discrete event simulation environment for neural networks. The use of an object-oriented language provides the data abstraction facilities necessary to support modification and extension of the simulation system at a high level of abstraction. Furthermore, the ability to specify concurrent processing supports execution on parallel architectures. The use of this system is demonstrated by simulating a specific neural network model on a general-purpose parallel computer.**

Index Terms— **Concurrent simulation, neural networks, nonlinear dynamical systems, object-oriented programming, parallel processing.**

I. Introduction

ARTIFICIAL neural networks are biologically inspired computing models characterized by large numbers of densely connected computational elements. The tremendous interest that has recently surfaced regarding these models has led researchers to propose their use in a wide range of application areas. Because of the analytical intractability of these models, a large portion of the research in the field of neural computation involves simulation. Furthermore, the parallel nature of neural network models makes them quite amenable for simulation on parallel computing architectures. In fact, the exploitation of this inherent parallelism is considered a necessity if very large neural network models are to be investigated [1]. In this paper we consider the development of simulation capabilities that support both rapid prototyping of neural network models, as well as their execution on parallel computing platforms. The design choices made during this development were influenced by the flexibility versus speedup trade-offs discussed below.

Manuscript received August 28, 1991; revised March 12, 1992. recommended by E. Gelenbe. This research was supported in part by grants from Sandia National Laboratories under Contract 87–5002, the Florida High Technology and Industry Council, and the Division of Sponsored Research at the University of Central Florida. Recommended by E. Gelenbe.

G. L. Heileman is with the Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131.

M. Georgiopoulos is with the Department of Electrical Engineering, University of Central Florida, Orlando, FL 32816.

W. D. Roome is with AT&T Bell Laboratories, Murray Hill, NJ 07974.

IEEE Log Number 9200611.

The target architecture for implementing parallel neural network simulations will either be a special-purpose or general-purpose parallel computer. Maximum speedup is currently achieved by designing special-purpose computers that directly implement neural network models using VLSI techniques [2]–[5]; however, the ability to modify the model is sacrificed once the neural network is "cast" into hardware. In addition, the time required to design, develop, and test such a system is typically on the order of months. Thus, the use of special-purpose computers is most appropriate when the specific form of a neural network model intended for use in a particular application has been finalized.

The other alternative is to utilize general-purpose computing platforms that are designed to execute a wide variety of applications. In this case, the model is created in software, and some degree of speedup is sacrificed for increased flexibility. Furthermore, a number of different approaches to the simulation of a given model on a general-purpose parallel computer can be considered. In each case, the trade-off is again one of speedup versus flexibility. First, a model can be directly mapped onto the parallel hardware using a programming language specifically designed for that particular machine [6]–[8]. This will typically yield the largest possible speedup when utilizing a general-purpose computer. The difficulty with this approach is that it requires the implementor to possess an understanding of the underlying hardware, including the processor interconnection scheme and communication protocol. Consequently, this mapping process is both time consuming and machine specific. An alternative is to implement the application using a general-purpose parallel programming language. This allows the compiler to automatically map the application onto a particular computer, thereby allowing the program to run on any parallel machine for which the compiler is available. In general, a direct mapping is more efficient than this automatic mapping; however, the use of a general-purpose concurrent programming language allows one to specify parallel operations at a higher level of abstraction.

In this paper, we investigate the simulation of neural network models using a general and easily modifiable simulation model at the level of a general-purpose object-oriented concurrent programming language. We believe that this is an important level for performing neural network research because it allows investigators to easily and rapidly test new ideas while providing the processing power necessary to investigate nontrivial models. Furthermore, once a network architecture has been realized and tested through simulation, it is possible to directly implement it using special-purpose hardware to achieve maximum speedup.

The organization of this paper is as follows: In Section II we discuss concurrent object-oriented programming in general, and then consider a specific language, Concurrent C++. A general model for neural computation is then presented in Section III. This model forms the framework for the generic Concurrent C++ neural network simulation system discussed in Section IV. In Section IV we also discuss how the concurrent processes associated with this simulation model are scheduled and synchronized so that useful parallel simulations can be performed. In Section V we consider how the concurrent simulation model developed in Section IV can be extended through inheritance to implement specific neural network models. We then demonstrate the simulation of a specific model on a general-purpose parallel computer. Finally, in Section VI we conclude with a discussion of our results.

II. Concurrent Object-Oriented Programming

Concurrent object-oriented programming languages utilize object-oriented programming capabilities while providing the ability to specify concurrent execution. From a software engineering perspective, the data abstraction capabilities offered by the object-oriented methodology are important because they facilitate the building of reusable and easily extendable software modules. As such, object-oriented programming techniques are currently being considered for addressing issues involved in "programming in the large." Another advantage is that the decomposition techniques used in the design of an object-oriented program are consistent with those used in the design of concurrent programs.

The concurrent object-oriented language used to implement the simulation system discussed in this paper was the Concurrent C++ programming language developed at AT&T Bell Laboratories. This language was created by integrating the object-oriented language C++, and the concurrent programming language Concurrent C. Implementation issues involved in this merger are discussed in [9].

Below we briefly discuss some of the facilities available in Concurrent C++. We will proceed by first discussing those facilities particular to C++, followed by those particular to Concurrent C. All of the facilities discussed are available in Concurrent C++. More detailed discussions of the C++ programming language can be found in [10] and [11]. Detailed discussions of Concurrent C and Concurrent C++ can be found in [12].

Computation in object-oriented programs centers around the manipulation of class objects. A *class* can be defined as an implementation of an abstract data type [13]. Recall that an abstract data type includes a definition of both the data elements, as well as the operations that may be performed on those data elements. An *object* is a particular instance of a class. For example, we may implement a stack using a class. In this case, the class will define what type of data elements can be stored on the stack, and how the operations (i.e., push, pop, etc.) will be implemented. The routines used to implement these operations are referred to as *member functions*. An instance of the stack class (i.e., a stack object) must be created if we wish to use it. Messages sent to this object may invoke specific member functions that operate on the stack data elements.

In C++, class declarations consist of two parts: a specification and a body. The class specification serves as the "user interface" to the class. Class specifications have the form

```
class class name : derivation list {
    private:
    private members
    protected:
    protected members
    public:
    public members
};
```

where the members of a class may be either data elements or functions. Class members specified as private are not visible to elements external to the class, while public members are. Protected class members are treated as if they were private class members, with one exception. Objects of derived classes (discussed below) treat the protected members as if they were public members. The second part of the class declaration, the class body, consists of the bodies of member functions that were declared but not defined in the class specification.

The derivation list in the class specification supports *inheritance*—one of the most important concepts found in object-oriented programming languages. Inheritance allows one to create a new class from existing classes. In this case, the new class that is being declared (i.e., the *derived class*) is said to be derived from the existing *base classes*. A derived class inherits the members of its base classes, and may also add new members. A derived class may also redefine any member function provided by the base class by simply supplying a new member function that has the same name as the old member function in the base class. In this case, the new member function in the derived class is said to *overload* the member function with the corresponding name in the base class. This allows different meanings to be attached to the same member function name; which member function is invoked when the name is called in a program depends upon the specific class being used. Furthermore, if the overloaded function is declared virtual in the base class, then the overloaded function will be dynamically bound to an object at run-time. This trait, known as *polymorphism*, is utilized extensively in the neural network simulation discussed in this paper.

Most concurrent programming languages are based on the use of *processes*. A process has its own thread of execution, stack, and machine registers. Thus, two processes may execute simultaneously on separate processors, or they may be time-sliced on a single processor. The concurrent programming facilities provided by Concurrent C are extensions of the *Communicating Sequential Processes* [14] and *Distributed Processes* [15] models. These models also form the basis for the concurrent programming facilities offered by the Ada programming language.

In Concurrent C, a process definition consists of a specification and a body. They have the form

```
        process spec process name(parameter types
                                        and names) {
          transactions
```

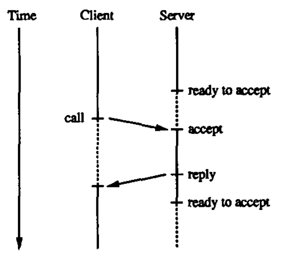


Fig. 1. A "time-line" diagram that illustrates how processes interact during a synchronous transaction call. A solid line indicates that a process is running, while a dotted line indicates that it is waiting.

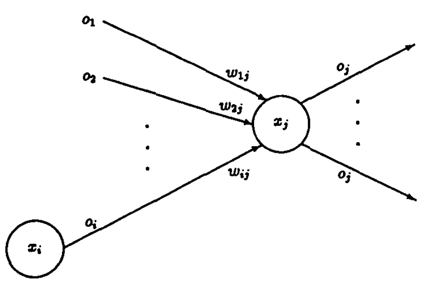


Fig. 2. A general model for computation in neural networks.

```
};
process body process name(parameter names)
{
  process code
}
```

The process specification contains all the information necessary to create and interact with processes of the type being defined, while the process body contains the code (along with the associated declarations and definitions) that is executed by a process of that type.

Only the information given in the process specification is visible to other processes. This information includes a list of available transactions. Concurrent C processes communicate by using transactions. These transactions can be either synchronous or asynchronous. In the synchronous case, illustrated in Fig. 1, a client process sends a message to a server process requesting it to perform some service. The client process then must wait for the server process to accept the message and perform the desired service. Upon completion, the server process returns some data to the client process which is then free to resume execution. Thus, a synchronous transaction requires both a synchronization between two processes, as well as a bidirectional exchange of information. Note in Fig. 1 that the client process is "blocked" while the server is servicing the transaction. This is not the case for asynchronous transactions, the client process does not wait for the server process to receive the message, and it is not possible for the server process to return results to the client process. Thus, an asynchronous transaction involves a unidirectional exchange of information, and no synchronization.

The Concurrent C++ programming language merges C++ and Concurrent C to produce a language with both data abstraction and concurrent programming facilities. The advantages of using the data abstraction facilities offered by classes in sequential programming apply equally to concurrent programming. Specifically, classes can be used in Concurrent C++ to hide the implementation details of a process, and to ensure that the proper protocol for interacting with a process is observed. In addition, the ability to use inheritance facilitates the building of reusable concurrent program modules.

In most concurrent object-oriented programming language models, objects are treated as sequential processes that respond to messages sent to them. Furthermore, is it assumed that every object may execute its operations concurrently [16]. This is not necessarily true in Concurrent C++: an object may execute an operation concurrently only if a process is somehow associated with that operation.

III. A General ModelFor Neural Computation

In this section we define a general model for neural computation that serves as the framework for specifying virtually any neural network paradigm. As we shall see in the following sections, the ability to specify such a general model has important implications in the object-oriented design of the simulation system.

The architecture of a neural network model can be described by a weighted directed graph in which the nodes of the graph represent neurons, and the weighted edges represent a set of internal dynamical parameters that generally correspond to synaptic weights. A portion of such an architecture is depicted in Fig. 2.

The state or *activation value* of each node in this system can be modeled by a dynamical variable. We will represent the activation value of the jth node, v_j, using the variable x_j. Computation in a neural system is usually viewed as an evolution through time of these node activation values. Thus, if we assume continuous-valued states and equations of motion, the state of the system may be described by the following equations:

$$\tau_x \frac{d}{dt} x_j(t) = \mathcal{G}_j(x_j(t), u_j(t)) \tag{1}$$

where τ_x is a positive numerical constant that defines the time scale over which the activation values change, and $u_j(t)$ is the *net input* to node v_j at time t. The form of the $\mathcal{G}$ function in (1) will be determined by the particular neural network model under consideration. Furthermore, for discrete-time models, (1) is often approximated using first-order finite difference equations. Particular forms for (1) can be found in [17].

Before discussing the generation of a node's net input value, we first must consider how a node generates an output value. Associated with each node v_j there is an *output function*, $f(x_j)$, that is responsible for mapping the current state of the node's activity x_j to an output signal o_j as shown in Fig. 2. Commonly used forms for the function f are the logistic function,

$$f_L(x) = (1 + \exp^{-\alpha x})^{-1} \tag{2}$$

and the hardlimiting threshold function,

$$f_H(x) = \begin{cases} \beta, & \text{if } x \geq \delta \\ \gamma, & \text{if } x < \delta \end{cases} \tag{3}$$

where typically $\beta > \gamma$, and δ is the threshold value. Note that for large α, (2) can be approximated by (3) with $\beta = 1$ and $\gamma = 0$.

The output of node v_i is multiplied by weight w_{ij} to produce an input to node v_j, and the collection of inputs to v_j are combined according to some operator to produce the net input u_j. Typically, u_j is expressed as the following propagation rule:

$$u_j(t) = \sum_i o_i(t) w_{ij}(t) \tag{4}$$

where w_{ij} represents the value of the weighted edge incident from v_i and incident to v_j, and the sum is over all nodes that are incident to v_j. In some models, positive and negative weight values are considered excitatory and inhibitory connections, respectively, while in other models, there are separate net input calculations for excitatory and inhibitory inputs.

We now consider the manner in which the dynamics of system (1) can be modified so that learning occurs. The parameters that are adjusted through learning in a neural network are the weights w_{ij}. Note that (1) is dependent on w_{ij} through u_j. Thus, a learning rule must specify the adaptive dynamics that allow the weights of dynamical system (1) to be modified so that for an initial input, the steady-state output of the system represents the desired response. A constraint that must be observed in all neural network systems is that the modification of a given weight must be based solely on information that is locally available to it. This constraint, known as the *locality constraint*, implies that

$$\tau_w \frac{d}{dt} w_{ij}(t) = \mathcal{H}_j(w_{ij}(t), g(v_j(t))) \tag{5}$$

where τ_w is a positive numerical constant that defines the time scale over which the weights change, and $g(v_j(t))$ is a function that is computed based on information that is available to node v_j at time t. The particular forms of the $\mathcal{H}$ and g functions are determined by the learning rule being employed in the neural network model under consideration. Again, there are analogous discrete-time representations for neural network learning rules.

At this point, a discussion of time scales is in order. If the neural model is to be treated as a true dynamical system, then the system of coupled differential equations that describe the activation and weight dynamics must be executed simultaneously. In this case, the relaxation time for the node activation values should be much faster than the relaxation time of the weights. This ensures that the system parameters are adapted based on the steady-state response of the nodes, and not their transient response. Choosing $\tau_w \gg \tau_x$ in (1) and (5) enforces this behavior (see [18] for a more complete discussion of this issue). In many models, however, the dynamics of learning are separated from the activation dynamics of the nodes. In these models there are typically two phases of operation: a training phase in which the network weights are adjusted, and a performance phase in which the network weights are held constant while input patterns are presented and network outputs computed. Since the learning and activation dynamics are effectively separated in this scheme, issues of time scales are not important.

```
typedef trans async (*tptr)();

class node {
  protected:
    node_process np;
    int num_inputs, num_outputs;
    double output, activity;
    class input_edge *input_connect;
    class output_edge *output_connect;
  public:
    void Input(int InputNum, double Val, tptr Tp);
    void ComputeNodeOutput(tptr Tp);
    void UpdateNodeWeights(double StepSize, tptr Tp);
    void PropagateOutputs(tptr Tp);
    virtual double ComputeActivity(double StepSize);
    virtual double ComputeOutput();
    virtual void UpdateWeights(double StepSize);
    // remaining member functions not shown
};
```

Fig. 3. The specification of the generic node class.

IV. The Concurrent Base Classes

This section describes the basic components of the simulation system. These include a base class that was developed to represent a generic node, as well as another base class that uses the node class to provide the architectural framework for a network and the primitives for accessing network elements. These classes provide the scaffolding upon which more complex neural network models can be built through the use of inheritance. Below we present the interfaces to the Concurrent C++ base classes used in the simulation system, and we discuss how the concurrent processes associated with these classes are organized so as to exploit the parallelism available in neural network models.

The Class Interfaces

We begin by discussing the manner in which network nodes are represented. Fig. 3 shows the specification of **node**, a class used to represent generic node elements. The simulation system is constructed so that the **node** class must serve as the base class for any specific node model that we wish to simulate.

Many of the member functions in the **node** class require a transaction pointer as one of their parameters. Thus, in Fig. 3 we use the **typedef** facility to declare a new type name. This declaration makes the name **tptr** a synonym for a pointer to an asynchronous transaction. A pointer to any asynchronous transaction containing an empty parameter list can be assigned to a variable of type **tptr**. The use of transaction pointers allows process interaction points to be dynamically specified. We will subsequently show how transaction pointers are used as a synchronization mechanism in the simulation system.

Note that the protected member **np** in Fig. 3 is of type **node_process**, a process discussed in detail below. Thus, every object of class **node** will have a concurrent process associated with it. In addition, the protected members **activ-**

```
class network {
  protected:
    scheduler_process scheduler;
    int num_nodes, num_layers;
    int *nodes_per_layer;
    int **connection_matrix;
    class node **nd;
    // remaining data members not shown
  public:
    void AssignInputs(int PatternNum);
    // remaining member functions not shown
};
```

Fig. 4. The specification of the **network** class.

ity and **output** are used to store the activity state x_j, and the output state o_j of the node. The remaining protected members are used to store information related to the input and output connections associated with a node.

A number of the member functions in the **node** class are declared as virtual. These include the functions that are used to compute the activity and output of a node, and the function used to update the weights incident to a node. These are the functions that will be overloaded in derived classes allowing specific node models to be simulated. The significance of using virtual functions is that they allow a general neural network class to be constructed independent of any particular node model.

Specifically, class **network** in Fig. 4 is patterned after the general model presented in Section III, and will be used through inheritance to implement specific neural network models in Section V. Thus, **network** serves as a generic base class for neural networks. Most of the data members in this class are used to store information related to the architecture of a network. For example, **num_nodes** stores the number of nodes in the network, and **nodes_per_layer** is a pointer to a dynamically allocated array that stores the number of nodes in each layer of the network. A two-dimensional array is dynamically allocated and used to store the specific connection pattern of a network—**connection_matrix** is a pointer to this array. The variable **nd** is used to point to an array of pointers to **node** objects. A pointer to any object derived from the node class can be assigned to one of these array elements. Furthermore, the virtual functions associated with the class of the derived object can be invoked through the pointers stored in this array. This allows multiple types of nodes to exist within a single network, and also allows us to write the member functions in the **network** class so that they operate on generic node objects. When any one of these member functions is actually called in a program, the form of the nodes in the network will determine the actual operations that are performed.

Finally, a concurrent process, **scheduler**, is associated with the **network** class. The manner in which this process interacts with other processes in the simulation system is discussed next.

Process Scheduling and Synchronization

There are numerous ways to decompose a simulation for processing on multiple processors. The decomposition technique that has shown the greatest potential in terms of exploiting inherent parallelism is called the *distributed model components* approach [19]. In this approach, each component in the system being simulated is assigned to a different process. This decomposition technique is consistent with object-oriented design in which each system component is represented by an object.

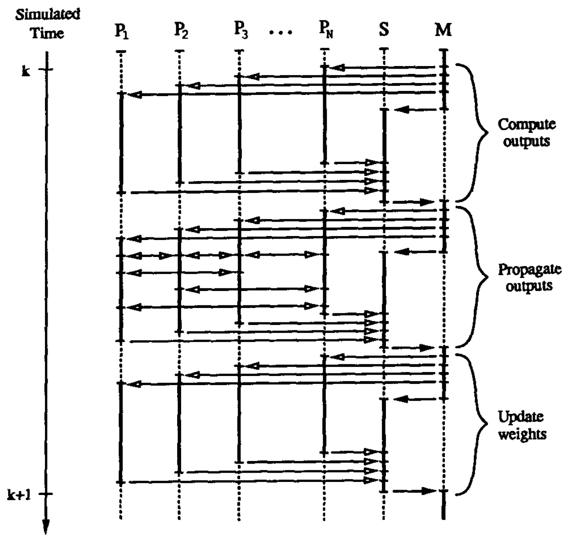


Fig. 5. An idealized "time-line" diagram that illustrates how process interact during the typical phases of a single iteration of a neural network simulation. P_1–P_N are node processes, S is the scheduler process, and M is the main process. A solid line indicates that a process is *running*, while a dotted line indicates that it is waiting. Open arrows represent asynchronous transactions, and closed arrows represent synchronous transactions. See the text for a description of the process interactions.

When using a distributed model components decomposition in discrete event simulation, specific schemes for synchronization and deadlock handling must be employed. Different approaches are taken depending upon whether the simulation time advances in fixed increments (time-driven) or moves from one event time to the next (event-driven), and whether these advances occur synchronously or asynchronously. For neural network simulations, we are concerned with using discrete event simulation to simulate an inherently continuous-time system by discretizing time. Thus, the time-driven approach is most natural. Below we describe a synchronous time-driven discrete event simulation approach to the simulation of neural network models. In this approach, the simulation at a particular time step typically proceeds in three phases: a state update phase, which involves computing the outputs of the network nodes at the current time; a communication phase, which involves propagating these outputs: and a weight update phase, in which the weights are modified according to the network learning rule.

Recall that a process is associated with each **node** object, and that a scheduler process is associated with a **network** object. An illustration of how these processes interact during the typical phases of a neural network simulation discussed above is given in Fig. 5. This figure shows the "time-lines" of the processes during a single iteration of the simulation. A solid line indicates that a process is running, while a dotted line indicates that a process is waiting. The time-lines of the processes associated with the N nodes in a network are labeled P_1 through P_N, while the time-lines for the scheduler and main processes are labeled S and M, respectively. The main process is the process associated with the particular neural network model being simulated. As mentioned in Section II, processes interact through transactions. In Fig. 5, an open arrow is used to represent an asynchronous transaction, and a closed arrow is used to represent a synchronous transaction. Note that only the main and scheduler processes interact through synchronous transactions. These interactions serve as the synchronization mechanism that to enforces the proper linear ordering on the processes in the simulation. These interactions are discussed in more detail below.

Let us consider the three phases depicted in Fig. 5 that occur during a typical iteration of a neural network simulation. First, the output states of the nodes at simulated time k are computed, assuming an initial set of inputs has been supplied. In order to maximize concurrency, the main process issues a sequence of asynchronous transactions calls to the node processes requesting them to compute their outputs. Because these calls are asynchronous, the main process is not blocked after issuing the initial transaction call, but instead may continue sending transaction calls to other node processes. This allows the computation of the node outputs to be overlapped in time.

Before proceeding to the next phase (where the nodes propagate their outputs to adjacent nodes) we must guarantee that all nodes have computed their outputs at time k. That is, a synchronization step is required. If we proceed to the next phase without synchronization, then it is entirely possible for a node process to receive a request to propagate its output before the request to compute its output has been received. Thus, the output at time $k - 1$ would be propagated, instead of the output at time k, and the proper linear ordering for that process is not maintained.

This synchronization step is implemented using a call back mechanism. Each asynchronous transaction call issued by the main process supplies a transaction pointer to the receiving node process. At the completion of their processing activities, each node process issues an asynchronous transaction call through this transaction pointer. This transaction call is simply an acknowledgment that can be received by the scheduler process. Thus, after the main process issues the initial set of asynchronous transaction calls, a synchronous transaction call is made to the scheduler process. This call forces the main process to wait until the scheduler process has received acknowledgments from all node processes. At this point, the synchronous transaction is completed, and the main process may proceed to the next phase. Therefore, the main process is blocked from proceeding to the next phase until all node processes have completed the processing required to compute their output state.

In the next phase, the main process again issues a series of asynchronous transaction calls to the node processes—in this case instructing them to propagate their output values to all adjacent nodes in the network. Thus, in Fig. 5 we see that the main process issues these asynchronous transactions (again, supplying a transaction pointer to each node process), and then initiates a synchronous transaction with the scheduler process, causing the main process to move into a wait state. Each node process then passes its output, as well as a transaction pointer, to its adjacent nodes. When a node receives an output value, it notifies the calling process that it has received the output by invoking an asynchronous transaction through the transaction pointer provided by the calling process. This call back mechanism allows each node to determine when all of its outputs have been received. At this point the node process informs the scheduler process that it has completed communicating its outputs. This is accomplished by invoking an asynchronous transaction through the transaction pointer supplied by the main process. When the scheduler process is notified that all nodes have propagated their outputs, it completes the synchronous transaction with the main process, allowing the main process to move on to the next phase. The use of the call back mechanism in this phase allowed the communication events required to propagate node output values to be overlapped in time.

```
process spec scheduler_process()
{
  trans async ackn();
  trans void wait(int num_msgs);
};

process body scheduler_process()
{
  for (;;)
    select {
      accept wait(num_msgs) {
        for (int i=0; i<num_msgs; i++) {
          accept ackn();
        }
      }
      or terminate;
    }
}
```

Fig. 6. The specification and body of the scheduler process.

In the final phase, the network weights are adjusted through learning. This phase is similar to the first phase in that the main process issues a series of asynchronous transaction calls that instruct each node to adjust the weights that are incident to it according to its learning rule. Then the main process moves into a wait state until this processing is completed. The call back mechanism is also employed here so that the computation required by each node process can be overlapped in time.

The scheduler process is given in Fig. 6. This process contains one synchronous transaction **`wait()`**, and one asynchronous transaction **`ackn()`**. The process body is composed of an infinite loop that contains a select statement. The select statement is used to select between alternative actions. In this case there are two alternatives: either accept a **`wait()`** transaction call, or terminate the process. The terminate alternative will only be executed if either all other processes in the simulation have terminated, or they are all waiting at a terminate option.

As was illustrated in Fig. 5, the scheduler process is used to synchronize the various phases of the simulation. The synchronous transactions issued by the main process in Fig. 5 are **`wait()`** transactions. After this transaction call is

```
process spec node_process(class node *nptr, int num_inputs, int num_outputs,
                          input_edge *input_connect, output_edge *output_connect)
{
  trans async input(int input_num, double val, tptr np_ackn);
  trans async ackn();
  trans async compute_output(double step_size, tptr tp);
  trans async propagate_outputs(tptr tp);
  trans async update_weights(double step_size, tptr tp);
  // remaining transactions not shown
};
```

Fig. 7. The specification of the node process.

```
process body node_process(nptr, num_inputs, num_outputs, input_connect, output_connect)
{
  int i, num_trans;
  double activity=0, output=0;
  tptr np_ackn=((process node_process)c_mypid()).ackn;
  for (;;) {
    select {
      accept compute_output(step_size, tp) {
        activity = nptr→ComputeActivity(step_size);
        output = nptr→ComputeOutput(step_size);
        (*tp)();    // transaction call to scheduler process
      }
      or accept propagate_outputs(tp) {
        num_trans = 0;
        for (i=0; i<num_outputs; i++) {
          if (*(output_connect[i].id) == nptr)    // self-connection
            input_connect[output_connect[i].input_num].value = output;
          else {
            (*(output_connect[i].id))→Input(output_connect[i].input_num, output, np_ackn);
            num_trans++;
          }
        }
        for (i=0; i<num_trans; i++)
          accept ackn();
        (*tp)();    // transaction call to scheduler process
      }
      or accept input(input_num, val, tp) {
        input_connect[input_num].value = val;
        (*tp)();    // transaction call to a node process
      }
      or accept update_weights(step_size, tp) {
        nptr→UpdateWeights(step_size);
        (*tp)();    // transaction call to scheduler process
      }
      or terminate;
    }
  }
}
```

Fig. 8. The body of the node process.

received, the scheduler process remains in a loop until the appropriate number of asynchronous **`ackn()`** transactions have been received from the node processes. At this point, a reply is sent to the main process so that it may continue processing (i.e., the synchronous **`wait()`** transaction is completed).

The most important processes in the simulation system are the node processes. The process specification and body of the node process are given in Figs. 7 and 8, respectively. Note that a pointer to the node object associated with the node process is supplied as an input parameter.

Below we discuss the transactions responsible for the three phases in the neural network simulation discussed above. First consider the asynchronous **`compute-output()`** transaction. In Fig. 7 we see that one of the parameters that must be supplied to this transaction is a transaction pointer. This is the transaction pointer that is supplied by the main process, and used to call the **`ackn()`** transaction in the scheduler process. In Fig. 8, the body of the accept alternative for this transaction involves three operations. The first updates the activity of a node according to the value returned from the **`ComputeActivity()`** member function. Because **`Compute-Activity()`** is a virtual function, the type of object assigned to the **`nptr`** pointer will determine what operation is actually performed. Next, the output of the node is computed using the **`ComputeOutput()`** virtual member function. Again, the operation that is used to compute the output is dependent upon the type of object assigned to the **`nptr`** pointer. Finally, the transaction associated with the transaction pointer **`tp`** is invoked, sending an acknowledgment to the scheduler process that a particular node has completed its output computation.

In the next phase, shown in Fig. 5, the nodes communicate their outputs to adjacent nodes. This is accomplished using the **`propagate_outputs()`** transaction in the node process. Note in Fig. 7 that this transaction accepts a transaction pointer as a parameter. Fig. 8 shows the accept alternative for the **`propagate_outputs()`** transaction. Two loops are involved. The first iterates over the number of output connections that a node has, and if a connection is a self-connection, then no communication event is necessary. The current output of the node is simply assigned to the appropriate input of the node. If the connection is to another node, then the **`Input()`** member function of the adjacent node object is called and supplied with a transaction pointer. This member function simply calls the **`input()`** transaction of the process associated with that object. The accept alternative for this transaction is also shown in Fig. 8. It involves two statements: the first assigns the output value supplied as a parameter in the transaction call to the appropriate input connection, the next invokes the transaction associated with the transaction pointer that was also supplied as a parameter. Returning to the **`propagate_outputs()`** accept alternative in Fig. 8, we see that the second loop executes until every node process that was passed the output value returns an acknowledgment that the output was received. After this loop has completed, the transaction pointer supplied by the main process is used to communicate to the scheduler process that a particular node has completed the propagation of its outputs.

The final phase shown in Fig. 5 is implemented using the **`update_weights()`** transaction shown in Fig. 7. In the accept alternative for this transaction, shown in Fig. 8, the first statement calls the virtual member function **`UpdateWeights()`** associated with the node object, and the second statement notifies the scheduler process when the node has completed updating its weights.

The next section demonstrates how the concurrent base classes presented in this section can be used to simulate specific neural network models.

V. A Representative Model And Its Parallel Simulation

Below we demonstrate how the simulation environment presented in the previous section can be used to simulate a specific neural network model. We then present simulation results obtained from this model's execution on a parallel computer. The model we chose to simulate is intended to

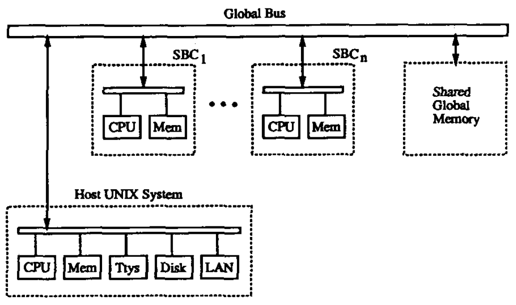


Fig. 9. A schematic diagram of the parallel computer that the concurrent simulations were executed on.

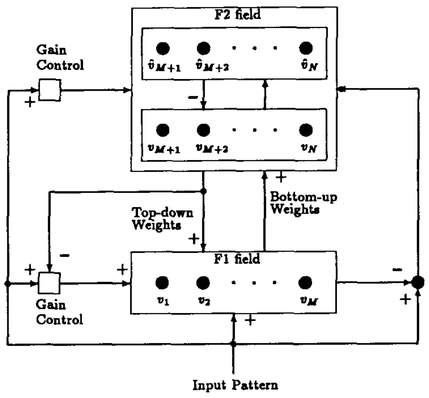


Fig. 10. The architecture of an ART1 network.

demonstrate the flexibility of this simulation system. In particular, a continuous-time model that has a relatively complex weight-update rule, and whose nodes are not homogeneous was chosen. The simulation environment was also used to implement discrete-time neural network models (e.g., the discrete-time Hopfield network [20]).

The parallel computer used to execute these simulations is depicted in Fig. 9. The architecture is a general-purpose bus-based shared memory multiprocessor system. Each processor in the system is a single-board computer (SBC) containing a 25 MHz Motorola 68030 microprocessor, a Motorola 68882 floating-point unit, and 8 Mb of on-board RAM. Access to local memory occurs via the SBC's local bus—this limits contention for the global bus. For the simulations performed here, the system was configured with six SBC's. Each SBC has a multi-tasking operating system that offers simple task management services, such as task creation and destruction [21]. Concurrent C++ is implemented on top of this operating system, and each Concurrent C++ process is mapped to a task in the underlying operating system. This allows the Concurrent C++ process to use the underlying SBC operating system facilities in a natural fashion, and limits the context switching overhead.

The architecture of this machine, as well as the manner in which Concurrent C++ is implemented, favors simulations that involve coarse-grained processes (i.e., processes in which the ratio of the amount of computation to the number of communication events is large). In order to increase the granularity of the processes in the simulation system, the **node** process (see Fig. 8) was modified so that the **propagate_outputs()** transaction only initiates communication events with other processes in the system if the output of the node associated with the transaction call has changed since the previous call to **propagate_outputs()**. This optimization allowed significant performance improvements in many of the networks we simulated. We now present the neural network model that is used below to demonstrate the use of the simulation system.

The ART1 Network

A neural network architecture for the learning of recognition categories was derived by Carpenter and Grossberg [22]. This architecture was termed ART1 in reference to the *adaptive resonance theory* introduced by Grossberg [23]. It was shown that ART1 self-organizes and self-stabilizes its recognition codes in response to arbitrary orderings of arbitrarily many binary input patterns [22]. The neural network simulated here is actually an extension of the of the ART1 network that, under certain parameter constraints, behaves exactly like Carpenter and Grossberg's original model [24], [25]. These extensions allow the ART1 model to be implemented *solely* as a set of concurrently executing nonlinear differential equations. (The simulation system can also be used to implement the original model if it is so desired.)

The architecture of this augmented ART1 network is shown in Fig. 10. It consists of two subsystems. The attentional subsystem contains an input representation field F1, as well as a category representation field F2. The orienting subsystem contains a reset node that interacts with the attentional subsystem to mediate an internally controlled search process. In particular, the F1 field contains a single layer of nodes that are used to represent an input pattern. The F2 field contains two layers of nodes. The first layer of nodes is used to categorize or code the input patterns appearing at the nodes in the F1 field, and the second layer (the $\hat{v}$ nodes in Fig. 10) are inhibitory nodes that are used in conjunction with the reset node (v_r in Fig. 10) to ensure that the proper nodes in the first layer of the F2 field are chosen to code the input patterns. Each node in the F1 field is connected via bottom-up weights to all nodes in the first layer of the F2 field, and each node in the first layer of the F2 field is connected via top-down weights to each of the F1 field nodes. Furthermore, each inhibitory node in the second layer of the F2 field is connected to a single node in the first layer of the F2 field, and the reset node v_r receives input from both the input pattern and the F1 field nodes, and passes its output to the inhibitory nodes in the F2 field. Below we summarize the form of the differential equations that define this ART1 model. These equations are presented in more detail in [22] and [25].

The activity of the network nodes in the F1 field and the first layer of the F2 field is described by the following differential equation:

$$\tau_x \frac{d}{dt} x = -x + (1 - Ax)J^+ - (B + Cx)J^- \qquad (6)$$

```
class threshold_node : public node {
  protected:
    double upper_bound, lower_bound, threshold_value;
  public:
    void ComputeOutput();
    // remaining member functions not shown
};

void threshold_node::ComputeOutput()
{
  if (activity ≥ threshold_value)
    output = upper_bound;
  else
    output = lower_bound;
}
```

Fig. 11. The threshold node class.

where x is the nodal activity; A, B, and C are network parameters; and J^+ and J^- represent the total excitatory and inhibitory net input to the node, respectively. Equation (6) is called a *shunting* differential equation because J^+ J^- multiply the node of activity x. Notice that if $A > 0$ and $C > 0$, then the activity of the node remains in the bounded range $[-BC^{-1}, A^{-1}]$ no matter how large J^+ and J^- become. Note also that the activity of the node decays to a resting level of 0 when $J^+ = J^- = 0$. J^+ and J^- are computed differently depending upon whether the node is in the F1 or F2 field. In both cases, these computations involve nonlinearities.

The activity of inhibitory node $\hat{v}_j$ in the F2 field satisfies

$$\tau_x \frac{d}{dt}\hat{x}_j = -[1 - g(I)]\hat{x}_j + g(I) f_H^r(x_r) f_H^2(x_j) \qquad (7)$$

where

$$g(I) = \begin{cases} 1, & \text{if } \sum_{i=1}^{M} I_i \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

and I_i is the ith component of the input pattern.

The reset node satisfies

$$\tau_x \frac{d}{dt} x_r = -A_r x_r + \mathrm{U}[P \sum_{i=1}^{M} I_i - Q \sum_{i=1}^{M} f_H^1(x_i)] \qquad (8)$$

where P and Q are network parameters, and U is the unit step function.

The value of the bottom-up weight, w_{ij}, associated with an edge connecting node v_i in the F1 field to node v_j in the first layer of the F2 field is determined by the following differential equation:

$$\tau_w \frac{d}{dt} w_{ij} = K f_H^2(x_j) \left[E_{ij} w_{ij} + f_H^1(x_i)\right]. \qquad (9)$$

where K is a network parameter, and the exact expression for E_{ij} can be found in [22].

A similar differential equation determines the value of the top-down weights w_{ji}. A description of how the parameters in these differential equations should be chosen, as well as theorems relating the initial value of the weights to network performance, can be found in [22] and [25].

Simulation Results

All of the nodes in the ART1 network compute their outputs using the hardlimiting threshold function of (3). The **threshold_node** class shown in Fig. 11 was created to

```
class F1_node : public threshold_node {
  protected:
    double A, B, C, D, tau, K, E, wt_tau;
    ode* diff_eq;
    ode** w_diff_eq;
  public:
    void ComputeActivity(double StepSize);
    void UpdateWeights(double StepSize);
    //remaining member functions not shown
};
```

Fig. 12. The class specification for the nodes in the F1 field of the ART1 network.

represent these types of nodes. Specifically, this class was created by inheriting the generic node class. Notice that additional data members were added to represent the β, γ, and δ values for the threshold node (i.e., **upper_bound** stores β, **lower_bound** stores γ, and **threshold_value** stores δ). The body of the virtual **ComputeOutput()** member function is also shown in Fig. 11.

Each specific node type in the ART1 network simulation was created by inheriting the **threshold_node** class. As an example, consider the class specification of the F1 field nodes shown in Fig. 12. Most of the protected data members in this class are used to store the parameter values found in (6) and (9). In addition, two of the data members store differential equation objects that are used in the numerical solution of (6) and (9). Specifically, **ode** is the base class associated with a hierarchy of numerical methods for solving ordinary differential equations. In Fig. 12, **diff_eq** is a pointer to an object of this base class. This object pointer is used by the **ComputeActivity()** member function to compute a node's activity according to (6). Thus, the form of the object assigned to the **diff_eq** pointer will determine what numerical technique is used to approximate the differential equation. Likewise, the **w_diff_eq** data member in Fig. 12 is used to store the address of a pointer to an array of pointers to **ode** objects. These objects are used by the **UpdateWeights()** member function to approximate the differential equations that determine the weight values associated with impinging connections. The significance of this approach is that the classes of nodes used in the simulation of the ART1 network do not have to be modified and recompiled if one wishes to experiment with different numerical techniques. Instead, pointers to objects of the new technique can be assigned to these data members, and they will be used in computing the activity and weight values of the node. A similar class specification exists for each type of node in the ART1 network simulation.

The ART1 neural network class shown in Fig. 13 is created by inheriting the **network** class given in Fig. 4. An additional member function is also added so that the network can iterate its node and weight equations according to the model presented above. The body of this **Iterate()** member function is shown in Fig. 13. This member function is responsible for initiating each of the phases depicted in Fig. 5. First, every node in the network is instructed to compute its output by invoking the **ComputeNodeOutput()** member function for each node object. This member function is responsible for invoking the **compute_output()** transaction of the process

```
class art1_ann : public ann {
  public:
    void Iterate(double StepSize);
    // remaining member functions not shown
};

void art1_ann::Iterate(double StepSize)
{
  int n;
  tptr tp = scheduler.ackn;
  for (n=0; n<num_nodes; n++)
    nd[n]→ComputeNodeOutput(StepSize, tp);
  scheduler.wait(num_nodes);
  for(n=0; n<num_nodes; n++)
    nd[n]→PropagateOutputs(tp);
  scheduler.wait(num_nodes);
  for(n=0; n<num_nodes; n++)
    nd[n]→UpdateNodeWeights(StepSize, tp);
  scheduler.wait(num_nodes);
}
```

Fig. 13. The ART1 network class.

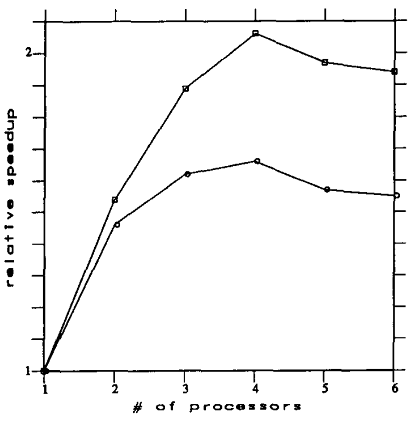


Fig. 14. Relative speedup curves for the concurrent ART1 simulations. The o's mark the nonoptimized simulations, and the □'s mark the optimized simulations.

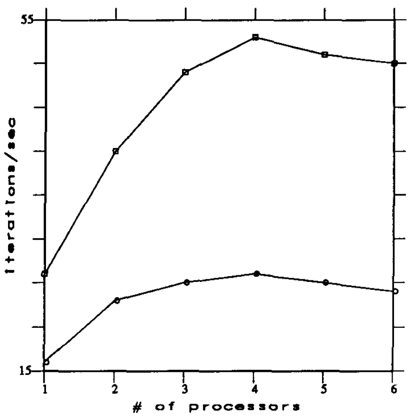


Fig. 15. Processing speed versus the number of processors used in the concurrent ART1 simulations. The o's mark the nonoptimized simulations, and the □'s mark the optimized simulations.

associated with a node object. As discussed previously, this transaction in turn invokes virtual member functions that compute the node's output according the type of node it is. Next, the **scheduler** process associated with the network (see Fig. 4) is instructed to wait until all nodes have completed computing their outputs. Each subsequent phase in the iteration proceeds in a similar manner.

In order to demonstrate the usefulness of these classes, an ART1 network containing 4 nodes in the F1 field, a reset node, and 8 nodes in the F2 field (4 nodes in the first layer and 4 nodes in the second layer) was simulated. The node differential equations were numerically approximated using an **ode** object that implements the Runge–Kutta method. A step size of 10^{-5} was used. All of the node activity and weight values were updated at each time step. Three input patterns were presented to the network: $I^1 = 1000$, $I^2 = 0000$, and $I^3 = 1100$. I^1 was presented for a simulated time of 1 s, followed by I^2 which was presented for a simulated time of 0.2 s, and then I^3 which was also presented for 0.2 s. This results in a total of 140000 iterations or time steps. Note that I^2 is a null pattern that is used between the presentation of other "interesting" patterns. That is, the presentation of pattern I^2 can be interpreted as the absence of an input pattern.

The network parameters for this simulation were chosen to satisfy the constraints presented in [25]. Fig. 14 shows the relative speedups that were obtained on the parallel computer shown in Fig. 9 utilizing from one to six processors. Speedup values for N processors were calculated by determining the ratio of the time taken to execute a simulation on a single processor to the time taken to execute the same simulation on N processors. The curve marked with o's in Fig. 14 is for simulations in which the outputs of the nodes were propagated at each time step, while the curve marked with □'s is for the optimized simulations in which the output of a node was only propagated if it was different from its output at the previous time step. Note that a speedup of over 2 is obtained with only four processors using the optimized simulation.

A further demonstration of the performance of these simulations is given in Fig. 15. In this figure the processing speed, in terms of iterations per second, is shown for both the optimized and nonoptimized approaches. An iteration is defined as the processing required to compute one simulated time step (see Fig. 5). Note that for a single processor, the optimized simulation is more than 1.5 times as fast as the nonoptimized solution. Furthermore, as each additional processor is added (up to four) this performance difference increases. For more than four processors, the performance difference is constant. This figure demonstrates how the simulation techniques presented here can be used to reduce the communication overhead in parallel implementations of neural network models.

Additional information about processor utilization during these simulations is contained in Table I. Table I(a) shows the percentage of time the processors were busy during the six simulation runs in which the outputs of the nodes were propagated at each time step.

This includes the time that a processor is waiting to become bus master. Lower number processors have priority for becoming bus master (i.e., bus mastership is granted via a daisy chain). Notice that higher numbered processors in Table I(a) are busier than the corresponding processors in Table I(b).

TABLE I
(a)PROCESSOR UTILIZATION DURING THE SIMULATION RUNS IN WHICH THE OUTPUT OF NODES WERE PROPAGATED AT EACH TIME STEP. (b) PROCESSOR UTILIZATION DURING THE SIMULATIONS RUNS IN WHICH THE OUTPUT OF NODES WERE PROPAGATED ONLY IF THEIR OUTPUTS DIFFERED FROM THOSE AT THE PREVIOUS TIME STEP.

	%busy / processor					
# processors	1	2	3	4	5	6
1	100	-	-	-	-	-
2	100	97	-	-	-	-
3	98	95	91	-	-	-
4	90	84	90	95	-	-
5	69	79	84	90	93	-
6	98	95	87	74	71	82

(a)

	%busy / processor					
# processors	1	2	3	4	5	6
1	100	-	-	-	-	-
2	98	87	-	-	-	-
3	94	92	72	-	-	-
4	88	73	72	87	-	-
5	67	70	62	70	83	-
6	94	90	55	50	50	65

(b)

This indicates that performance degradation is related to bus contentions during interprocessor communications. However, if bus contention was the only factor limiting parallelism, then the higher numbered processors (with lower priority for bus access) would be the busiest. This does not happen. In the simulations using six processors, the first two processors are the busiest. This indicates that there is a degree of program-limited parallelism in these simulations.

In order to gain a better understanding of the type of performance that can be expected from this simulation system for different neural network models, let us consider Fig. 5 again. Recall that in this figure a solid line indicates a running process, while a dashed line indicates that the process is waiting. The horizontal arrows in this figure represent transaction calls. That is, they represent the communication overhead associated with the simulation. If we assume a fixed amount of communication overhead, then the amount of speedup that can be obtained is directly related to the degree of overlap (in time) of the solid lines in this figure. The more compute intensive the nodes' processing activities are, the longer the solid lines in Fig. 5 become, and consequently the degree of overlap increases. This means that relative to the communication overhead, a larger percentage of the processing time is devoted to parallel computation. Thus, we would expect that more complicated (i.e., compute intensive) neural network models would achieve higher speedup results. This expected behavior has been verified by results obtained from additional simulations.

VI. CONCLUSION

A framework for the concurrent simulation of neural network models as discrete event nonlinear dynamical systems was presented, and the simulation of a specific neural network model was demonstrated on a general-purpose parallel computer. The basic components of the simulation system are two generic base classes that capture the functionality of general neural network models. Specific neural network models are created through the inheritance of these classes. The most complex issues involved in the development of the simulation system were those related to the synchronization of the concurrent processes in the simulation, as well as deadlock avoidance. The base classes encapsulate these concurrent processes, thereby allowing users to construct novel neural network models from a higher level of abstraction. Specifically, the user does not have to deal with synchronization and deadlock avoidance issues when constructing neural network models. The flexibility of this system was exhibited by simulating a specific nontrivial neural network model on a general-purpose parallel computer.

The parallel computer used to execute the simulation program was a bus-based shared memory system. The main advantage of these types of systems is that they are easy to build with off-the-shelf components; however, bus contention is a limiting factor in bus-based parallel computers. Typically only a small number of processors can be effectively utilized by such computer systems. The use of additional processors leads to diminishing returns in terms of processor utilization, and therefore adversely affects speedup. The results we obtained indicate that the simulation system would scale to larger parallel computers that utilize switching networks for interprocessor communication.

It should be mentioned that this system can also be used to simulate neural networks whose learning rule is based on the popular back-propagation algorithm [26]. Although the back-propagation learning rule violates the locality constraint discussed in Section III, a number of techniques have been developed that remove this problem (see [27] and [28]). These forms of back-propagation learning that employ local techniques are well-suited for the concurrent simulation system discussed here.

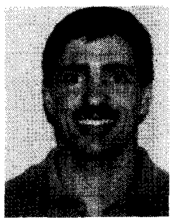
ACKNOWLEDGMENT

The authors wish to thank W. Horne and J. Howse for reading the manuscript and offering valuable comments. The authors also thank D. Hush, C. Abdallah, E. Schamiloglu, and G. Donohoe of the BSP Group for their helpful discussions and suggestions.

REFERENCES

[1] DARPA Neural Network Study, B. Widrow, Study Director. Fairfax, VA: AFCEA International Press, 1988.
[2] J. Alspector and D. Hammerstrom,"Electronic and optical implementations sessions," in *Proc. Int. Joint Conf. Neural Networks,*, vol. 1, pp. 415-592, 1991.
[3] R. P. Lippmann *et al., Advances in Neural Information Processing Systems 3.* San Mateo, CA: Morgan Kaufmann, 1991, pp. 993-1052.
[4] C. Mead, *Analog VLSI and Neural Systems.* Reading, MA: Addison Wesley, 1989.
[5] E. Sánchez-Sinencio, Ed., *IEEE Trans. Neural Networks*, Special Issue on Neural Network Hardware, vol. 2, pp. 192-251, 1991.
[6] G. L. Heileman *et al.,*, "A neural net associative memory for real-time applications," *Neural Computation*, vol. 2, pp. 107-115, 1990.

[7] D. A. Pomerleau *et al.*, "Neural network simulation at WARP speed: How we got 17 million connections per second," in *Proc. IEEE Int. Conf. Neural Networks*, vol. II, pp. 143–150, 1988.
[8] X. Zhang *et al.*, "An efficient implementation of the back-propagation algorithm on the connection machine CM-2, in *Advances in Neural Information Processing Systems 2*. San Mateo, CA: Morgan Kaufmann, 1990, pp. 801–809.
[9] N. Gehani and W. D. Roome, "Concurrent C++: Concurrent programming with class(es)," in *Software—Practice & Experience*, vol. 18, pp. 1157–1117, 1989.
[10] S. B. Lippman, *C++ Primer*, 2nd ed. Reading, MA: Addison-Wesley, 1991.
[11] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Reading, MA: Addison-Wesley, 1991.
[12] N. Gehani and W. D. Roome, *The Concurrent C Programming Language*. Summit, NJ: Silicon Press, 1989.
[13] B. Meyer, *Object-oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall, 1988.
[14] C. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall, 1985.
[15] P. Brinch Hansen, "Distributed processes: A concurrent programming concept," *Communications of the ACM*, vol. 21, 1978.
[16] G. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, pp. 125–141, 1990.
[17] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms, and architectures," *Neural Networks*, vol. 1, pp. 17–61, 1988.
[18] F. J. Pineda, "Dynamics and architecture for neural computation," *J. Complexity*, vol. 4, pp. 216–245, 1988.
[19] R. Righter and J. C. Walrand, "Distributed simulation of discrete event systems," *Proc. IEEE*, vol. 77, pp. 99–113, 1989.
[20] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," in *Proc. National Academy of Science USA*, vol. 79, pp. 2554–2558, 1982.
[21] W. D. Roome, "The CTK: An efficient multi-processor kernel," AT&T Bell Laboratories, 1986.
[22] G. A. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics, and Image Processing*, vol. 37, pp. 54–115, 1987.
[23] S. Grossberg, "Adaptive pattern recognition and universal recoding II: Feedback, expectation, olfaction, and illusions," *Biological Cybernetics*, vol. 23, pp. 187–202, 1976.
[24] G. L. Heileman and M. Georgiopoulos, "The augmented ART1 network," in *Proc. Int. Joint Conf. Neural Networks*, pp. 467–472, 1991.
[25] ——, "A real-time representation of the ART1 network," *rep. no. EECE 91–001*, Univ. New Mexico, Jan. 1991.
[26] D. E. Rumelhart *et al.*, "Learning internal representation by error propagation," pp. 318–362, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA: MIT Press, 1986.
[27] L. B. Almeida, "A learning rule for asynchronous perceptrons with feedback in a combinatorial environment," in *Proc. IEEE Int. Conf. Neural Networks*, vol. II, pp. 609–618, 1987.
[28] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Proc. Int. Joint Conf. Neural Networks,*, vol. 1, pp. 593–606, 1989.



Gregory L. Heileman (S'86–M'89) received the B.S. degree from Wake Forest University in 1982, the M.S. degree in biomedical engineering and mathematics from the University of North Carolina in 1986, and the Ph.D. degree in computer engineering from the University of Central Florida in 1989.

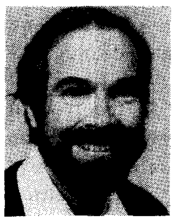
He is currently an Assistant Professor in the Electrical and Computer Engineering Department at the University of New Mexico, Albuquerque. His research interests include neural networks, pattern recognition, learning theory, parallel computation, and object-oriented simulation.

Dr. Heileman is a member of the IEEE Computer Society, the Association for Computing Machinery, and the International Neural Network Society.

Michael Georgiopoulos (S'84–M'86) received the Diploma in electrical engineering from the National Technical University of Athens, Greece, in 1981, and the M.Sc. and Ph.D. degrees in electrical engineering from the University of Connecticut, Storrs, in 1983 and 1986, respectively.

In 1987, he joined the University of Central Florida, Orlando, where he is now an Assistant Professor in the Department of Electrical Engineering. His research interests are in the areas of spread spectrum communications and neural networks.

Dr. Georgiopoulos is a member of the Technical Chamber of Greece and the International Neural Network Society.



William D. Roome (S'72–M'74) received the B.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1970 and 1974, respectively.

He has been a member of the technical staff at AT&T Bell Laboratories since 1974. He has also worked on Concurrent C, a time-oriented file server, a shared-memory database machine, and several real-time kernels. His current research interests include operating systems, distributed systems, programming languages, and database machines.