A Neural Network Simulator for the Connection Machine [1]

N Asokan

Ravi Shankar

Chilukuri Mohan

Kishan Mehrotra

Sanjay Ranka [2]

School of Computer and Information Science

Syracuse University, Syracuse, NY 13244-4100

[1]This work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to DARPA and the Air Force Systems Command, Rome Air Development Center (RADC), Grifiss Air Force Base, New York under contract # F 306002-88-C-0031

[2]Author for correspondence. e-mail address: ranka@top.cis.syr.edu

Abstract

In this paper we describe the design, development, and performance of a neural network simulator for the Connection Machine (CM)[3]. The design of the simulator is based on the *Rochester Connectionist Simulator(RCS)*. RCS is a simulator for connectionist networks developed at the University of Rochester. The CM simulator can be used as a stand-alone system or as a high-performance parallel back-end to RCS. In the latter case, once the network has been built by RCS, the high-performance parallel back-end system constructs an equivalent network on the CM processor array and executes it. The CM simulator facilitates the exploitation of the massive parallelism inherent in connectionist networks. It can also enable substantial reduction in the training times of connectionist networks.

[3]Connection Machine is a registered trademark of Thinking Machines Corporation.

1 Introduction

1.1 Connectionist Networks and RCS

Studies of the behavior of animal brains have inspired the use of massively parallel computational models of intelligent behavior, called *connectionist models*. Connectionist models assume that the processing takes place through a large number of simple processing elements, or *units*. Each unit can be connected to some or all of the other units via excitatory or inhibitory weighted links. Such connections, taken together with any external inputs, determine the *level of activation* for each unit. The network of units update their levels of activation, either synchronously or asynchronously, depending on their inputs.

In recent years, there has been a flurry of activity in the area. Consequently, mechanisms for the specification and simulation of such networks have become necessities. Rochester Connectionist Simulator is a tool that provides such mechanisms. RCS is designed to run on sequential machines. It provides a library of functions and a user-interface. The network can be built either by writing a program that calls the appropriate library functions or by issuing relevant commands from the user interface. It can then be executed via the user-interface. A variety of user commands allow the user to examine and set various attributes of the network.

1.2 Motivation and scope of parallelism

The representation of a sizable connectionist network for a typical application like image processing may have thousands of units and tens of thousands of interconnections. On a sequential machine, each iteration in the execution of the network amounts to a complete traversal of the representation graph. Thus the execution is necessarily slow.

However, the operation of the original connectionist model itself is inherently parallel. The sequential operation is forced by the machine on which the simulator is being implemented. Each iteration of execution consists of a large number of updates which can be performed simultaneously and independently. This makes a strong case for performing the execution on a parallel machine which does not impose any "sequentialism".

Some work was done in developing a parallel version of RCS to run on the BBN Butterfly Multiprocessor[1]. In this work, we have attempted to develop a similar simulator for the Connection Machine. The Connection Machine is a parallel computer system with a SIMD architecture, comprising up to 65,536 bit-serial processing elements. Each processor has up to 65,536 bits of local memory. Each group of 16 processors is equipped with a router node which provides highly optimized inter-processor communication.

2 Representation

2.1 Data structures in the serial version

The network paradigm used in RCS is depicted in Figure 1. Each unit has a number of sites where incoming links terminate. Each link originates from a unit. During execution, the link function is applied to the output of the source unit and the result is presented at a site in the destination unit. At each site in a unit, all inputs presented by the incoming links are collected and the associated site function is applied to them. Finally, at each unit, the outputs of its sites are gathered and the associated unit function is applied to them. The result is adapted as the current level of activation/output of the unit.

Units, sites and links are declared as records with the various fields. Units are kept in an array. Each unit has a pointer to a list of sites. Each site, in turn, has a pointer to a list of links. It must be noted that the pointers are uni-directional. Figure 2 shows the representation adapted in the sequential implementation.

2.2 Data structures in the parallel version

In the parallel implementation, each record is assigned to a processor. Therefore, we may imagine that each unit, site and link has processing power of its own. All steps mentioned in the previous section can now be performed in parallel. For instance, all links can now read the outputs of their source units in parallel. Similarly, all links can perform a random-access-write with

an appropriate combining function to propagate their values to the sites. This implementation has the following requirements:

- Each link has a pointer to its source unit.
- Each link(site) has a pointer to its parent site(unit).
- Each link(site) knows the function corresponding to the parent site(unit), so that the appropriate combiner for the random access write can be identified.

Figure 3 illustrates the representation of the paradigm for the parallel implementation. The reversal of the site and link pointers is due to the requirements listed above.

3 Building the network

The simulator provides a library of functions to construct and execute the network. In this section, we describe the parallel versions of the library functions.

Parallel structures of the appropriate type are declared to hold the components(units, sites, links) of the network. A parallel structure has an instance in every processor in the array. Thus, a parallel structure can be viewed as a layer[4] that spans across the processor memories. A particular layer begins at the same address in the local memory of each processor.

[4]This use of the term *layer* is not to be confused with a *level* of units in a neural net

3.1 Making a unit

The function *CMakeUnit* makes a unit for the parallel back-end. The function is prototyped as shown below:

```
CMakeUnit(type, func, ipot, pot, data, out, istate, state)
```

This is fully compatible with the RCS version. Each new unit is assigned an instance of the same parallel structure. When all the instances of a parallel structure are used up, a new parallel structure is created. A front-end list of pointers, called *unit_list*, keeps track of the addresses of the parallel structures used to hold units. *CMakeUnit* returns a unique index for each unit. The address of the parallel structure and the address of the processor can be extracted from this index.

3.2 Adding a site

The function *CAddSite* adds a site to a unit that has been already made. It is prototyped as follows:

```
CAddSite(unit, name, func, data)
```

As before, this is fully compatible with the RCS version. The front-end list of pointers to parallel structures is called *site_list*. In addition, to the parameters passed to the function, the sites need to know the unit-functions of their respective parents. Since, each site has a pointer to its parent-unit, this assignment can be performed by a random-access-read. This is done in parallel at the end of network building phase in the function *Csi_finish_make*.

3.3 Making a link

The function *CMakeLink* makes a link between two units. The function is prototyped as:

```
CMakeLink(from, to, site, weight, data, func)
```

This too is fully compatible with the RCS version. The front-end list of pointers is called *link_list*. Each link needs to know the location and site function of its parent site. This information is extracted using the address of the target unit and the name of the parent site. Note that the name of the parent site is not unique.

The parallel representation of a connectionist network can now be built by writing a program that makes appropriate calls to these functions.

4 Execution

The execution is carried out by calling the function *CStep*. *CStep* takes the number of iterations as argument. The algorithm appears in figure 4. Computations are performed in all selected processors. Initially all processors are selected.

```
for step_ct := 1 to NoSteps do
  Initialize values of all active sites.
  for lc := 1 to NoLinkLayers do
     Select active links in link layer ‘lc’
     for uc := 1 to NoUnitLayers do
          Read output of the source unit into value field

          for fc := 1 to NoLinkFunctions do
              Select links holding link-fn ‘fc’
              Apply link-fn ‘fc’ to value field and
                       set data field to the result
          endfor fc
     endfor uc
     Initialize outputs of all active units.
     for sc := 1 to NoSiteLayers do
         Select active links pointing to active sites in site layer ‘sc’
         for fc := 1 to NoSiteFunctions do
            Write value field to parent-site using site-fn ‘fc’
            as combiner
         endfor fc
    endfor sc
  endfor lc
  Initialize values of all active sites.

  for sc := 1 to NumSiteLayers do
    Select active sites in site layer ‘sc’
      for uc := 1 to NumUnitLayers do
        for fc := 1 to NumUnitFunctions do
          Select active sites with ‘fc’ as parent-unit-fn
             Write value field to parent-unit using unit-fn ‘fc’
             as combiner
        endfor fc
      endfor sc
  endfor lc
endfor step_ct
```

Figure 4. Network Execution Algorithm

In the initialization step, the link data and site values are always initialized to zero. The unit output is initialized to the initial potential *init_potential*. Thus the user could choose between sustained input presentation and momentary input presentation by setting the *init_potential* field accordingly.

5 Interfacing with RCS : *The Bridge*

The parallel functions developed are sufficient to build and execute a simulated network. However, configuring them as a back-end and interfacing with RCS is desirable for several reasons. RCS has a well-developed user-interface and a graphics package. Besides, the Connection Machine is an expensive resource, whereas RCS can run on ordinary sequential machines.

The bridge between RCS and the CM simulator has been realized by adding three commands to the RCS user interface. The command *cdump* dumps the current network representation in RCS onto a set of front-end files. The command *cgo* builds the equivalent network on the Connection Machine, executes the number of iterations specified in the command line and dumps the results back on to the front-end files. The command *cload* reads the information from the front-end files and sets the appropriate fields of the network representation.

The additional commands were inserted into the RCS user-interface by including the appropriate functions into the RCS source code file.

6 Load Balancing

Load balancing is a desirable attribute of parallel programs. In the system described so far, no restrictions have been imposed on the kinds of components (units, links, sites) present in any layer. As a result, during each step of the execution, two kinds of iterations are involved - first, the iteration through layers, and then the iteration through all possible component functions for each layer. Iterating through all the combining functions is unavoidable since the Connection Machine has an SIMD architecture.

We achieve load balancing by grouping components(units, links, sites) with the same parent-component function in a separate layer. By balancing the load in this manner, iteration through the layers is combined with the iteration through functions.

7 Performance

A Hopfield Net[3] was simulated using CM simulator and RCS. The results are summarized in figure 5.

Number of units	*Execution time(seconds)*								
	10 Steps			*100 Steps*			*1000 Steps*		
	RCS	8K	32K	RCS	8K	32K	RCS	8K	32K
50	0.15	0.34	0.34	1.55	3.67	3.44	15.52	34.62	34.50
100	0.65	0.65	0.41	6.47	6.51	4.05	65.10	66.91	40.49
150	1.63	1.11	0.44	16.09	11.36	4.36	160.55	112.09	43.09
200	2.91	2.20	0.78	29.18	24.79	7.77	290.21	217.09	77.67
250	5.39	2.60	0.85	52.95	26.22	8.47	534.89	292.94	84.84
300	7.67	3.68	1.29	75.77	37.37	12.86	755.11	370.70	127.70

Figure 5. Comparison of the performance of RCS with the CM simulator

RCS was run on a VAX-8800 running ULTRIX. The CM simulator was run on the Connection Machine (Model CM-2) at the Northeast Parallel Architectures Center. The CM simulator timings represent timings taken after load balancing, with 8192 and 32768 processors. It must be noted that the VAX-8800 processor (rated at 6 MIPS[4]) is much more powerful than a single bit-serial CM processor.

8 Related Issues

We have striven to ensure compatibility with RCS as far as possible. But at times, this concern did undermine the elegance and efficiency of the parallel functions. For instance, even though a network may have a large number of units and interconnections, they usually follow some underlying coarse-grained pattern. It would be possible to write functions to build several components that belong to the same coarse-grained pattern simultaneously. This direction may be worth pursuing because one of the major bottlenecks of RCS is the time taken to build a large network. It is however necessary to systematically identify the common coarse-grained patterns.

We have so far assumed that the network being built is completely specified. In each step of the execution, the output of the units are computed using random-access-write within each level of the network, and random-access-read between levels. However, when the weights along the links of the network are not available a training procedure like *back-propagation* has to be used. The present Connection Machine representation of the network is quite suited for implementing back-propagation. Back propagation of errors is done using random-access-read within each level of the network, and random-access-write between levels.

9 Conclusion

We have designed and implemented a neural network simulator for the Connection Machine. The performance of the CM simulator was compared with the performance of simulators running on sequential machines. The system designed is expected to be very useful for designing connectionist networks for applications such as speech analysis and image processing, where the number of iterations before convergence is likely to be very high. Once training algorithms are added, the system can be used as a training ground for large networks which can be ported back to sequential machines .

References

[1] Feldman, Jerome A. et. al. 'Computing with Structured Connectionist Networks', *CACM*, February 1988,Vol 31 #2,170-187.

[2] Goddard, Nigel H. et. al. 'Rochester Connectionist Simulator',Technical Report, March 1988.

[3] Lippman, Richard 'An Introduction to Computing with Neural Nets', *IEEE ASSP*, April 1987.

[4] Digital Equipment Corporation, *VAX Systems and Options Catalog*, January 1989.