CHAPTER 13

P3: A Parallel Network Simulating System

D. ZIPSER and D. RABIN

Research on parallel distributed processing is to a large extent dependent upon the use of computer simulation, and a good deal of the researcher's time is spent writing programs for this purpose. Virtually all the PDP systems described in this book require special-purpose computer programs to emulate the networks under study. In writing programs of this type, it is usually found that the basic algorithms of the PDP network are easy to program but that these rather simple "core" programs are of little value unless they are embedded in a system that lets the researcher observe and interact with their functions. These user interface programs are generally tedious and very time consuming to write. What is more, when they are directed toward one particular system they can be quite inflexible, making it difficult to easily modify the PDP network being studied. Also, because of the time involved, particularly for interactive graphics programs, the researcher often makes do with very limited facilities for analyzing the performance of the network. In this chapter we will describe a general-purpose parallel system simulator called P3. It was developed with PDP research explicitly in mind and its major goal is to facilitate simulation by providing both the tools for network description and a powerful user interface that can be used with any network described using the tools. There are many problems to be faced and tradeoffs to be made in designing such a system but in the process of doing this we feel that not only has a useful system been developed, but also that we have learned a great deal about the whole problem of PDP simulations.

In P3 networks, each computing element, called a *unit,* contains a computer program that reads inputs from connections and sets outputs on other connections, possibly also modifying some local state parameters. The major components of P3 are:

- The *plan language*, which describes the collection of units in a model and specifies the connections between them. This description is called a "plan."
- The *method language,* an extension to LISP, which implements the internal computational behaviors of the units in a model.
- The *constructor*, which transforms the plan and associated methods into a computer program and, when run, simulates the network.
- The *simulation environment*, which provides an interactive display-oriented facility for observing and testing P3 models.

Input to units described in a P3 plan can come only from other units in the plan. That is, there is no "outside world" in a P3 plan language description of a network. This means that at the level of description of the P3 plan language, the P3 network is closed. Access to the *outside* world must occur *inside* a unit through its method. Methods may access the world outside the P3 system through any available computer peripheral. The only thing that methods are not allowed to do is to reconfigure with the P3 system itself or communicate with other methods through "underground connections" not mentioned in the P3 plan.

In any simulation, the relationship between real and modeled time is of key importance. A real unit, such as a neuron, would read inputs continuously and update its outputs asynchronously, but this cannot be simulated exactly on a digital computer. Many simulations use a simple synchronous approximation to real time. However, sometimes this produces unwanted artifacts and a closer approximation of asynchrony is required. Often, in fact, what the investigator really wants to do is to experiment with the effect of different kinds of time simulation on the network under study. Since there is no way for the system designer to know in advance all the possible ways that the investigator will want to handle time, some strategy has to be used that allows great flexibility. The approach taken by P3 is that this flexibility can come through the methods that can use conditional updating. The P3 system itself is completely synchronous and updates all units on each cycle. Since updating a unit involves invoking its method, the question of whether or not the outputs of a unit actually change on any P3 cycle can be

decided by the method. For example, to model asynchronous updating, each unit can have an additional input that controls whether or not it is updated on a cycle. Then the decision as to which units are to be updated can be given to a control unit that is connected by a separate line to the update inputs of all the other units. The method program inside this control unit decides which units in the network will be updated on each cycle. Note that this approach is very flexible since small changes in the method program of the control unit can implement a large range of possible update time schemes.

A typical P3 plan might contain a large number of simple neuron-like units forming the core of the network together with a few special purpose units to generate input to the core network and control its function. The master control unit, used above to implement asynchronous updating, is an example of this kind of special-purpose unit. They can also be used to sequence simulated experiments and to interpret output of other units. How all this can be done will become clearer as we describe the use of P3 in detail. They key point here is that the P3 "style" is to include *within* the P3 plan all aspects of the simulation including input to and control of the core network. This approach simplifies the problem of constantly having to interfere special-purpose routines to a general-purpose modeling environment.

It often happens that networks are modular, that is, made up of distinct subnetworks. P3 facilitates the use of modularity by allowing subnetworks to be treated as single processing units. This feature is of particular use when several P3 units are used to simulate a single object such as a "realistic" neuron. The modular feature also facilitates "top-down" and "structured" definition of the plan even when the underlying networks are not particularly modular.

The P3 plan language has an additional feature that is not directly concerned with describing the functional aspects of a parallel network. Every unit in a P3 plan has a location in a three-dimensional Euclidean reference frame call *P3 space.* This means that every P3 plan not only describes a network, but it also describes a geometrical structure. Since the functioning of a P3 network does not depend on its geometrical structure, it might seem odd to go to all the trouble of describing the geometry. There are two main reasons for locating P3 units in space. The first reason is to facilitate visualizing a P3 network while observing its function during the simulation of a model. The units can be placed so that they appear at the same relative positions on the computer display during simulation as they have in the investigator's conceptual image of the model. The second reason to give each unit a position in space is to make it possible to specify connections between units implicitly on the basis of their spatial locations rather than explicitly. This latter feature is of particular importance when modeling systems in

which the connectivity is described in terms of geometrical relations. This is often the case when dealing with realistic neuronal modeling, especially of primary sensory processing structures.

The P3 Plan Language

The job of the P3 plan language is to describe the units and connections that constitute the network being simulated. To do this, the language uses a small but rich set of statements that make it possible to succinctly describe large groups of complex, connected units. The three fundamental constituents of the plan language are the UNIT TYPE, UNIT, and CONNECT statements. The UNIT TYPE statement names and describes a kind of unit. The UNIT statement instantiates and names actual units. This statement can instantiate either a single unit or a whole array of units of the same type. The CONNECT statement makes connections. Since the statement can be used inside of loops, a single connect statement can make an arbitrarily large number of connections using the available array features.

A unit in P3 can have any number of inputs and outputs together with any number of parameters. Before the start of a simulation, values must be given to all parameters and to all outputs. Each value is always a single computer word in length. The interpretation of this word depends on how the methods use it. As the simulation proceeds, these initial values are continuously updated. Taken together, the values of the parameters and the outputs constitute the state of the system at any time during simulation. The major difference between parameter values and output values is that outputs are available to other units in a network through connections, while the value of parameters can only be read by the unit to which they belong. P3 units can have two classes of parameters: *unit parameters* and *terminal parameters*. The unit parameters apply to the whole unit, for example, the threshold in a linear threshold unit. The terminal parameters are associated with individual inputs or outputs and correspond, for example, to weights.

An important function of the P3 plan language is to describe the connections between units. Since units can have multiple inputs and outputs there has to be some way to name them so that the CONNECT statements will know which connections to make. These names are also used within the method programs to read inputs and set outputs. The basic form of the CONNECT statement is

(CONNECT < unit-name > OUTPUT < output-name >
TO < unit-name > INPUT < input-name >)

For units with only a few inputs or outputs each input or output can be given a separate name. When a unit has a large number of inputs or outputs it is more convenient to group them together in input or output arrays. The individual items in these arrays are referenced by giving the array name and a set of subscript values. These arrays can be used in iterative statements in plans and methods.

An output value can serve as input to any number of units, i.e., the fan-out is arbitrarily large. Each individual input can receive only one value. This is easy to enforce as long as it is known that just one connection is to be made to each input. This works well in many cases but it often happens that it is very hard or impossible for the programmer to know exactly how many connections will be made. This is the case, for example, when connection decisions are being made implicitly by some computational procedure such as "connection by location" or random connection. To overcome this, P3 secretly treats each individual input as an array and automatically adjusts its size to fit the number of inputs. This process is transparent to the programmer which means that multiple connections can be made freely to a single input. There is a special iteration statement in the method language to access these multiple inputs. Each individual input that actually gets generated is called a TERMINAL and there are procedures for associating parameters with terminals and initializing their values.

The method programs that implement the internal functionings of units are written in the form of ordinary computer programs in an appropriate language. In the current implementation, which runs on the Symbolics 3600, the language is LISP. In order to allow the methods to use the values of inputs and parameters in their computations, a set of special access statements is incorporated into this system and is available to LISP programs. These statements make it possible for methods to read and set inputs, outputs, and parameters more or less as if they are ordinary variables.

In order to illustrate how P3 works, we will describe a model of a simple competitive learning network of the type described in Chapter 5. The basic network contains two types of units: a pattern generator to supply stimuli and a cluster of competitive learners connected to it, which spontaneously discover some features of the patterns. Since learning is spontaneous and does not require a teacher, the functioning of the network is simple and straightforward. The pattern generators sequentially produce output patterns that serve as input stimuli to the cluster units. Each pattern is an activation vector specifying which of the inputs are active and which are not. Each cluster unit produces an output indicating its response to the current stimulus which is transmitted to all other members of the cluster to create a "winner take all" network. The cluster unit which wins is the only one that learns and it

uses the weight redistribution procedure described in the competitive learning chapter, that is,

$$\Delta\omega_{ij} = \begin{cases} 0 \text{ if unit } j \text{ loses on stimulus } k \\ g\dfrac{c_{ik}}{n_k} - g\omega_{ij} \text{ if unit } j \text{ wins on stimulus } k \end{cases}$$

where c_{ik} is equal to 1 if in stimulus pattern S_k, element i in the lower layer is active and zero otherwise, and n_k is the number of active elements in pattern S_k (thus $n_k = \sum_i c_{ik}$).

The first step in creating a P3 plan is to supply the UNIT TYPE statements. The UNIT TYPE statement for the pattern generator is given below:

```
(unit type dipole
    parameters flag i1 j1 i2 j2
    outputs (d array i j)
    method <update routine code in lisp>)
```

In this, and all our other examples, words in italics are part of the P3 plan language while the nonitalicized words are supplied by the user. The UNIT TYPE statement gives the type a name that will be used throughout the plan. The name for the pattern generator type is "dipole." There are five parameters that are used internally for pattern generation. The technicalities of the use of these parameters is irrelevant here. The UNIT TYPE statement describes the output of the unit. This output is a two-dimensional array of lines called "d." This array of outputs is the retina on which stimulus patterns are generated which serves as an input to the competitive learning cluster units. The "i" and "j" that follow the word *array* are dummy variables that tell P3 how many dimensions the array has. The actual size of the array is variable and is initialized when we instantiate units of the type dipole. Note that the unit type dipole had no inputs since it is itself the source of patterns.

The second basic unit type is the competitive learning unit, which in our plan we call "competitor." The unit type statement for it is given below:

```
(unit type competitor
    parameters p g flag
    inputs (C array i j terminal parameters W)
        (i-A)
    outputs (o-A)
    method <lisp code>)
```

Note that the input array "C" of this unit corresponds exactly in form to the output array "d" of the dipole unit described previously. This correspondence will make it possible to make one-to-one connections between the output of dipole type units and the input of competitor type units. Also notice that a terminal parameter "W" has been associated with the input array "C." The competitor unit needs an additional input called "i-A" which will receive information from the outputs of all the other members of the cluster.

We have described the two unit types we will need. We can now go ahead and instantiate units of these types. The statement that creates a pattern generator unit of type dipole is shown below:

```
(unit stimulus of type dipole
    at (@ 0 0 0)
    outputs (d array (i 0 5) (j 0 5))))
```

The unit statement names the unit it is creating. This is the name of a real unit that is actually going to exist in our model and it is the name that will be referred to when this unit is connected to other units. For P3 to build such a unit, it has to be told the type. There can be any number of units of the same type and they can all have different names. Since every real unit in P3 has a location in P3 space, we must specify it in the unit statement that instantiates the unit. The *at* clause is used for this. The *at* is followed by a location specifier that simply evaluates to the x-, y-, and z-coordinates of the unit in P3 space. For simplicity we locate the pattern generator at the origin of P3 space which will initially be located at the center of the display window when we simulate the model. Since we are building a real unit, we have to give a size to its array of output lines. This is done in the *outputs* clause of the UNIT statement. Each subscript specifier consists of a subscript name and initial value, which in the current implementation must be 0, and final value, which in this example is 5 for both the "i" and the "j" subscripts. This statement will generate a 6×6 array of output lines on connector "d."

Now that we have a source of patterns, we need to create a cluster of units that will receive these patterns. The statement that instantiates these units is given below:

```
(unit cluster array (k 0 – cluster-size 1) of type competitor
    at (@ ( * k (+ cluster-size 4))(+ cluster-size 10) 0)
    initialize (g = 0.05)
    inputs (C array (i 0 (– stimulus-size 1))(j 0 (– stimulus-size 1)))
```

In this case, we are not instantiating a single unit but an array of units. In the competitive learning model, the learning cluster always consists

of two or more units, so we want a way to vary the number of units in a cluster. In the first line of the unit statement we give the name cluster to the array and then we indicate the size of the array with a subscript specifier. The name of this subscript is "k"; its initial value is 0. Its final value is one less than the global constant "cluster-size." The value of cluster-size, which will occur at various points in the plan, is set by a statement at the beginning of the P3 plan that determines the value of global constants. This feature means that we can change the parameters such as cluster-size globally throughout the plan by only fiddling with a single value. The upper bound of the stimulus input line array has also been set with the use of a global constant "stimulus-size" rather than with an integer as was done previously. Also notice that the variable "k" is used in an *at* clause to place each unit of the array at a different place in P3 space.

Our next task is to connect the units together in the appropriate fashion. We have two classes of connections: those that go from the stimulus generator to the learning cluster and those that interconnect the units within the learning cluster. Each of these classes of connections has many individual connections within it, but these individual connections can be specified algorithmically in such a way that only a few CONNECT statements are needed to generate the entire network. What is more, the algorithmic specification of these connections makes it possible to change the size of the cluster or the size of the stimulus array without altering the CONNECT statements at all. The code required to connect the stimulus to the learning cluster is given below:

```
(for (k 0 (+ 1 k))
   exit when (> k cluster-size) do
   (for (i 0 (+ 1 i))
      exit when (> i stimulus-size) do
      (for (j 0 (+ 1 j))
         exit when (> j stimulus-size) do
         (connect unit stimulus output d i j
            to unit (cluster k) input C i j
            terminal initialize (W = (si:random-in-range
               0.0 (// 2.0 (expt (+ stimulus-size 1) 2)))))))))
```

There are three nested loops. The first ranges over each member of the cluster, and the next two range over each dimension of the stimulus array. Inside these three nested loops is a single CONNECT statement. The CONNECT statement has the job of initializing the value of any terminal parameters. In our model we have a very important terminal parameter, "W," the weight between a stimulus line and a cluster unit, which we want to initialize to a random value which sums

to one for the whole input array. This is accomplished by setting the initial value of "W" with a LISP function that evaluates to the required quantity. In general, in a P3 plan wherever a number is required, a function (in our case a LISP function) that evaluates to a number can replace the number itself. The sum of the random numbers generated by our simple LISP function is not exactly one, but only averages one. This is satisfactory for the competitive learning algorithm because it is self-normalizing and will force the sum to one in the course of learning.

The connections that link the members of a cluster are a bit more complex. Each member of the cluster must receive input from all other members except itself. The code for doing this in a completely general way for clusters of any size is given below:

```
(for (k 0 (+ 1 k))
   exit when (> k cluster-size) do
   (for (j 0 (+ 1 j))
      exit when (= j k) do
      (connect unit cluster k output o-A
         to unit cluster j input i-A))
   (for (j (+ k 1) (+ 1 j))
      exit when (> j cluster-size) do
      (connect unit cluster k output o-A
         to unit cluster j input i-A)))
```

The idea here is that we first connect each unit to those units whose subscripts are lower than it and then to each unit whose subscript is higher than it. This requires two separate loops, each with its own CONNECT statement, both nested within an outer loop that ranges over all units in the cluster. Note that this is a case of making multiple connections to a single input line. We don't have to know how many connections there are because within the method there is code that will examine all connections on this line to decide if the unit has won. This feature is very useful and can be applied whenever a method needs to know the value of an input but not its originating unit.

We have now specified all the features of a plan that describes the basic competitive learning network. Of course, this plan can only be used to construct a running model if we have available the appropriate method programs. Since these are ordinary computer programs written in LISP, we won't analyze them in detail. The code for the methods used here is given in the appendix of this chapter, which shows a complete plan for a simulation of competitive learning. It is worthwhile, however, to see how the method language accesses the inputs and outputs of the units about which we have been saying so much in the development of the plan.

The only difference between the arguments to a P3 method and the arguments to a normal LISP function is that the P3 arguments are accessed by special access functions. For example, to get the value of a parameter, the following form is used:

(*read-unit-parameter* flag)

This form returns the current value of flag. To read an input from an array of input lines the following form can be used:

(*read-input* (C i j))

In this case the value of "i" and "j" must be bound at the point in the program where an expression using them occurs. There are corresponding forms for reading terminal parameters, setting outputs, and setting parameter values.

The P3 Simulation System

The P3 simulation system is the environment in which models in P3 are simulated. It is highly interactive and makes extensive use of the window system and the "mouse" pointer of the Symbolics 3600. The first step in simulating a model is to compile the methods and construct the plan. The constructor is a program similar in purpose to a compiler. However, the input is a P3 description of a network, rather than a computer language description of a program. The output of the constructor is a data structure containing all the relevant information about the network that can be used by the P3 simulation system to run a simulation of the model. As with any form of computer programming, a model must be debugged before it can be simulated. There are really two levels of debugging for network models. First, the user wants to know that the network that has been created is connected up in the way intended. Once this has been established, the actual functioning of the network can be debugged. P3 provides tools for both of these phases of the debugging process.

To check the correctness of connections, P3 provides a display that shows each unit in the model at its location in P3 space. The user interacts with this display with a mouse pointing device. Clicking on a particular unit provides a menu that enables the user to trace out any of the connections emanating from that unit. This facility for tracing out connections, one at a time, has proved much more useful than simply presenting a user with the wiring diagram of the model. Once the user is convinced that the constructed model corresponds to the envisioned network, the job of analyzing the function of the model can begin.

Analyzing the running of a complex simulation is a demanding task. It is in this analysis that we have found that all the features of the P3 system come together and begin to justify their existence. Because every object in the model has a location in P3 space that corresponds to the user's mental image of the network, the simulation system can display values representing the state of the system at locations on the screen that have meaning to the user. This means that during the course of a simulation, meaningful patterns of P3 variables can be displayed. This approach is widely used in analyzing the function of parallel systems. What P3 has done is to standardize it and relieve the user of the need to implement the details of this display strategy for each new model.

In the current implementation of P3, each object in the model is represented at its designated location by a small rectangular icon. By the use of a mouse pointer driven menu system, the user can assign the icon representing a unit the variable whose value is to be displayed. Thus, for example, the icons representing the input terminals of a cluster unit in our example can be assigned either the value of the input to that terminal or the value of the weight on that terminal. These assignments can be made or changed at any time during a simulation run. They can be set to be updated continually as a simulation proceeds, or they can be examined in detail when the simulation is temporarily interrupted. The current P3 implementation displays the relevant state values at two possible levels of precision. The approximate value of the state value is indicated by the degree of darkening of the icon. There are five levels of intensity. Their range is under user control and can be changed at any time. This enables the user to adjust the range so that the difference between the lightest and the darkest icons will optimize the information content of the display. There is also a high precision display that permits the exact value of any P3 variable to be examined.

Figure 1 shows how the screen of the Symbolics 3600 looks after 588 P3 cycles of simulation of a competitive learning model with a 6×6 stimulus array and a dipole stimulus. There are six windows displayed and each shows a different aspect of the simulation. Window A shows the three units in the model at their respective positions in P3 space. The upper narrow rectangle is the pattern generator. It is not displaying any value. The lower two rectangles represent the two cluster units. They are displaying the approximate value of their outputs by the size of the contained black rectangle. Clearly the unit on the left has a lower output value than the one on the right. Window B shows the output of the pattern generator unit, which was called "stimulus" in the plan. The lines form a square array because that is the way they were specified in the plan. The two dark rectangles show the current dipole

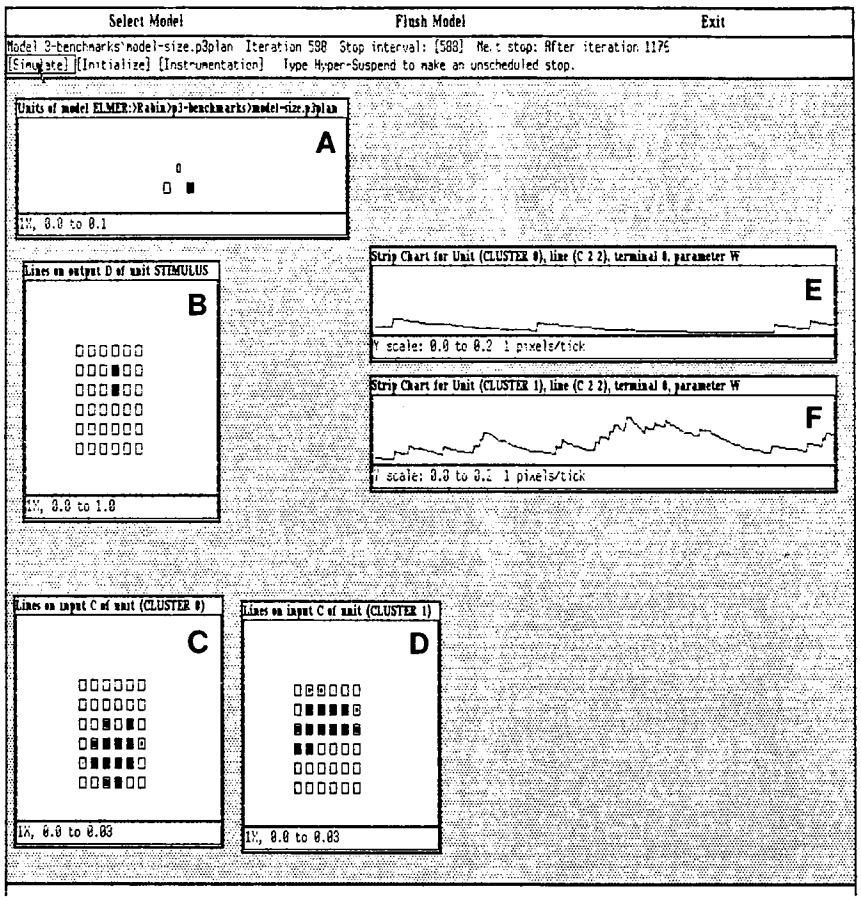

FIGURE 1. Display of Symbolics 3600 during P3 session. The mouse arrow is pointing to the "simulate" command in the upper left. Clicking a mouse button will start simulation.

pattern. Windows C and D show the approximate values of the weights on the input line arrays of each of the two cluster units. The fact that competition has pretty well separated the weights along a horizontal line is clearly visible from these two windows. Windows E and F are "strip chart" records that produce a graphical record of any P3 variable. The strip charts have been set up to record the value of a pair of corresponding weights, one from each unit in the cluster. Time increases to the right so the initial state is at the extreme left side of the strip charts. It is interesting to note that one of the weights became dominant for a while but at later times seems to have lost its dominance.

In addition to the special functions of P3, the user also has available all the powerful program development tools of the Symbolics 3600. For example, suppose that the user believes that an observed bug is due to an error in the code of a method. It is possible to interrupt the simulation, go directly to the editor buffer that contains the method code, alter it, recompile the alteration, and then return to the simulation at exactly the point at which it was interrupted. This facility has proved invaluable in debugging.

As we work with the P3 simulation system, we constantly find new features that are useful in the analytical process. We view the implementation of each of these new analytical techniques as analogous to adding a new instrument to a laboratory. Thus, we call the features of P3 that enable the user to analyze a functioning model "instruments." Each of these instruments can be called up at any time. Every instrument has a window that displays the results of the instrument's analysis. For example, one instrument is the "strip chart recorder" used in Figure 1. The strip chart recorder has a probe that can be connected to any particular state variable of any unit. Since multiple instances of any instrument can be created, any number of strip charts can be running at the same time. In addition to instruments that display the values of variables, we also envision a class of instruments that record these variables. Clearly, it is very important for a serious modeler to be able to record the results of a simulation. The instrument concept will enable the modeler to record just those variables required. This is a very important feature since simply recording the entire state of the model as it develops in time would produce an overwhelming flow of data.

Performance

So far we have said nothing about the speed at which simulations run. This is a problem of tremendous importance for PDP models. Big models inherently run slowly on serial computers. Generally, parallel programming systems like P3 stress ease of model definition and simulation. How much penalty must we pay in model performance? There is always some performance penalty for a general-purpose system. For any given piece of computer hardware, it is generally possible to write a specially tailored program that will run some particular model faster than any general system will run it. However, this special tailoring itself takes considerable time and makes it much harder to change the details of the model structure. Thus, we envision that programs like P3 will be useful in the early stages of model development when the size

of the models are modest and there is frequent need for changes in structure. When the structure and parameters of a model have been decided upon and it is necessary to scale the model up and have it run extremely rapidly, it may in some cases be advantageous to write a special program to implement the model.

The general-purpose systems, however, have several things going for them with respect to model performance. First of all, since the data structure has the same form for models, it is possible to put a lot of effort into optimizing running speed for the particular hardware on which the system is implemented. This optimization only has to be done once rather than for each model. A second way in which general-purpose systems can improve performance is through the use of special-purpose hardware. The models generated by the P3 system are inherently parallel models and map well to some parallel computer architectures. The one way to get blinding speed from parallel models is to implement real parallelism in parallel computers. In some cases, array processors can also be highly beneficial. Since all the P3 models are of the same sort, a constructor can be made that will provide the appropriate data structures to run any P3 model on these kinds of hardware. This will make the hardware transparently available to the user of systems like P3. This, we believe, is a significant plus, since it is notoriously difficult to program any particular application for array processors or truly parallel hardware.

In conclusion, the P3 system illustrates some of the general issues that arise in any attempt to simulate PDP models, and provides a number of useful tools that can greatly facilitate model development. General-purpose systems like P3 have promise for speeding and facilitating the programming of parallel models and the ultimate ability to run these models very fast using specialized hardware.

APPENDIX A

```
;;;
;;;          ----------------------------------------------------
;;;
;;;                  P3 Plan for Competitive Learning
;;;   (NOTE the use of the "plan constant" and "include" statements.)
;;;          ----------------------------------------------------
;;;
;;;*****************************************************************
;;;                            Unit types
;;;*****************************************************************

;;;******** Dipole pattern generator ********
(unit type dipole
   parameters flag i1 j1 j1 i2 j2
   outputs (d array i j)
   include dipole-generator) . . . (see code file on p. 506)

;;;******** Learning unit ********
(unit type competitor
   parameters p q flag
   inputs (C array i j terminal parameters W)
      (i-A)
   outputs (o-A)
   include comp-learn) . . . (code on p. 504)

;;;*****************************************************************
;;;                            Unit instances
;;;*****************************************************************

   (plan constant stimulus-size = 6) (plan constant cluster-size = 2)

;;;******** Dipole pattern generator********
(unit stimulus of type dipole
   at (@ 0 0 0)
   outputs(d array (i 0stimulus-size) (j 0stimulus-size) linesat (@ i j 0)))
```

```
;;;******** Learning units ********
(unit cluster array (k 0 (- cluster-size 1)) of type competitor
   at (@ (* 1 (+ cluster-size 4)) (+ cluster-size 10) 0)
   initialize (q = 0.05)
   inputs (C array (i 0 (- stimulus-size 1)) (j 0 stimulus-size)))

;;;*************************************************************************
;;;                              Connections
;;;*************************************************************************

;;;******** Stimulus to both clusters ********
(for (k 0 (+ 1 k))
   exit when (= k cluster-size) do
   (for (i 0 (+ 1 i))
      exit when (= i stimulus-size ) do
      (for (j 0 (+ 1 j))
         exit when (= j stimulus-size) do
            (connect unit stimulus output d i j
               to unit cluster k input C i j
               terminal initialize
               (W = (si:random-in-range
                  0.0 (// 2.0 (expt (+ stimulus-size 1) 2)))))))

;;;******* Interconnect the clusters to implement competition ********
(for (k 0 (+ 1 k))
   exit when (= k cluster-size 1) do
   (for (j 0 (+ 1 j))
      exit when (= j k) do
      (connect unit cluster k output o-A
         to unit cluster j input i-A))
   for (j (+ k 1) (+ 1 j))
      exit when (= j cluster-size 1) do
      (connect unit cluster k output o-A
         to unit cluster j input i-A)))
```

APPENDIX B

```
;;;
;;;          ------------------------------------------------------
;;;
;;;                Competitive Learning:  Methods
;;;
;;;          ------------------------------------------------------
;;;
;;;*************************************************************************
;;;          Method for unit in cluster of competitive learners
;;;*************************************************************************

method
   (let ((imax (input-dimension-n C 1))
      (jmax (input-dimension-n C 2))
      (win t)
   (N 0))

   ;; ******** Is this a learning iteration? ********
   (cond

      ;; ******** No ********
      ;; Accumulate the weighted sum of the pattern inputs into
      ;; unit parameter p, and set the competition output p-A to
      ;; that value
      ((> (read-unit-parameter flag) 0)
         (loop initially (set-unit-parameter p 0)
         for i from 0 below imax do
            (loop for j from 0 below jmax do
               (set-unit-parameter p
                  (+ (read-unit-parameter p)
                     (* read-terminal-parameter (C i j) W)
                     (read-input (C i j))))))
            finally (set-output o-A (read-unit-parameter p)))

   ;;******** Flip the iteration-parity flag ********
   (set-unit-parameter flag 0))
```

```
;;******** Yes ********
;; Figure out whether this unit wins on this cycle. Winning
;; requires that this unit's parameter p be greater than those
;; for the other units of this type. Those values are available
;; on the terminals of input i-A.
;; NOTE: On iteration 0, everything is 0, so no unit thinks it
;; wins, and hence all avoid learning.
(t


;; ******** Find out whether we won ********
;; Win was initialized to t in the let at the top level of this method.
(for-terminals k of input i-A
   (if (<= (read-unit-parameter p)
      (read-input (i-A terminal k)))
   (setq win nil)))
(when win


   ;; ******** Accumulate sum of all inputs into N ********
   ;; This will become a normalizing constant.
      (loop for i from 0 below imax do
         (loop for j from 0 below jmax do
            (setq N (+ N (read-input (C i j))))))


;; ********Compute new weights ********
;; But only if the total input was greater than 0.
(if (> N 0)
   (loop with q-factor = (read-unit-parameter g)
      for i from 0 below imax do
      (loop for j from 0 below jmax do


         ;; ******** Compute one new weight ********
         (let* (old-weight
            (read-terminal-parameter
               (C i j) W))
            (new-weight
               (+ old-weight
                  (* g-factor
                     (- (// (read-input (C i j)) (float N))
                        old-weight)))))
```

```
                ;; Update the terminal parameter to the new weight
                (set-terminal-parameter
                   (C i j ) W
                   new-weight))))))

        ;; ******** Flip the iteration-parity flag ********
        (set-unit-parameter flag 1))))

;;;*****************************************************************
;;;
;;;              Dipole pattern generator method
;;;*****************************************************************

method

;;******** Do we need a new pattern on this iteration? ********
(cond

   ;; ******** Yes.  Erase old dipole and make new one. ********
   ((< (read-unit-parameter flag) 1)
      (let (( imax (- (output-dimension-n d 1) 2))
         ( jmax (- (output-dimension-n d 1) 2)))
      (set-output (d (read-unit-parameter i1) (read-unit-parameter i1)) 0)
      (set-output (d (read-unit-parameter i2) (read-unit-parameter i2)) 0)
      (set-unit-parameter i1 (+ (random imax) 1))
      (set-unit-parameter j1 (+ (random jmax 1))
      (cond ((> (random 2) 0.5
         (cond ((> (random 2 0.5)
            (set-unit-parameter i2 (+ (read-unit-parameter i1) 1)))
            (t
            (set-unit-parameter i2 (- (read-unit-parameter i1 1))))
         (set-unit-parameter i2 (read-unit-parameter j1)))
         (t
         (cond ((> (random 2) 0.5)
            (set-unit-parameter i2 (+ (read-unit-parameter j1) 1)))
            (t
            (set-unit-parameter i2 (- (read-unit-parameter j1) 1))))
         (set-unit-parameter i2 (read-unit-parameter i1))))
   (set-output (d (read-unit-parameter i1) (read-unit-parameter j1)) 1)
   (set-output (d (read-unit-parameter i2) (read-unit-parameter j2)) 1)
   (set-unit-parameter flag 1)))
   (t
   (set-unit-parameter flag 0)))
```