Technical Report CS/E-88-028 August 1988

ANNE:
ANOTHER NEURAL NETWORK EMULATOR

Casey S. Bahr
Oregon Graduate Center
Dept. of Computer Science & Engineering
Beaverton, Oregon 97006-1999
(503) 690-1151

ANNE:
ANOTHER
NEURAL NETWORK
EMULATOR

August 4, 1988

ABSTRACT

ANNE:
*A*NOTHER *N*EURAL *N*ETWORK *E*MULATOR

Casey S. Bahr
Oregon Graduate Center
August 4, 1988

Supervising Professor: Dan Hammerstrom

ANNE is a neural network simulation system designed for an MIMD distributed-memory computer and implemented on Intel's iPSC. ANNE is one part of a neural network hardware development system that includes a network description language, a standardized network structural format, and a utility to map network connection nodes to physical processor nodes.

ANNE features user-variable, message-driven synchronization between iPSC nodes. This synchronization technique relies on a replication of the simulation clock among the iPSC nodes and the cube host processor. Each node clock runs independently of other node clocks for some number of cycles, before synchronizing with the global host clock. Messages between network connection nodes must cross processor boundaries are packaged according to one of two methods: synchronous packetization (SP) or asynchronous packetization (AP). These messages are timestamped according to the clock of the local physical node, and this timestamp is used to determine a message's "alignment" upon arrival.

Performance results were obtained using several back-propagation networks and small "receptive field" networks. The SP and AP methods are constrasted as well as preliminary results from a direct porting of ANNE to an iPSC/2.

TABLE OF CONTENTS

LIST OF FIGURES iii

LIST OF TABLES iii

LIST OF GRAPHS iv

CHAPTER

1. INTRODUCTION 1
   1.1 Connectionism 1
   1.2 Neural Network Definition 2
   1.3 CAP 2
   1.4 The Target Machine 4
   1.5 Overview 5

2. DESIGN ISSUES 6
   2.1 Development Cycle for Neural Network Simulations 6
   2.2 Design Limitations 8
      2.2.1 Memory 8
      2.2.2 Inter-processor Communication 9
   2.3 Design Assumptions 10
      2.3.1 Network Size 10
      2.3.2 Internal Network Communications 10
      2.3.3 Simulation Accuracy 11
      2.3.4 User's Knowledge of Network Structures 12
   2.4 Timing and Synchronization Techniques 12
      2.4.1 Misra's Algorithm 13
      2.4.2 Time Warp 13
      2.4.3 Why not Misra? 13
      2.4.4 ... and Time Warp? 14
      2.4.5 Message-driven Synchronization 15
   2.5 Software Architecture 17

3. IMPLEMENTATION 20
   3.1 Beaverton Intermediate Form 20
      3.1.1 Objects in BIF 20
      3.1.2 BIF Syntax 22

3.2 Building the Network 22
3.2.1 Loading and Parsing BIF 22
3.2.2 Network Data Structures 23
3.3 Network Emulation 25
3.3.1 User Network Procedure 26
3.3.2 ANNE System Calls 26
3.3.3 Timing, Synchronization, and Checkpointing 28
3.3.3.1 Local Synchronization 28
3.3.3.2 Global Synchronization 30
3.3.4 CN-to-CN Communications 30
3.3.4.1 Output msg Table 31
3.3.4.2 Msg Packetization Methods 32
3.3.4.3 Sending and Receiving msg Packets 32
3.3.4.4 Msg Delay 33
3.3.5 Network I/O 33
3.3.6 User Interface 34
3.3.6.1 User Commands 34
3.3.6.2 Terminal Display 37

4. PERFORMANCE TESTING 38
4.1 Test Networks 38
4.2 Xps Tests 41
4.3 Functional Components Performance 44
4.3.1 Loading and Parsing Performance 47
4.3.2 Other Components 48
4.4 Cpc Tests 52
4.4.1 Cpc Results for the iPSC 56
4.4.2 Cpc Results for the iPSC/2 56
4.5 Network Locality Testing 63

5. CONCLUSION 69
5.1 Summary 69
5.2 Future Work 70

REFERENCES 73

LIST OF FIGURES

2.1 Development Cycle in CAP Simulation System 7
2.2 ANNE's simulation clocking model 16
2.3 Conceptual model of the timestamp window 17
2.4 Simulator components 18
3.1 Conceptual structure of a CN and its sub-components 21
3.2 Structure of a single CN after parsing 23
3.3 Example of Local CN Table in an HN 24
3.4 Portion of a user's network procedure 27
3.5 Cycle_params structure 29
3.6 Msg table at HN 0 31
4.1 Back-propagation network mapped to two HNs 39
4.2 Receptive field network 40

LIST OF TABLES

4.1 Xps tests 41
4.2 Msg packet sizes for n16 44

LIST OF GRAPHS

4.1 iPSC xps performance: SP method 42
4.2 Xps vs. number of external connections per HN 43
4.3 iPSC xps performance: AP method 45
4.4 iPSC/2 xps performance: SP method 46
4.5 iPSC/2 xps performance: AP method 47
4.6 Functional component performance on the iPSC: SP method 49
4.7 Functional component performance on the iPSC: AP method 50
4.8 Functional component performance on the iPSC/2: SP method 51
4.9 Functional component performance on the iPSC/2: AP method 52
4.10 Cpc results on the iPSC: SP method 54
4.11 Seconds elapsed for cpc tests on the iPSC: SP method 55
4.12 Cpc results on the iPSC: AP method 57
4.13 Seconds elapsed for cpc tests on the iPSC: AP method 58
4.14 Cpc results on the iPSC/2: SP method 59
4.15 Seconds elapsed for cpc tests on the iPSC/2: SP method 60
4.16 Cpc results on the iPSC/2: AP method 61
4.17 Seconds elapsed for cpc tests on the iPSC/2: AP method 62
4.18 Speedup vs. locality on the iPSC: SP method 64
4.19 Processor efficiency vs. locality on the iPSC: SP method 65
4.20 Speedup vs. locality on the iPSC: AP method 66
4.21 Processor efficiency vs. locality on the iPSC: AP method 67
4.22 Speedup vs. locality on the iPSC/2 68

CHAPTER 1

Introduction

1.1. Connectionism

Recently the interest in massively parallel computational models, commonly referred to as connectionist models or neural networks, has increased tremendously. This interest is due to a number of reasons, not the least of which are these networks' abilities to efficiently compute a variety of problems considered to be resource intensive for conventional serial computation. Connectionist models that feature a distributed data (knowledge) representation also promise a high degree of fault tolerance even in the face of faulty structural components, or "noisy" input data. This promise does not mean that neural network models can enjoy universal application in all areas of computation. Rather, connectionist models may be viewed as complementary to von Neumann computational models. They often exhibit optimal performance when solving problems for which serial computation is most inefficient, but they are unsuitable for the many problems that require precise, deterministic solutions at which conventional digital computation excels.

Examples of the best applications for neural networks may be categorized into two classes [LaF86]. The first class consists of difficult constraint satisfaction problems, such as The Traveling Salesman [HoT]. The second class of problems relates to content addressable memories; i.e., being able to extract stored information by association rather than by explicitly naming the information's location. Many connectionist models extend this latter property to the point of automatically producing generalizations of the knowledge explicitly stored in the network [MRH86a].

In both classes of problems the function performed by these networks is that of finding a "best match" solution. The neural network approach differs from conventional computational paradigms, where exact results are obtained through a series of highly specified steps. The goal of connectionist models is not necessarily to obtain *the* single, correct answer to a problem, but rather to obtain *an* answer among several that is "close" (within some error acceptable to the application). To accomplish this goal, connectionist models use a process of satisfying multiple, simultaneous, and mutual constraints by relaxation of the network toward some attractive point in a problem's solution space. This relaxation is often measured in terms of a global network energy value [Hop82]. This process is implemented through a highly parallel interaction among a large number of computationally simple units.

The connectionist approach has been inspired by observations of the brain's ability to perform such complex tasks as vision, hearing, and motor control with units (neurons) that are orders of magnitude slower (milliseconds) than the silicon devices in today's digital computers. Conversely, some researchers hope that the study of connectionist models will shed light on some of the workings of the human

brain. However, the inspiration for research more logically flows from neuro-science towards connectionist theory, since current network models have a complexity far less than that possessed by even elementary structures in the human brain. Many people feel uncomfortable with the term "neural network" when describing these models, preferring instead to call them connectionist networks. Nonetheless, the term "neural network" persists in the field and in turn is applied throughout this paper. However, it is by no means implied that these network models actually resemble the brain in their workings, except at a gross level. Terms such as neuron, axon, or synapse are therefore avoided in describing the components of these network models.

1.2. Neural Network Definition

Since the main purpose of this thesis is to describe an implementation of a neural network emulator, a detailed description and analysis of the large variety of neural network models and their algorithms is not given. It is assumed that the reader already has a working knowledge of these models.[1]

This paper adopts a common definition of neural network models that use a distributed representation of their memory. This definition says that such a model consists of a network of computationally simple units that interact with one another by exchanging data over weighted connections (links). A standard requirement is that the information exchanged between these units is minimal and non-symbolic. The information in a network is distributed among the units (or connection nodes) in the network, most notably in the weights associated with each unit's links. A network "learns" by the application of a learning rule that makes adjustments to these weights. In addition to a learning function, each unit normally possesses a few simple functions, which might include an activation function, a threshold function, or an error function.

1.3. CAP

There are a large number of network models and learning algorithms that fit this definition. Research of these models relies on simulations running on conventional computers. Due to the nature of the neural computational paradigm (distributed state and "program"), serial computer architectures are unsuited for expressing the full potential of neural networks. Neural network research needs hardware that matches the massively parallel characteristics of these networks.

Some neural network designs have been cast in silicon. Researchers at AT&T Bell Labs have implemented, for instance, a Hopfield chip containing 512 nodes. Other chips have been produced for the areas of vision and speech processing [Bro86a][Bro86b][MeM88]. These chips, however, are limited in their practical application due to their small size and network-specific design. Few have faced the problems of large-scale implementation of neural networks in silicon. Scaling problems

[1] For the reader not familiar with neural networks an excellent introduction is provided by Rumelhart and McClelland [RuG86].

are due mainly to the high connectivity required between connection nodes in many neural networks [BaH88]. A wafer-scale integrated (WSI) chip architecture, specifically designed to emulate a variety of connectionist designs, would provide an engine to explore the possibilities of neural networks. Such an implementation is the aim of the *C*ognitive *A*rchitecture *P*roject (CAP) at OGC.

Before designing such an architecture, the CAP group must depend on neural network simulation, using conventional computer technology, in order to discover which of many neural network models are best suited to our prototype applications. The group's approach to the necessity of simulation is to create "in house" multipurpose software rather than individual network simulations. Ad hoc simulations offer little as a consistent measure of each model's effectiveness.

The CAP group is developing software for an integrated WSI neural network architecture development system, including a high-level *N*etwork *D*escription *L*anguage (NDL), and three simulators. The *H*ierarchical *A*rchitecture *S*imulator (HAS) is aimed at specifically modelling the group's proposed architectural features for a silicon neural network emulator [Jag88]. ANNE (*A*nother *N*eural *N*etwork *E*mulator), is designed as a general-purpose neural network emulator with a bias towards networks with localized communications. The third simulator, ***Fltsim*** (***F***au***lt*** ***Sim***ulator), introduces stochastically determined faults into the network models run by both HAS and ANNE [May88a][May88b]. Each simulator utilizes an intermediate network specification produced by NDL.

The CAP group has a bias toward networks that exhibit high *locality* of communications between connection nodes.[2] However, our first applications use models that are non-localized, such as back-propagation [Hin87]. Such models are best handled by techniques similar to full matrix computation, whereas non-localized models are efficiently simulated with sparse matrix techniques. Current neural network hardware generally takes the full matrix approach, utilizing digitial signal processing chips [Wor88]. These "neurocomputers" can be considered first generation hardware solutions to neural network simulation.

Our group is developing a second generation approach. Our prototype hardware design is specialized for fast, virtual connection node processing. In general, though, this design embodies full matrix techniques for network emulation. Long-range plans call for the development of third and fourth generation neural hardware that more closely models biological neural networks, in particular cortex [LyB86][HSA84][Lin88]. Sparse matrix techniques work well for such networks due to their lower, more localized, connectivity characteristics.

Due to our bias toward localized networks, we have chosen to represent networks as collections of connection node objects, each with its local segment of an entire network's connection matrix. This representation is the basis for our intermediate network specifications and the internal network representations for HAS

---

[2] Intuitively, high locality refers to networks whose connection nodes communicate almost exclusively with neighboring or near-neighboring connection nodes. Networks in which connection nodes communicate mainly with distant nodes exhibit low locality. For a formal definition of locality see [Ham86].

and ANNE. Thus, ANNE utilizes something akin to sparse matrix techniques in its treatment of neural network connection matrices.

1.4. The Target Machine

The fine-grained parallelism inherent in neural networks, make ANNE and HAS well-suited for implementation on an MIMD computer. Furthermore, our network representations influence the type of MIMD machine to chose. The choice of machines was between a loosely-coupled machine such as the iPSC® and a tightly-coupled one such as one of Sequent's Balance® or Symmetry® Series[3]. The choice of the iPSC as the implementation vehicle was influenced by several factors, not the least of which is its geographical proximity. The latter consideration aside, there are issues of simulation scaleability, ease of use, machine cost, and future extensibility.

With Intel's and Sequent's machines the latter issue is moot. Both companies continue to design their computers to be upward-compatible. Shared memory systems have the advantage in terms of ease of use, although this advantage will be minimized with the advent of high speed concurrent I/O systems for distributed systems. Distributed systems have the advantage in cost per processor. The main issue, of scaleability, depends on the amount of core memrory available, the inter-processor communications medium, the expected locality of communications within the neural networks being simulated, and the nature of the network representation.

In general, distributed memory systems are able to provide more core memory than shared memory systems. Furthermore, distributed memory implies increased system memory bandwidth. This increased bandwidth is most often obtained over multiple, serial channels, which are slower than the high bandwidth busses normally found in shared memory systems. However, in a shared memory environment connection updates would pass twice over a single bus. Moreover, neural network computation is characterized by frequent memory references, which exhibit poor data reference locality. In such a situation, a shared memory system suffers from bus and data lock contention. This situation is aided little by high-speed caching [Wor88].

The performance of distributed memory systems also suffers if the memory references in a neural network simulation must frequently cross processor boundaries. However, a good partitioning of highly localized networks over a distributed memory machine minimizes inter-processor memory access. No such partitioning will improve the performance of a shared memory system for localized networks. Thus, a distributed memory machine is favored, which updates connections simultaneously across a neural network with reduced message passing between processors. Although not dealt with in this thesis, sequential and shared memory parallel versions of ANNE are planned. They should make for interesting comparisons.

---

[3] iPSC is a registered trademark of Intel Corporation. Balance and Symmetry are registered trademarks of Sequent Computer Systems, Inc.

1.5. Overview

The objective of this research is the design and implementation of a general-purpose neural network simulator, ANNE, as part of an extensive WSI development system for neural networks. The following two chapters cover the design and implementation details of ANNE, including a look at the user interface and a brief description of the development cycle for ANNE's neural network simulations. Chapter 4 presents the results of validation tests and their analysis. The final chapter discusses future work including improvements and extensions to ANNE as well as discussion of ANNE's most serious limitations and the direction a next generation simulator might take.

CHAPTER 2

Design Issues

ANNE offers the designer of experimental neural networks a virtual machine on top of a distributed memory multiprocessor computer. ANNE provides a pre-built framework for network loading and initialization, and pre-defined communication and synchronization facilities between connection nodes in a network. ANNE also provides a debugging interface for observing and manipulating network data and network runtime parameters.

HAS simulates neural networks via a virtual architectural model. ANNE's purpose is to shake out the major bugs in a neural network design before moving it to HAS for specialized performance testing. Both simulators must solve problems concerning data distribution in the iPSC nodes, updating this data, and synchronizing neural network nodes in an asynchronous environment. Both ANNE and HAS have adopted similar approaches in their solutions to these problems [BHJ88].

Each simulator uses its own version of one basic method of synchronization between neural network nodes. This method allows the user to alter the "tightness" of synchronization between individual node processors. Thus, the user directly affects the degree to which individual hypercube nodes can operate independently before explicit synchronization with one another. This synchronization method has the potential for speeding up simulator performance.

2.1. Development Cycle for Neural Network Simulations

A brief description of how a particular neural network design is implemented using the CAP development system is presented to illustrate the environment in which ANNE operates. Seeing the "big picture" helps make later descriptions of ANNE's limitations and design assumptions more meaningful. The description that follows focuses on ANNE, but the steps taken using HAS are similar. A pictorial view of this development cycle is shown in figure 2.1.

The development cycle for a neural network design in the CAP system requires the following steps:

(1) A high-level description of the connectivity of a neural network is developed using CAP's network specification utility, NDL (pronounced "noodle") [Joh88a][Joh88b].

(2) NDL is compiled into an intermediate structural form known as BIF (*B*eaverton *I*ntermediate *F*orm).[1] BIF describes a network's state and connectivity.

[1] As part of the next chapter an overview is given of the objects represented in BIF. BIF's technical specification is forthcoming in [Bah88].

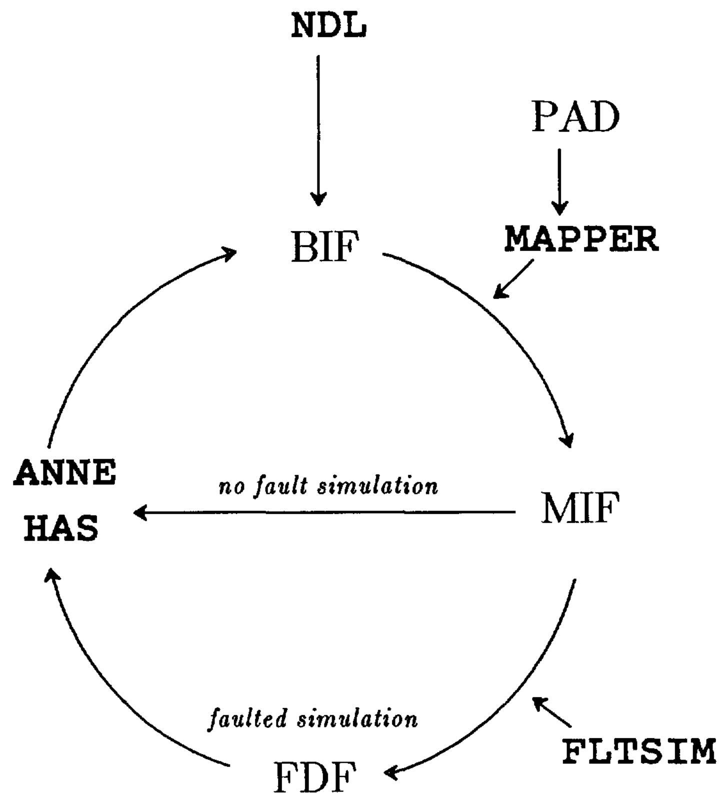


Figure 2.1: Development Cycle in CAP Simulation System

(3) Connection nodes described in a BIF file are augmented with physical processor numbers by a BIF mapping utility, *Mapper* [Bai88]. Mapper assigns connection nodes to physical processors. It uses a *P*hysical *A*rchitecture *D*escription (PAD) file in attempting near optimal connection node mappings for a target machine (for ANNE and HAS, the iPSC).

(4) The mapped BIF (or MIF) is read by either ANNE or HAS to construct an internal representation of the network. Fltsim uses MIF as input to produce a *F*ault *D*escription *F*ile (FDF), which is referenced by Fltsim routines at simulation runtime.

In addition to specifying the structural characteristics of the neural network using NDL, the user writes two segments of code specifying the behavioral characteristics of the network. One segment, known as the *network procedure*, resides in the iPSC nodes. This code describes one complete cycle of network operation, i.e., it contains the network node and link functions and the "script" for interactions between connection nodes. The second code segment, the *convergence procedure*,

resides in the iPSC host processor and determines, globally, if the user's network has converged. This procedure is activated whenever the network sends external network output to the host. The user's host code might determine convergence, for example, based on a comparision between a network's output vector and a predetermined target vector. The user code is independent of the particular configuration of the iPSC chosen for mapping the BIF file. Once written, the user need not change her network code to suit different dimensions of the hypercube; she need only re-map the original BIF file.

When ANNE is invoked, the user directs the simulator to parse a BIF file. After network building is complete the user specifies the input and output vector files (and possibly a target vector file if supervised learning is to take place), and certain simulation parameters. When ANNE is running, the user's code guides the communications between connection nodes and sequences local network cycles at each iPSC processor. During a network simulation the user may suspend the network and access fields in the network's structure. The user can also direct ANNE to save the current state of the network in a new file in BIF format.

2.2. Design Limitations

ANNE's current capabilities are, in part, bounded by two major limitations of the iPSC hardware at OGC. These limitations are memory space and the time to route and deliver messages between hypercube processor nodes. A third factor limiting ANNE's performance is related to the task of performing accurate, timed simulations in a distributed, non-shared memory, processing environment. This last point is especially important, since even if the limitations of memory and message time were relaxed there still remains the difficulty of distributed synchronization between hypercube processors. A synchronization scheme that sequentializes the interaction between hypercube nodes to any large degree defeats the purpose of implementing the simulator as several parallel processes.

These three factors alone might seem to preclude the use of the iPSC for ANNE without a substantial performance penalty. The remainder of this section discusses the first two limiting factors of memory space and message passing time. A brief discussion concerning how ANNE synchronizes events (communication) between connection nodes within neural networks is deferred to the third section in this chapter. Section 2.3 also presents the qualifying assumptions and adaptations made in ANNE's design to accomodate these limitations.

2.2.1. Memory

For a fixed percentage of interconnect in a given network, the number of connections (or links) grows geometrically in proportion to the number of connection nodes (CNs). For neural networks represented in BIF, the number of links in the network is a proportional approximation of the network's memory requirements. Due to the tendency of neural networks to be memory-intensive, space to store network structures is at a premium. Also, loading large networks into hypercube node (HN) memory is a potential bottleneck in the initialization phase of simulations, since the the HNs have no direct file I/O capabilities. The network structure must pass from the host processor to the HNs via a single message channel. (To avoid confusion

between nodes in a user's network and the nodes of the iPSC, connection nodes are henceforth called CNs, while the hypercube nodes are referred to as HNs.)

Once the network structure is in place, there is another serious memory limitation at run time. Buffer space is necessary in which to store the messages passed between CNs. It is possible during a single cycle of a network to generate a CN message (output) for every link in the network. The iPSC system pre-allocates buffer space for HN messages. This space is partly utilized, by ANNE, as an input buffer. Additional buffers are allocated, by ANNE, for sorting CN output messages prior to their placement into the HN buffers.

As the number of links grows, so does the demand on ANNE's and the HN's buffers. Depending on the specific network configuration, and the communication patterns between CNs that pass messages across HN boundaries, these demands vary widely, even within a single network during the course of a single simulation. Thus, the number of links in a network has a dual impact on the available space to store and simulate networks.

One aspect of the neural network computational strategy is advantageous in terms of memory requirements. Due to the relatively simple CN functions that are normally used, the space needed to store user object code is minimal. This assumption is true for the network models most likely to be emulated by ANNE. These models' object code consists of small functions shared by a large number of the CNs in the network. This assumption goes hand in hand with the concept of distributed representations of knowledge within neural networks [MRH86b].

2.2.2. Inter-processor Communication

The time to generate and pass iPSC messages between HNs is the greatest intrinsic limitation on ANNE's performance. The iPSC has inter-HN message latency times of $\sim$ 2 ms for a 1K buffer, the largest discrete data unit passed between HNs [Int85]. The latency period increases linearly up to the maximum buffer size of 16KB.[2] The time to generate and deliver a single CN message by ANNE is significantly less. Thus, the time required for message passing is often an upper bound on ANNE's overall performance.

For larger networks, efficient use of the message passing facilities of the iPSC becomes more important. In the work described here, the solution sought to compensate for HN message delays has been to minimize the number of HN messages as much as possible. Communication is not considered to be a severe constraint, since ANNE is intended for the simulation of networks with highly localized communication patterns. Network communication requirements are discussed shortly.

[2] Future hypercubes are improving this constraint. Intel Scientific's newest machine, the iPSC/2, has message latency times that range from $\sim$ 2 ms for a 1K buffer to $\sim$ 7 ms for 16KB. Moreover, HN messages are virtually unlimited in size [Nug88].

2.3. Design Assumptions

This section and its sub-sections cover fundamental assumptions about ANNE's design. Three of these assumptions concern: (1) the size of the user's networks, (2) the networks' communications requirements, and (3) the expected behavior of distributed neural networks. This third assumption is crucial to ANNE's CN synchronization scheme.

Finally, there is an assumption about the programming skills of an ANNE user that simplifies the interface between ANNE's communication and synchronization facilities and the user's network code.

2.3.1. Network Size

Given the present configuration of OGC's hypercube (32 iAPX 286-based nodes), it was assumed that ANNE have the ability to simulate networks containing a quarter of a million links. This number might translate, for instance, into a network of one thousand CNs with a CN-to-CN interconnect density of 25% (meaning that every CN is connected on average to one quarter of the other CNs in the network). Throughout the design of ANNE a network of this configuration has been considered the worst case. This size network is not overwhelming by present neurocomputer standards, but it is enough to provide an adequate assessment of ANNE's design before porting ANNE to another computer with more memory resources, such as Intel's newest hypercube, the iPSC/2®[3].

Little more was done to improve network storage space in ANNE than had already gone into the design of BIF, the network connectivity specification parsed by ANNE into internal data structures. For specific networks BIF could be more streamlined. The tradeoff in the BIF model is between a complex syntax allowing "tight" specifications of individual networks, and a simpler, more general format. The latter was chosen and this philosophy is carried through to ANNE's network representations.

Related to the choice of a generalized BIF is the issue of the user's mental model of network design. In part, BIF's structure (the available fields and the field layout) is designed in a user-friendly and straightforward manner, since the user has full access to the network data model. A more efficient encoding of BIF in ANNE would be at the expense of requiring an interface for the user between the BIF model and ANNE's internal representation of that model.

2.3.2. Internal Network Communications

Concerns about network size aside, an assumption was made concerning the overall communication requirements of neural networks. It is assumed that the models ANNE simulates are homogeneous in their inter-HN communication patterns. The homogeneity of the inter-HN communications for a given network simulation depends on both the ability of the BIF mapping utility to find an optimal mapping of the CNs onto the HNs, and the dynamics of the user's network at runtime.

[3] iPSC/2 is a trademark of Intel Corporation

Furthermore, a network's *locality* strongly affects the amount of HN-to-HN communications required by a network. Locality is inversely related to inter-HN communications requirements in that a connection node with high locality has less (if any) need to communicate with connection nodes on another physical processor.

Neither ANNE nor Mapper can be relied on to make up for network configurations that have intrinsically lopsided inter-HN communications. Fortunately, most network models do not exhibit such behavior. Moreover, the CAP group's future interests lie mainly with high locality network models, so that ANNE's limited abilities to simulate networks with poor locality are not of great concern now.

The bottom line on overcoming performance limitations imposed by inter-HN messages is to reduce the number of these messages. The BIF mapper does this indirectly to some degree by trying to eliminate, statically, extra-HN communication as much as possible. ANNE's design attempts to reduce the number of HN-to-HN messages directly by exploiting the fact that CN messages need only carry minimal information. Therefore, iPSC messages are capable of holding a number of these smaller messages. Up to 63 inter-CN messages are packed into a single 1K byte iPSC message. Another way to minimize message traffic between HNs is to use multiple copies of all CN structures [Pla87]. This technique requires more memory, and a great deal of complexity to maintain coherency between CN copies. Such a scheme also makes the CN-to-HN mapping task more difficult. This scheme is not used here, but is discussed in the final chapter under future work.

Network models based on a localized representation of knowledge in the individual CNs could be simulated by ANNE as long as such a model conformed to the need for balanced communications between physical processors. However, these types of network models often require the exchange of symbolic information between CNs, and thus might find ANNE's built-in premise of non-symbolic CN messages cumbersome. Such models would suffer in performance, and be implemented more efficiently as stand-alone programs.

2.3.3. Simulation Accuracy

The architecture of the hypercube is best suited for problems consisting of independent processes for each node, or interdependent processes requiring a minimum of inter-HN communication. A distributed simulation of neural networks is likely to have inter-HN communication, despite the assumptions discussed in the previous section, thus requiring HN synchronization. It is costly to tightly synchronize HNs in order to produce a completely accurate simulation.

An *accurate* simulation is defined here as a simulation that produces results identical with those that could be obtained using a sequential process. Other researchers have proposed solutions to the problem of producing accurate simulations in the context of a distributed, non-shared memory system. Why those solutions were found unsuitable are covered in more detail shortly.

A less stringent method for synchronization between the CNs in ANNE takes advantage of the demonstrated robustness of most neural network models. The performance of these models degrades gracefully in the face of incomplete or noisy

inputs, missing links, or faulty CNs. This advantage means that the simulations can tolerate a degree of error in the strict sequencing of events and still produce a *valid* simulation. That is, the results of the simulation are valid if the network produces answers within some error value of the answers produced by an accurate simulation and that these answers are reproducible, compared again to the accurate results $\pm$ the same error value. The error value is determined by the user.

ANNE's lenient view towards the accuracy of its simulations permits greater asynchrony between network CNs and consequently the HNs. This higher degree of asynchronous operation averages out differences between the time taken by HNs to process their local network cycles, and leads to a simpler design of the entire synchronization scheme.

Given the three assumptions just outlined, it can be seen that the limitations of the 286 hypercube are not overwhelming. The issues discussed so far are less significant when considering a port of ANNE to Intel's iPSC/2, which has more memory and improved message passing capabilities. In the meantime, small networks, such as that used to demonstrate NetTalk [SeR], are adequate for yielding useful results.

2.3.4. User's Knowledge of Network Structures

An ANNE user has full read and write access to all data fields represented in any BIF network specification. BIF's design assumes users are familiar with neural network models in terms of CNs and links. This familiarity leads to a natural understanding of BIF, and consequently the structure of ANNE's networks. Furthermore, it is assumed that the user has enough programming experience, not necessarily in 'C', to be undaunted by complex access paths to network data. Data access is assisted through numerous macros that provide short-hand specification of individual data fields. Full access to ANNE's network representation minimizes the user interface to this data and provides the user maximum flexibility in network manipulation.

The user is able to approach a network algorithm's design in a top-down fashion aided by several ANNE system calls provided specifically for use in the network code. An earlier design alternative was to have the user write routines that were applied to single CNs. Each CN was to have a set of functions bound to it and these would be called in order. This method leads the user to tackle the network algorithm bottom-up, CN by CN. Instead, ANNE's model encourages the user to to write network routines that apply to groups of one or more CNs. An example of a user's network algorithm is presented in the chapter on ANNE's implementation.

2.4. Timing and Synchronization Techniques

Two alternatives for the design of ANNE's timing and synchronization methods were considered. Both methods are used for event-driven, distributed simulations. The first, by Jayadev Misra [Mis86], does without a shared global clock or shared event list. Instead, simulation time is kept by timestamping messages used to communicate between simulation objects and by following a certain messsage exchange protocol. The second method, Time Warp [JeS82], also uses timestamped

messages and, in addition, features a method of rolling back the simulation when out-of-order messages are encountered. The implementation of both methods depends on sophisticated distibuted synchronization techniques.

2.4.1. Misra's Algorithm

In Misra's approach, each object being simulated has a separate input message queue for each object from which it receives messages. Within each input queue messages are ordered by their timestamps. The message at the head of the queue carries the most recent time. To ensure that messages from all objects are processed in timed order, Misra's algorithm requires that all objects produce some message at each tick of an object's local clock. If an object is not scheduled to produce any message at a particular time step, it must produce a *null message*, which requires no action, but carries that object's current clock value.

The production of null messages serves two purposes. First, it allows the receiving object to determine which sending object has produced the most current event. Second, it avoids deadlocks of the "deadly embrace" type in which two or more simulation objects are waiting on each other to produce output, but each must receive input from the others.

2.4.2. Time Warp

Jefferson and Sowizral's method for distributed synchronization, Time Warp, differs from Misra's in that objects are restricted to a single input queue into which timestamped messages are deposited without time-ordering them. Also, unlike Misra's method, an object can block on an empty input queue.

Time Warp uses a *rollback* of events to restore correctness of a simulation should a message appear out-of-order in an object's input queue. Rollback requires both the saving of past state for objects and the issuance of *anti-messages* to cancel the effects of out-of-order messages. Anti-messages may in turn require the rollback of other objects' states and so on. The authors contend that the time saved in processing incoming events without the overhead of ordering these events offsets the cost of this rollback procedure. This method relies on the hope that a simulation is more or less well-behaved and that the ordering of messages is of high precision as a simulation progresses. Thus, Time Warp is sensitive to the probabilities of receiving out-of-order messages, making its performance application dependent.

2.4.3. Why not Misra?

If each CN's output is considered to be an event it is easy to see that a large number of messages are generated, in a network where the CNs have a large fan-out. For the worst-case network, if all the CNs fired simultaneously, 250,000 messages are created throughout the system. These messages almost fill the available HN message buffer space over the iPSC.

Assume that some CNs delay firing so that 50% fire at each step in a simulation. This is not an uncommon situtation in many neural network designs. Misra's algorithm still requires that a quarter of a million messages be generated, though half of them are null. If a uniform dispersal of these messages is assumed throughout

a 5 dimensional configuration of the iPSC (32 processors), these extra 125,000 messages take roughly an extra 100 milliseconds to pass through the cube's communication channels per simulation cycle. This calculation does not take into account channel contention nor the processing time for each message upon its arrival. This additional time is more significant if the messages are queued at the input and ordered by timestamp as Misra's algorithm requires.

Not only are twice as many messages generated at each simulation step using Misra's algorithm, but, if a good mapping of the network over the iPSC has been obtained, the CNs most likely to communicate non-essential information (null messages) are furthest away. In other words, CNs are waiting the longest time for meaningless messages for the sake of strictly ordering all events in the network. In a hypercube of dimension 5, a particular HN's nearest 16 neighbors (including itself) are an average of about 1.7 cube links from each other, whereas the furthest 16 HNs are twice this distance. Few neural networks would map to the hypercube network so as to require CNs to communicate heavily with their nearest neighbors and still have links with the furthest, but this is essentially the effect of Misra's algorithm as it applies to neural network simulation.

The semantics of a null message, by Misra's definition, are not clear when applied to the high connectivity and parallellism of most neural nets. Misra's approach is designed for situations in which only a single message among an object's input queues is used to initiate events in that object. The messages arriving at a CN are usually coincident and interdependent. They are not events to be chosen individually and processed one at a time. Often they are summed, or operated on as a whole in some other way, to provide input for the entire CN. In this context, single messages do not have a great deal of importance, so the sending of null messages is not cost-effective, especially if there are a large number of them.

2.4.4. ... and Time Warp?

Time Warp seems to offer a more reasonable model for neural network simulations with its single input queues and absence of null messages. Despite its promise, a large number of coincident messages pose an even greater problem than in Misra's technique. The authors claim that coincident messages can be handled, though they do not describe exactly how. It is assumed that the coincident messages are used in their spatial order, which seems simple enough. But it is difficult to see how Time Warp might efficiently handle the occurrence of a single out-of-order message in a queue of several messages having an earlier time. The number of regenerated messages and anti-messages that would be required in the case of the rollback of even portions of a neural network is formidable.

Time Warp could be adapted for our purposes by defining "out-of-order" to mean for a message what it means in ANNE (allowing messages to fall within a time range). Aside from this, however, the requirements of saving past state and messages is untenable in a system in which the knowledge of the network is held in hundreds of thousands of links, not just the state of relatively few CNs. Even saving a single past state, which Jefferson and Sowizral say is least efficient, because it requires more anti-messages to be sent at rollback, requires almost as much storage as the original network structure.

2.4.5. Message-driven Synchronization

Both Misra's technique and Time Warp are designed for general-purpose large-grain simulations. These algorithms must be concerned with the strict requirements of an accurate, event-driven simulation. Both methods require substantial runtime overhead to generate and manage the events in neural network simulation with potentially thousands of messages at each clock cycle. This overhead is costly in terms of both memory space and processor time. For neural network simulation this extra overhead is not cost-effective. The large number of coincident messages (messages with the same timestamp and destination) also raises concerns about the workability of either algorithm. Furthermore, Misra's algorithm and Time Warp appear to require algorithms and data structures too complex to implement in a reasonable amount of time.

ANNE utilizes a timing and sychronization scheme that is essentially *message-driven*, though at a higher level it is also clock-driven. This scheme is tailored for localized, fault-tolerant networks. It is not offered as a general-purpose simulation technique. Our scheme yields minimal message overhead, faster simulations than possible with the alternative methods just described, and no possibility of deadlock. This technique lets the user tune synchronization parameters to suit the particular network at hand. It does all this through simple methods requiring little extra overhead in terms of time or memory.

ANNE pays for its lack of complexity in its synchronization scheme by not always being able to emulate networks in a completely correct manner, but compensates for this by allowing the user, with a little experimentation, to tune the simulations to her satisfaction for accuracy and performance.

The basic premise that makes this method possible is that CNs are able to produce output even if new inputs have not arrived or have arrived out of order. From the beginning of the simulation until the end there is an input value available at each CN's input links, whether it is a previously used value or a new one. This condition removes the possibility of deadlock among the CNs, since no CN ever blocks indefinitely, waiting for input messages from another CN. Thus, it is no longer necessary to consider sending null messages, since a CN continues to operate without checking for the earliest message along its links.

Under this synchronization scheme, simulations are run using two closely coupled clocks, one global clock in the host processor, and a local clock that is replicated in each HN. One clock step equals one cycle of the network being simulated. At the host level the clock value refers to the global state of the network, whereas the value of an HN clock refers to the state of its local portion of the network. These two clocks use the same metric, but are incremented in different ways.

All HN clocks are set at a *global synchronization point*; when HN clocks synchronize with the host clock. The time between global synchronization points is the *synchronization interval.* The host clock is incremented by the value of the synchronization interval. HN clocks are incremented by one for each cycle of their local portion of the network. Between global synchronization points each HN is free to run on its local clock without regard to the local time on other HNs. Global synchronization points are determined by the user. See figure 2.2.

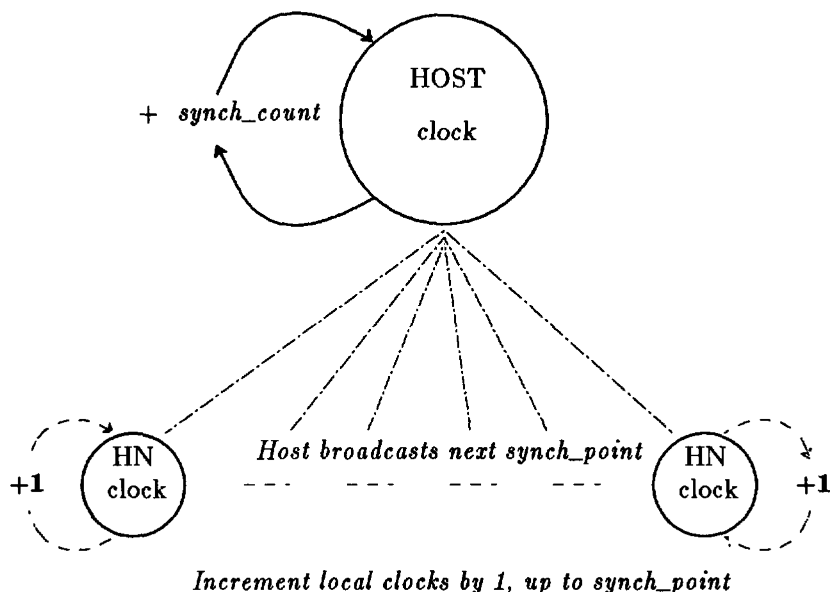


Figure 2.2: ANNE's simulation clocking model

From cycle to cycle HNs may differ in the speed at which they complete their local simulation cycles. This speed variance is HN *clock drift*. Assuming Mapper has done its job, the amount of drift should be minimal. Varying the time between global synchronization points varies the average clock drift among processors. In general, longer times between global synchronization points diminish the effect of clock drift. The user can collect statistics to get a feel for the amount of clock drift for a particular network and set of simulation parameters.

The user's control over the number of local clock cycles that HNs complete between global synchronization points is advantageous during the learning phase of many network models. Often during learning, network input vectors are clamped for many (thousands) of network cycles before the network converges [HSA84]. The user can vary the HN clock drift based on observations of the convergence behavior.

Messages sent between CNs are timestamped with their HN's clock value. The use of timestamps in ordering messages at a receiving CN is not absolute. Instead, for arriving CN messages a timestamp *window* is defined that "slides" along with the HN clock as shown in figure 2.3. This window determines the "alignment" of an arriving input message. If the message does not fit in the window, the CN uses the old value on that link, and the current message is discarded.

In each HN process, message generation and message reception are interleaved, in order to avoid HN message buffer overflow and to balance communications

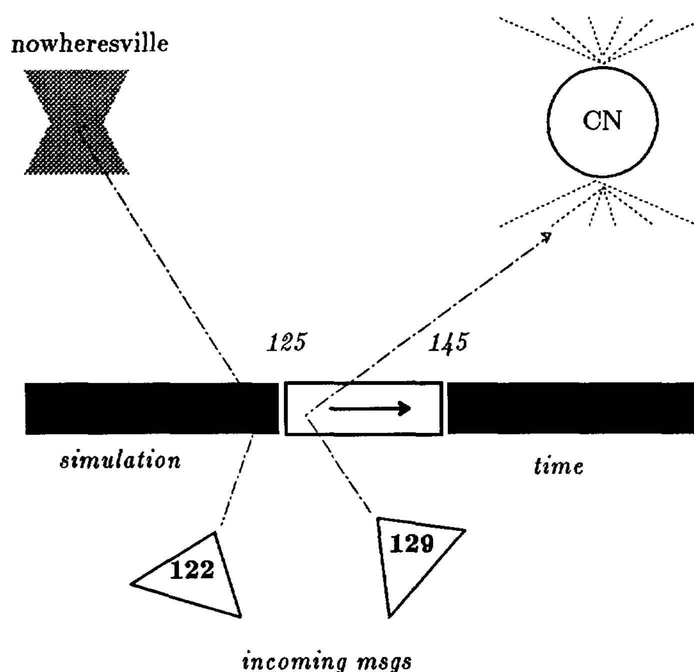


Figure 2.3: Conceptual model of the timestamp window used in ANNE

among HNs. After generating all the output messages for a single CN during one clock cycle, the HN process enters a message reception interval. The length of this interval is called the message probe *timeout.* If any incoming messages are detected during the message reception interval, the timeout is reset and those messages that have arrived are processed. Afterwards, continued message reception is driven by the arrival of more messages, otherwise the message reception interval expires, and new CN outputs are generated. This message-driven behavior tends to make the entire network self-synchronizing at the HN level. This synchronization technique is described in more detail in Chapter 3.

2.5. Software Architecture

Before going into the details of ANNE's implementation, a brief outline of its organization is presented. ANNE has three major functional components. These are the *user interface*, the *network builder*, and the *network runtime system* or *network control.* Only the user interface resides solely in the host of the iPSC, while the latter two components each have one sub-component in the host and in the HNs. The user interface interacts with both the network builder and network control, though only minimally with the builder. How these three components fit together is

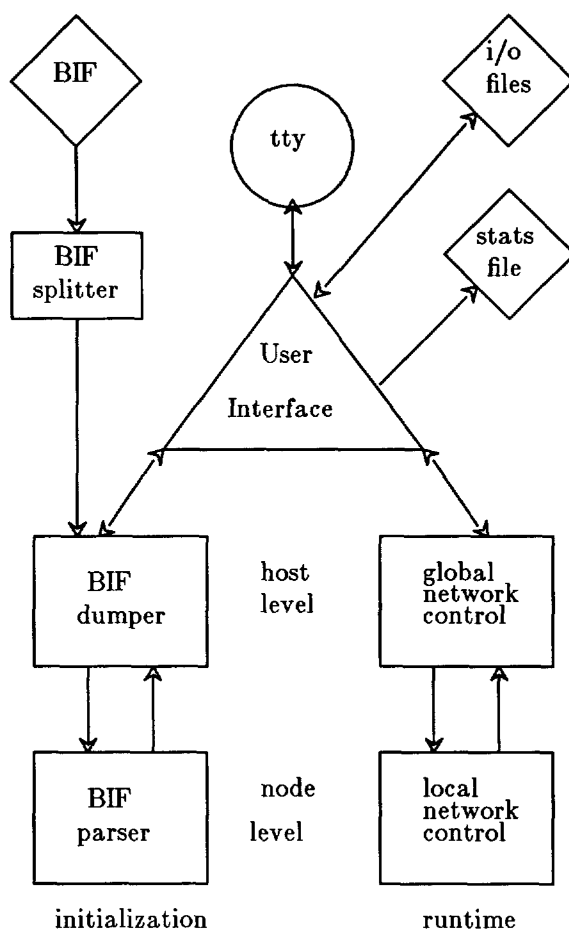


Figure 2.4: Simulator components

displayed in figure 2.4.

The network builder's (both host and node components) task is straightforward. This part of ANNE has as input the BIF description of the network to be simulated. An auxiliary utility, outside of ANNE, post-processes the original BIF file into multiple subfiles to be distributed to the HNs. These subfiles are parsed in parallel by the HNs into the internal data structures representing the user's network. The network builder also constructs auxiliary data structures used at network runtime.

After the network structure is loaded into the HNs, the simulator enters its runtime mode when the network control component of ANNE takes over. Its sub-

components in the host and the HNs work together to run the network and provide the user with access to the network structure. These sub-parts constitute the nuts and bolts of the network runtime system.

At the host level, network control is responsible for global synchronization of the HNs and global network I/O operations. It is the go-between for the user interface and network control in the HNs, functioning synchronously with each in turn. At runtime, network control delivers to the network an input vector and possibly a target vector. An output vector, representing the network's global results, is collected by the host in segments from each HN and passed to the user's convergence procedure. When the network is suspended (after network initialization, or at a breakpoint), network control in the host passes to the HNs requests to read network data and assign data to network fields. It also delivers new simulation parameters, including a new clock value.

In the HNs, network control, in addition to interacting with the host, runs the network according to the user's network procedure. This procedure guides inter-CN communications and performs local network computations.

The main tasks of the user interface are providing control over simulation parameters, accessing network data, and displaying these data and simulation state. Besides the three simulation parameters mentioned earlier, the user has control over the inter-HN message packet size and simulation breakpoints. The user interface allows multiple runs with a single network loading, and the specification of new input and output vector files.

The following chapter discusses how ANNE's functional components have been implemented, including the ANNE system calls available to the user to write the network procedure. It also introduces the simulator's command set.

CHAPTER 3

Implementation

First in this chapter, a brief description of BIF is given to assist in understanding ANNE's internal data structures and algorithms, which are described later. Then there is a detailed discussion of ANNE's two major modes of operation: 1) that of constructing the network and auxiliary data structures, and 2) executing the constructed neural network model. Access to and control over both modes, via the user interface, is also covered.

3.1. Beaverton Intermediate Form

The inspiration for BIF's original format grew from the network representation used for the ISCON simulator developed at University of Rochester [SSB83][Fan86]. BIF has since evolved through several revisions as the CAP group has customized it to fit their needs.

3.1.1. Objects in BIF

A BIF file represents the connectivity specification of a neural network. This file contains the "raw" structure of the user's network in terms of three basic objects: CNs, sites, and links. Functions associated with each of these objects are not specified within the BIF file. The association of CNs (or groups of CNs) with particular functions occurs within the user's network procedure by direct manipulation of the network data structures that are derived from the BIF specification.

A BIF file has two parts. The first contains a listing of the *CNgroups*, each of which consists of a unique group index, a string name, and two initialization values for CNs belonging to the group. Each CN carries an index corresponding to the group to which the CN belongs. The group name allows the user to address groups of CNs symbolically. At present there are only two initialization fields in the CN group specifications, vestiges from early BIF versions. These fields can be used to re-initialize the *state* and *potential* fields in CNs. Re-initialization fields such as these are used for a "reset" of the network at runtime. The second part of a BIF file consists of individual CN records. These records are composed of a hierarchy of CNs, sites, and links. Sites nest within CNs, and links nest within sites (see figure 3.1). Input or output sites are not listed in any particular order.

Essentially, a CN contains state information that is global to itself and its sub-parts. Some of this information is directly related to the characteristics of the network model, such as a CN's *output*, *potential*, *error*, etc. Other fields are pertinent to a simulator's internal data structures. These include fields indicating on which physical processor a CN resides, and to which group of CNs a CN belongs.

The primary function of a site is to group the links entering into or emanating from a CN. Sites are specified in BIF as being either input or output, although this

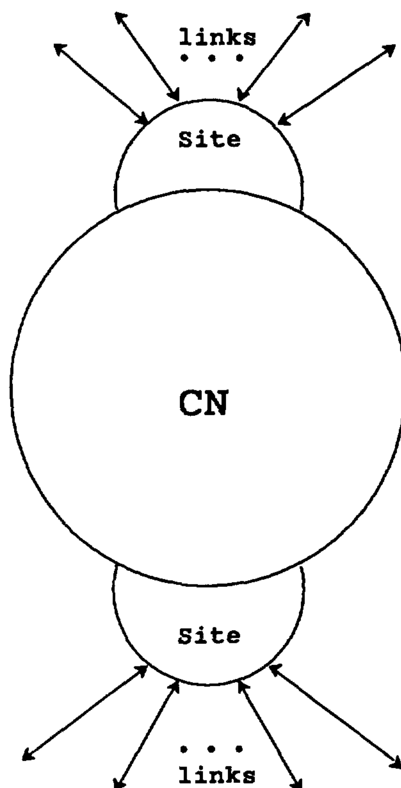


Figure 3.1: Conceptual structure of a CN and its sub-components.

designation is ignored in ANNE, where sites are bidirectional. In a typical case for an input site (or bidirectional site being used for input) a site's link values are summed together and the resulting intermediate value is used in internal CN calculations. An output site may be used to broadcast the CN's output value (if any) along the links attached to the output site.

In BIF, each link specification within a single site is the specification of one of two ends of a single connection between two CNs. Each end contains the address (index) of the CN, site, and link to which it connects at its other end. All links have a *weight* that can be used to modify values passing along the link. In other words, at each of two CNs, which are connected to one another, there is complementary address information and identical weight information in their links. This bidirectional nature of the links greatly facilitates the specification of networks such as back-propagation and Hopfield. It is also utilized during weight updates to keep weights consistent at both ends of a link. The weights are sent as messages along the link in this update technique.

3.1.2. BIF Syntax

Six reserved words plus four bit flags are used to facilitate BIF parsing. There is one bit vector per CN, site, and link. Three reserved words are used to delimit the beginning and end of the CN groups part and each group within this part (**sgbk**, **egbk**, and **egrp**). In a similar manner, the other three reserved words mark the start and end of the CN record section (**scbk**, **ecbk**) with a word to end each CN (**endc**). In each site and link specification there is a bit that tells the parser when it is reading the last site or link in the current sub-block. Two additional bits indicate which of certain optional fields are present in CN and link records.

The use of a small number of reserved words and a few bits to aid parsing is due to a desire to keep the parsers simple (and quick), and in the interests of compact files. Moderately large networks could easily require BIF files of several megabytes. It may seem contradictory that BIF files in current use are ascii. Early BIF files were binary, but have since been made human readable, while the group is engaged in developing the simulation environment. Given our current BIF, these files, and the parsers, can easily be moved to a binary format later.

3.2. Building the Network

Constructing the network data structures takes three basic steps. The first is the loading of the BIF network specification into the HNs. Next, the HNs parse the BIF specification into local network data structures to hold the CNs, their sites and links. Finally, auxiliary data structures are built in both the host and HNs to aid in executing the network procedure.

3.2.1. Loading and Parsing BIF

Reading a BIF specification was originally done exclusively in the host. The ascii data was converted to binary and shipped one CN at a time to the HNs. Loading networks of $\simeq$ 2,000 links was noticeably slow with this method and it was easy to extrapolate that the load time for large networks would be hours. A scheme of parallel parsing was devised that improved the network loading time on average by a factor of 10.

In this scheme, an auxiliary utility splits an already mapped BIF file into as many subfiles, plus one, as there are HNs to be utilized. Each subfile holds the CN records to reside on a particular HN. In other words, the CN blocks are sorted into separate files according to their *procid* fields, which contain the HN processor number given to the CN by Mapper. Each subfile is designated by the name of the original BIF file appended with an HN number. One additional file carries the CN group information augmented by the CN indices belonging to each group and a table showing to which HN each CN is assigned. This latter information eliminates an additional scan of the BIF file by the host process at BIF load time.

The host sends each BIF subfile to the HN corresponding to the subfile's name suffix. The HNs use the same parsing mechanism as used in the old host parser, adapted to read from a buffer instead of a file. An entire subfile can be dumped into each HN despite the 16K byte limit for HN messages. HN message buffering is done automatically by the operating system into a large (100K+ bytes) space. This buffer

space can be increased to approximately 200K bytes when loading the HN portion of ANNE's object code. No buffer management currently exists for BIF subfiles larger than 200K bytes, but the amount of space now available is adequate for the test networks used in this thesis.

3.2.2. Network Data Structures

As each HN reads its BIF subfile, it builds its local portion of the neural network one CN at a time. An entire CN with sites and links is placed into a temporary list structure before allocating contiguous arrays to hold the CN's sub-parts. First to be read are a CN's "global" fields. These are assigned to the fields of a single structure. In addition to these CN fields, there is a field for the number of total sites belonging to this CN and a pointer to the CN's site list. As each site is encountered in the CN specification, a site structure is allocated to hold the site fields read in from the BIF file. This structure also contains a field for the number of links belonging to the site, a pointer to the next site structure, if any, and a pointer to the

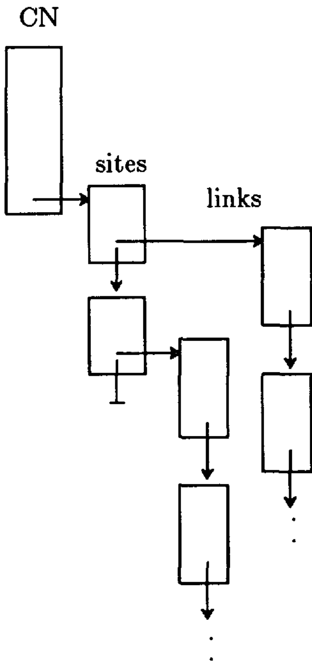


Figure 3.2: Structure of a single CN after parsing its BIF specification and before assigning it to the local CN table.

site's links list. Each link is then read in and a linked list of links constructed for the site. Once an entire CN structure has been constructed (see figure 3.2), site and link arrays of the correct size are allocated and the structures in the site and link lists are transferred to these arrays. The original list structure is then deallocated.

The final CN structure, site, and link arrays are entered into a local table, as seen in figure 3.3. The CN table at each HN is pre-allocated to fixed length. Local CN records are placed in the table according to their unique index in the network. This scheme eliminates one level of indirection in accessing the CN fields at the cost of some wasted space for non-local CN table entries. Site and link structures were also placed into arrays to reduce the time in accessing their fields.

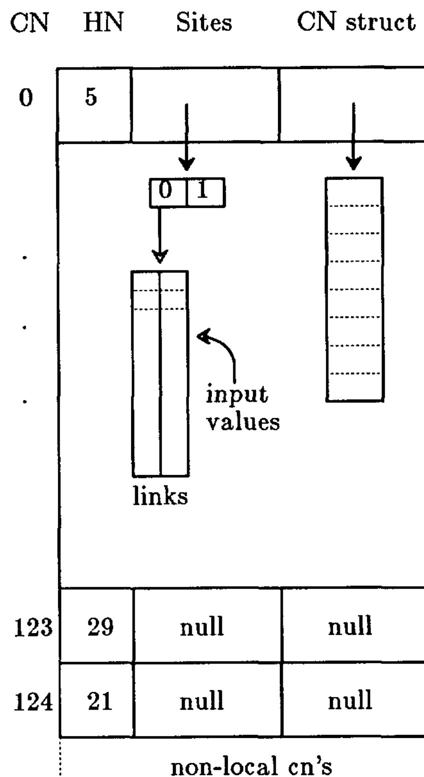


Figure 3.3: Example of Local CN Table in an HN, showing CN 0 mapped to HN 5

The first entry in a CN table indicates on which HN a CN resides in the system. There is an entry for each CN in the entire network. These entries make up a map used to direct CN output to the appropriate HN. The next entry for a CN is a pointer to the list of sites for that CN. Through this pointer and a site's links pointer, ANNE gains access to the link fields. Last, but not least, the final CN table entry is a pointer to the CN structure. For a non-local CN table entry, the site and CN pointers are null.

As an example of how a link's weight field would be accessed, ANNE uses the index of the CN to which a link is attached to gain access to the proper CN table entry. Then the site index of the link is used to select the correct link array: `CNtbl[`*i*`].sites[`*j*`].links[`*k*`].weight`, in 'C' vernacular. There is no need to search for the correct link structure, it is addressed directly. It takes several index computations to reach the links array, but thereafter all the links can be accessed quickly, for instance when updating all of a CN's weights.

At this point an estimate can be made of the size of a CN table. This estimate assumes a roughly even distribution of 1,000 CNs over the hypercube, so that there are about 33 CNs per HN. The distribution will vary depending on the network and how the BIF file was partitioned by Mapper. Pointers are four bytes in length. HN fields are 1 byte. There are 1,000 table entries, though 967 of these are non-local entries. Thus, at this point the table has consumed 1,000 bytes of HN numbers plus 8,000 bytes of table pointers.

Each CN structure present in the table consumes a minimum of 24 bytes. Depending on field alignment, slightly more bytes are used. With padding, each CN might require, roughly, 30 bytes. Again, as in the worst-case network proposed in the previous chapter, this hypothetical network has an interconnection density of 25%. However, there are twice this many link terminations, since link specifications are shared between CNs, giving a total of 500 link structures for each CN. Each link structure consumes 14 bytes, and with padding about 16 bytes. It is assumed there are 2 sites per CN, used for input and output, and that these take a total of 12 bytes per CN.

There are 999 bytes of CN global information, 360 bytes of sites, and 264,000 (8000 x 33) bytes of link information at each HN, for a subtotal of 265,359 bytes of useful information. In addition there are 9,000 bytes in HN indices and null table entries. In total then, the table requires roughly 274K bytes.

Note that despite the large number of empty table entries (97%), they only comprise 3% of the total space occupied in this example. The reduced indirection in accessing the fields in a local entry easily offsets this small cost. The other major consumer of memory is the HN object code (ANNE plus the user's network procedure), but this code takes less than 40K bytes. With 391K of usable memory space in each HN (not counting message buffer space) there is adequate memory space. There is even room to expand the iPSC message buffer space if needed.

3.3. Network Emulation

ANNE's "network engine" is a runtime framework for the user's network code. This framework includes routines to set the simulation time clocks, and to

synchronize HNs and CNs. It also includes system calls to effect CN-to-CN communications and produce global network output.

3.3.1. User Network Procedure

A network's structure is specified in a BIF file through use of the NDL compiler. The user specifies network operation by writing two 'C' procedures to be loaded with ANNE's object code. The first of these is the user's *network procedure.* This code is loaded with ANNE's HN image. The second program is the *convergence procedure,* which ANNE's host code accesses when determining global network convergence. The convergence procedure is called at global synchronization points, when the HNs send output results from their local sub-networks to the host. This procedure has access to the network's global output vector and possibly a target vector (the desired output vector).

The network procedure has two parts. The first is an initialization procedure that ANNE calls once during a single simulation run. This procedure, *Init_user_fx(),* allows the user to set up local variables or structures that are used during subsequent operation of the network. The second part of the network procedure is a function *User_fx().* It describes a single complete cycle of the user's network. This procedure is the script for ANNE to follow in delivering input to the network, propagating that input through the network, and finally producing the network's global output. To aid in writing the network procedure, ANNE supplies procedural calls to effect communication between network connection nodes as well as external network I/O.

The user must implement the network procedure using a "go-to" paradigm. In other words, to effect communications between CNs, or from CNs to the host, the user's code makes calls to *send* CN output, but no system calls are provided that explicitly handle functions for either local CN or global network *input.* Input operations are handled internally by ANNE, transparent to the user. The user must treat CN message reception as being immediate.

In actual practice ANNE does not guarantee that the input values for any CN are the "latest" values, but does guarantee that some value exists at every input link. Depending on the particular simulation, this practice might exhibit differences in actual CN output values for different runs with the same input data. How these differences effect the *validity* of the simulation is dependent on the fault-tolerance of the network and the application.

3.3.2. ANNE System Calls

ANNE provides a small number of system calls for the user's network procedure. One set of calls are necessary to access ANNE's CN communication mechanisms, and to perform link weight updates, possibly across HN boundaries. Another set of system calls are used at the convenience of the user. Included in this latter set are standard site functions and 'C' macros to simplify access to network data structures.

Many of the system calls in the first set of procedures address entire groups of CNs by their CN group name. Therefore, there is a procedure for returning a list of

```
void Init_user_fx() {
    /* get the different CN groups */
    cns_in  = Get_cnlist("input");
    cns_hid = Get_cnlist("hidden");
    cns_out = Get_cnlist("output");
}
void User_fx() {
    int cnx;

    /* get global input, pass to next layer */
    input_site_fx();
    Send_group_output("input",TOP);

    /* sum weighted inputs at hidden layer  */
    other_site_fx(cns_hid,BOTTOM);
    /* pass input through "squashing" func. */
    squash(cns_hid,BOTTOM);
    /*  pass squashed value to output site  */
    assign_to_outsite(cns_hid,TOP);
    /*    send this output to next layer    */
    Send_group_output("hidden",TOP);

    /* sum weighted inputs at output layer, */
    /*     continue as for hidden layer     */
    other_site_fx(cns_out,BOTTOM);
    squash(cns_out,BOTTOM);
    assign_to_outsite(cns_out,TOP);
    /*  send output layer's output to host  */
    Send_net_output();
}
```

Figure 3.4: Portion of a user's network procedure for a feed-forward network with three layers. Procedure calls beginning with an upper-case letter and in **this font** *are ANNE system calls, those in lower-case are written by the user.*

CN indices when passed a CN group's name. This call is useful for individual operations on CNs within a single group. Calls that operate on groups of CNs are designed to encourage the user to make a functional partitioning of the network procedure according to the CN groups listed in the network's BIF specification.

Here is a partial listing of the ANNE system calls provided for the user's *User_fx()* procedure. Figure 3.4 shows a simple example of such a procedure.

`Send_node_output(cn_index,site_index);`

Sends the output from a single CN along all the output links belonging to the named site.

`Send_group_output(group_name,site_index);`

Performs the same operation as `Send_node_output()` for a named group of CNs.

`Send_net_output(filename);`

Used specifically for global network output to the host process. The output vector is written by the host to the named file.

`Update_node_weights(cn_index,site_index);`

If the user has set the link weights on a particular node and the link is used bidirectionally the user makes this call to transmit the new weights to the other end of the links. The CN and site indices in the parameter list name the group of links that transmit their weights.

`Update_group_weights(group_name,site_index);`

Performs the same function as `Update_node_weights()` for the named group of CNs.

`cnlist * Get_cnlist(group_name);`

Returns a structure that holds both a list of the local CN indices belonging to the group name and the number of CNs in the list.

From within the network procedure the user has full access to CN structures, including sites and links. Addressing these structures in 'C' closely follows BIF structures. The user may perform any operation on the data that she wishes, though the user must match fields' data types and take care not to corrupt local data structures.

3.3.3. Timing, Synchronization, and Checkpointing

There are two simulation clocks and two levels of synchronization during a simulation. CNs synchronize to a local clock on each HN and the HNs synchronize to the global clock of the host. At a global synchronization point the two clocks have equal values. This occurrence is known as a *synch_point.*

A "run" is the presentation of a set of input vectors from a single input file. Before and during a single network run the user may manipulate several parameters governing a simulation's behavior. These parameters are included in a single structure that is passed between the host and the HNs at *synch_points.* This structure is presented in figure 3.5, and its fields discussed shortly.

3.3.3.1. Local Synchronization

Whenever a new input vector is presented to a network the global and local clocks are set to zero. Both the host and HNs are at their first *synch_point.* The

```
struct cycle_params {
     short global_clock,   /* host clock value          */
     local_clock,          /* HN clock value            */
     msg_window,           /* CN message window         */
     synch_count,          /* # of HN clocks to run     */
     synch_point,          /* next global synch point   */
     checkpoint,           /* next suspension point     */
     mintime,              /* oldest msg to accept      */
     maxtime,              /* newest msg to accept      */
     maxmsgnum,            /* # msgs per iPSC message   */
     timeout;              /* # probes for input msgs   */
     char outfile[NAMELEN];/* net output file name      */
};
```

Figure 3.5: Cycle_params structure containing simulation parameters.

next *synch_point* is determined by the simulation parameter *synch_count*. *Synch_count* is equivalent to the synchronization interval. Once a network is activated, each HN simulates its portion of the network asynchronously for *synch_point* cycles, incrementing its local clock value by one after each call to the user's network procedure. The host clock is incremented by *synch_count*, thus being preset to the next global synchronization point.

Messages passed between CNs carry the time of their HN clock. When a CN receives a message from another CN, the receiving CN compares the message's timestamp to its own HN clock. The difference (if any) between the received message's timestamp and the local time is the "alignment" of the message. How this alignment is intrepreted depends on another user-settable parameter, *msg_window*. *Msg_window* tells CNs what message timestamps are acceptable. If the absolute alignment of an incoming message is greater than *msg_window* the message is discarded.[1] Otherwise, the value passed in the message is stored in the destination link. In the structure *cycle_params*, *mintime* and *maxtime* define the boundaries of the message window derived from *msg_window*. CN messages bound for a CN residing on the same HN are the exception to the method of CN communication just described. These messages bypass this mechanism and are delivered directly to the local CN.

[1] One variation would be to only apply *msg_window* to messages in the "past", and to buffer "future" messages.

CN message generation and reception alternate within an HN during network simulation. *Timeout* determines the number of calls made to the iPSC system call *probe()*, which is used by ANNE to detect new incoming CN messages. After each call to *probe()*, *timeout* is decremented. Upon the detection of CN input, a message is processed, *timeout* is restored to its original value, and message reception continues. If no messages are received, the HN returns to the message generation interval.

3.3.3.2. Global Synchronization

At a *synch_point*, final probe is made for CN input messages. CNs in each HN continue to process incoming messages that are within *msg_window* of their local clock, but cannot produce additional CN-to-CN outputs. HNs signal the host upon reaching a global synchronization point by sending their portion of the global network output. The global network output received from a network is sorted by CN index by the host. The user's convergence procedure uses this output to determine if the network has converged. To avoid interference due to the asynchronous operation of the HNs, a network is restricted to sending global output only at global synchronization points.

After receiving all the network output, several things may happen at the host level. If the network has converged, then a new input vector, and new target vector, if any, is delivered to the network before the HNs resume. If the network did not converge, a new *synch_point* is calculated, equal to the last *synch_point* plus *synch_count*, and network simulation continues.

It might happen also, that the network has reached a *checkpoint* set previously by the user, or that the user has issued a "stop" command since the network last reached a synchronization point. In either case, the simulation is suspended. Suspension of the network occurs, too, if the set of input vectors are exhausted, in which case the user specifies a new (or previous) input file for additional network runs.

3.3.4. CN-to-CN Communications

Messages between CNs not resident on the same HN must be passed using the iPSC message passing facilities. Most of the time used by the iPSC to send an inter-HN message is taken in message set-up. Therefore, it is advantageous to pack as much information as possible into a single iPSC message. Other than an address, necessary to the delivery of a message between CNs, the information exchanged between CNs is a single value. Therefore, these messages are small. To reduce message traffic between HNs, CN outputs are packaged as sub-messages that are part of one HN message. These CN-to-CN sub-messages are called *msgs* here to distinquish them from HN messages.

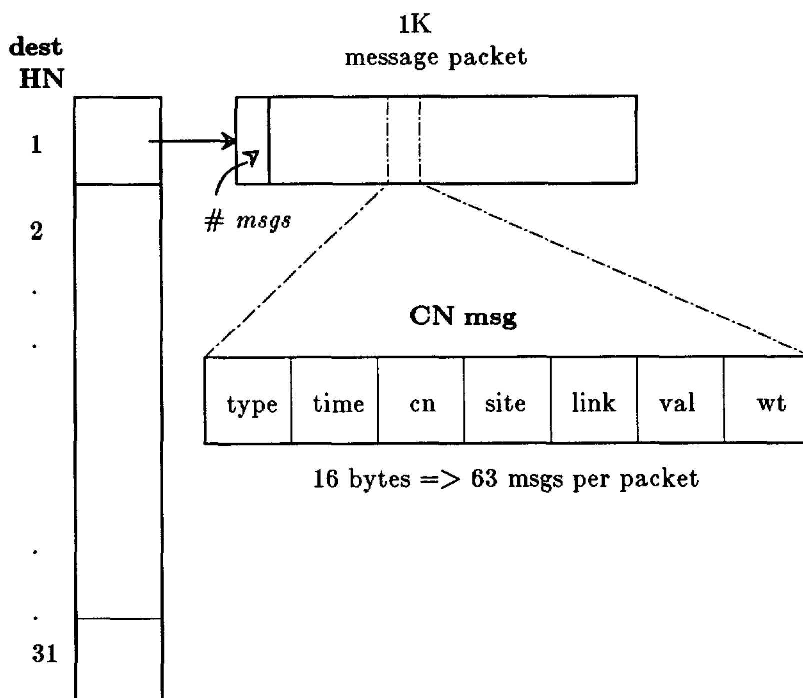
Each *msg* is 16 bytes in length. A *msg* includes seven fields: the *msg* type, a timestamp, a destination CN index, a destination site index, a destination link index, an "output" value, and a "weight" value. What either the output or weight fields in *msgs* actually contain is determined by the user. For example, the output field might be a CN's error value, and the weight field might be transmitting a weight change value instead of an absolute weight value.

Except for the weight value, which is single precision floating point, all fields are two byte integers. The type field determines which of the latter two fields is being used in the *msg*, since the same CN-to-CN message mechanism is used for passing output values and link weights. Which HN a message is bound for is determined by the global CN-to-HN map incorporated into an HN's local CN table. Where the value in the message is finally delivered is determined directly by the destination indices in the *msg*.

3.3.4.1. Output msg Table

At each HN there exists an output *msg* table (see figure 3.6), each entry of which points to a 1K *msg* packet that is shipped as a single HN message. Each packet contains a count of the number of *msgs* it currently contains. Sixty-three *msgs* are packed into a 1K iPSC message.

There are as many entries in the output *msg* table as there are nodes in the hypercube, less one. So, for instance, there are 31 table entries for a 5d hypercube. A thirty-second entry is not necessary, since local *msgs* are delivered directly,



3.6: Msg table at HN 0

bypassing the iPSC message mechanism. The table is indexed directly by the HN index found in the local CN table corresponding to the CN that is to receive the *msg*. At most, the entire output structure consumes about 31K of data space at any one time since *msg* packets are shipped when, or before, they reach the maximum length set by the user. The packet length is stored in the *cycle_params* field *maxmsgnum*.

The time to pack and unpack *msgs*, to maintain the output message table, and the time HNs must wait for the first *msg* to arrive, is offset by fewer HN messages being sent and stored in the system. For a full *msg* packet, HN message time is reduced by over an order of magnitude. The user can determine empirically an optimal size for the *msg* packets. The best size may depend on the network structure, activity, and the iPSC configuration used. The default *maxmsgnum* setting is for full packets.

3.3.4.2. Msg Packetization Methods

Besides setting the maximum size of *msg* packets, ANNE's implementation gives the user two major options controlling the behavior of *msg* packaging. The difference between these options has to do with the relationship between the local simulation time and the timestamps on the *msgs* at the originating HN. In the first method, *synchronous packetization* or *SP*, all *msg* packets are sent to their destination HN during a single clock cycle. With SP, any remaining *msg* packets are flushed from the output message table at the end of each network cycle, regardless of their current size. Thus, all the *msgs* contained in each packet carry the same timestamp.

In contrast, *asynchronous packetization* or *AP*, is designed to obtain maximum utilization of *msg* packet space. The AP method only ships *msg* packets when they have reached the maximum capacity set by the user in *maxmsgnum*. No packet flushing occurs at the end of each network cycle. *Msgs* with different timestamps may be found within the same packet. AP, in general, will improve the speed of simulations at the cost of increased asynchrony in CN-to-CN communications. For the feed-forward mode of most neural network models, the SP method is most appropriate.

3.3.4.3. Sending and Receiving msg Packets

The sending of output *msgs* is interleaved with receiving *msgs* from other HNs. Whenever a single message packet is sent, the HN's message space is probed for packets meant for the HN's local CNs. If a packet is present, then it is processed before continuing to send another output packet. Thus, the system message space is utilized partially as ANNE's input message structure. Furthermore, by alternately sending and receiving packets an overflow of the system message space is avoided.

It is unfortunate that there is no way on the iPSC to allow ANNE to efficiently perform the interleaving of CN message output and reception by the use of separate logical processes. While it is certainly possible to create separate logical processes on each HN, there are no inter-process software signal facilities other than

the iPSC message mechanism itself.[2] Nor do facilities exist for processes to share data segments, which might at least allow a crude technique for inter-process signalling. Thus, send and receive interleaving has been accomplished by multiple calls to a CN *msg* receive sub-routine at strategic places in the node level code.

During one local clock cycle a CN may receive multiple *msgs* along a single link, depending on the setting of *msg_window*. The exception to this exists for the specialized "output" CNs, which send the global network output to the host. Not only can they not send more than a single output per local network cycle, but such output is forbidden, and probably unnecessary, except at a point of global synchronization.

3.3.4.4. Msg Delay

Some network models utilize the concept of delays in CN output. Delay in ANNE is simulated by not producing *msgs* from that CN if the CN's delay field has a value greater than zero. The delay field is set by a user's delay function. If a CN's delay is greater than zero, ANNE decrements the field at the end of each local cycle. Whether or not a CN calculates new output values while a delay is in effect is up to that CN's output function.

The use of output delay is important, for instance, in temporally-dependent models such as Hopfield's, where random CN output generation is employed [Hop82]. Using ANNE, a random delay could be associated with each CN locally and a complete cycle of the network spread over as many simulation cycles as the value of the greatest delay. This technique ensures that all the CNs fire during each network meta-cycle. The global synchronization points would be distanced accordingly.

3.3.5. Network I/O

Since the hypercube nodes have no direct I/O connections with the outside world, vector input and output is handled through the host. The host provides the network with input vectors and maintains a record of its output, which is done via Xenix files. An ANNE system call (`Send_net_output()`) is used to send output vectors to the host and then to a file. This file output call is formatted as a request to the host from the nodes. An HN sends a typed message containing a list of CN indices and corresponding output values. The host sorts the HN output values by CN index and writes them to the filename passed to `Send_net_output()`. Input vectors are delivered to the network automatically, provided that the network converged and more input vectors are available. In the user's input file, the user is responsible for matching the number of values in each vector to the number of CNs belonging to the "input" CN group. The number of elements in the output vector need not concern the user once it has been specified in BIF.

Input and output vectors are read and written by ANNE in increasing order of CN indices. Let **I** stand for an input vector and **Ii** stand for the set of indices belonging to those CNs that are used as a network's global input. Then, for example, if the input CNs have indices **Ii** = {3,11,17,20,21,22}, these CNs are assigned

[2] The ability to set up handler routines based on message types on the iPSC/2 corrects this shortcoming.

input values from vector **I** as follows: CN[3].in = **I**[0], CN[11].in = **I**[1], CN[17].in = **I**[2], and so on. A global output vector available to the user's convergence procedure is assigned from the set of output CNs in a similar fashion.

3.3.6. User Interface

ANNE's user interface is command-driven and ANNE has three distinct modes with three corresponding command line interpreters. The fact that there are three interpreters is "translucent" to the user. That is, even though it is transparent that commands are being interpreted by three routines, it is readily apparent that ANNE enters distinct "states" during a simulation run. These states are represented by network initialization, network suspension, and the network execution.

When a simulation begins there are two commands the user can give, either to build the network, load and initialize, or to quit. Once the network is built, the network is in a suspended state, as it is at a *checkpoint* during runtime. When the network is suspended, the user may reset runtime parameters, such as *msg_window*, *synch_count*, or *checkpoint.* The user must re-specify a new checkpoint if it does not conform to the current *synch_count.* The user might also examine, modify, save the current state of the network, or quit the simulation. In order to start the simulation, one command is used to specify certain "global" parameters (such as the name of the input vector file). Another command prompts for "local" simulation parameters (such as the next *checkpoint*) before starting the network. When the network is running the user can stop the network at the next global synchronization point regardless of the setting of *checkpoint.*

The most significant synchronization parameters set at runtime are *synch_count*, *msg_window*, *timeout*, and *maxmsgnum.* The ability to set these parameters aids the user in tuning simulations for optimum performance.

For the highest speed the user sets both *synch_count* and *maxmsgnum* high, and *timeout* low (e.g., zero). To gain more accurate results, as compared to a strict sequence of events, the synchronization interval and message wait time are set conversely. *Msg_window* has minor effect on the raw speed of the simulations. Its effect, together with the size of *msg* packets, is more noticeable in the convergence results of the networks.

The upper limit on *synch_count* is dependent not only on how often the user wishes to check on network progress, but also on how quickly the network is apt to converge for a particular set of inputs. Local pieces of the network may converge separately, but global evaluation of the entire network's output is only made in the host at a global synchronization point. If the synchronization points are too far apart the network pieces may be "spinning their wheels" in extra cycles after the network has already converged as a whole. If the synchronization interval is too small more time is spent by the HNs synchronizing with the host.

3.3.6.1. User Commands

buildnet

Constructs and initializes the network and auxiliary data structures. Leaves the

network in a suspended state. **buildnet** has several command line flags associated with it for timing measurements, creating a log file, running FltSim, creating a verbose listing of BIF files, and for debugging ANNE itself.

newrun
Begins a simulation run. The user is prompted for the names of the input vector, the target vector, if any, and output vector file names. The user sets the size of *msg* packets and the length of timeout for receiving *msgs*.

startnet
Activates the simulation. The user is prompted to set *synch_count*, *msg_window*, and the next *checkpoint*. The *checkpoint* is checked to see that it corresponds to a global synchronization point. A checkpoint may be set such that the simulation works in step mode, that is, it only progresses one clock cycle before suspending.

stopnet
Suspends the network at the next global synchronization point whether or not it has reached a preset *checkpoint*. It may be restarted with the **startnet** command.

savenet
Saves the current network structure (including modified weights, etc.) in a new BIF file, which can be used later for a new simulation. The user is prompted for the name of the save file.

show
Displays the state of the "local" simulation parameters and lists the currently active *traces* (see below).

quit
Causes ANNE to exit. Once a network is activated (by **startnet**) summary statistics are sent from the HNs and written to a summary file, then ANNE exits.

help
Prints a list of ANNE's commands.

The user accesses data in the nodes through a similar mechanism as used for performing global network I/O. Though in this case the requests come from the host level instead of the node level. The user gains access to network data via *CN maps*. These are named groups of CNs. For example, one map may be of the CNs in the hidden layer of a back-propagation network. The default CN maps, as mentioned above, are the `CNgroup` blocks. Attached to each CN group is a list of the CNs that belong to it. By giving the name of a CN group to certain user operators, either via the terminal or at the node level through the user network code, entire groups of CNs are addressed.

At a suspension point in the simulation a user might print the values of some network fields. The host formats such a command as requests to HNs containing the CNs listed in the CN map. Each request contains a CN map name, e.g. "input", a structure name, such as "cn", "site", "link", an offset into the structure, and the width of the requested field in bytes. The offset and width are retreived from a hash table in the host that is indexed by each field's name. A 'C' union is employed to carry network data of different formats between the host and HNs. Currently, only the retrieval and assignment of "cn" fields are fully implemented.

The CN map name is matched to an HN's local map of CN indices. An HN responds to the host's data request by attaching the requested field values to each CN index in a copy of the appropriate local CN map and shipping this structure back to the host. At the host, the data values from each HN are re-assembled according to the global CN map and printed to the terminal screen. In a similar fashion, assignments to fields in the network are made.

The following are user commands that can be invoked when the network is in a suspended state. They operate on CN maps. Currently only the first four of these commands have been fully implemented.

print *<struct_name>* *<field_name>* *<CNmap_name>*
Prints the current value of *struct_name.field_name* for each CN in the named CN map.

assign *<struct_name>* *<field_name>* *<CNmap_name>*
Assigns a value to *struct_name.field_name* in each CN in the named CN map. The user is prompted for a CN index and enters a value for each CN.

trace *<struct_name>* *<field_name>* *<CNmap_name>*
Acts similarly to **print** except that the current values for each CN in the CN map are printed at each global synchronization point, according to *synch_count*.

untrace *<trace_number>*
Deletes a particular trace by number. The **show** command lists all active *traces* with a corresponding *trace_number*.

reset {*<CNmap_name>* | *all*}
Resets the link weights in a particular CN map or the entire network (not implemented).

eucldist *<CNmap_name>* *<CNmap_name>*
Gives the Euclidean vector distance between two CN maps, for example, the hamming distance (not implemented).

intersect *<CNmap_name>* *<CNmap_name>*
Gives the conjunction of two CN maps (not implemented).

union <*CNmap_name*> <*CNmap_name*>
Gives the disjunction of two CN maps (not implemented).

makemap <*CN_range*> <*CNmap_name*>
Defines CNmaps other than the default CN maps built when the simulator is initialized (not implemented).

3.3.6.2. Terminal Display

The current implementation of ANNE's display is rudimentary. There are no bells and whistles whatsoever. Some care has been taken to display information in a readable manner on the terminal screen. At each *synch_point* the current values of the simulation parameters are displayed along with the current state of the network: converged, suspended, or continuing. The user is free to add other information to the display screen through the convergence procedure. All output merely scrolls up the screen as a simulation progresses.

CHAPTER 4

Performance Testing

The main goal of testing ANNE is to determine the suitability of its unique simulation strategy when executed by a distributed memory processor model. In particular, these tests demonstrate the effectiveness of ANNE's methods of time synchronization, message packetization schemes (both SP and AP), and the costs of inter-processor communications versus network locality. Testing consisted of four parts:

(1) Measure ANNE's raw performance as determined by the number of connections per second (*xps* or "zips"). Xps is the rate at which CN-to-CN *msgs* are processed during network operation and provides a direct measure for processor efficiency and speedup.

(2) Test the performance of the four main functional components: BIF loading, CN-to-CN communication, inter-HN synchronization, and network computation (user time). CN-to-CN communication is further broken down into times for generating *msgs*, sending *msgs*, receiving *msgs*, and delivering *msgs.*

(3) Determine the number of cycles per convergence (*cpc*) for a specific back-propagation network. This metric illustrates how the convergence behavior of a highly synchronous, low locality network model is affected as it is spread over increasing numbers of HNs as a result of ANNE's loosely-coupled synchronization strategy.

(4) Demonstrate the effects of network locality on ANNE's performance using special, varying locality, *receptive field* networks.

In each of the above categories the SP and AP methods are contrasted. Preliminary results from tests on an iPSC/2 are also presented. ANNE's current iPSC/2 version is a direct porting of ANNE's iPSC code. That is, only minimal re-coding of ANNE was done to make it functional on the iPSC/2. No advantage was made of the many enhancements available on Intel's newest hypercube. A more optimized porting will surely result in significant performance improvement compared to our early results, but the testing accomplished thus far on the iPSC/2 illustrates several interesting trends.

4.1. Test Networks

The data model used in the first three parts of testing is a three layer back-propagation network. For such a network the interconnection density between layers is 100%; each CN in a layer is connected to every CN in an adjoining layer. For the xps and functional component testing, each layer in the network has an equal number of CNs. The names of these networks are prefaced with an "n" followed by a number indicating how many CNs are in each layer. Thus the network

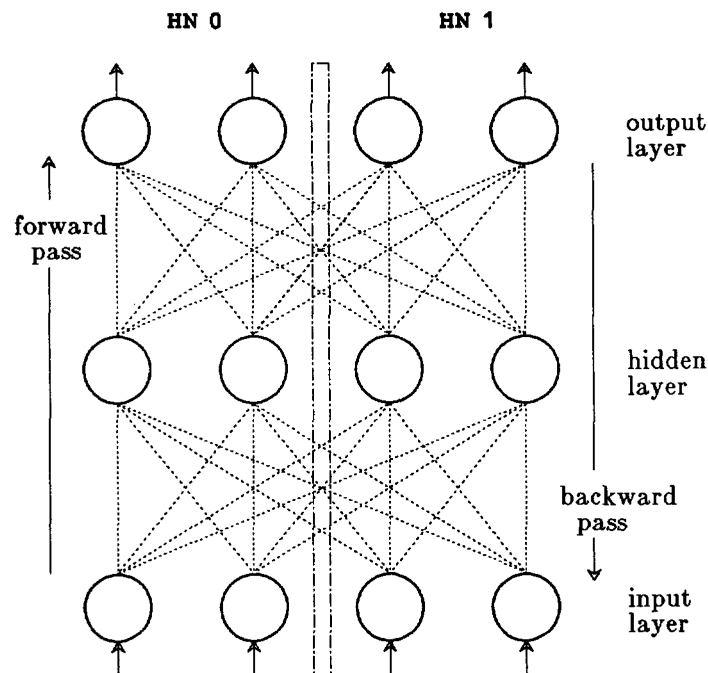


Figure 4.1: Back-propagation network mapped to two HNs, having 4 CNs per layer.

depicted in figure 4.1 is referred to as *n4*. For xps testing, 5 such networks were used: *n8, n16, n32, n64,* and *n128,* each containing links totalling 144, 544, 2,112, 8,224, and 33,024, respectively. Functional component testing used only *n32*. These networks were partitioned among HNs by vertical slicing, as shown in figure 4.1.

A back-propagation network used for character recognition was utilized for cpc testing. This network contains 88 CNs in its input layer, and 16 CNs each in its hidden and output layers, for a total of 1,768 links. This network, henceforth referred to as *alfanet,* receives 8 by 11 binary matrix representations of the first 16 letters in the alphabet as input. Accompanying each input vector is a binary target vector that has a single component turned "on", while all other components are "off". For example, the target vector for the letter 'D' is

"0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0"

When completely trained, a single output CN will be set high (~1) corresponding to the letter presented as input.

Special networks were utilized for the fourth set of tests. Here, these *receptive field networks* are not intended to have any meaningful computational properties. They act as connectivity frameworks on which to test the effect of network

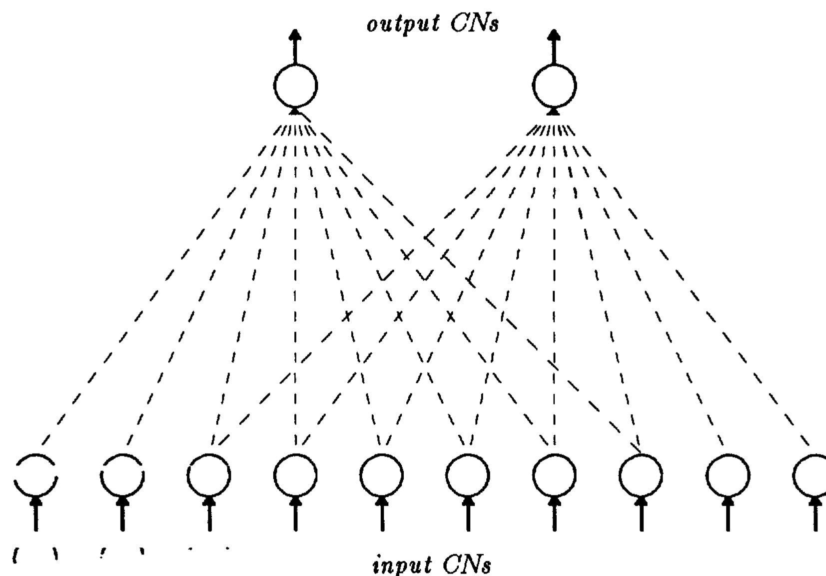


Figure 4.2: Receptive field network with a field of 8 for each output CN and a shift of 2.

locality on ANNE's raw performance. Figure 4.2 illustrates a small example of such a network. Each network consists of two layers, an input and output layer. A vector is presented to the input layer, some ersatz function takes place on the value input to each CN in this layer, and the transformed vector is propagated to the output layer for further processing.

The parameters in designing these networks are the number of input and output CNs, the width of each output CN's receptive field, and the *shift* of each output CN's receptive field relative to its neighbor's field. For the tests described here the number of output CNs was held constant at 32, and the receptive field size was always 8. Specifying a different shift then automatically determines the number of input CNs. The network in figure 4.2 has a shift of two.

An increase in the shift implies an increase in these networks' locality, also. The actual ratio of local connections to extra-HN connections for each of these networks depends on the configuration of the iPSC at runtime, but for a given number of HNs, higher shifts mean higher locality. These networks were mapped in a similar fashion as the back-propagation networks. That is, they were partitioned vertically into equal or nearly equal portions.

Each network name is prefaced with the letters "rn", followed by numbers indicating the number of inputs and the shift of the network. Thus, network *rn70.2* has 70 input CNs with an output field shift of 2. Other networks used in these tests

are *rn132.4*, *rn194.6*, and *rn256.8*. *rn256.8*, since it has a shift value that does not allow any overlap in receptive fields is actually composed of 32 separate sub-networks.

4.2. Xps Tests

Xps tests used 5 networks over 6 dimensions of the iPSC, while iPSC/2 testing used *n8*, *n16*, and *n32* over 3 dimensions. All the tests were run using the feed-forward mode of the networks. In feed-forward mode, an input vector is presented, allowed to propagate through the network *synch_count* times, and the CNs in the output layer send their values to the host. A *synch_count* of 100 was used to minimize global synchronization overhead and maximize xps performance.

The xps tests are summarized in table 4.1. Some tests were not completed for *n32*, *n64*, and *n128*. For the latter two networks smaller dimensions of the iPSC could not be used, since the BIF sub-files for these networks were too large. The 32 processor test for *n32* using SP simply failed to run for various combinations of simulation parameters. Tests in which there would be less than 3 CNs per HN were not attempted.

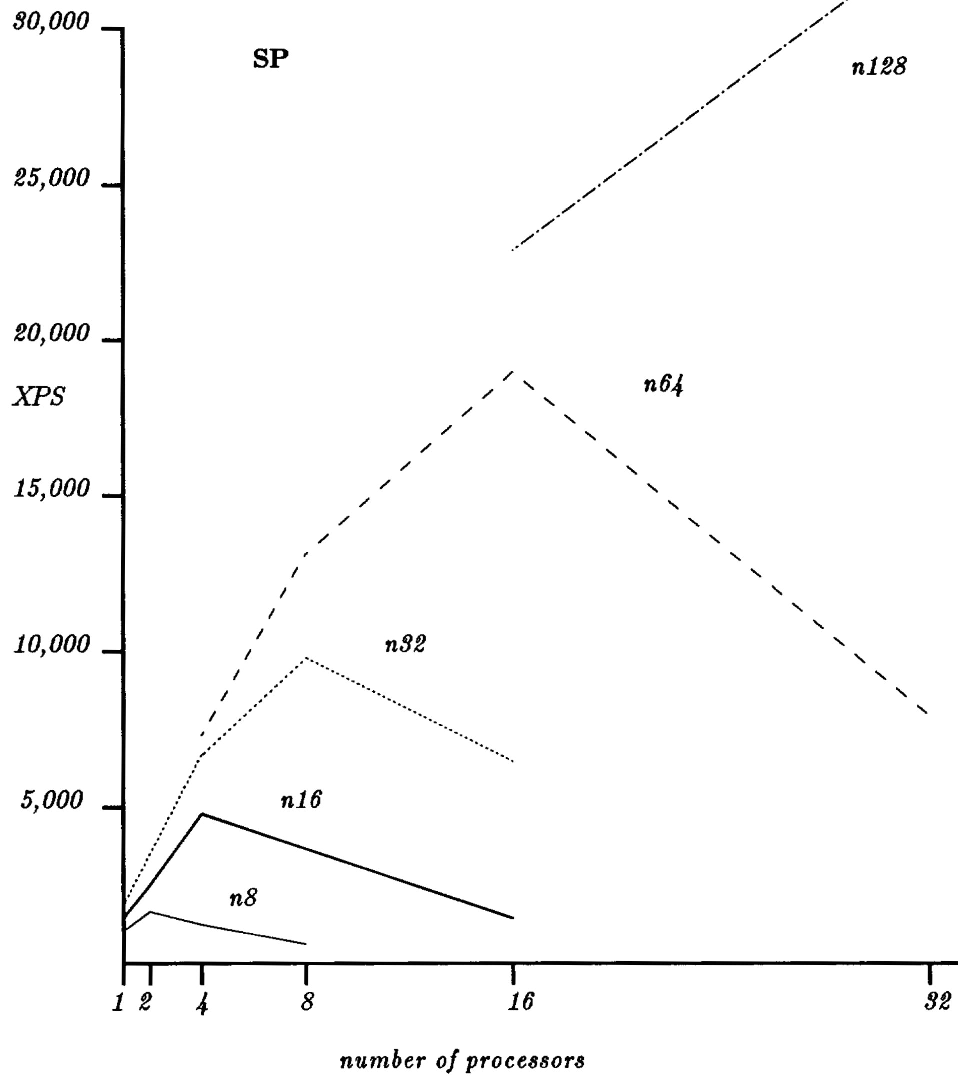
The results of xps testing on the iPSC using SP are shown in graph 4.1. With the exception of *n128*, each of the test networks show a steady, though tapering, increase in xps, up to a marked peak. After each peak, performance drops off dramatically. These results highlight the effect of communication locality. As each network is spread over more HNs, communication costs between HNs increases. This communication overhead diminishes the increased computational power of more processors, and accounts for a slight downward trend in performance, seen in the first part of the graphs for *n32* and *n64*. More rapid decreases in performance result from insufficient utilization of *msg* packets in the SP method.

Locality effects are more clearly illustrated in graph 4.2, where the results of graph 4.1 are shown relative to each dimension of the iPSC. Here it is clear that performance using SP is directly related to the number of connections that must cross HN boundaries.

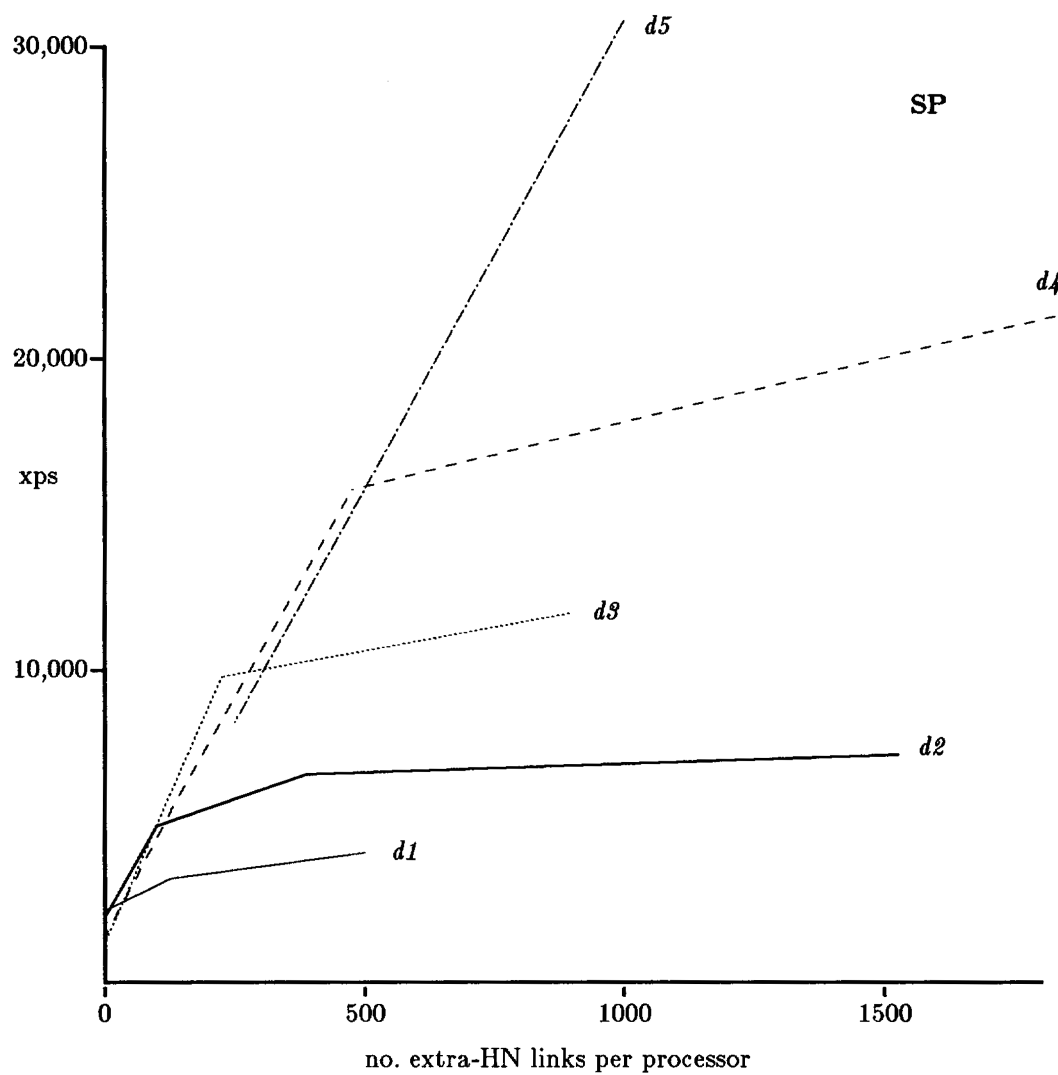
Table 4.1 - Xps tests

synch_count = *msg_window* = 100 S = too small
maxmsgnum = 63, msg *timeout* = 0 L = too large

nets & cube config.		CNs per HN / actual msg packet size					
name	dimension ☞	0	1	2	3	4	5
n8:	8x8x8	24/0	12/32	6/8	3/2	S	S
n16:	16x16x16	48/0	24/63	12/32	6/8	3/2	S
n32:	32x32x32	96/0	48/63	24/63	12/32	6/8	3/2
n64:	64x64x64	L	L	48/63	24/63	12/32	6/8
n128:	128x128x128	L	L	L	L	24/63	12/32



Graph 4.1: iPSC xps performance: SP method



Graph 4.2: Xps vs. number of external connections per HN

Table 4.2 - *msg* packet sizes for *n16*		
dim	# mid + flush packets out/HN (size)	# packets in/HN
1	2(63) + 1(2)	3
2	0 + 3(32)	9
3	0 + 7(8)	49
4	0 + 15(2)	225

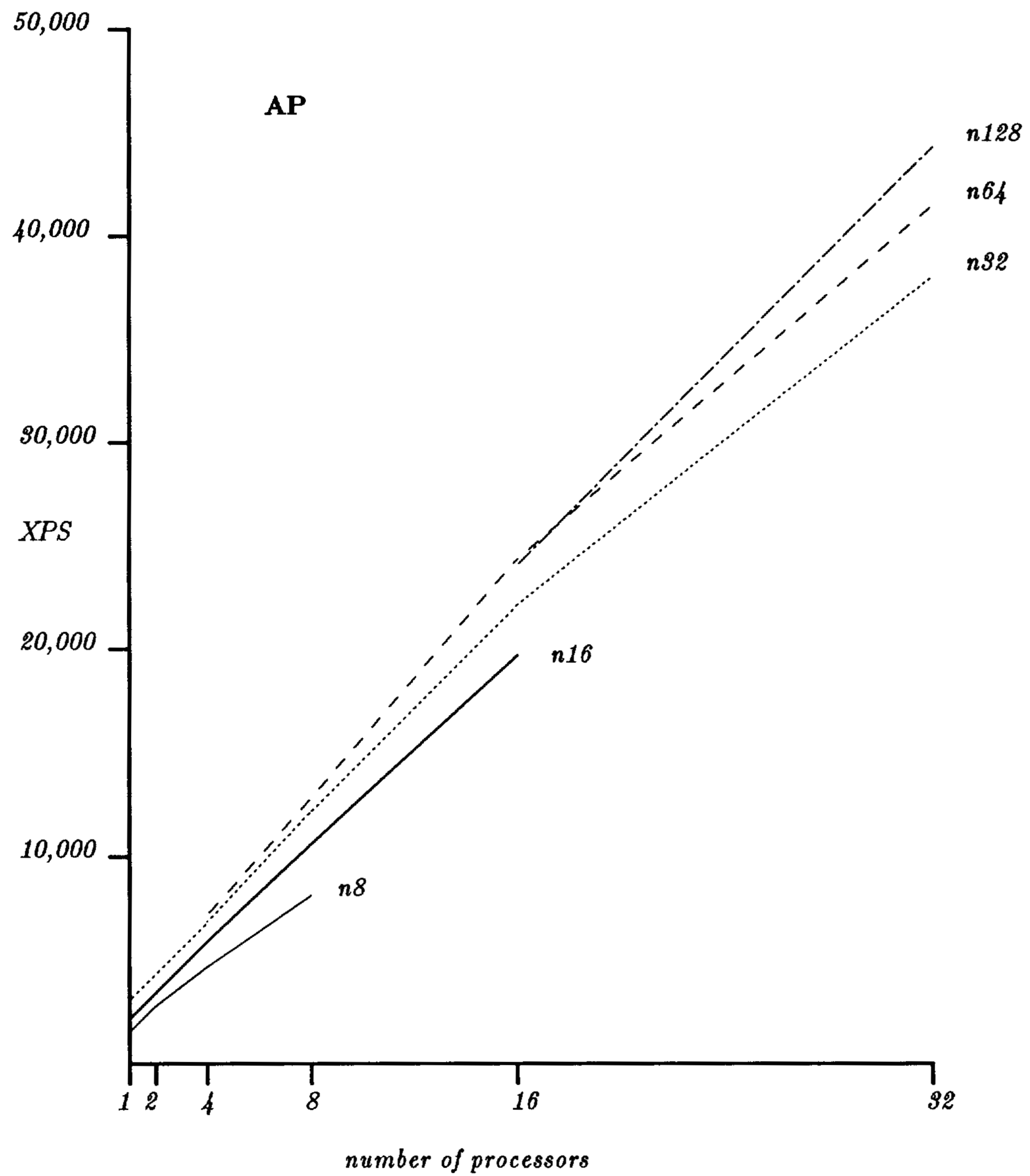
The performance of a particular test network is greatly influenced by the number of *msg* packets produced during each clock cycle. With SP, packets left at the end of a local clock cycle are flushed, regardless of their size. Thus, some packets may be partially empty. Table 4.2 illustrates this point for the test network *n16*. With two processors each HN produces two full *msg* packets in the middle (labelled *#mid* in the table heading) of each clock cycle, and 1 packet of two *msgs* at the end of each cycle (*flush*). With 4 HNs, 3 half-full *msg* packets are produced for each clock cycle, and each are sent at the end of a local network cycle. The number of packets per clock more than doubles when *n16* is mapped to 8 HNs, and increases five-fold for iPSC dimension 4. More importantly, the number of incoming packets at each HN increases dramatically. Reductions in performance for the other networks are due to similar sudden increases in the number of *msg* packets during each clock cycle.

Using the AP method, which guarantees that only packets containing *maxmsgnum msgs* are sent, ANNE's xps performance improves markedly. For these tests *maxmsgnum* equaled 63. Graph 4.3 shows the xps results using AP. A uniform increase in performance is seen for all the test networks, *n8* through *n128*. With AP, ANNE also posseses more stability, thus allowing the completion of the 32 HN xps test for *n32*, which was not possible with SP.

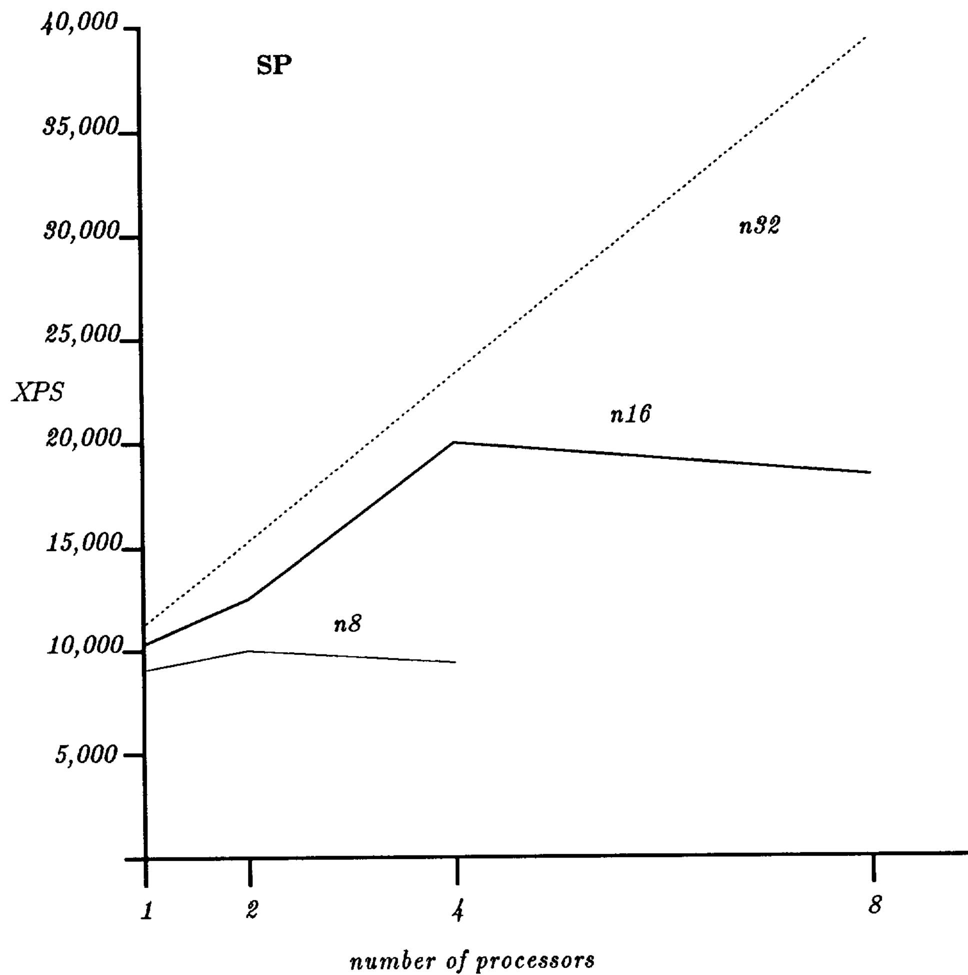
Graph 4.4 illustrates the xps results for 3 test networks on the iPSC/2 using SP. ANNE shows over a four-fold increase in raw performance on this machine. *n8* and *n16* still show a drop in performance (*n32* has just reached its peak in this graph) due to the number of *msg* packets being produced, but the drop-off is not nearly as severe as for the iPSC. The improved performance of ANNE on the iPSC/2 is due to the increased speed of the 386-based HNs, and improved inter-HN message routing. The same overall increase in xps performance afforded by the AP method on the iPSC, is seen, naturally, on the iPSC/2. Unfortunately, by the time the tests illustrated in graph 4.5 were made only 4 HNs were available on the iPSC/2 being used. Thus only the line for *n8* shows real improvement relative to the SP data.

4.3. Functional Components Performance

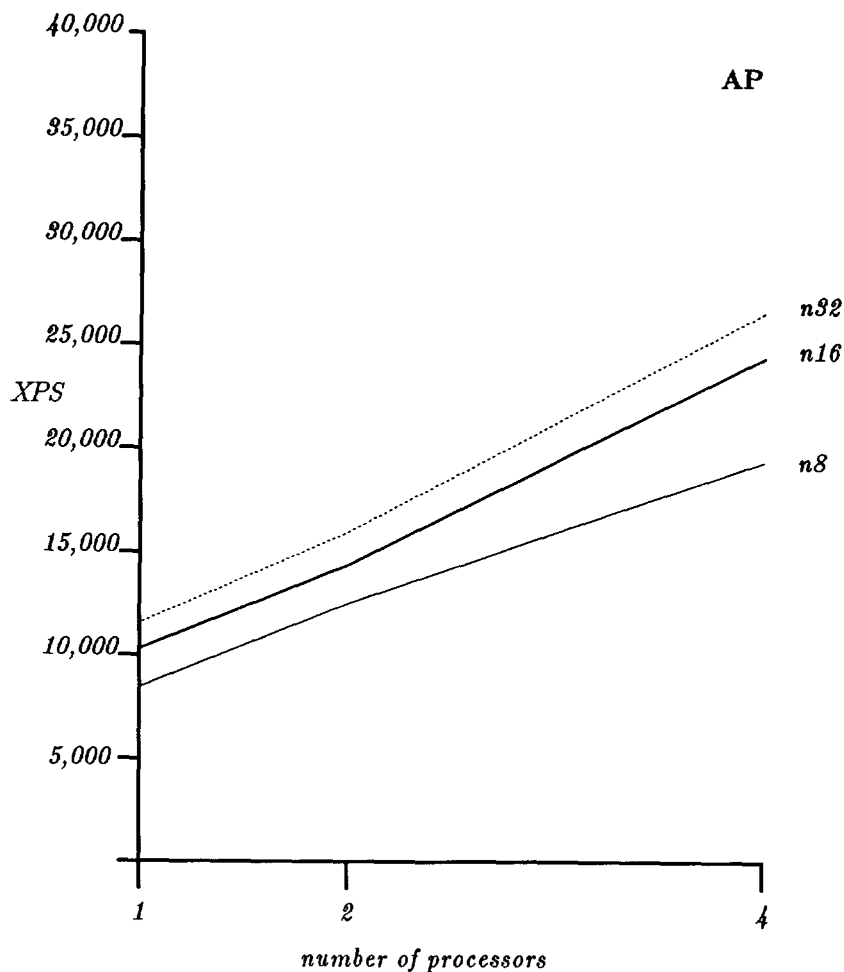
n32 was used for testing the performance of ANNE's functional components. This network was simulated using cube dimensions 0 through 5 on the iPSC and cube dimensions 0 through 2 for the iPSC/2.



Graph 4.3: iPSC xps performance: AP method



Graph 4.4: iPSC/2 xps performance: SP method



Graph 4.5: iPSC/2 xps performance: AP method

4.3.1. Loading and Parsing Performance

Loading the BIF files for *n32* on the iPSC is done in nearly constant time. BIF loading times ranged from 7 seconds for a single HN to 11 seconds for 16 HNs. The overall time needed to parse and initialize an entire network is called the network *scan time*. Through 4 HNs there is a near linear speedup in scan time. For 8 and 16 HNs, scan time levels off to a little over 5 seconds. No further decrease in the scan time for these iPSC dimensions occurs, because the scanning operation is sandwiched between two parts of the overall loading operation. After sending the BIF sub-files, each HN begins scanning its own sub-file. The first HNs loaded may be idle after scanning these files if they finish scanning before the host has completed loading of the remaining BIF sub-files. Only after the host has completed this phase

will it send the HNs their copies of the *CNgroups* file. Thus, scan time is no longer reduced by having more processors once scan time is less than the overall loading time. In this case the sequential portion of network loading dominates the parallel portion.

Together, the load and scan times can be combined into a single rough measure: links per second. Using this measure the rates for dimensions 0 through 4 are 64, 96, 146, 143, and 129 links per second, respectively. Due to a small number of samples, and to the fact that the iPSC/2 host was being utilized by multiple users, its load/scan rates are less reliable. The scan rates for the iPSC/2 ranged from 235 to 768 links per second.

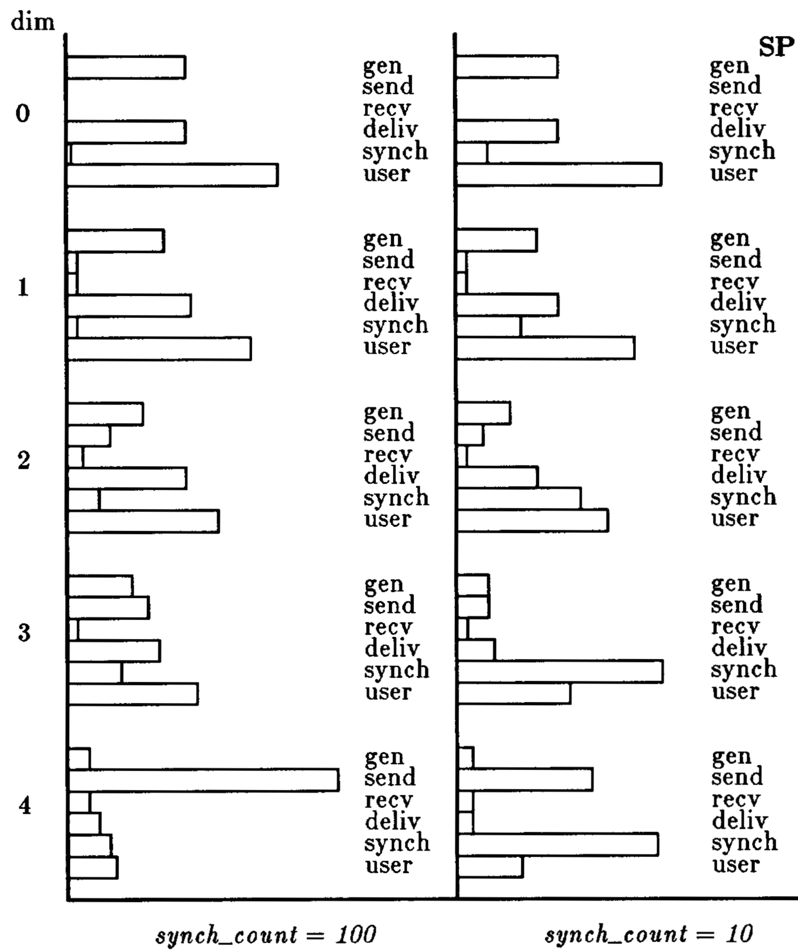
4.3.2. Other Components

Communication, synchronization, and user time were measured using the *n32* network. The performance of each of these components is illustrated by four sets of histograms starting with graph 4.6. There are two sets of tests in each graph each with synchronization intervals of 10 and 100, and with equal *msg_window* settings. The graphs represent the percentage of time taken per HN by six activities at the HN level, with the exception of *user*, which also includes some host time.

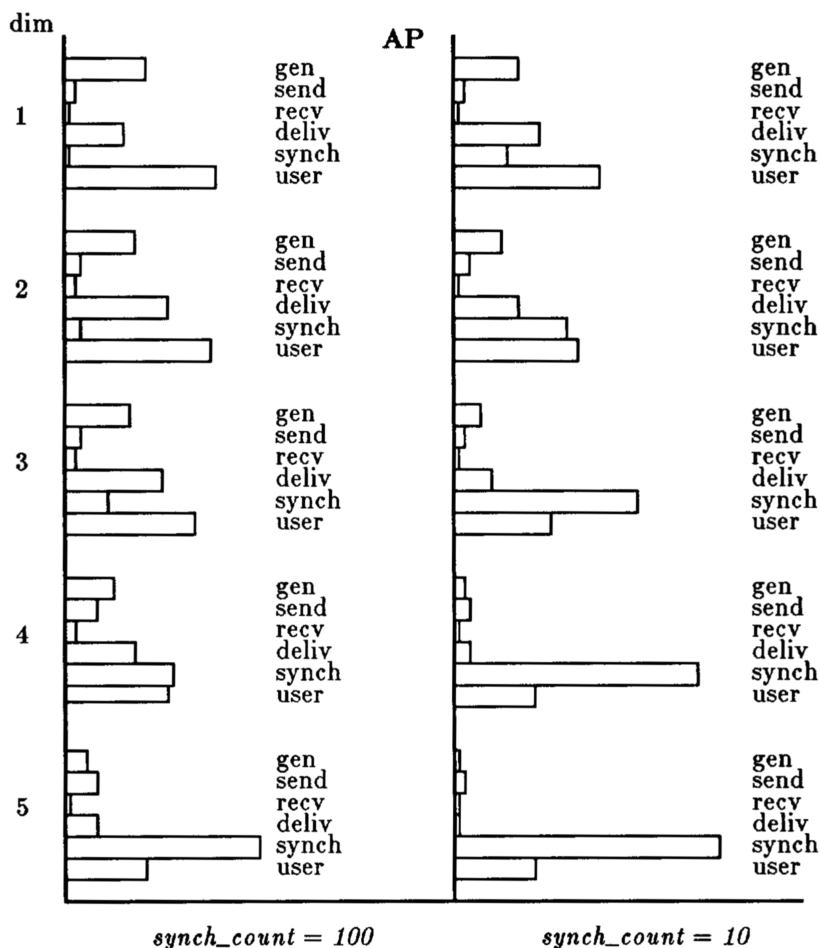
Six shorthand names label the horizontal bars in the graphs. *Gen* represents the actions taken to generate CN messages, such as locating a links array, adding a *msg* to a *msg* packet, etc. *Send* is the act of handing off a *msg* packet to the iPSC messaging system, including waiting for a clear messsge channel. *Recv* involves probing for incoming *msg* packets and waiting for the iPSC to assign these packets to a buffer in ANNE. Unpacking *msgs* and assigning their values to the appropriate link is the function labelled *deliv*. *Synch* measures the time used by HNs waiting for other HNs to complete their network cycles at global synchronization points. All other activity, labelled *user*, is the result of network computations done by the user's network procedure in the HNs and the convergence procedure in the host.

A comparison of the two histograms of graph 4.6 shows four interesting trends. First, the percentage of time taken by *recv* is relatively constant, regardless of the *synch_count*. Second, *gen*, *deliv*, and *user* times benefit from an increasing number of HNs. This benefit is paid for by a third trend, a sharp rise in the percentage of time taken to send *msg* packets. *Send* time consumes about 62% of the total time for a *synch_count* of 100 with 16 HNs. Last, decreasing the *synch_count* increases the cost of synchronization, as should be expected.

The behavior of *send* in graph 4.6 is explained almost entirely by the iPSC's difficulty in handling the large inter-HN communication load brought on by the combination of the SP method and the high interconnection density of a back-propagation network. Channel contention becomes the limiting factor as more and more packets are pushed into the system. And as the iPSC dimension grows, more of these packets are shared by multiple HNs, since the iPSC uses a store-and-forward scheme for passing messages beyond immediate neighbors. *Send* performance is slightly better for a shorter synchronization interval, because at synchronization points, HN message channels have time to clear and thus, channel contention is eased at the start of the next network cycle.



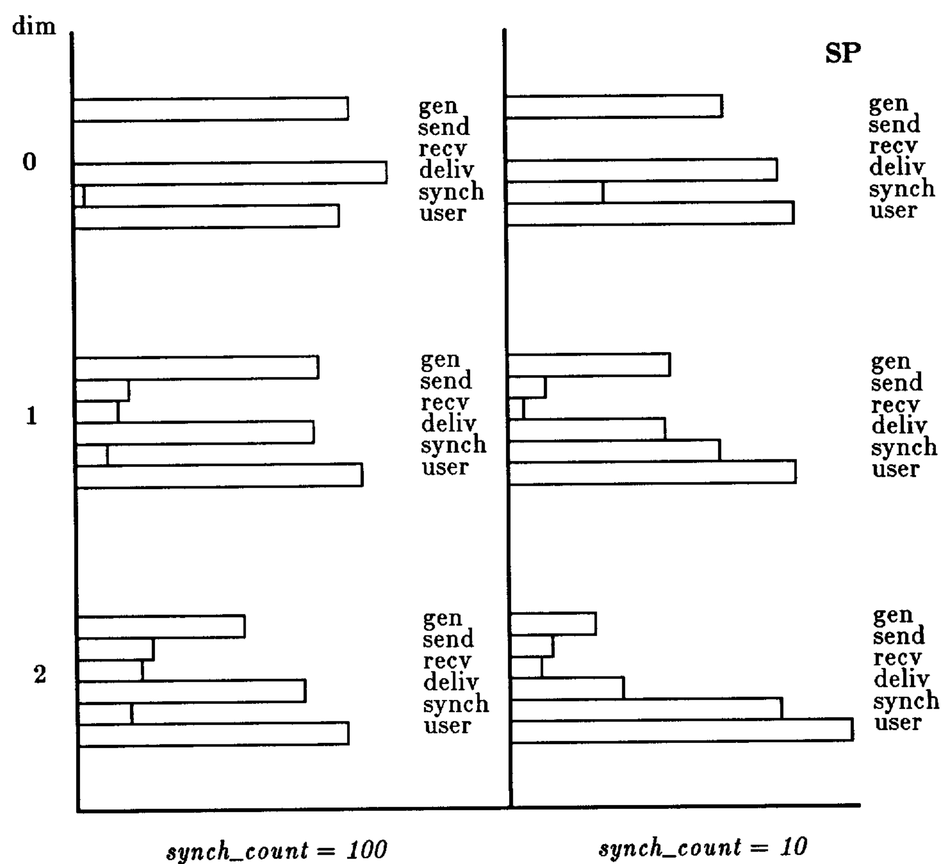
Graph 4.6: Functional component performance on the iPSC: SP method.



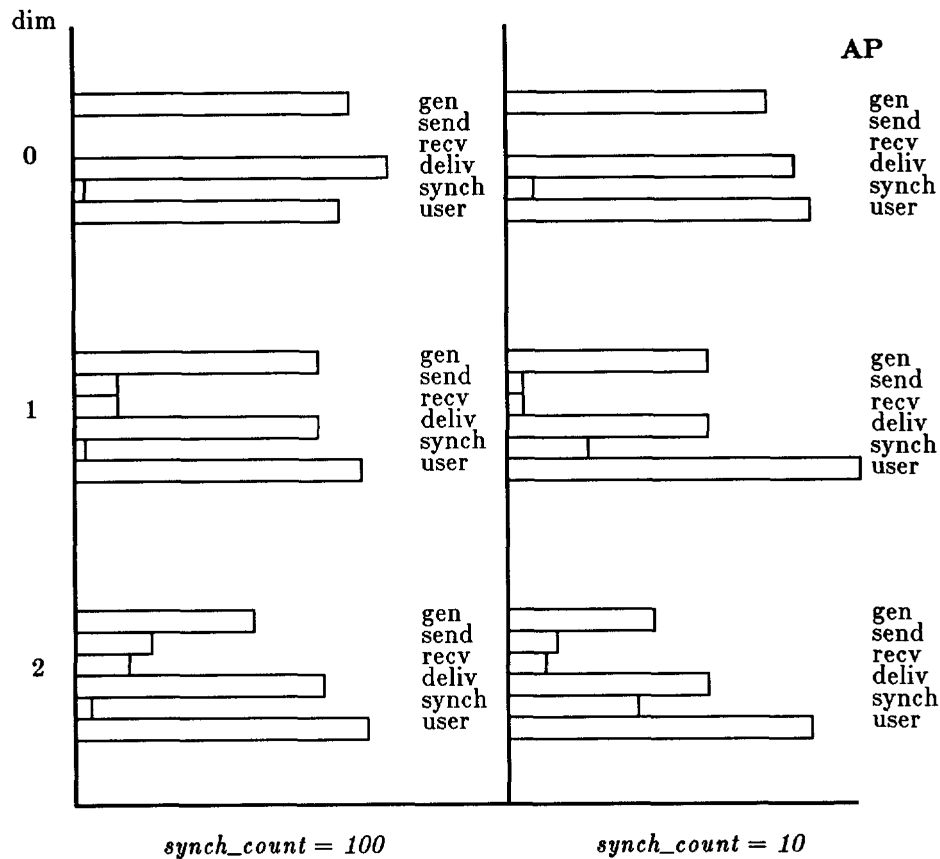
Graph 4.7: Functional component performance on the iPSC: AP method.

The histograms in graph 4.7 give the results of tests using the AP method in ANNE. The first two trends noted in graph 4.6 hold for graph 4.7, also. Packet sending performance, however shows a marked improvement with the full utilization of *msg* packets. With the percentage of time taken by *msg* sending reduced, the time taken for HNs to globally synchronize with the host increases. For two HNs, the ratio of *synch* times between a *synch_count* of 100 and a *synch_count* of 10 is nearly 10 to 1. For larger cube dimensions a *synch_count* of 100 is inadequate to dilute the effect of sequentializing the interaction between the host and many HNs. With 32 HNs, 53% of run time is taken by synchronization for a *synch_count* of 100 versus 73% of run time for a *synch_count* of 10.

Functional component testing of ANNE for the iPSC/2 is shown in graphs 4.8 and 4.9. The analysis of these data suffers from a limited number of tests and a host



Graph 4.8: Functional component performance on the iPSC/2: SP method.



Graph 4.9: Functional component performance on the iPSC/2: AP method.

machine in multi-user mode at the time of testing. However, the patterns exhibited are quite similar to those seen in the iPSC graphs for cube dimensions 0 through 2. As on the iPSC, *synch* time again corresponds directly to the *synch_count*, and the time allotted to the user's code remains fairly constant, reaching approximately 42% for two HNs. One notable difference between the two machines' data is the smaller allotment of *synch* time for 4 HNs on the iPSC/2 as compared with the iPSC. Four HNs is insufficient to analyze the effects on ANNE of the iPSC/2's faster inter-HN communications.

4.4. Cpc Tests

Measuring cycles per convergence used *alfanet*. The set of weights used were identical at the start of each test. The only starting condition on the weights was that they be random and relatively small, in the range ± 0.400. For the iPSC, the results for cube dimensions 0 through 3 are presented. For the iPSC/2 dimensions 0

through 2 were used. Four parameters, *maxmsgnum, msg timeout, synch_count,* and *msg_window,* were varied to test their effects on convergence behavior. Except for *synch_count,* however, the other three parameters are irrelevant when running ANNE on a single HN. Cpc results are shown in the eight graphs which follow.

Preliminary testing of *alfanet's* convergence found that a strictly uniform list of simulation parameter settings for each cube dimension was not effective. Parameters that gave good results for one cube dimension might fail dismally for another dimension, leaving little to compare. Instead, a wide variety of parameter values were experimented with and from these tests a representative sampling of results were chosen. Typical values used for the four parameters listed above ranged as follows:

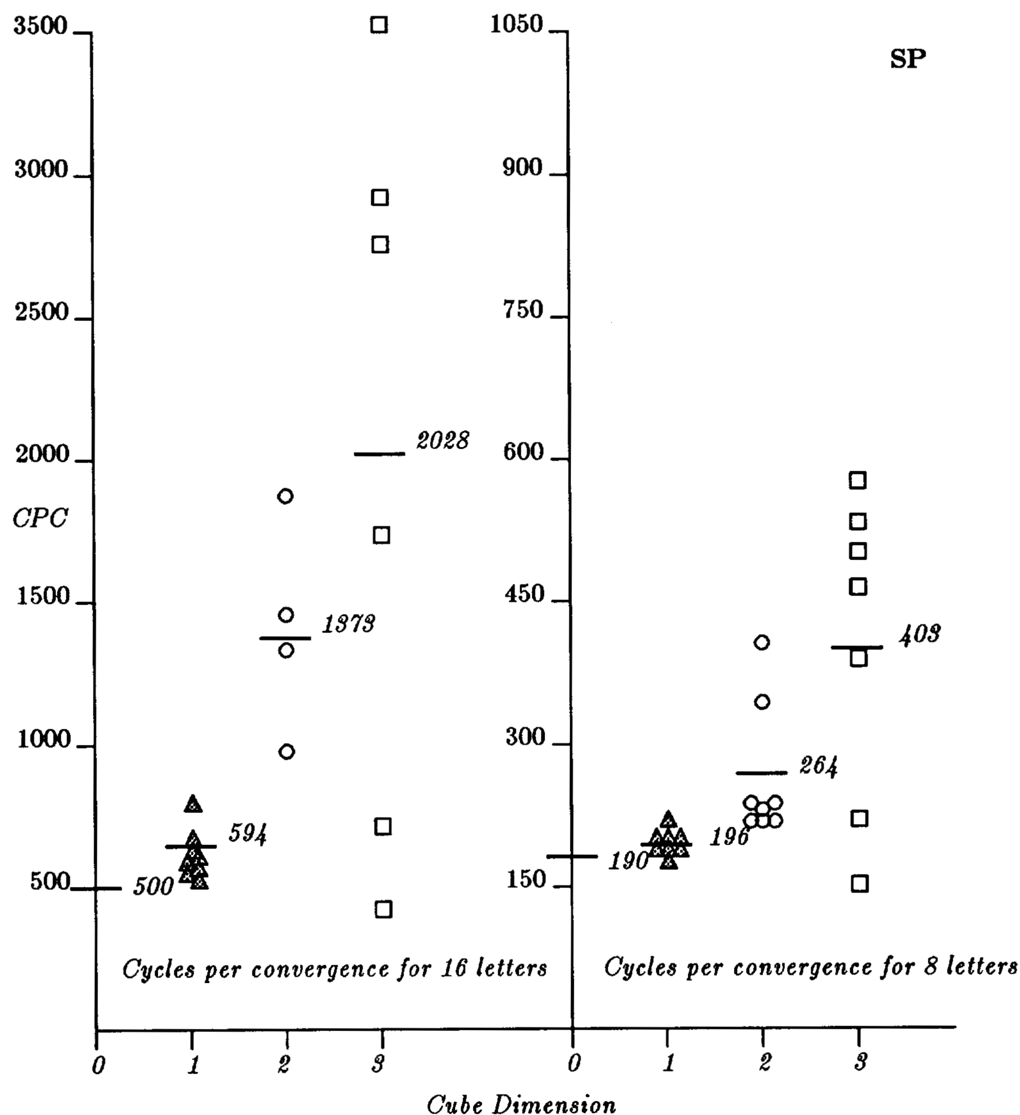
maxmsgnum: 13, 26, 52, and the maximum value, 63.
msg timeout: 0, 5, 10, 20, 50, and 100.
msg_window: 0, 1, 5, 10, and 20.
synch_count: 1, 10, and 20.

Tests which failed are not presented.

Tests failed for one of two reasons: (1) The chosen parameters crashed the machine, or (2) a test was considered non-completeable due to the imposition of a *1000 cycles rule.* In general, as letters were presented in order (from 'A' to 'P') it took an increasing number of cycles for *alfanet* to converge on a particular letter. Early testing revealed that if the convergence of a particular letter did not complete within 1000 cycles, then the network would not converge on that letter nor subsequent letters in less than several thousand cycles, if at all. This type of behavior nearly always occurred at or subsequent to the 14th letter ('N').

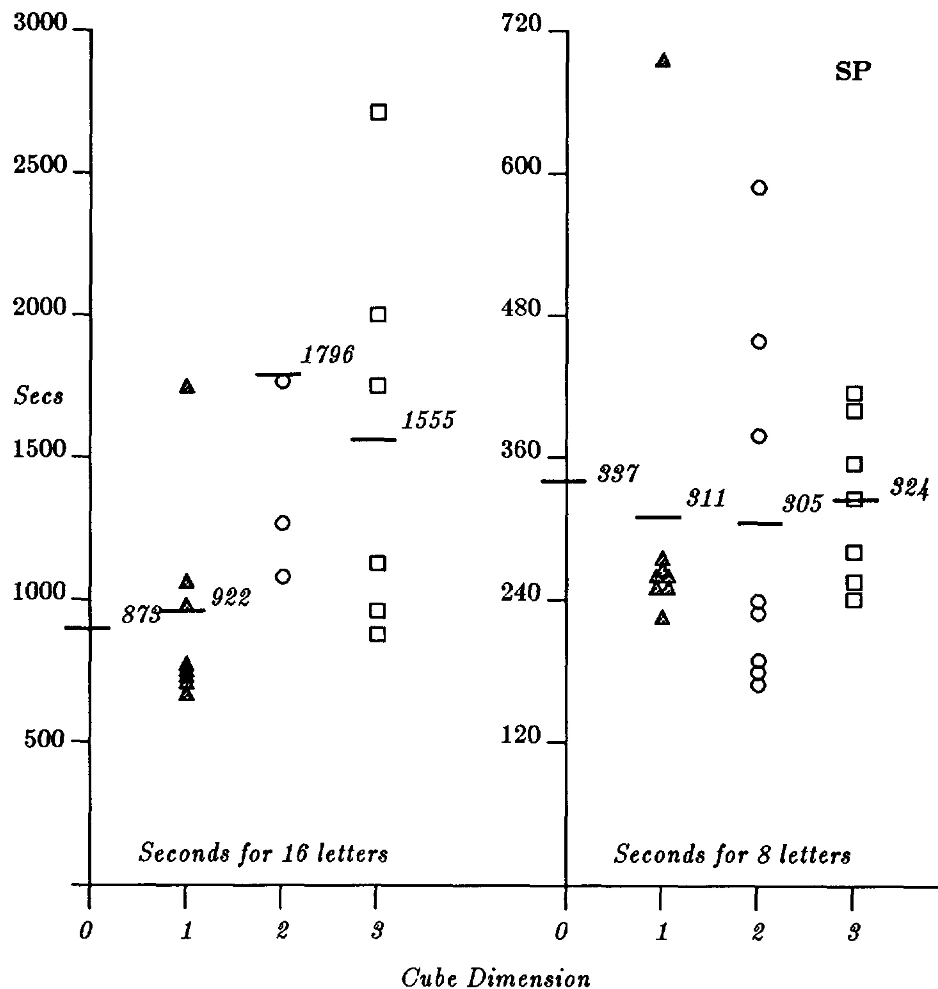
The test method employed here, converging on letters one at a time, is one of two network training methods often employed. The other method of "teaching" a back-propagation network to recognize characters is to present it with each letter, and the corresponding target vector, for one to several iterations of the network before moving to the next set of input and target vectors. Switching from one letter to the next is done arbitrarily, regardless of the actual convergence behavior of the network for a given letter. However, in the test method here, each input and target vector is clamped to the network until the entire network converges on the input letter within a specified error value. The error value used was 0.4. The network converges when the appropriate CN, corresponding to the target vector component equalling 1, has a value ≥ 0.60, and the other output CNs are ≤ 0.40. An entire test consisted of clamping each of the first 16 letters of the alphabet until each had converged, individually, according to this method. To help clarify the results of cpc testing with 16 letters, the results from the first 8 letters in each test are presented alongside the 16 letter data. Test results were more consistent for the first 8 letters. Tests using one HN are used as the base comparison for the multiprocessor tests.

The detailed results of cpc testing are somewhat inconclusive in terms of the settings of individual synchronization parameters for both the iPSC and the iPSC/2. There appears to be a weak correlation between better convergence and longer *msg timeouts* or shorter synchronization intervals. Larger settings of *msg_window* had, in general, a more favorable effect when used with larger cube dimensions. With more



Graph 4.10: Cpc results on the iPSC: SP method.
Tests for each cube dimension are arranged in vertical
lines. Each dimension is represented by a different symbol.
Horizontal bars mark the average value of tests for each dimension.

than two HNs, the convergence behavior of the character recognition network is, at best, unpredictable. It is clear that beyond cube dimension 1 the convergence of *alfanet* takes considerably more cycles as more HNs are used.



Graph 4.11: Seconds elapsed for cpc tests on the iPSC: SP method. Tests for each cube dimension are arranged in vertical lines. Each dimension is represented by a different symbol. Horizontal bars mark the average value of tests for each dimension.

The poor behavior of *alfanet* with ANNE is not totally unexpected. A back-propagation network presents ANNE with a real challenge due to its high interconnection density, low to moderate locality, and highly synchronous algorithm. The convergence results presented here are remarkable considering ANNE's design bias toward sparsely connected, high locality network models, and its semi-synchronous

nature. The results are also a credit to the robustness of neural network models in general.

4.4.1. Cpc Results for the iPSC

Graphs 4.10 and 4.11 show the results of cpc testing for the iPSC using SP. The tight clustering and the moderate increase in the cpc values seen for two HNs in the first graph indicates that ANNE's synchronization scheme works well for this most basic case of multiprocessing. However, the results for all 16 letters in graph 4.10 shows an approximate doubling of the average cpc for each subsequent cube dimension used. For 8 letters the cpc results are better, indicating that perhaps there is a second order effect of the letter presentation procedure being exhibited, or that an "unseasoned" network is less sensitive to ANNE's synchronization model. In any case, graph 4.11 illustrates that increased processing power is not enough to overcome the increase in cpc for all 16 letters. This graph shows the total elapsed time, in seconds, for each of the previous iPSC cpc tests. For 8 letters, only moderate improvement in convergence time results.

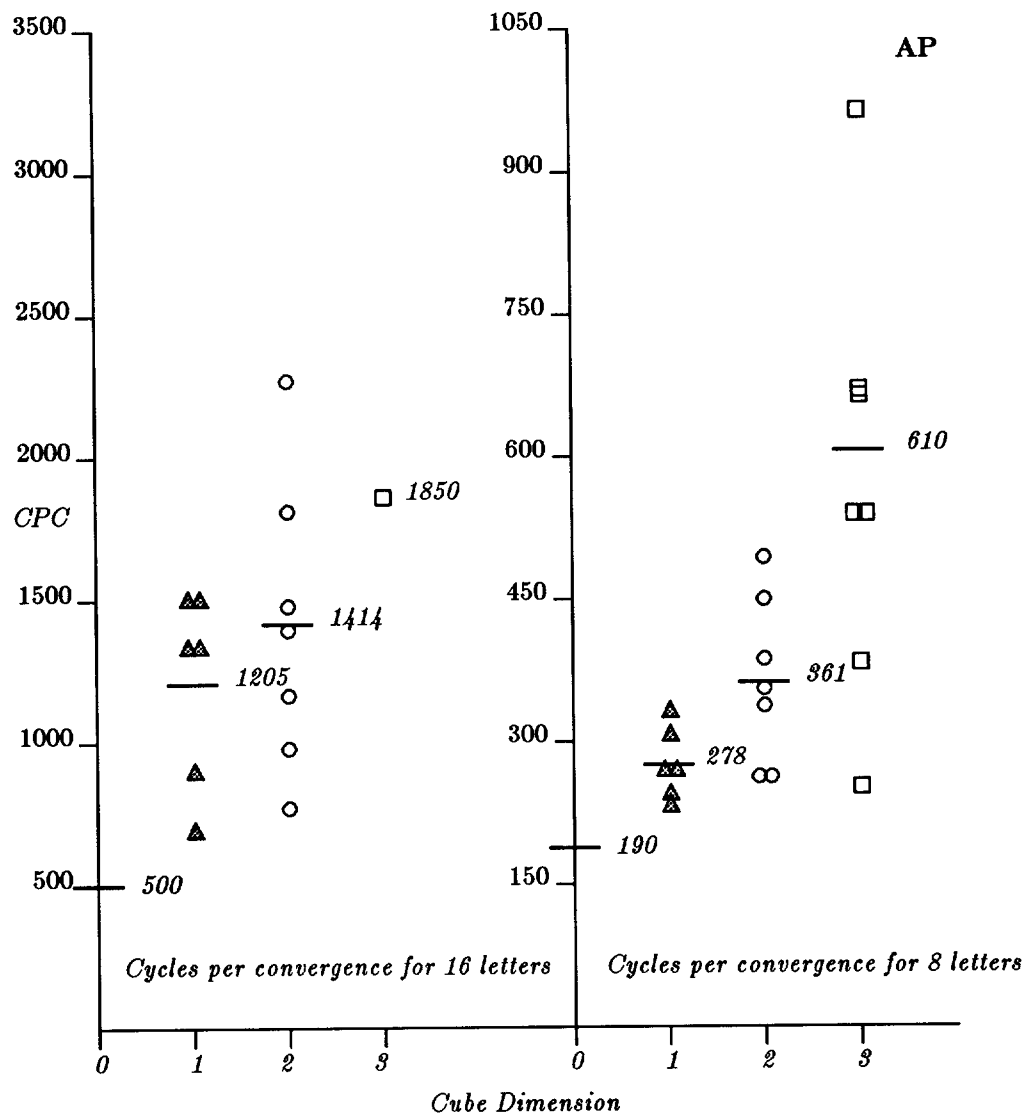
The use of the AP method helps the cpc situation not at all (see graphs 4.12 and 4.13). The tight clustering of results for 2 HNs is is not evident in the data for 16 letters. With AP the trend for both 16 and 8 letters is similar to that found with SP, in that as larger cube dimensions are utilized the cpc continues to rise sharply. In fact, only one of several tests using 8 HNs completed according to the 1000 second rule. With the exception of the time data for cube dimension 2 in graph 4.13, convergence time for all 16 letters continues to increase for more HNs. The time data for 8 letters is tightly grouped, but the average times are relatively flat, despite the number of HNs put to the task.

4.4.2. Cpc Results for the iPSC/2

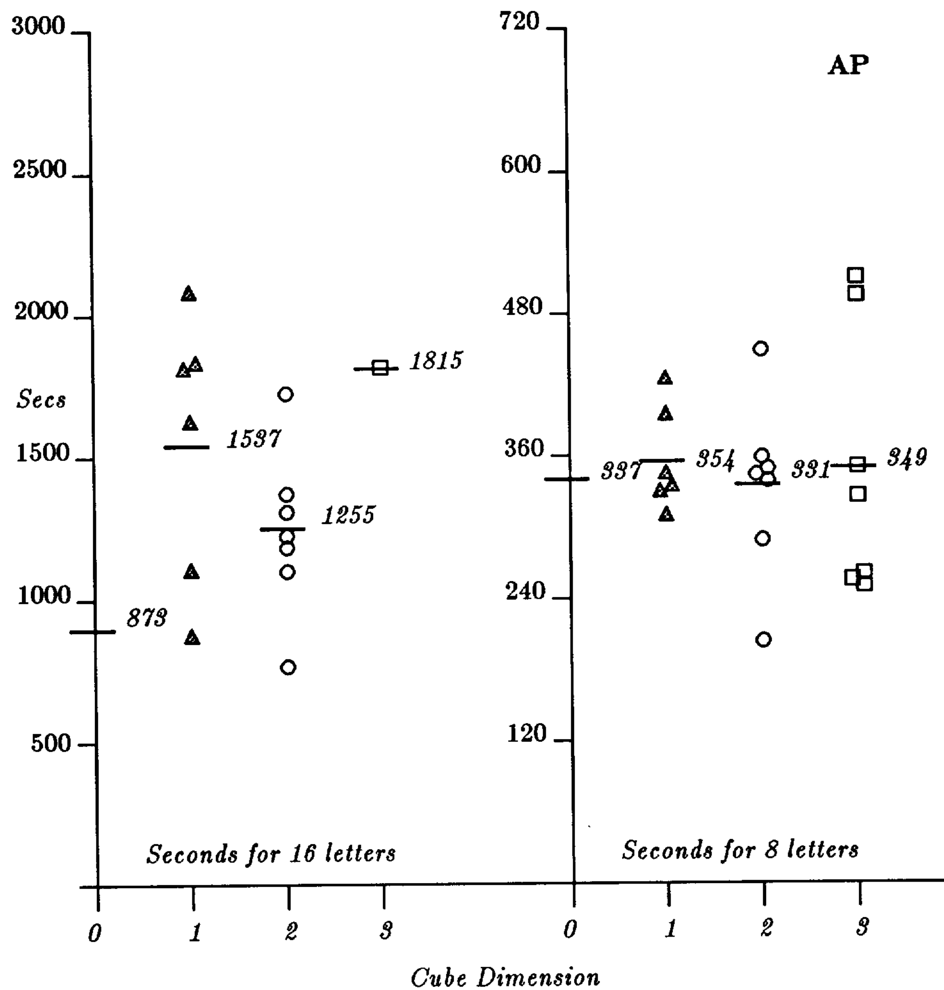
The limited number of cpc tests for the iPSC/2 show trends similar to the results for the iPSC, though the iPSC/2 data are more encouraging. The average number of cycles to converge over 16 letters and 8 letters using SP (see graph 4.14) increases with the number of HNs used, but less severely than for the iPSC. Correspondingly, in graph 4.15, the average convergence time for 16 letters increases more slowly as compared to the iPSC data, and actually decreases for 8 letters. Convergence efficiency per HN, however, is still plainly low on the iPSC/2.

Both the cpc and elapsed time data using the AP method on the iPSC/2, seen in graphs 4.16 and 4.17, exhibit trends similar to those on the iPSC. The results in both graphs, for cube dimension 1, and 16 letters, are widely scattered, implying a great deal of non-determinancy in *alfanet's* behavior for such low parallelism. This is especially surprising in light of the tight clustering of cpc values for 8 letters using 2 HNs. Remarkably, this behavior improves for four HNs.

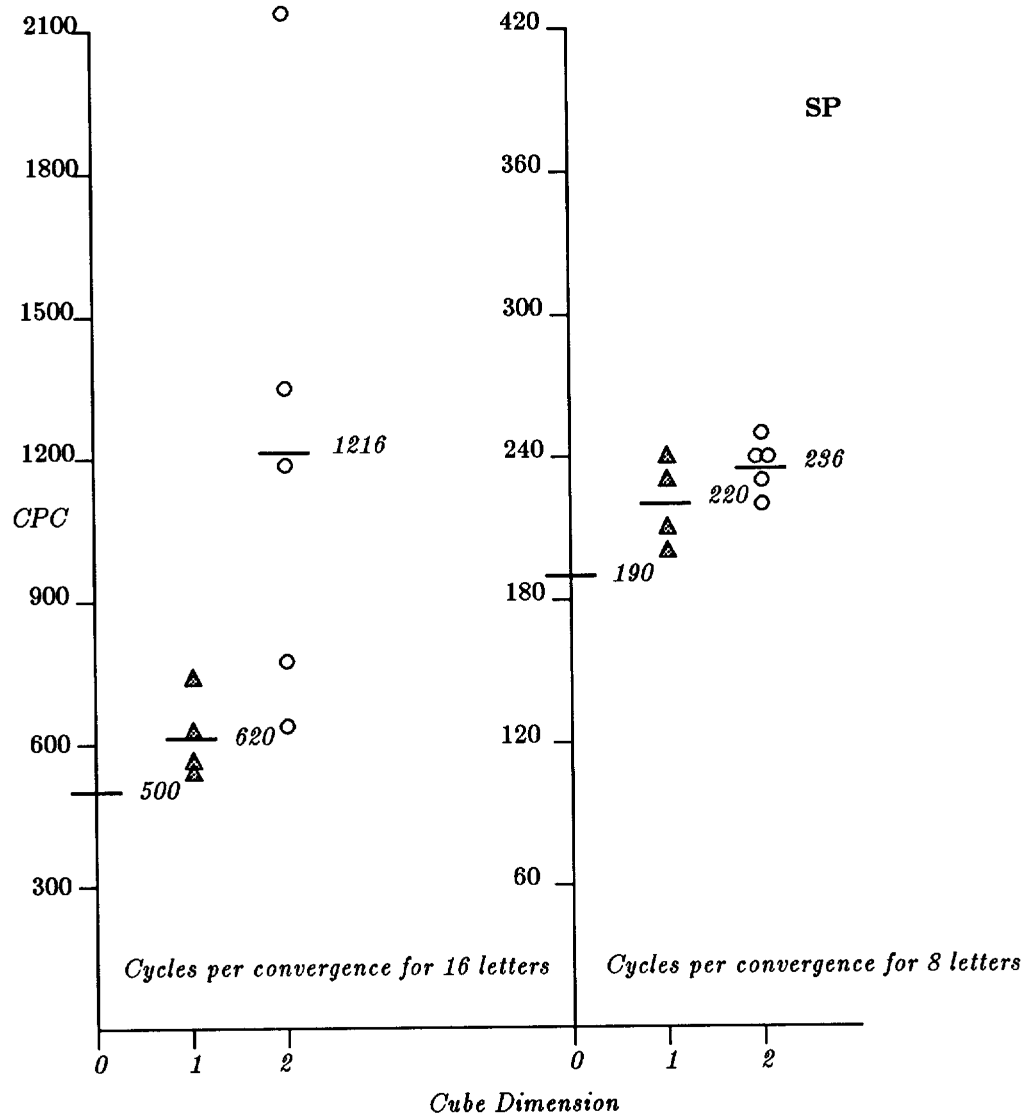
Despite what might be expected from the AP method, considering its dramatic effects on raw performance for low locality networks, it is flawed in that it may tend to "over-synchronize" the communications between CNs. If *msg* packets fail to fill during a single clock cycle then there is a lag in fresh CN output activity following each *synch_point* along links that cross HN boundaries. CNs repeatedly



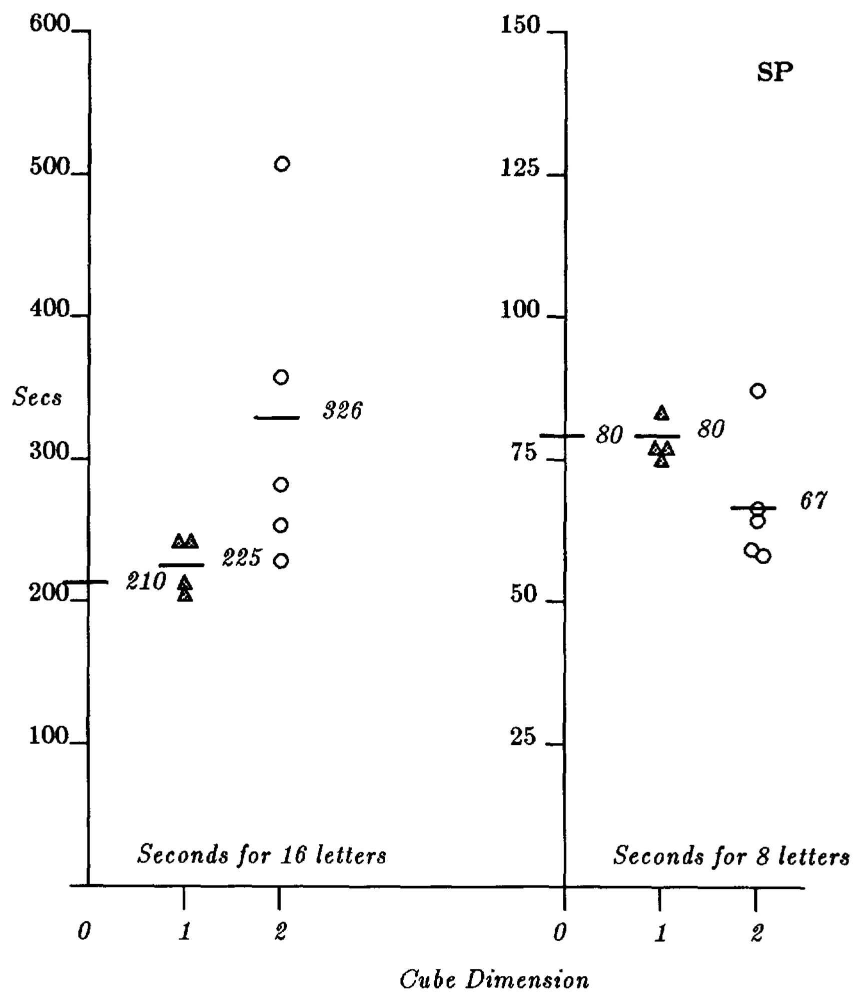
Graph 4.12: Cpc results on the iPSC: AP method.
Tests for each cube dimension are arranged in vertical lines. Each dimension is represented by a different symbol. Horizontal bars mark the average value of tests for each dimension.



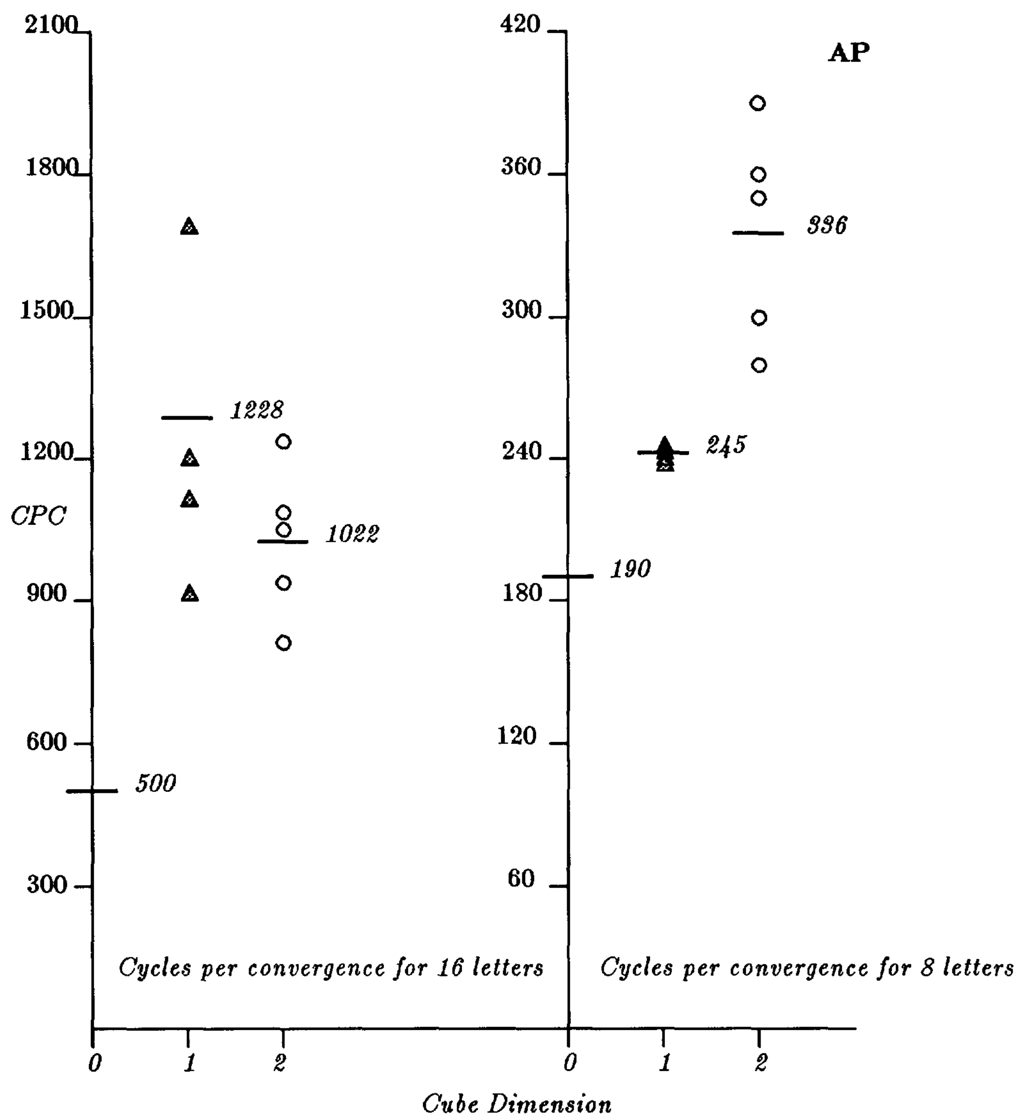
Graph 4.13: Seconds elapsed for cpc tests on the iPSC: AP method. Tests for each cube dimension are arranged in vertical lines. Each dimension is represented by a different symbol. Horizontal bars mark the average value of tests for each dimension.



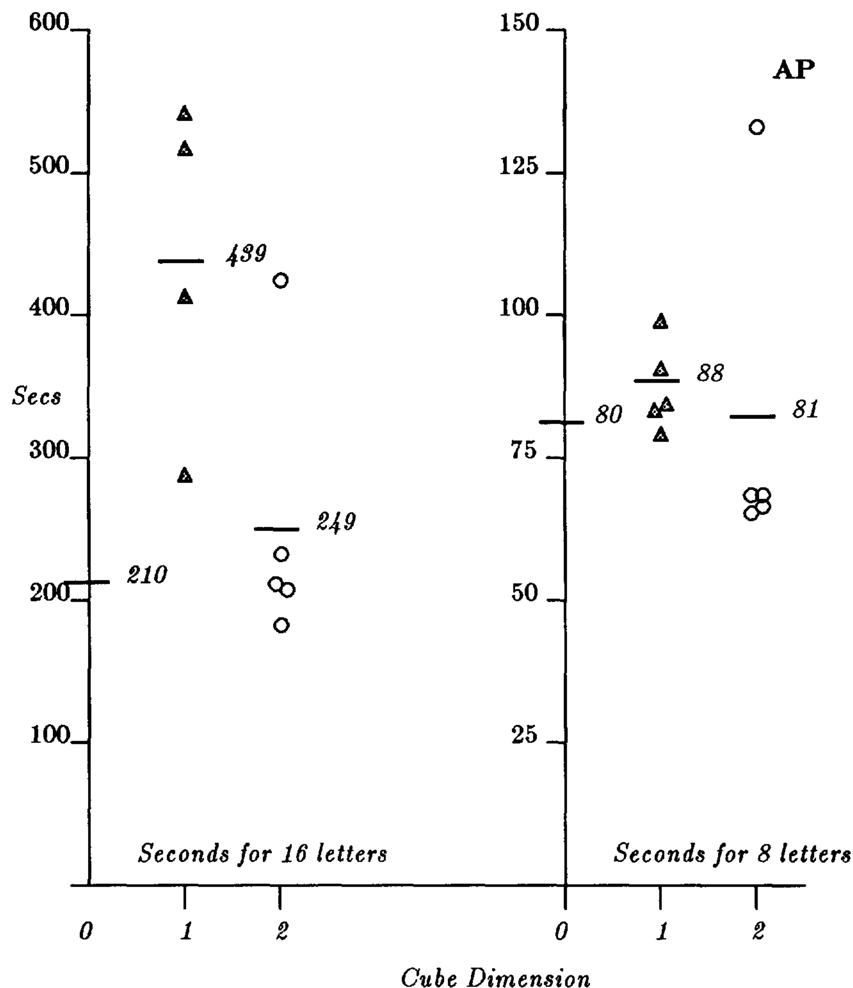
Graph 4.14: Cpc results on the iPSC/2: SP method. Tests for each cube dimension are arranged in vertical lines. Each dimension is represented by a different symbol. Horizontal bars mark the average value of tests for each dimension.



Graph 4.15: Seconds elapsed for cpc tests on the iPSC/2: SP method. Tests for each cube dimension are arranged in vertical lines. Each dimension is represented by a different symbol. Horizontal bars mark the average value of tests for each dimension.



Graph 4.16: Cpc results on the iPSC/2: AP method. Tests for each cube dimension are arranged in vertical lines. Each dimension is represented by a different symbol. Horizontal bars mark the average value of tests for each dimension.



Graph 4.17: Seconds elapsed for cpc tests on the iPSC/2: AP method. Tests for each cube dimension are arranged in vertical lines. Each dimension is represented by a different symbol. Horizontal bars mark the average value of tests for each dimension.

use stale input data from such links and produce new packets with duplicate, stale output values before the first *msgs* arrive. This characteristic of AP *should* be most prevalent with *alfanet* for larger dimensions of the iPSC and iPSC/2, where *msg* packets are slower to fill. But the results from graphs 4.16 and 4.17 seem to indicate that the most detrimental effects on convergence occur for two HNs.

It may be that the over-synchronization of communication between CNs brought on by AP can take better hold in the case of two HNs. For cube dimension 1, previous convergence tests using SP have already shown a high tendency to be

more synchronized. Summary data indicate that the stale *msg* percentage (the percentage of *msgs* that failed to fall within *msg_window*) using 4 HNs is double to triple the percentage when using two. With AP, the rejection of more *msgs* when using more HNs could result in a better mix of values reaching CNs along extra-HN links. This effect would then act to "prime" the network with an increasing variety of output and weight values, which in turn would lead to a more energetic search of the network's solution space.

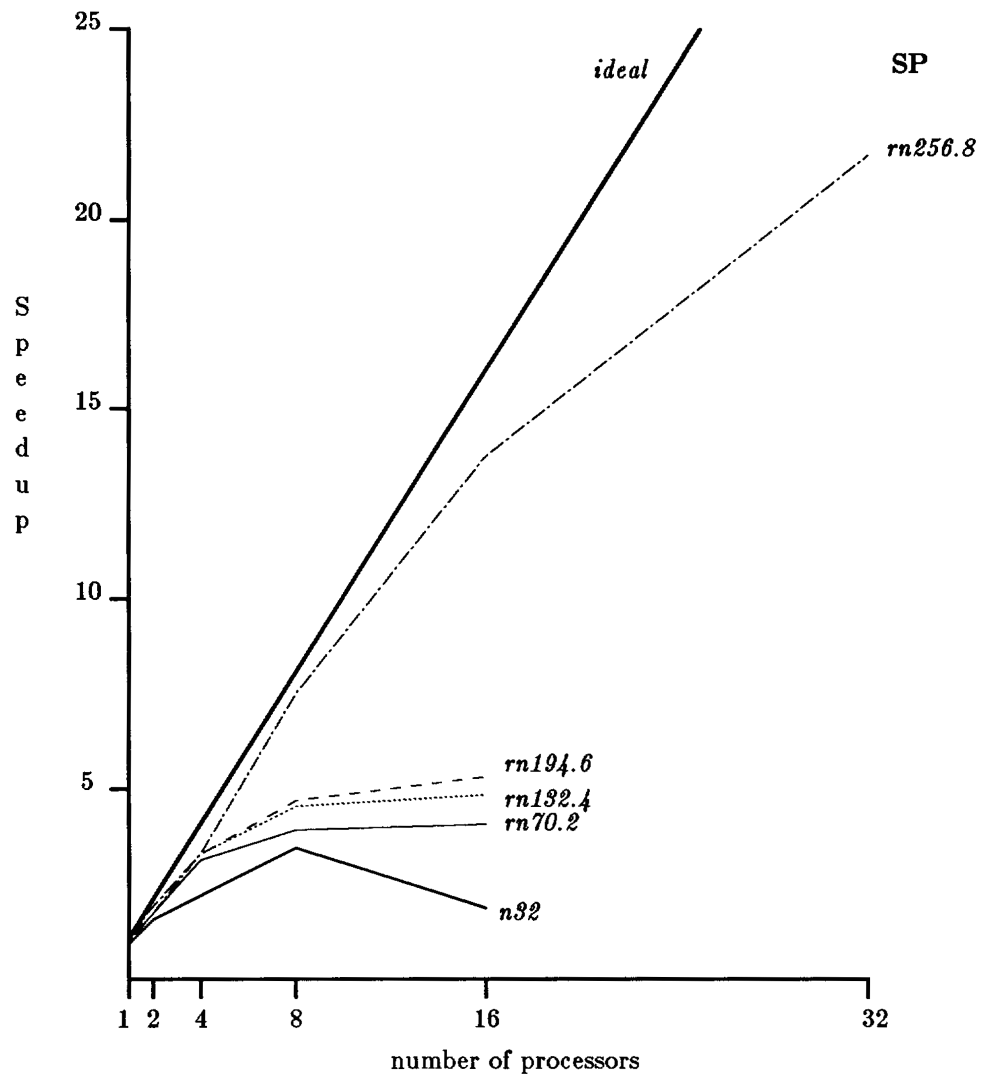
4.5. Network Locality Testing

As noted previously, in the section describing the test networks, the relative communication locality of the receptive field networks is indicated by their suffix number. Thus *rn256.8* has a higher measure of locality than *rn194.6*, etc. All the locality tests were run in a similar fashion as those for the xps measurements for back-propagation networks. That is, a large synchronization interval was chosen so as to obtain near optimal xps performance. Graphs 4.18 and 4.19 show the speedup and processor efficiency for these networks on the iPSC using SP. The results for *n32* are repeated here for contrast with a low locality network containing approximately the same number of links. SP overloads the iPSC inter-HN communications, so results are not shown for most of these networks using 32 HNs.

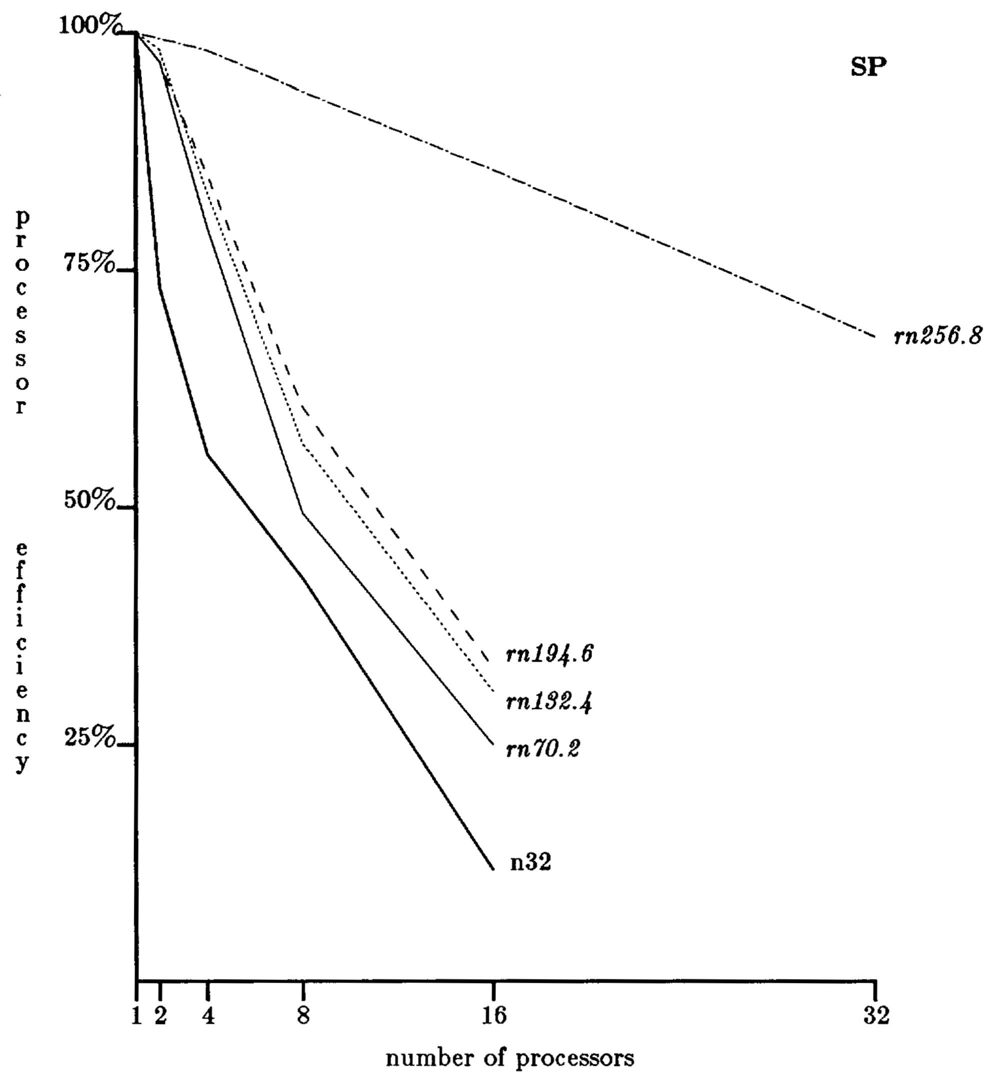
Despite increasing locality, the performance of these networks, with the notable exception of *rn256.8*, is damped by the SP method. The combination of SP and low inter-HN connectivity leads to inefficient utilization of *msg* packets. For example, *rn194.6* sends fewer inter-HN *msgs*, but about the same number of packets as the other, less localized networks. Of course, for *rn256.8* there is no inter-HN communication overhead, so it attains a speedup of about 25 with 32 HNs. The slight dropoff in its speedup line is a result of the highly sequentialized operation of 32 HNs synchronizing with the host processor relative to the perfectly parallel operation of the network itself. A very large *synch_count* would eliminate this effect.

The AP method better exhibits the effects of network locality on peak xps performance. In general, graph 4.20 demonstrates better than a three-fold increase in speedup for the larger dimensions of the iPSC. Furthermore, the differential in speedups between networks of different localities is highlighted for 32 HNs. Even the back-propagation network, *n32*, obtains about 50% processor efficiency for cube dimension 5.

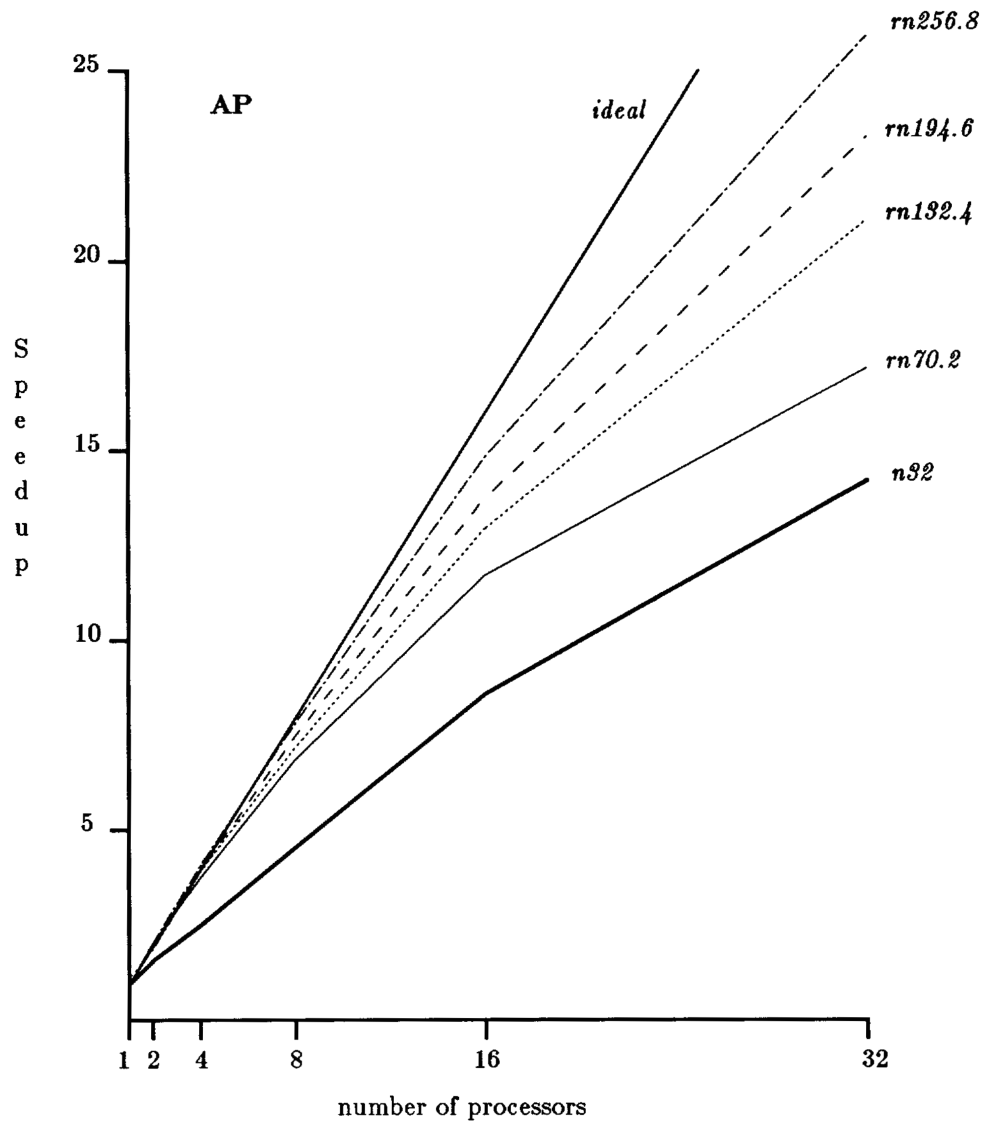
The limited number of locality tests that were accomplished on the iPSC/2 (graph 4.22) illustrate the advantage in raw performance that AP offers over SP, even here where inter-HN communication performance is less a problem than on the iPSC. But this data is inconclusive as to the effects of network locality for larger cube dimensions.



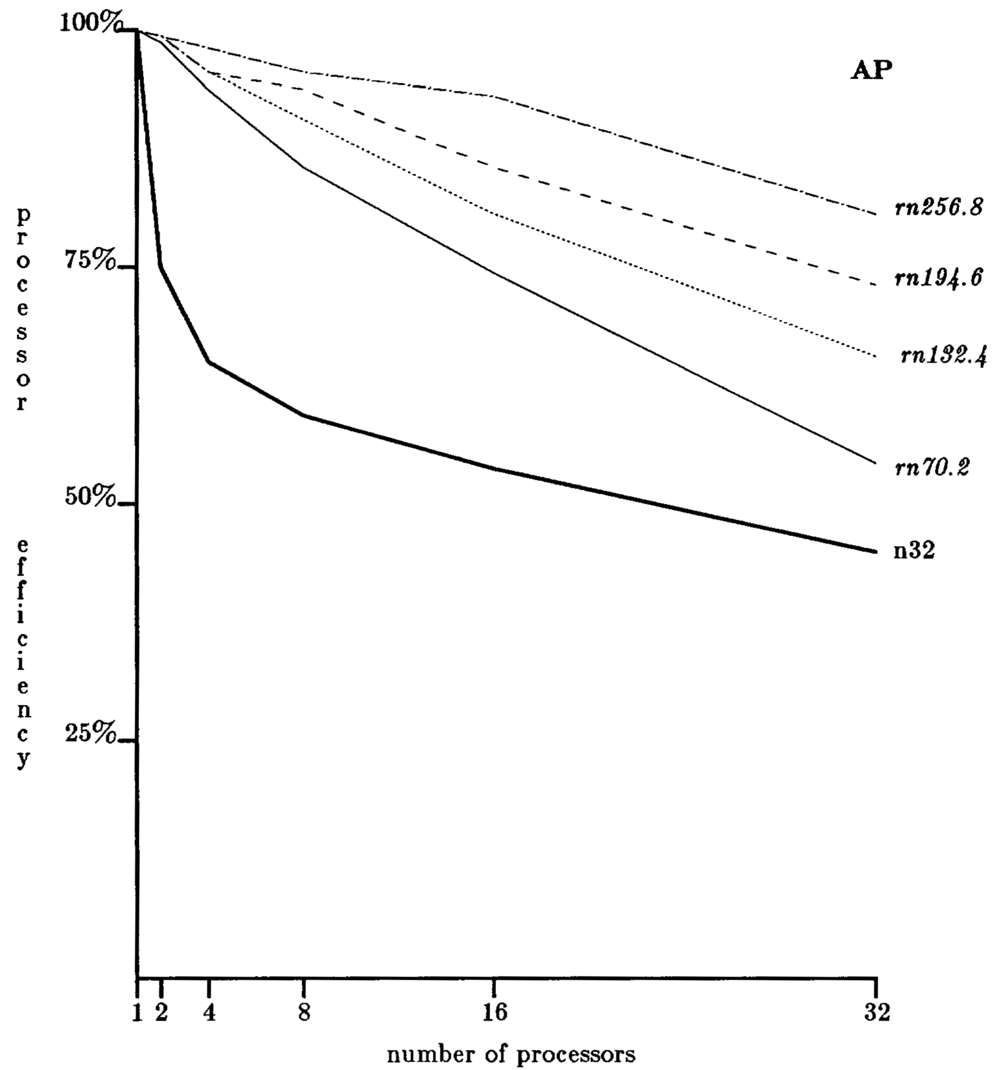
Graph 4.18: Speedup vs. locality on the iPSC: SP method.



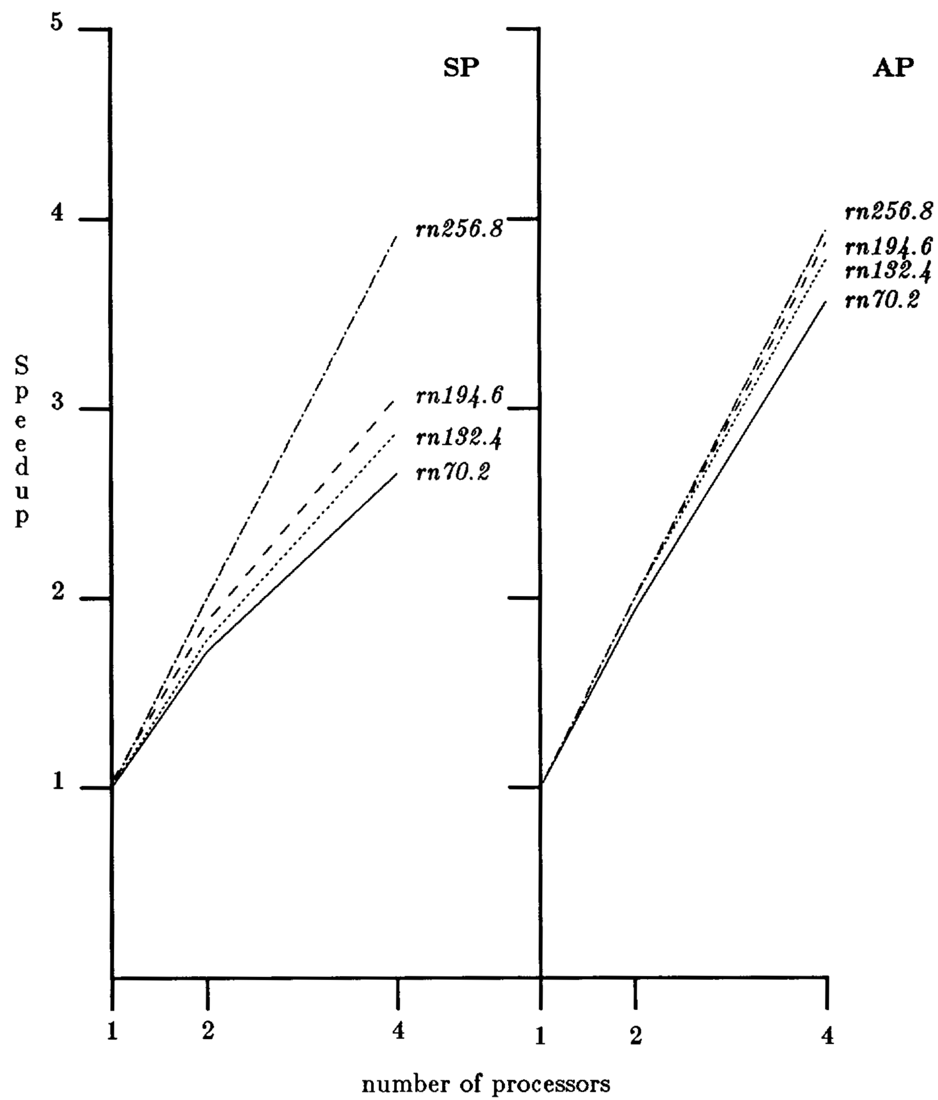
Graph 4.19: Processor efficiency vs. locality on the iPSC: SP method.



Graph 4.20: Speedup vs. locality on the iPSC: AP method.



Graph 4.21: Processor efficiency vs. locality on the iPSC: AP method.



Graph 4.22: Speedup vs. locality on the iPSC/2

CHAPTER 5

Conclusion

5.1. Summary

This thesis describes ANNE, a parallel, general-purpose neural network simulator designed and implemented to run on versions of the Intel Scientific Hypercube computer. ANNE models neural networks internally as connection node (CN) objects with distributed sub-arrays of the weight matrices commonly used in neural network representations. A unique scheme of synchronization between Hypercube nodes (HNs) is employed, which exploits the fault-tolerance of distributed neural network models.

The memory and inter-processor communication limitations of the iPSC were discussed, which led to three main design assumptions about ANNE. These assumptions concerned the maximum size of networks to simulated, the homogeniety of inter-processor communications, and the fault-tolerance of the neural networks for which ANNE was to be used. Subsequently, two existing synchronization schemes for distributed event-driven simulations, Misra's algorithm and Time Warp, were discussed and found to be unsuitably costly and complex for general-purpose neural network simulations. An alternative message-driven synchronization technique, as used in ANNE, was described, including two variations in inter-CN message packaging: synchronous and asynchronous packetization, known respectively as SP and AP.

Following the discussion of ANNE's design considerations, the specifics of ANNE's implementation were covered. This discussion was prefaced by a brief description of BIF, an intermediate structural specification of neural networks used by several of the CAP group's simulators. The details of BIF loading and parsing in ANNE were examined, followed by a discussion of the procedures and structures used to effect inter-CN and inter-HN communication and synchronization. Last was a description of ANNE's user interface, from the level of writing neural network simulation code to ANNE's command level during simulation runtime.

The three main design assumptions presented in Chapter 2 proved to be somewhat optimistic as they relate to the iPSC. The xps tests illustrated that, in practice, the upper size limit for neural networks in ANNE is ~2,000 links per HN. The projected limit was ~7,500 links per HN. Two miscalculations contributed to this shortfall. One, the amount of extra object code contained in the library routines loaded by the iPSC along with ANNE's HN image was underestimated. Second, empirical study revealed that larger networks required more than twice the normal allotment of iPSC message buffers at runtime, especially when distributed over large dimensions of the iPSC. Both of these factors subtracted from the projected amount of memory available for network storage.

The assumptions concerning the homogeniety of inter-HN communication and the fault-tolerance of neural networks in the face of noisy input data appear to have been more well-founded. Both assumptions greatly eased the otherwise complex implementation of ANNE on OGC's hypercube. Moreover, the stress tests with back-propagation models on ANNE verify the value of the communication and synchronization techniques, which depend on these assumptions. Although the test results are poor in some respects, this is not totally unexpected given that the techniques employed in ANNE are covering much new ground.

Chapter 4 presented the test suite used to explore performance characteristics of ANNE and the results of those tests. Several variations of a back-propagation network model were used. Both the SP and AP methods were compared. Tests on the iPSC were complemented with preliminary testing of a direct porting of ANNE to the iPSC/2.

The basic results from this testing suggest that SP is not particularly well-suited, in terms of raw performance, for networks with either a high or low interconnection density. ANNE's synchronization scheme in SP mode does show promise relative to the convergence behavior of back-propagation, especially when inter-HN communication costs are minimized as with the iPSC/2. For the AP method, the converse is true. AP greatly enhances ANNE's raw performance, most notably for larger dimensions of the iPSC. But, with AP, convergence behavior is little improved, and in many cases it is made more erratic.

In short, the combination of back-propagation and ANNE yielded some complex results that are not easily explained. The information capacity of back-propagation models relative to their size and configuration is still being explored. Indeed, the relationship of synchronicity, network size, and convergence behavior is important to many network models and often not fully understood.

It is significant, however, that a poor locality, highly synchronous network was able to converge under ANNE's simplified, semi-synchronous, inter-CN communication model. ANNE has clearly demonstrated the robustness of neural network models in taking on such a challenging network model for multi-processing. Moreover, research with ANNE has contributed a new direction in thinking about the synchronization of fine-grained, massively-parallel models of computation without the need for complex event-driven simulation techniques. Important work remains in testing ANNE with more recent network models that feature more localized connectivity and asynchronous operations between network nodes. BIF models do not, as yet, exist for the networks being derived from research in the neurobiological sciences. However, ANNE offers a unique system for internally structuring such models in a general way.

5.2. Future Work

The most rudimentary improvements to ANNE simply involve completing the implementation of the runtime commands listed in Chapter 3. Of these the most useful would be the **makemap** command, allowing the specification of CN maps at runtime. Other basic functional improvements to ANNE include: (1) a buffer management scheme for loading BIF sub-files larger than the pre-allocated buffer

space for each HN, (2) giving the user the ability to create her own set of runtime parameters that can be manipulated at the host level, and (3) a graphic representation of the data in the network, especially the link weights. The latter feature might be implemented through a scripting mechanism in ANNE that would periodically write weight values to a history file to be later "played back" through a high-resolution terminal.

A necessary feature to add is the ability to re-load the network procedure at runtime. Currently, the user must exit ANNE, revise and compile her network code, link this code with ANNE, and re-start the simulator. Runtime re-loading of the network procedure might be accomplished by pre-allocating a buffer for the user's object code and providing entry and return linkage to this code. The location of ANNE's system calls and local CN table could be passed as parameters via a special routine called from the user's *Init_user_fx* routine.

A more fundamental change to ANNE, as mentioned in Chapter 2, involves replicating every structure of CN global data, contained in each local CN table entry, across all HNs. This change to ANNE's data model needs to be optional, since its benefits are greatest for networks with low locality. Replication of CN data would significantly reduce the number of *msgs* produced at each CN, from one for every output link to one per CN. However, the single CN output message produced under this scheme is broadcast to all HNs in the network. For networks exhibiting high locality and sparse interconnect, which was our main goal, ANNE's current communication scheme is appropriate.

In order to use ANNE more effectively for networks with poor locality, an alternative to global CN replication would be to allow more than one network to share the host processor simultaneously. In particular, when using large synchronization intervals, the host processor sits idle for long periods with a single network. Multiple networks could share the hypercube and more fully utilize the host. Each network would be allocated to a unique set of HNs. The status of each network could be maintained in network control blocks, much as a time-share operating system maintains multiple job control blocks. Or, a simpler approach would replicate the same network over many HNs, giving each its own set of training vectors. The weight changes in each network could then be combined into a single trained network. A relatively new technique of simulating a back-propagation network in a systolic fashion, using linear arrays of processors, has proved to be very effective [PGT88].

ANNE's iPSC version should be viewed as a developmental prototype, which has provided valuable research toward new semi-synchronous, fine-grain simulation techniques. Its raw performance on the iPSC, using 32 HNs, while about three-fold that of an unadorned network program on a VAX 11/780, is 2 orders of magnitude less than that claimed by the latest SAIC machine or Hecht-Nielsen Corporation's ANZA+. Of course, SAIC's and HNC's solution to neural network simulation is done with DSP hardware, and is tailored for full matrix network models whose weight matrices can be highly vectorized. Such networks exhibit poor locality. This type of hardware would do poorly for the sparse connectivity and highly localized cortical models that are included in the long term goals of the CAP group.

Furthermore, this type of hardware does not have the system overhead contained in ANNE's software. For instance, there are numerous consistency checks still done in ANNE related to network structure and CN communications that could be made optional or eliminated. The inter-HN communication bottleneck must be viewed, though, as the greatest hindrance to ANNE's performance in this multiprocessor implementation, especially for low-locality networks. It is unclear how new generation, biologically-based network models will affect the difference in performance numbers between an improved version of ANNE and the current hardware solutions to neural network simulation.

A modified version for the iPSC/2 (and beyond) should be developed. With optimization, ANNE's shortcomings in performance and accuracy will be less significant. The communication-bound nature of large networks on ANNE currently can be relieved by exploiting two major improvements in the iPSC/2: inter-processor messages can be of unlimited length, and messages over ~1,500 KB achieve the shortest latency period. ANNE's *msg* packetization techniques must be made more sophisticated to take advantage of larger packet sizes as the data using AP has shown. In addition, ANNE on the iPSC/2 could handle much larger networks and realize further performance enhancements through the use of available vector boards, faster floating point co-processors, and a high-speed node I/O system that is under development.

References

[BHJ88] Bahr, C., Hammerstrom, D. and Jagla, K., *Concurrent Neural Network Simulation: Two Examples Within A Single, Integrated Neural Network Hardware Development Environment*, IASTED Applied Simulation and Modelling Conference, Galveston, Texas, May 1988.

[Bah88] Bahr, C., "ANNE User Manual," Tech. Report CS/E-88-029, Dept. of Computer Science/Engineering, Oregon Graduate Center, Beaverton, OR, 1988.

[BaH88] Bailey, J. and Hammerstrom, D., *Why VLSI Implementations of Associative VLCNs Require Connection Multiplexing*, IEEE International Conference on Neural Networks, San Diego, CA, July 1988.

[Bai88] Bailey, J., "A VLSI Interconnect Structure for Neural Networks," Ph.D. Dissertation, Dept. of Computer Science & Engineering, OGC, 1988. In preparation.

[Bro86a] Brown, C., "Hopfield's Nerve Nets Realize Biocomputing," *Computer Engineering*, April 1986.

[Bro86b] Brown, C., "Neural Circuits Will Improve Vision, Speech Systems," *Computer Engineering*, April 1986.

[Fan86] Fanty, M., *A Connectionist Simulator for the BBN Butterfly Multiprocessor*, Department of Computer Science, Univ. of Rochester, NY, January 1986.

[Ham86] Hammerstrom, D., "A Connectivity Analysis of Recursive, Auto-Associative Connection Networks," Tech. Report CS/E-86-009, Dept. of Computer Science/Engineering, Oregon Graduate Center, Beaverton, Oregon, August 1986.

[HSA84] Hinton, G. E., Sejnowski, T. J. and Ackley, D. H., "Boltzmann Machines: Constraint Satisfaction Networks that Learn," Technical Report CMU-CS-84-119, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA 15213, May 1984.

[Hin87] Hinton, G. E., "Connectionist Learning Procedures," Tech. Rep. CMU-CS-87-115, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, PA, June 1987.

[Hop82] Hopfield, J. J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proc. Natl. Acad. Sci. USA,* vol. 79(April 1982), pp. 2554-2558.

[HoT] Hopfield, J. J. and Tank, D. W., "Computing with Neural Circuits: A Model," *Science*, vol. 233.

[Int85] Intel, iPSC User's Guide, October 1985.

[Jag88] Jagla, K., "A Broadcast Hierarchy Simulator for the Intel iPSC," CSE Technical Report, Oregon Graduate Center, Department of Computer Science/Engineering, Beaverton, OR, March 1988. In preparation.

[JeS82] Jefferson, D. and Sowizral, H., "Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control," Tech. Rep.-83-204, Univ. of

Southern California, Computer Science Dept., December 1982.

[Joh88a] Johnson, M. A., "NDL User's Manual," CSE Technical Report, Oregon Graduate Center, Department of Computer Science/Engineering, Beaverton, OR, July 1988. In preparation.

[Joh88b] Johnson, M. A., "NDL Reference Manual," CSE Technical Report, Oregon Graduate Center, Department of Computer Science/Engineering, Beaverton, OR, July 1988. In preparation.

[LaF86] Lapedes, A. and Farber, R., "Programming a Massively Parallel, Computation Universal System: Static Behavior," LA-UR-1179, Los Alamos National Laboratory, Los Alamos, NM, April 1986.

[Lin88] Linsker, R., "Self-organization in a Perceptual Network," *IEEE Computers*, vol. 21, 3 (March 1988), pp. 41-51.

[LyB86] Lynch, G. and Baudry, M., *Structure-function Relationships in the Organization of Memory*, Center for the Neurobiology of Learning and Memory, Univ. of California, Irvine, 1986.

[May88a]
May, N., "Fault Simulation of a Wafer-Scale Neural Network," Tech. Report CS/E-88-020, Dept. of Computer Science/Engineering, Oregon Graduate Center, Beaverton, Oregon, May 1988.

[May88b]
May, N., "FltSim Detailed Description and Operation," Tech. Report CS/E-88-021, Dept. of Computer Science/Engineering, Oregon Graduate Center, Beaverton, Oregon, May 1988.

[MRH86a]
McClelland, J. L., Rumelhart, D. E. and Hinton, G. E., "The Appeal of Parallel Distributed Processing," in *Parallel Distributed Processing*, vol. 1, P. R. Group (ed.), 1986.

[MRH86b]
McClelland, J. L., Rumelhart, D. E. and Hinton, G. E., "Distributed Representations," in *Parallel Distributed Processing*, vol. 1, P. R. Group (ed.), 1986.

[MeM88] Mead, C. A. and Mahowald, M. A., "A Silicon Model of Early Visual Processing," *Neural Networks*, vol. 1, 1 (1988), pp. 91-97.

[Mis86] Misra, J., "Distributed Discrete-Event Simulation," *Computing Surveys*, vol. 18, 1 (March 1986), .

[Nug88] Nugent, S. F., *The iPSC/2 Direct-Connect Communications Technology*, Intel Scientific Computers, Inc., 1988.

[Pla87] Plate, T., "A design for the simulation of connectionist models on coarse grained parallel computers.," MCCS-87-106, Computing Research Laboratory, Dept. 3CRL, New Mexico State Univ., November 1987.

[PGT88] Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S. and Kung, H. T., *Neural Network Simulation at Warp Speed: How We Got 16 Million Connections Per Second*, IEEE International Conference on Neural Networks, San Diego, CA,

July 1988.

[RuG86] Rumelhart, D. E. and Group, J. L. M. P. R., in *Parallel Distributed Processing*, vol. 1, 1986.

[SeR] Sejnowski, T. J. and Rosenberg, C. R., "NETtalk: A Parallel Network that Learns to Read Aloud," Tech. Rep. JHU/EECS-86/01, John Hopkins University.

[SSB83] Small, S. L., Shastri, L., Brucks, M. L., Kaufman, S. G., Cottrell, G. W. and Addanki, S., "ISCON: A Network Construction Aid and Simulator for Connectionist Models," Tech. Rep. 109, Univ. of Rochester, Dept. of Computer Science, April 1983.

[Wor88] Works, G. A., *The Creation of Delta: A New Concept in ANS Processing,* 1988 ICNN Proceedings, San Diego, Calif., 1988.