A Scalable Parallel Formulation of the Backpropagation Algorithm for Hypercubes and Related Architectures *

Vipin Kumar, Shashi Shekhar and Minesh B. Amin
kumar@cs.umn.edu shekhar@cs.umn.edu amin@cs.umn.edu
Department of Computer Science, University of Minnesota, Minneapolis, MN - 55455

Abstract

In this paper, we present a new technique for mapping the backpropagation algorithm on hypercubes and related architectures. A key component of this technique is a network partitioning scheme which is called *checkerboarding*. Checkerboarding allows us to replace the all-to-all broadcast operation performed by the commonly used vertical network partitioning scheme, with operations that are much faster on the hypercubes and related architectures. Checkerboarding can be combined with the pattern partitioning technique to form a hybrid scheme which performs better than either one of these schemes. Theoretical analysis and experimental results on nCUBE2^{TM}[†] and CM5^{TM}[‡] show that our scheme performs better than the other schemes, both for uniform and non-uniform networks.

1 Introduction

The Backpropagation algorithm (BP)[1] is one of the most popular neural network learning algorithms. It has been used in a large number of applications [2, 3, 4, 5]. This algorithm is computation intensive and as a result there has been a great interest in developing parallel formulations of this algorithm for a variety of parallel computers.

BP can be parallelized either by network partitioning or by pattern partitioning. In network partitioning schemes, nodes and weights of the neural network are partitioned among different processors and thus the computations of node activations, node errors and weight changes are parallelized [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. In pattern partitioning, individual weight changes due to various learning patterns are computed concurrently [17, 18, 19, 20]. Pattern partitioning and network partitioning can also be combined [6, 11, 12, 14, 20, 21] to form hybrid schemes. Several machine architectures including linear arrays, meshes and hypercubes have been explored to implement parallel BP.

In this paper, we present a new technique for mapping the backpropagation algorithm on hypercubes and related architectures. A key component of this technique is a network partitioning scheme which is called *checkerboarding*. The major communication-intensive operation in the commonly used vertical sectioning scheme [6, 9, 10, 11, 12, 13, 14, 15, 17, 22] is the all-to-all broadcast. In this operation, each of the P processors has to broadcast its local m units of information to all the other processors. This takes $\bigcirc(mP)$ time on linear array, on mesh *as well as* on hypercube. Hence, the vertical sectioning scheme performs equally well (or equally poorly) on all these architectures.

*This work was supported by Army Research Office grant # 28408-MA-SDI to the University of Minnesota, by a grant from the Graduate School of University of Minnesota and by the Army High Performance Computing Research Center at the University of Minnesota. The authors are listed in no particular order.

†nCUBE2 is a trademark of the Ncube corporation

‡CM5 is a trademark of the Thinking Machines corporation

The checkerboarding scheme allows us to replace the all-to-all broadcast operation by concurrent non-interfering single source broadcasts, which are much faster on hypercubes. Furthermore, our method can use a larger number of processors without incurring higher communication costs than the vertical sectioning scheme, and shows strictly better performance on hypercubes with more than 16 processors. Our scheme also performs better than a pattern partitioning scheme for a large class of problems. In addition, our scheme can be combined with the pattern partitioning scheme to form a hybrid scheme which performs better than either one for a wider variety of cases. Although the scheme is natural for the hypercube architecture (e.g., nCUBE, Intel iPSC^{TM}[§]), it is equally suitable for Fat-tree based architectures such as CM5. Our scheme is applicable only to fully connected networks; i.e., each node in a layer (except the output layer) is connected to all nodes in the next layer. The number of nodes in each layer can be the same (uniform network) or different (non-uniform network).

Experimental results on nCUBE2^{TM} and CM5^{TM} show that our scheme performs better than the other schemes for both uniform and non-uniform networks. Furthermore, it provides very high overall performance on existing commercial parallel computers. For example, an unoptimized version of our hybrid parallel formulation on a 256 processor CM5 (without vector units) performs over 50 million weight changes per second (or over 160 million connections per second) for non-uniform networks. A summary of results of this paper appear in [23].

The checkerboarding scheme has been used in other parallel formulations of BP for the mesh architecture [21] and systolic arrays [16]. But if these formulations are mapped onto hypercube, then their performance approaches that of vertical sectioning (in the case of [16]) or the hybrid of vertical sectioning and pattern partitioning (in the case of [21]). The communication pattern used in our scheme is significantly different from those of other checkerboarding schemes[16, 21], leading to improved performance on hypercubes[¶].

Most previous parallel formulations of BP on hypercube or on related architectures used vertical sectioning [15], pattern partitioning [18, 19], or a hybrid of vertical sectioning and pattern partitioning[11, 12]. For hypercubes, the authors are aware of only one exception [8] in which each node and weight of the BP network is mapped onto a separate processor of CM2^{TM}[‖]. As discussed in [11, 12], this method incurs too much communication overhead.

The paper is organized as follows. Section 2 overviews the serial BP algorithm and its existing parallel formulations. Section 3 discusses existing parallel formulations of BP for hypercube and presents our new parallel formulations. Section 4 contains the theoretical and experimental comparison of checkerboarding, vertical sectioning and pattern partitioning. Section 5 describes a hybrid scheme combining checkerboarding with pattern partitioning, along with experimental results. Section 6 shows the performance of the proposed schemes on non-uniform networks. Section 7 considers scalability and optimality issues. Section 8 contains concluding remarks.

[§] iPSC is a trademark of the Intel Scientific Computers

[¶] In Kung's scheme[16], multiplication of a $I \times I$ matrix to a $I \times 1$ vector takes $\bigcirc(I)$ time on a $I \times I$ mesh-connected systolic array. In contrast, the same computation using our scheme takes $\bigcirc(\log(I)^2)$ time on a hypercube of $(\frac{I}{\log(I)} \times \frac{I}{\log(I)})$ processors. The real benefit of Kung's scheme occurs only when many matrix-vector multiplications are pipelined. This is possible only with the *set-training* regime, and is most effective when the number of training patterns is $\bigcirc(I)$ (where I is the number of inputs). In particular, with the per-pattern training regime, Kung's scheme[16] is strictly inferior to our scheme on hypercube, *as well as* to adaptation of our scheme on a mesh computer with worm-hole routing.

[‖] CM2 is a trademark of the Thinking Machines corporation

2 Backpropagation Learning Algorithm

BP trains a given feedforward neural network for a given set of learning patterns. The trained network is able to map the input portion of each learning pattern to the output portion of the same learning pattern. The training of a neural network can be viewed as discovering values for its weights.

To simplify the discussion, we focus on a uniform multilayered feedforward neural network, in which the number of nodes, I, in each layer is the same. We discuss the case of non-uniform networks in section 6. Let J be the number of layers, and I be the number of nodes per layer. Figure 1 shows a uniform network with 3 layers (J = 3) and 3 nodes per layer (I = 3). The layers are labeled from 1 through J. The input layer is labeled 0, and is not counted in J. Each node i in layer j is associated with an activation value a_i^j and an error value δ_i^j. The activation values associated with the nodes in a layer form an activation vector. Z1, Z2 and Z3 represent the activation vectors for the different layers for a given pattern in Figure 1. Similarly, the error vectors associated with different layers are represented by E1, E2 and E3. A node i in layer j is connected to all nodes at layer $j-1$ and $j+1$. A weight, $w_{i,k}^{j,j+1}$ is associated with the connection from node i in layer j to node k in layer $j+1$. The weight matrix is shown in the right half of Figure 1. There are JI^2 non-zero weights in the network. The non-zero entries of the weight matrix are grouped as submatrices U1, U2 and U3. Each submatrix represents the weights on various edges between the nodes in adjacent layers. For example, U1 represents the weights on edges from the nodes in input layer to the nodes in layer 1. U2, and U3 are defined similarly.

BP has two phases: *Forward Phase*, and *Backward Phase*. In the forward phase, the input portion of a learning pattern is fed to the network. It is propagated through the layers to compute the activations of the nodes in each layer. The difference between the activation of nodes in the highest layer and the output portion of the learning pattern defines error for the nodes in the highest layer. In the backward phase, the error is propagated backwards from the highest layer to the lower layers to compute the errors assignable to nodes in various layers. This computation is accompanied by the computation of weight changes that will reduce the total error at the nodes in the highest layer.

The forward and backward phases are carried out for every learning pattern. The error in the nodes at the highest level is accumulated over all the patterns. The computed weight change for each weight is also accumulated over all patterns, and each weight in the network is changed by the accumulated weight changes. This completes one iteration for all patterns. Now, if the accumulated error for nodes in the highest level is above a given target, this iteration is repeated again. BP terminates when the accumulated error in the nodes at the highest layer falls below a given target. The details of BP are shown in Figure 6 in Appendix A.

The bulk of the computations done by BP can be viewed as a sequence of matrix-vector multiplications, as shown in Figure 1. The input portion $Z0$ of a learning pattern is presented to the nodes in layer 0. The activation vector $Z1$ is computed by multiplying $Z0$ with the weight matrix $U1$[1]. This process is repeated for computing $Z2$ (= product[$U2$, $Z1$]), and $Z3$ (= product[$U3$, $Z2$]) thus completing the computation of activation vectors for each layer. This is followed by error computation for the highest layer (layer 3) and the backward phase. The error vector $E3$ is computed as the vector difference between the output portion of the learning pattern and $Z3$. The error vector $E2$ is computed by multiplying $E3$ with $U3$, i.e. $E2$ = product[$E3$, $U3$][2]. This process

[1]The activation vector computation is somewhat higher than the simple matrix-vector product. The exact expression used is $Z1$(i) = f(product[$Z0$, $U1$](i)), where is f is a sigmoidal function.

[2]The error vector computation differs from the simple vector matrix product. The exact expression used is $E2$(i)

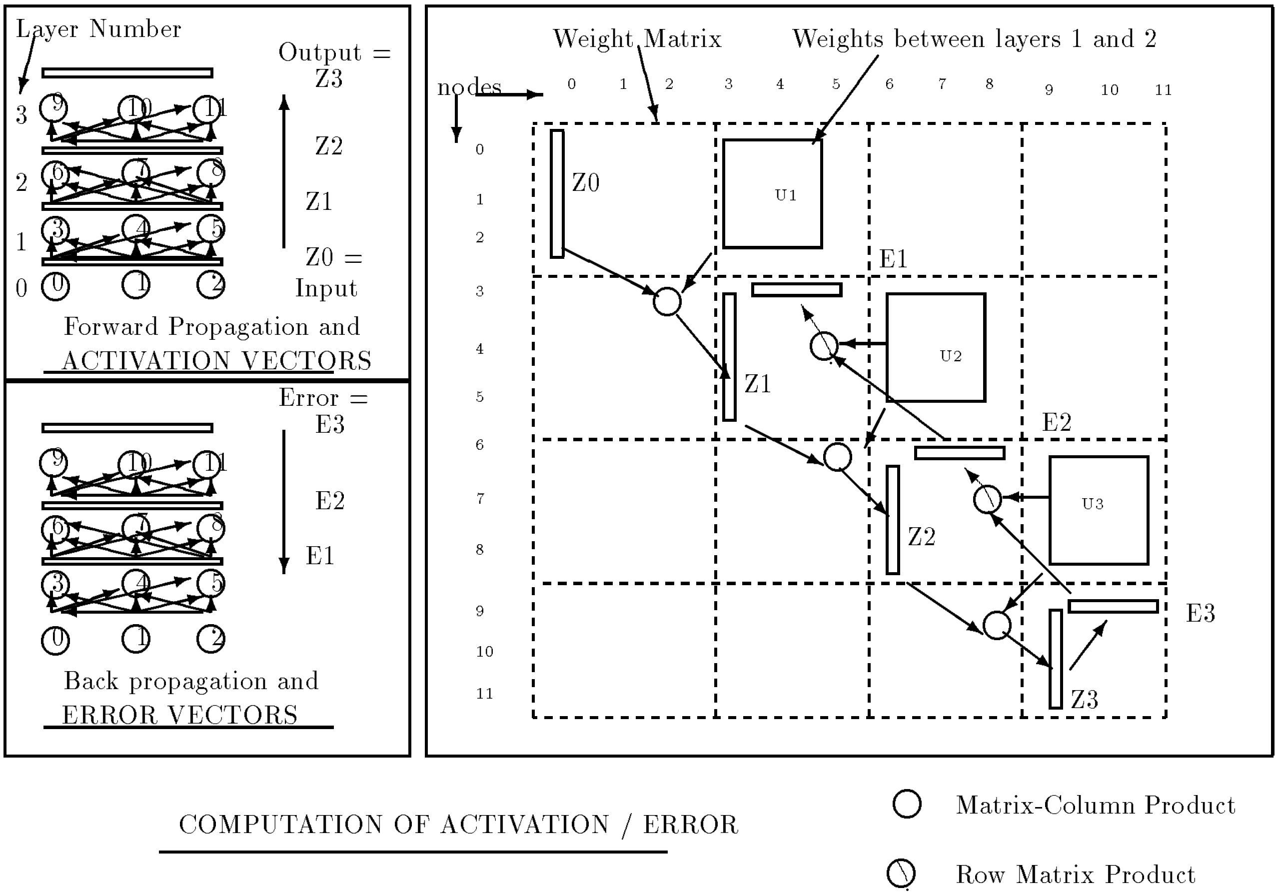

Figure 1: Backpropagation as a sequence of matrix-vector operations

is repeated to compute the error vectors for other layers, i.e. $E1$ = product[$E2$, $U2$]. The backward phase also computes changes in the weights to reduce errors. The weight changes are computed by evaluating the cross products of activation and error vectors. For example, the weight changes for elements of $U3$ are computed via Δ U3 = cross_product[$Z2$, $E3$]. The cross_product of two vectors is defined in the following manner: $\Delta U3[i,j]$ = $Z2$(i)*$E3$(j), where the subscript i, and j refer to the node numbers given in Figure 1. The weight change computation for weights in other layers are carried out in a similar manner; e.g. $\Delta U2$ = cross_product[$Z1$, $E2$], and $\Delta U1$ = cross_product[$Z0$, $E1$].

The dominant computation per iteration is represented by the sequence of matrix-vector operations described above. The overall runtime for each iteration of BP can be approximated as LJI^2t_c, where I^2t_c is the computation cost of all matrix-vector operations associated with each layer for one pattern, and L represents the number of learning patterns. This cost model can be used to analyze and compare the parallel formulations of the backpropagation in spite of the following simplifications. BP does more computation than the sequence of matrix-vector operations depicted in Figure 1. For example, it computes sigmoidal functions and the magnitude of the error vector for the highest layer. It adjusts weights after examining the entire set of learning patterns

= (1 - $Z2$(i))*$Z2$(i)*product[$E3$, $U3$](i) as shown in Figure 6. That is, the i^{th} element of the vector product[$E3$, $U3$] is multiplied by a function of the activation of the node (i.e. by (1 - $Z2$(i))* $Z2$(i)).

in each iteration. Lastly, the number of matrix-vector products in the forward phase is one more than the number of matrix-vector products during the backward phase.

2.1 Training Regimes

In set-training regime described in Appendix A, the weight changes for all the learning patterns are accumulated before changing the weights. Many neural network software (e.g. [24]) support an alternative training regime, called per-pattern training. In this training regime, the set of patterns are permuted to create an ordering among the learning patterns. The patterns are then examined in sequence. Weight changes are computed for a pattern, and the weights are changed before examining the next pattern. This semantics prevents parallelization of weight adjustments due to different patterns.

Pattern-partitioning schemes to parallelize BP are not applicable to the per-pattern training regime for neural networks. Network partitioning schemes can, however, be used for both per-pattern training regime and for the set-training regime. We will focus on the set-training regime in this paper.

2.2 Application Domains of Backpropagation

The applications of neural network learning algorithms can broadly be divided into two groups: recognition and generalization[3] [3]. In recognition problems, a set of learning patterns are used to train the network. The trained network is expected to recall the output part of a previously seen learning pattern, when the input portion of the same learning pattern is presented. Pattern recognition applications[25] are examples of recognition problems. On the other hand, in generalization problems neural networks are tested with input portions of new testing patterns. The network is expected to predict the output portions of the testing pattern, which have not been shown to the network during the learning phase. Generalization problems include assignment of bond-rating [4] and economic prediction [5].

Usually a large neural network is used in recognition problems. Furthermore, recognition problems have a small number of learning samples. For example, in the protein-folding problem, L = 14 and I = 1000 [25]. The recognition problems can be characterized by L $\ll I^2$. In contrast, generalization problems are characterized by a large number of learning samples. The number of learning samples should be an order of magnitude larger than the number of weights to be learned. This is required for reasonable confidence in the learned weights for generalization over unseen testing patterns. These problems can be characterized by L $> 10 I^2 J$ [3, 4].

2.3 Parallel Formulations of Backpropagation

Here we survey existing schemes to parallelize BP for the fully connected multi-layer networks. In these networks, the adjacent layers are completely connected. For schemes that primarily deal with randomly sparse networks, see [7, 26, 27, 28, 29].

Parallelization schemes for BP can be classified into three broad categories: network partitioning schemes, pattern partitioning schemes, and hybrid schemes. The network partitioning schemes take advantage of the parallelism in the computation of node activations and node errors by distributing both the nodes and the weights of the network among different processors. Nodes can be partitioned in different ways. Complete partitioning assigns one node per processor [6, 8]. Vertical sectioning

[3] Many researchers use alternative categories, namely classification and regression. We choose recognition and generalization categories to establish the dominance regions of alternative parallel formulations.

divides the nodes in a layer among the processors, and each processor gets some nodes from each layer [6, 9, 10, 11, 12, 13, 14, 15, 17, 22].

The weights can be partitioned in four different ways: complete partitioning, inset grouping, outset grouping and checkerboarding. Complete partitioning [8, 16] allocates one processor per weight in the network. It allows maximum concurrency in the computation of the various terms of the node activation and node error. However, it requires communication to accumulate the terms. Inset and outset grouping schemes are often used in conjunction with the vertical-sectioning scheme. Inset grouping schemes form sets, which represent the collection of weights on the incoming edges for a specific node. The inset of a node is allocated to the processor possessing the node. Inset grouping reduces the need for communication in computing node activations during the forward propagation. It has been used in [6, 11, 12, 14]. Outset grouping schemes form sets, which represent the collection of weights on the outgoing edges from a specific node. The outset of a node is allocated to a processor possessing the node. Outset grouping eliminates the need for communication to compute error vectors during the backward phase. It has been used in [15]. Both inset and outset grouping can be used together to improve the efficiency of both forward and backward phases [9, 10, 13]. However, this scheme duplicates each weight on two processors, increasing the work during weight updates. It can also be shown that the communication cost for inset grouping is the same as the communication cost incurred by the replication scheme, through the use of multiply-accumulate-rotate operation [11]. Checkerboard partitions the weights by grouping the rows and columns of the weight matrix. It has been used for systolic arrays [16], and for transputers [21] connected in a mesh configuration. Given enough processors, checkerboarding reduces to a complete partitioning of weights.

Pattern partitioning replicates the network nodes and weights at each processor. It divides the pattern set equally among all processors. Each processor carries out the forward and the backward phase for the local set of patterns. Each processor also accumulates the weight changes according to the local patterns. Then, the processors communicate to accumulate the weight changes for updating weights. This scheme is preferred for problems with a large set of learning patterns on machines supporting efficient broadcast operation. This scheme has been used in [18, 19, 17, 20]. The computation in different layers of BP can be pipelined [6, 14]; i.e., while one pattern is being processed for some layer, a different pattern can be processed for a preceding layer. Even though pipelining partitions the network horizontally, it is more appropriate to view it as pattern partitioning.

Hybrid schemes combine pattern partitioning with network partitioning. For example, pipelining can be combined with vertical sectioning [6, 14]. Other examples include the combination of vertical sectioning with simple pattern partitioning [12]. We can view the forward and backward phase computations for a pattern as a sequence of weight-matrix to activation/error-vector products. The hybrid schemes that process multiple patterns simultaneously can be viewed as a sequence of matrix-matrix products. This allows the possibility of using parallel algorithms for computing a sequence of matrix operations [30, 21].

An orthogonal way of classifying the approaches to parallel BP is based on the architectures on which they are implemented. Complete node partitioning and inset weight partitioning with pipelining have been explored for linear arrays in [14]. The vertical-sectioning of nodes with duplicate inset/outset weight partitioning is used for mesh-connected transputers in [9, 10]. Mesh-based machines have been explored by many researchers. A matrix-matrix multiplication-based formulation is given in [21] for mesh architectures. Other schemes for mesh include those proposed in [13, 6, 16]. Techniques for hypercube and related architectures have been explored in [8, 11, 12, 15, 18, 19].

3 Hypercubes and Backpropagation

In a hypercube-connected parallel computer, each processor is directly connected to $\log(P)$ other processors whose addresses differ by exactly one bit. The time to deliver a message of length m words between two adjacent processors is given by $(t_s + t_w m)$. Here, t_s is the message startup time and t_w is per-word communication time. A message containing m words can be broadcast from a processor to all other processors in time $(t_s + t_w m) * \log(P)$, as a binary tree of depth $\log(P)$ can be mapped on to a hypercube containing P processors [31][4].

We first consider the following two important parallel formulations of BP before describing our new scheme:

- Partitioning of patterns
- Partitioning of Neural Networks by Vertical Sectioning

To simplify the discussion, we defer the discussion of hybrid schemes using both pattern partitioning and network partitioning to section 5.

3.1 Pattern Partitioning

In BP, under the set-training regime (see section 2.1) the computation of weight change for different patterns can be carried out in parallel without requiring any communication. But, the step of accumulating the weight changes requires communication among the processors.

In this scheme, the whole network is replicated on each of the P processors. Each processor then performs learning using $\frac{L}{P}$ patterns. For each iteration of BP, the following steps are performed on each processor:

1. Each processor carries out forward and backward phases to compute the sum of the weight changes due to the $\frac{L}{P}$ local learning patterns. This step takes time $\frac{L}{P}JI^2t_c$ approximately[5], where t_c is the computation cost per weight per pattern for each iteration of the set-training regime.

2. The average weight changes for all L patterns for each of the JI^2 weights is computed. To do this, a binary tree is imposed on P processors. The root of this tree accumulates and computes the average weight changes. This step takes $[t_s + JI^2t_w]\log(P)$ communication time.

3. The average weight changes computed by the root processor are sent back to all processors to compute the new weight matrix at each processor. This step takes $[t_s + JI^2t_w]\log(P)$ communication time.

Hence, the parallel runtime for the BP algorithm for one iteration and for L patterns is given by:

$$\frac{L}{P}JI^2t_c + 2[t_s + JI^2t_w]\log(P)$$

The main features of this scheme are as follows:

[4] Johnson and Ho present faster broadcast algorithms for hypercubes in [32]. These can be used to improve the performance of our new scheme and pattern partitioning. For simplicity, we only consider the standard broadcasting algorithm from [31]. The relative performance of different schemes will be similar even with the new broadcast algorithms.

[5] In steps 2 and 3, when weight changes are accumulated and reset, an addition is done for each weight. We ignore this cost to simplify the discussion.

- It is usable only for set training regime and is not applicable to a per-pattern training regime.
- The maximum number of processors that can theoretically be used by this mapping scheme is equal to the number of learning patterns (L).
- As P approaches L, the computational component ($\frac{L}{P}JI^2t_c$) can become much smaller than the communication time ($2[t_s + JI^2t_w]\log(P)$). Hence, the speed-up will usually peak for $P < L$.

3.2 Network Partitioning using Vertical Sectioning

In a vertical-sectioning scheme, the nodes in the different layers of a uniform network are divided in an identical manner by allocating an equal number of nodes per layer to each processor. The processors also contain the incoming weights associated with the nodes that reside in them. Each processor computes node activations, node errors, and weight changes for one pattern at a time. As an example, consider the scheme given in [11]. A similar scheme is given in [9, 10]. In this

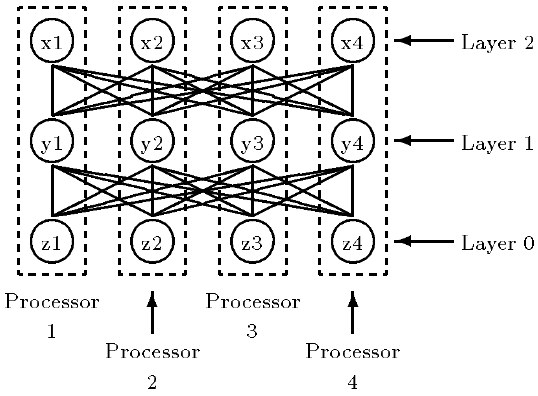

Figure 2: The Vertical Sectioning of the Network

scheme, I nodes of each layer are partitioned among P processors. Thus, each processor computes activations for $\frac{I}{P}$ nodes in each layer for the forward phase. The incoming weights for each of the $\frac{I}{P}$ nodes within each layer are also located at the same processor For an illustration, see Figure 2 which shows the mapping of a network with $I = 4$, $J = 2$ on 4 processors.

In the forward phase, to compute activation values for any node i in layer j, it is necessary to have the activation values of all nodes from layer $(j-1)$. Thus, the *all-to-all* broadcast operation [33], in which each processor needs to send its $\frac{I}{P}$ activations to all other P processors, needs to be performed. This operation takes $P[t_s + \frac{I}{P}t_w]$ for each of the J layers. Note that the run time of this operation remains the same for a linear array, mesh, systolic array or a hypercube[6]. Similar computation and communication takes place during the backward phase. Hence, the parallel runtime for one iteration and L patterns is:

$$\text{Parallel Runtime} = \frac{L}{P}JI^2t_c + 2LJP(t_s + \frac{I}{P}t_w) \tag{1}$$

[6]All-to-all broadcast can be carried out in $\bigcirc(\frac{P}{\log(P)}[t_s + \frac{I}{P}t_w])$ on a hypercube if simultaneous communication along all channels of hypercube is allowed [31].

3.3 Our Parallel Formulation

In the vertical-sectioning schemes, one processor performs the computation related to one or more nodes in each layer. In contrast, in our scheme, a set of processors is used to perform these operations. To do this, the weights related to a node are divided among a set of processors. Furthermore, the processors processing the incoming and outgoing weights for a node are arranged in a special manner to minimize communication costs incurred in the computation of node activations and errors. This increases the degree of concurrency of our scheme over the vertical-sectioning scheme. Our formulation is essentially an adaptation of the matrix vector multiplication scheme for hypercube presented in [31].

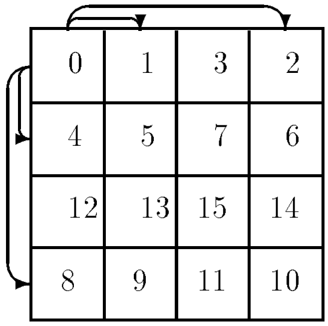

Figure 3: Mapping of 4*4 processor-grid onto a 16-node hypercube

Our scheme uses a well known embedding [34] of a $\sqrt{P}$ x $\sqrt{P}$ grid of processor on a P ($= 2^{2k}$) processor hypercube. The processors in each row and column of the grid form a $2^k = \sqrt{P}$ processor hypercube. Also, the communication interconnection subnetworks in the hypercube in different rows (and different columns) are disjoint. A sample mapping of a 4*4 processor grid onto a hypercube with 16 processors is shown in Figure 3. The processor grid is shown as a matrix with 4 rows and 4 columns. The number inside each grid cell represents the address of the processor in the hypercube. The processors in different rows (or columns) form independent hypercubes.

The partitioning of nodes and weights in our scheme is depicted in Figure 4(a). We partition each I x I weight submatrix (e.g. U1, U2, U3 in Figure 1) onto a grid of $\sqrt{P}$ x $\sqrt{P}$ which is embedded on the P processor hypercube as described above. Thus each processor contains $\frac{I^2}{P}$ weights from each layer. The I nodes for each layer are partitioned among the $\sqrt{P}$ processors that are on the diagonal of the $\sqrt{P}$ x $\sqrt{P}$ grid. Hence, each diagonal processor contains $\frac{I}{\sqrt{P}}$ nodes. Each row of $\sqrt{p}$ processors together contains I incoming weights for each of the $\frac{I}{\sqrt{P}}$ nodes (per layer) in its diagonal element. Similarly, each column of processors together contains I outgoing weights for each of the $\frac{I}{\sqrt{P}}$ nodes (per layer) in its diagonal element. We call our scheme checkerboarding due to the manner in which the weights are partitioned.

The communication needed to compute the activation vectors, error vectors and weight changes are now described. In the forward phase, we need to move the node activation values from the diagonal processor to each row as shown in Figure 4(b). This is done by broadcasting $\frac{I}{\sqrt{P}}$ values along each column (which is a hypercube of $\sqrt{P}$ processors), and this takes $[t_s + \frac{I}{\sqrt{P}} t_w] \log(\sqrt{P})$ time. Now, each row of processors has the node activation values from all nodes (and they already have all incoming weights for the $\frac{I}{\sqrt{P}}$ nodes). Each row of $\sqrt{P}$ processors now collectively computes node

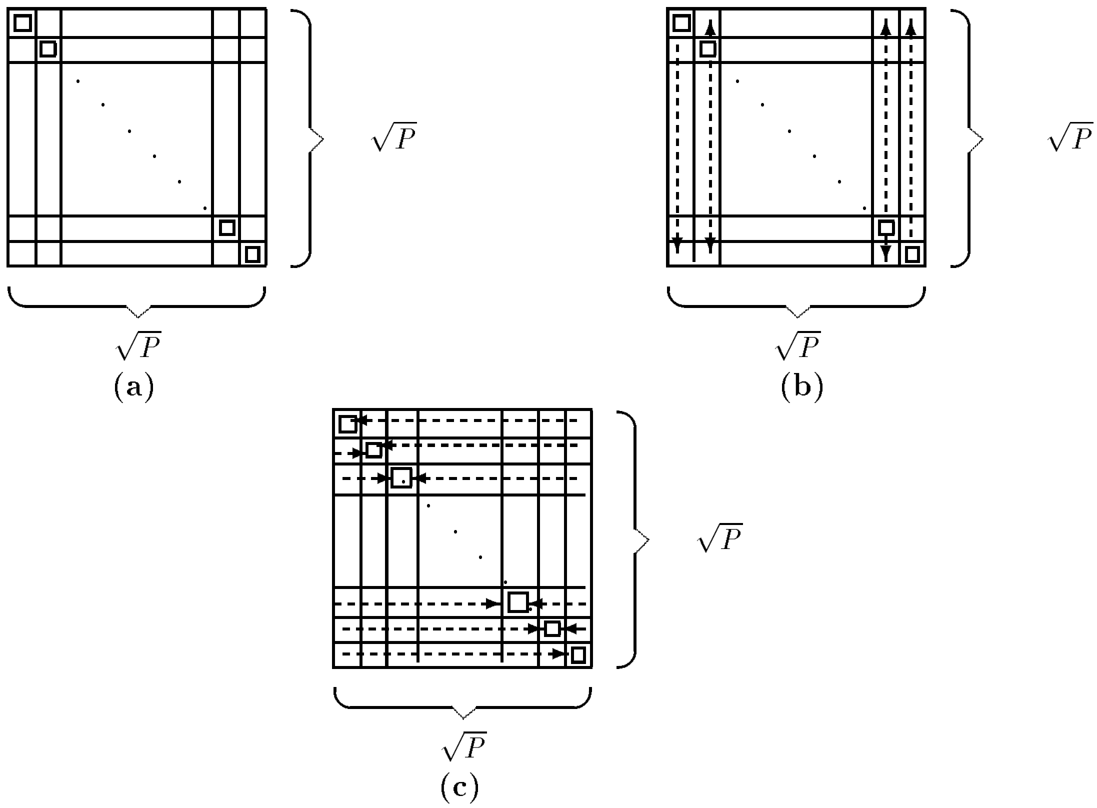

Figure 4: (a) Mapping $I*I$ submatrices onto a $\sqrt{P}*\sqrt{P}$ processor grid (b) Concurrent single source broadcast from diagonal processors (c) Accumulation of weighted sum at diagonal processors

activations for $\frac{I}{\sqrt{P}}$ nodes in the next layer by performing the required multiplications, and summing up the result at the respective diagonal processors. The communication pattern is illustrated in Figure 4(c). The time for this communication step is $[t_s + \frac{I}{\sqrt{P}}t_w]\log(\sqrt{P})$ for each layer and each pattern.

The backward phase is executed similarly, and communication time in this phase is identical to the communication time in the forward phase. Hence, total runtime for one iteration and L patterns is:

$$\text{Parallel Runtime} = LJ\frac{I^2}{P}t_c + 2LJ\log P t_s + 2LJI\frac{\log P}{\sqrt{P}}t_w \tag{2}$$

A formal description of the mapping and steps in our scheme is provided in Appendix E, to explain implementation details.

4 Performance Comparison

4.1 Framework for Comparison

Different network-partitioning schemes can be compared equally well on either per-pattern training iteration or on set-training iteration. On the other hand, per-pattern training iterations cannot be used to compare network-partitioning schemes with pattern partitioning-schemes to parallelize BP. However, the set-training iteration can be used as a benchmark task to compare alternative schemes to parallelize BP, since this task provides an equal amount of computation for all parallel formulations. Hence, in this paper set-training iteration has been used as the benchmark to compare pattern partitioning and network partitioning of BP as is done by [12, 21] among other papers. One full-iteration of the set-training regime is used as the benchmark task to compare the alternative parallel formulations. A full-iteration of the set-training regime of BP is made of the forward phase, the backward phase, and the weight change computations for all patterns and weight adjustments.

Because of our focus on hypercube-connected multi-computers, benchmark problems are chosen such that the cardinality of the set of learning patterns, as well as the number of nodes in any layer of the neural network, is a power of 2. This choice is made to simplify interpretation of the experimental results. If the size (number of nodes) of a layer is not a power of 2, then the network partitioning schemes (i.e. vertical sectioning and our scheme) increase the size of the layer by adding dummy nodes to the nearest larger power of 2. For example, the NETtalk network (203-60-26) is approximated by the network with the configuration (256-64-32) for network partitioning schemes. Pattern partitioning on hypercube is not affected by network architecture, but the number of learning patterns (L) is rounded to nearest larger power of 2.

Parallel run-time and speedup are used as the metric of performance for different parallel formulations. The parallel time for a parallel formulation refers to the execution time of the benchmark on P processors, where P is the number of processors used. Speedup is the ratio of serial runtime to parallel runtime. The serial time refers to the execution time of the benchmark task on a single processor. We provide a translation of the speedup measurements into weight updates per second and connections per second to be able to compare our results to related results [12, 22] in the literature.

Parallel formulations speed up the execution of BP by accelerating the execution of each iteration. Another way to speed up the backpropagation algorithm is based on increasing the rate of convergence (i.e. reducing the total number of iterations needed for convergence) by tuning the parameters (e.g. learning rate and momentum), or by selecting an appropriate training regime. Measurements of improvements due to change in the rate of convergence is outside the scope of this paper, and is not captured in our speedup metric.

4.2 Theoretical Comparison

In this section, we compare the cost of the checkerboarding scheme, the vertical network sectioning scheme and the pure-pattern partitioning scheme. We first show that the checkerboarding scheme is *almost always strictly better than the vertical-sectioning scheme*. We then compare the checkerboarding scheme to the pattern-partitioning scheme for some interesting cases.

4.2.1 Vertical Sectioning vs. Checkerboarding Scheme

Table 1 shows the coefficients of t_c, t_s and t_w in the runtime for one full iteration of BP, for vertical sectioning and checkerboarding. The expressions given in Table 1 assume P < I for vertical sectioning and P < I^2 for the checkerboarding scheme.

Note that while the coefficients for t_c in both the schemes are the same, the coefficient for t_s is smaller in the checkerboarding scheme. Also, the coefficient for t_w is smaller in the checkerboarding scheme when P $\geq$16[7], since the ratio $\frac{\log(P)}{\sqrt{P}} < 1$. Hence, given a hypercube with more than 16

[7]Recall that if communication along all channels of a hypercube is permitted, then the communication cost for vertical sectioning is $\frac{P}{\log(P)}[t_s + \frac{I}{P}t_w]$. In this case, P must be greater than 256 (for $\frac{\log^2(P)}{\sqrt{P}} < 1$) for checkerboarding to perform strictly better on t_w terms.

Scheme	t_c	t_s	t_w
Vertical Sectioning	$J\frac{I^2}{P}$	$2JP$	$2JI$
Checkerboarding	$J\frac{I^2}{P}$	$2J\log(P)$	$2JI\frac{\log(P)}{\sqrt{P}}$

Table 1: Coefficients of t_c, t_s and t_w in the runtime for Vertical and Checkerboarding schemes for 1 pattern and 1 iteration of BP

processors, our scheme will *always outperform* the vertical-sectioning scheme. Furthermore, our scheme can use up to I^2 processors, whereas vertical sectioning can use only up to I processors.

4.2.2 Comparison of Pattern Partitioning and Checkerboarding

Table 2 shows the coefficients for t_s, t_w, and t_c in the runtime for the pattern-partitioning and the checkerboarding network partitioning schemes. It assumes P < L for pattern partitioning scheme and P < I^2 for the checkerboarding scheme. The coefficient of t_w for pattern partitioning becomes too large as I^2 increases. On the other hand the coefficient of t_w for the checkerboarding scheme becomes too large as L increases. The coefficient of t_s for the checkerboarding scheme also increases with L; however, it can be made independent of L by sending and processing all the learning patterns together.

Clearly, neither of the schemes dominates the other in all cases. Next, we consider different classes of problems and discuss the relative superiority of different schemes.

Scheme	t_c	t_s	t_w
Pattern Partitioning	$JI^2\frac{L}{P}$	$2\log(P)$	$2JI^2\log(P)$
Checkerboarding	$JI^2\frac{L}{P}$	$2LJ\log(P)$	$2LJI\frac{\log(P)}{\sqrt{P}}$

Table 2: Cofficients of t_c, t_s and t_w in the runtime for pattern partitioning and checkerboarding for L patterns, and 1 iteration

- **Case 1: Per-Pattern Training Regime:** Pattern partitioning cannot be used to speedup BP in this case. However, the checkerboarding scheme can be used to speedup the computation of weight changes due to each pattern.
- **Case 2: Recognition Problems:** For recognition problems (see section 2.2), L tends to be smaller than I^2. In particular, if L < I, then the coefficient of t_w is always smaller for the checkerboarding scheme. Even if I^2 > L > I, the coefficient of t_w is smaller for checkerboarding for any P larger than some threshold. The threshold value of P depends upon the value of $\frac{I^2}{L}$ and on the value of t_s.
- **Case 3: Generalization Problems:** Generalization problems (see section 2.2), have a large number of learning patterns, L, which are larger than the number of weights, JI^2 in the neural network. In this case, pattern partitioning is strictly better than the checkerboarding scheme and can be shown to be better than any network partitioning scheme for hypercubes, meshes and linear arrays. Even for these problems, a hybrid of the checkerboarding scheme with the pattern-partitioning scheme can perform better than pure-pattern partitioning. This is discussed further in section 5.

4.3 Experimental Evaluation and Comparison

In this section, we present the performance of the checkerboarding scheme on uniform networks. To test the performance, we implemented BP on the nCUBE2^{TM} hypercube. nCUBE2 is a second generation hypercube containing up to 8192 processors. Each node of nCUBE2 is rated at approximately 8 MIPS / 2.5 MFlops (peak). In our experiments, we observe interprocessor communication startup time t_s to be around 180 μs, and the transmission time for 4-byte message t_w to be about 3 μs. The value of t_c (for our unoptimized code) was observed to be around $10\mu s$. BP was tested for uniform networks with an input layer and two other layers (*i.e.* $J = 2$), and a full-iteration was performed.

Table 3 shows the speedups obtained on nCUBE2 for checkerboarding for different values of P, and for network sizes in carrying out one full iteration of set-training regime of BP. Speedup on small networks is poor but it improves significantly for large networks.

	I	
P ↓	64	256
4	3.41	3.87
16	8.42	14.15
64	12.11	43.83
256	12.58	93.79

Table 3: Actual speedups obtained by checkerboarding

We were unable to get experimental speedup figures for $I > 256$, due to insufficient per-processor memory on nCUBE2 for determining serial execution time. To study the impact of larger I and P, we theoretically computed the parallel runtime and speedup for checkerboarding using a cost model described in Appendix B, which is slightly more detailed than the simplified model used for theoretical comparison. The constants used in this model were computed using the actual experimental values t_s, t_w and t_c for nCUBE2. These values are given in table 4. The readers can see that the experimentally obtained values (the columns for $I = 64, 256$ and row for $P = 4, 16, 64$) are fairly close to the values predicted by our model. From 4, it is clear that for sufficiently large sizes of networks, near-linear speedups can be obtained, even on very large number of processors.

	I			
P ↓	64	256	1024	4096
4	3.38	3.86	3.97	3.99
16	8.38	14.04	15.55	15.89
64	12.56	43.35	59.16	62.86
256	12.90	93.57	207.82	244.96
1024	11.55	128.98	596.56	919.14

Table 4: Analytically computed speedups for checkerboarding

Table 5 shows the speedups obtained on nCUBE2 for checkerboarding, vertical sectioning and pattern partitioning, for I equal 64 and 256, and for P equal 4, 16, 64 and 256. The number of training samples, L, was kept equal to P in each case. The entry for $P = 256$ and $I = 64$ for the vertical-sectioning scheme is empty, as $P \geq I$. From this table, it can be seen that the checkerboarding scheme is substantially faster than the vertical sectioning scheme for $P \geq 16$.

Actual Speedup (per training iteration) on nCUBE2 with J = 2						
$P \downarrow$	$I = 64$			$I = 256$		
	Checkerboarding	Vertical Part.	Pattern Part.	Checkerboarding	Vertical Part.	Pattern Part.
4	3.4134	3.5227	1.5257	3.8739	3.9466	1.465
16	8.4246	4.5937	3.93	14.1518	13.6783	3.813
64	12.1100	1.5262	10.49	43.8332	17.4662	11.3176
256	12.5814	-	36.8399	93.7938	5.8845	35.9651

Table 5: Speedup results for Checkerboarding, Vertical Partitioning and Pattern Partitioning

Note that the checkerboarding scheme is faster than the pattern-partitioning scheme for networks with a large number of nodes. The pattern-partitioning scheme is better than the checkerboarding scheme for networks with a smaller number of nodes. This is consistent with our theoretical analysis. We note that the vertical-sectioning scheme can use a maximum of I processors, but due to communication overhead, the speedup peaks at less than I processors. Similarly, the checkerboarding scheme can use a maximum of I^2 processors, but due to communication overhead, the speedup will peak at less than I^2 processors. Readers can verify that the speedup will peak at $\bigcirc(\frac{I^2}{K\log(I)})$ processors, where K is determined by t_c, t_s and t_w.

5 Hybrid of Checkerboarding with Pattern Partitioning

The checkerboarding network partitioning scheme can be combined with pattern partitioning in a natural manner. This hybrid scheme divides a given set of processors into a collection of clusters. The set of learning patterns are divided equally among the clusters. Network nodes and weights are replicated across clusters. Each cluster divides the network nodes and edges in a checkerboarding manner. Proper mapping of clusters on a hypercube makes it possible to eliminate communication conflicts within clusters for forward propagation and back propagation. It also eliminates communication conflicts, among nodes responsible for a given weight, across clusters.

We divide P ($= P_1 \times P_c$) processors on a hypercube into P_1 clusters of P_c processors, where P_1 and P_c are powers of 2: i.e. $P_1 = 2^a$ and $P_c = 2^b$ for some integers a and b. Let us number the processors by $P(i,j)$, where i is the cluster number and j is the rank of the processor within its cluster. This division of processors is such that the P_c processors in a cluster (i.e. $P(i,j)$ for fixed i) form a hypercube. Furthermore, the processors across clusters with a common rank, i.e. $P(i,j)$ for fixed j, also form a hypercube.

The learning patterns are divided into P_1 equal group patterns; i.e., the number of patterns in each cluster is $L_c = \frac{L}{P_1}$. Each cluster of P_c processors divides the network in an identical manner, via the checkerboarding scheme, over a $\sqrt{P_c} \times \sqrt{P_c}$ grid imposed over the cluster. Each cluster carries out forward and backward phases for the L_c patterns local to the cluster, accumulating the weight changes for all the JI^2 weights in the network. The division of the network among these $\sqrt{P_c} \times \sqrt{P_c}$ processors is just like in the checkerboarding scheme, hence each processor in a cluster keeps $J\frac{I^2}{P_c}$ weights and the corresponding weight changes. The communication time for the forward phase is $JL_c\log(P_c)[t_s + \frac{I}{\sqrt{P_c}}t_w]$. The communication time for the backward phase is the same. Therefore, the total communication cost for the forward and the backward phase is $C_1 = 2JL_c\log(P_c)[t_s + \frac{I}{\sqrt{P_c}}t_w]$. This phase is followed by the accumulation of weight changes across clusters. Each processor $P(i,j)$ communicates with the processors with the same rank, j, in various clusters, to accumulate the changes for its $J\frac{I^2}{P_c}$ weights. The changes are accumulated at a processor (say in cluster 0), and then broadcast back to the respective processors, which then independently recompute the weight matrix for the next iteration. The communication time for this step is $C_2 = 2\log(P_1)[t_s + J\frac{I^2}{P_c}t_w]$.

The total parallel computation time taken by each processor is $C_3 = LJ\frac{I^2}{P}t_c$. Therefore, the total runtime for the hybrid scheme can be approximated as $C_1 + C_2 + C_3$. The cost model for the hybrid scheme reduces to the cost model of the pure-pattern partition for $[P_c = P, P_1 = 1]$. It reduces to the cost model of the pure-checkerboarding scheme, for $[P_1 = P, P_c = 1]$. The runtime can be changed by varying the parameters P_1 and P_c such that $P = P_1 \times P_c$, and $P_1 = 2^a$ and $P_c = 2^b$. The hybrid scheme can use LI^2 processors, which is more than the number of processors used by either the pattern partitioning or the checkerboarding scheme. Actually, in order to ensure efficient use of the parallel computer, the number of processors should be no more than $\frac{L}{\log L}(\frac{I}{\log I})^2$. If more processors are used then the increased communication overhead will nullify the gains due to extra parallelism.

The hybrid scheme can mimic pure pattern partitioning by choosing $P_1 = 1$, and can mimic pattern partitioning by choosing $P_c = 1$. Thus, the hybrid scheme can perform at least as well as both the schemes. In addition, the hybrid scheme with $P \neq P_1$ and $P \neq P_c$ can outperform the pure schemes for a large class of problems. We show these cases via algebraic and experimental methods now. We summarize the terms for t_c, t_s and t_w in table 6, for the three schemes for comparison.

Scheme	t_c	t_s	t_w
Hybrid	$LJ\frac{I^2}{P}$	$2L_cJ\log(P_c) + 2\log(P_1)$	$2L_cJ\log(P_c)\frac{I}{\sqrt{P_c}} + 2J\log(P_1)\frac{I^2}{P_c}$
Pure Checkerboarding $[P_1 = 1, P_c = P]$	$LJ\frac{I^2}{P}$	$2LJ\log(P)$	$2LJ\log(P)\frac{I}{\sqrt{P}}$
Pure pattern partitioning $[P_1 = P, P_c = 1]$	$LJ\frac{I^2}{P}$	$2\log(P)$	$2J\log(P)I^2$

Table 6: The t_c, t_s, *and* t_w cofficients for the three schemes (one iteration of set training regime of BP)

We note from table 6 that the coefficient of t_c is identical for the three schemes. The coefficient of t_s is smallest for the pure partitioning scheme and is largest for the pure checkerboarding scheme. Furthermore, note that the coefficients of t_w are larger than the corresponding coefficients of t_s for each scheme. The t_w terms will dominate the t_s terms if $JI^2\frac{I}{\sqrt{P}} > \frac{t_s}{t_w}$ and $J\frac{I^2}{P_c} > \frac{I}{\sqrt{P_c}} > \frac{t_s}{t_w}$. The t_w term will dominate the communication cost for large networks and for a large number of learning samples. We will compare the t_w term of the alternative schemes in the following algebraic analysis to simplify the analysis.

Now we provide independent conditions under which the t_w term for the hybrid scheme is smaller than that for the pure pattern-partitioning and pure checkerboarding schemes. The intersection of these conditions represents the region of dominance of hybrid over both the schemes. The derivation of these conditions is provided in Appendix C.

The t_w of the hybrid scheme is smaller than the t_w term of pattern partitioning, if $L < IP_1\sqrt{P_c}$. This condition is met for recognition problems. It is also met for many generalization problems for a large value of P, or for large networks, or for small learning samples. Thus, we expect the hybrid scheme to outperform the pure pattern-partitioning scheme if a large number of processors are available, or if network is large, or if the number of training patterns is small. The t_w term of the hybrid scheme is smaller than the t_w term of pure checkerboarding if $L > I\frac{\sqrt{P_1}}{\sqrt{P_c}}$. This condition is met for generalization problems (e.g. with $P_1 = P_c$). and for some recognition problems. Thus we expect the hybrid scheme to outperform checkerboarding, if large number of processors are available or if the network is small, or if the learning sample set is large. Finally, combining both inequalities, we get

$$IP_1\sqrt{P_c} \;>\; L \;>\; I\frac{\sqrt{P_1}}{\sqrt{P_c}}$$

which is the region of dominance for the hybrid scheme over both the pure pattern-partitioning and the checkerboarding schemes. The condition holds for some values of P_1, P_c, and $P = P_1 P_c$, for given values of L and I. For example, the hybrid scheme can outperform the pure schemes for problems with $L > I$ using $P^{3/2} > \frac{L}{I}$ with $P_1 = P_c = \sqrt{P}$. Note that this region is not the only region in which the hybrid scheme outperforms the other two schemes.

Experimental Evaluation of Hybrid Scheme

Table 7 shows the speedups obtained for pattern partitioning, hybrid and checkerboarding schemes. The results were obtained using 256 processor nCUBE2 for two networks: 2×64 (*i.e.* $J = 2$, $I = 64$), and 2×256 (*i.e.* $J = 2$, $I = 256$). The second column shows the value for $\log P_1$ and $\log P_c$ in the format ($\log P_1 \times \log P_c$). As explained before, for pure partitioning, $\log P_c = 0$, and for pure checkerboarding , $\log P_1 = 0$. For the hybrid scheme, three combinations for $\log P_1 \times \log P_c$ were used, 6×2, 4×4, and 2×6. Note that all these hybrid combinations perform at least as well as the pattern partitioning and the checkerboarding schemes, with some combinations doing significantly better.

Actual Speedups for Uniform Network **on nCUBE2** $L = P = P1 * Pc = 256$			
		$J = 2$, $I = 64$	$J = 2$, $I = 256$
scheme ↓	$\log(P_1) \times \log(P_c)$	Speedup	Speedup
Pattern Partitioning	8x0	36.8399	35.9651
Pure Hybrid	6x2	110.0739	129.8600
	4x4	121.8266	192.3834
	2x6	48.1149	172.6645
Checkerboarding	0x8	12.5873	93.8585

Table 7: Speedup for the hybrid scheme

Alternative Hybrid Schemes

Other network partitioning schemes can be combined with the pattern-partitioning scheme to form alternative hybrid schemes. In particular, vertical sectioning can be combined with pure pattern-partitioning. The hybrid of checkerboarding with pure pattern partitioning can be shown to be better than the hybrid of vertical sectioning with pattern partitioning by extending the theoretical comparison discussed in section 4.2.

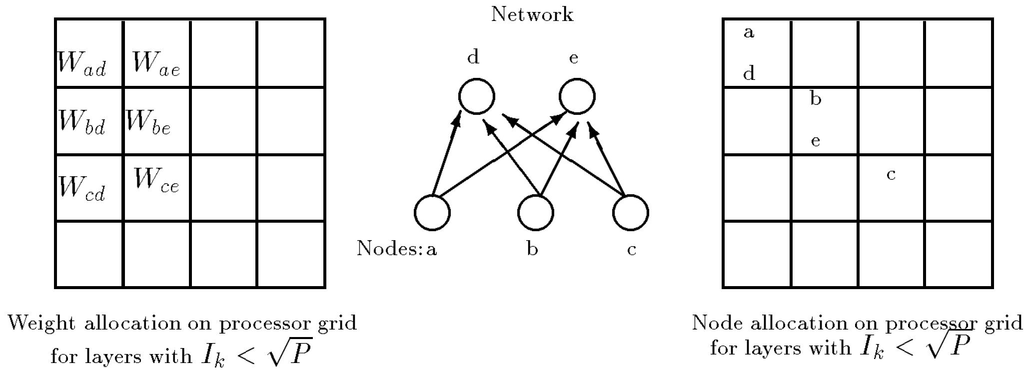

Figure 5: The partition for non uniform networks

6 Non-uniform Networks

Non-uniform networks are characterized by layers with different numbers of nodes. Non-uniform networks are interesting, since these are used in many applications [2, 25]. Let $I_0, I_1, \ldots I_J$ be the number of nodes in the different layers of the network. The network is *uniform* iff $I_0 = I_1 = \cdots = I_J$; otherwise, the network is called a *non-uniform network*. Our scheme can be extended to non-uniform networks. For per-pattern training, we again use a single[8] square grid of P processors for all layers of the network. If the nodes in layer k are less than the number of processors on the diagonal (i.e. $\sqrt{P}$), then $\sqrt{P} - I_k$ processors will not get any node from the layer k, as shown in figure 5. The processors with no real weights for the layer with $I_k < \sqrt{P}$ can be viewed as processors with zero weights, incoming/outgoing to a fictitious node.

The interesting issue in extending checkerboarding to non-uniform networks relates to the choice of the size $\sqrt{P}$ for the $\sqrt{P} \times \sqrt{P}$ processor grid. Choosing $\sqrt{P} = I_{max} = maximum[I_1, I_2, \cdots, I_J]$ provides the highest speedups but poor processor utilization. On the other hand, choosing $\sqrt{P} \leq I_{min} = minimum[I_1, I_2, \cdots, I_J]$ provides high processor utilization but relatively smaller speedups. This issue arises when extending any network partitioning scheme to non-uniform networks. For example, the choice of the size (P) of linear array in the vertical-sectioning scheme faces a similar problem. Choosing $P = I_{max}$ improves speedups but yields poor processor utilization.

The actual choice of P would depend upon the number of processors available and on the network size. For example, consider the mapping of a network of size $1024 \times 64 \times 4$ using our checkerboarding scheme. If $P = 4 \times 4$, then all the processors will do useful work, as the weight matrices of size 1024×64 and 64×4 can be easily mapped on to 4×4 processors. If $P = 8 \times 8$, then the weight matrix between the hidden layer and the output layer (64×4) will have to be padded with redundant entries to form a 64×8 matrix, causing redundant computation in the parallel formulation. In this case, the total amount of redundant computation is: $\frac{64 \times 4}{1024 \times 64 + 64 \times 4} \approx .4\%$. If $P = 64 \times 64$, then the amount of redundant computation is: $\frac{64 \times 60}{1024 \times 64 + 64 \times 4} \approx 6\%$.

Detailed algebraic analysis for non-uniform networks is given in Appendix D. As shown there, the relative performance of vertical sectioning, checkerboarding, pattern partitioning and hybrid schemes remains the same. Checkerboarding can perform better than pattern partitioning if the network size (i.e. the total numer of weights in the network) is sufficiently large. The hybrid of checkerboarding with pattern partitioning can perform better than either one.

[8] Alternatively, one may vary the length and breadth of the processor grid from layer to layer, for a non-uniform network.

Experimental Evaluation

We experimentally tested the performance of our hybrid scheme for non-uniform networks on nCUBE2 and CM5 parallel computer. CM5 contains up to 16384 processors connected via a pseudo a fat-tree. Since the hypercube topology maps naturally on a fat-tree (and vice-versa), the checkerboarding and the hybrid schemes can be easily implemented on CM5.

The speedups obtained on nCUBE2 and CM5 for non-uniform networks are shown in Table 8. These results were obtained for a network with $J = 2$ and $L = 256$. The first column of this table shows the network size in the form of $I_0 \times I_1 \times I_2$, where I_0, I_1 and I_2 are the number of nodes in the input, first and output layer respectively. As the hybrid scheme can give different speedups depending on the value of $\log P_1$ and $\log P_c$, we present in the fourth and sixth column the combinations that gave the best speedups. The actual values for $\log P_1$ and $\log P_c$ are shown in square brackets in the form $[\log P_1 \times \log P_c]$. Again, note that the hybrid performs at least as well as checkerboarding, and in some cases much better.

Speedup for Checkerboarding on Non-Uniform Networks (J=2, L=256)					
network	P	Speedup for nCUBE2		Speedup for CM5	
		Checkerboarding	Hybrid	Checkerboarding	Hybrid
$256 \times 64 \times 64$	64	20.1110	48.8500 $[4 \times 2]$	17.6533	48.5280 $[4 \times 2]$
$256 \times 64 \times 256$	64	25.4457	46.8091 $[4 \times 2]$	22.2775	43.4293 $[2 \times 4]$
$256 \times 64 \times 64$	256	25.4287	148.0692 $[4 \times 4]$	17.2962	140.3415 $[4 \times 4]$
$256 \times 64 \times 256$	256	35.9892	161.8324 $[4 \times 4]$	26.7363	144.2137 $[4 \times 4]$
$1024 \times 64 \times 1$	256	50.4172	157.5171 $[4 \times 4]$	42.6003	165.0833 $[4 \times 4]$
$1024 \times 64 \times 64$	256	53.3520	166.7024 $[4 \times 4]$	45.5122	176.3711 $[4 \times 4]$
$1024 \times 256 \times 64$	256	116.7895	215.1825 $[4 \times 4]$	117.6144	218.8961 $[4 \times 4]$

Table 8: Checkerboarding and Hybrid using Non-uniform networks on nCUBE2 and CM5

7 Scalability Analysis and Optimality Considerations

The scalability of an algorithm on a parallel architecture is determined by its capability to effectively utilize an increasing number of processors. For all four parallel BP schemes discussed, near-linear speedups can be obtained on an arbitrarily large number of processors, provided the network is large enough (for vertical sectioning, checkerboarding, and hybrid) or if the number of patterns is large enough (for pattern partitioning and hybrid). In that sense, all the schemes are scalable. But the degree of scalability of these schemes is actually quite different from each other. A number of metrics have been developed for characterizing the scalability of different parallel algorithms and architectures[35]. For fixed processor utilization, the rate of change of problem size as a function of number of processors is a good characterization of the *scalability of an algorithm* [36, 37, 38, 39]. An algorithm that requires a smaller change in problem size to obtain fixed efficiency is considered more scalable.

The problem size (i.e., serial run time) for each iteration of BP is LJI^2t_c. For pattern partitioning, the problem size can be increased as $O(L)$; for checkerboarding and vertical sectioning, it can be increased as I^2, and for hybrid it can be increased as LI^2.

The efficiency E of a parallel algorithm is defined to be $\frac{S}{P}$ = Serial Run Time / ($P*$ Parallel Run Time). By plugging the expressions for parallel run time for the different algorithms in the expression, we can derive the following statements for different schemes.

- For vertical sectioning, fixed efficiency can be maintained if I is increased as $O(P)$; i.e., if I^2 is increased as $O(P^2)$.
- For checkerboarding, fixed efficiency can be maintained if I is increased as $O(\sqrt{P} * \log P)$; also if I^2 is increased as $O(P(\log P)^2)$. Clearly this scheme is more scalable, as $O(P^2)$ grows much faster than $O(P(\log P)^2)$.
- For pattern partitioning, fixed efficiency can be maintained if L is increased as $O(P \log P)$. Hence this scheme is quite scalable, provided the number of patterns is large. Note that the scalability expressions of checkerboarding (or vertical sectioning) and pattern partitioning cannot be compared, as the problem size is increased in a different way in both schemes.
- For hybrid, fixed efficiency can be maintained if I is increased as $O(\sqrt{P} * \log P)$ or if L is increased as $O(P \log P)$. This scheme is more scalable then pattern partitioning and checkerboarding. To illustrate this, consider the following special case in which $I = L$ and J is a constant, and thus the problem size is $O(I^3)$. For this case, I should be increased as $O(\sqrt{P} * \log P)$ for checkerboarding, as $O(P \log P)$ for Pattern partitioning, and as $O(P^{\frac{1}{3}} \log P)$ for the hybrid scheme.

We also note that the checkerboarding scheme and its hybrid with pattern partitioning are optimal parallel formulation of backpropagation in the following sense. These schemes provide parallel time close to the asymptotic lower bound on the parallel time for computing each iteration of BP on parallel machines, where the atomic operations for each processor is limited to binary arithmetic operations. In such an environment, the lower bound on the parallel computation time for evaluating an arithmetic expression with N atomic operations is $\lceil \log_2(N) \rceil$ for any number of processors [40]. The number of arithmetic operations in the expression computed at the output nodes of a feed-forward neural network at the end of forward propagation is $\bigcirc(LI^2)$, assuming J to be a constant. Thus, the lower bound on carrying out each iteration of BP in set-training regime (see 2.1) is $\bigcirc(\log(L) + \log(I))$. Given enough processors ($P = LI^2$), a hybrid of checkerboarding with pattern partitioning ($P_c = I^2$, $P_1 = L$) can carry out each iteration of BP in in $\bigcirc(\log(L) + \log(I))$ time. Using a similar argument, the checkerboarding scheme can be shown to be an optimal network-partitioning scheme for the per pattern training regime.

8 Concluding Remarks

Our analytical and experimental results show that the checkerboarding network partitioning scheme is better than the commonly used vertical-sectioning scheme. It is also better than pattern partitioning for a large class of useful problems and a large number of processors. The checkerboarding scheme can be combined with pattern partitioning to provide very high overall performance on existing commercial parallel computers. For example, the observed value of t_c on the currently available scalar processors of CM5 (in our non-optimized code) is $4.3\mu s$. The processing time per node for only the forward phase is around $1.38\mu s$. Hence, the non-optimized version of our hybrid parallel formulation on 256 processor CM5 (with scalar processors) performs over 50 million weight changes or 160 million connections per second for the $1024 \times 256 \times 64$ network. With the upgrade time of processors to vector units, t_c as well as the overall performance may improve by one or two orders of magnitude. The proposed schemes scale well with problem size and achieve asymptotically optimal parallel execution time for backpropagation algorithm on uniform networks.

The parallelization schemes for non-uniform networks need further exploration. We plan to explore the trade-off between processor utilization and speed-up in our future work. We would also like to generalize our schemes by utilizing different lengths and breadths of processor grids for different layers of the network. It would also be interesting to compare the parallelization techniques for randomly sparse networks [7, 26, 27, 28, 29]. with our scheme for non-uniform networks.

9 Acknowledgement

We would like to thank Prof. Joydeep Ghosh (University of Texas at Austin), George Karypis and Ananth Grama for their useful comments during the drafting of this paper, and Dr. Robert Benner of Sandia labs for giving access to nCUBE2. Anonymous refrees provided valuable comments to improve the presentation. Ms. Christine McCarthy, Dept. of Composition, University of Minnesota provided editorial help to improve grammatical and stylistic aspects. We also thank [24], which was an excellent source on BP.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, chapter 8. MIT Press, 1986.

[2] T. J. Sejnowski and C. R. Rosenburg. Nettalk: a parallel network that learns to read aloud. Technical Report JHU/EECS-86/01, Johns Hopkins University - EE/Csci, 1986.

[3] S. Shekhar and M. B. Amin. Genralization performance of feed-forward neural networks. *IEEE Trans. On Knowledge and Data Engineering*, April 1992.

[4] S. Dutta and S. Shekhar. Bond-rating: A non-conservative application of neural networks. In *IEEE Intl. Conf. on Neural Networks*, July 1988.

[5] H. White. Economic prediction using neural networks: The case of ibm daily stock returns. In *IEEE Intl. Conf. on Neural Networks*, July 1988.

[6] W. Allen and A. Saha. Parallel neural network simulation using backpropagation for the es-kit environment. In *Proceedings of the 1989 Conference on Hypercubes, Concurrent Computers and Applications*, pages 1097–1102, 1989.

[7] W. M. Lin, V. K. Prasanna, and K. W. Przytula. Algorithmic mapping of neural network models onto parallel simd machines. *IEEE Trans. on Computers*, 40(12):1390–1401, December 1991.

[8] G. Blelloch and C. R. Rosenberg. Network learning on the connection machine. Technical report, MIT, November 1986.

[9] H. Yoon and J. H. Nang. Multilayer neural networks on distributed-memory multiprocessors. In *Proc. Intl. Jt. Conf. on Neural Networks (IEEE/EEC)*, pages 669–672, 1991.

[10] H. Yoon and J. H. Nang. A distributed backpropagation algorithm of neural networks on distributed-memory multiprocessors. In *Proc. Intl. Conf. on Parallel Processing*, pages 358–363, 1991.

[11] Xiru Zhang. An efficient implementation of the backpropagation algorithm on the connection machine cm-2. Technical Report RL89-1, Thinking Machines Corporation, August 29 1989.

[12] Xiru Zhang and Michael Mckenna. The backpropagation algorithm on grid and hypercube architectures. Technical Report RL90-9, Thinking Machines Corporation, 1990.

[13] F. Baiardi, R. Mussard, R. Serr, and G. Valastro. Feedforward neural networks on message passing parallel computers. In *Proc. 2nd Italian Workshop on Parallel Architectures and Neural Networks.* (Ed. E. R. Caianielo) World Scientific, Singapore, 1990.

[14] M. Marchesi, G. Orlandi, F. Piazza, and A. Uncini. Linear array architecture implementing the back-propagation neural network. In *Proc. 2nd Italian Workshop on Parallel Architectures and Neural Networks.* (Ed. E. R. Caianielo) World Scientific, Singapore, 1990.

[15] D. S. Newhall and J. C. Horvath. Analysis of text using a neural network: A hypercube implementation. In *Proceedings of the 1989 Conference on Hypercubes, Concurrent Computers and Applications*, pages 1119–1122, 1989.

[16] S. Y. Kung and J.N. Hwang. A unified systolic architecture for artificial neural networks. *Journal of Parallel and Distributed Computing*, 6:358–387, 1989.

[17] K. Joe, Y. Mori, and S. Miyake. Simulation of a large-scale neural network on a parallel computer. In *Proceedings of the 1989 Conference on Hypercubes, Concurrent Computers and Applications*, pages 1111–1118, 1989.

[18] M. Witbrock and M. Zagha. An implementation of back-propagation learning on gf11, a large simd parallel computer. Technical Report CMU-CS-89-208, CMU, December 1989.

[19] J. Bourrley. Parallelization of a neural learning algorithm. In *Proc. 1st Euro. Workshop on Hypercube and Distributed Computers.* (Ed. F. Andre) North-Holland, 1989.

[20] B. K. Mak and O. Egecioglu. Communication parameter tests and parallel backpropagation algorithms on ipsc/2 hypercube multiprocessor. In *IEEE Frontier*, pages 1353–1364, 1990.

[21] A. Petrowski, L. Personnaz, G. Dreyfus, and C. Girault. Parallel implementations of neural network simulations. In *First European Workshop on hypercube and distributed computers*, pages 205–218. Elsevier Science Publishers B.V. (North-Holland), 1989.

[22] D.A.Pomerleau G.L.Gusciora, H.T.Kung and D.S.Touretzky. Neural network simulation at warp speed: How we got 17 million connections per second. In *IEEE 2nd Intl. Conf. on Neural Networks*, pages 143–50, 1988.

[23] V. Kumar, S. Shekhar, and M. B. Amin. A highly parallel formulation of backpropagation on hypercubes: A summary of results. In *Proc. Intl. Conf. on Neural Networks (IJCNN)*, November 1992.

[24] D. E. Rumelhart J. L. McClelland. *Explorations in Parallel Distributed Processing.* MIT Press, 1988.

[25] G. L. Wilcox, M. Poliac, and M. Liebman. Protein tertiary structure prediction using a large backpropagation network. In *IJCNN (IEEE/EEC)*, pages 365–369, 1990.

[26] B. W. Wah and L. Chu. Efficient mapping of neural networks on multicomputers. *IEEE International Conference on Parallel Processing*, pages 234–241, 1990.

[27] M. Misra and V. K. Prasanna Kumar. Neural network simulation on a reduced mesh of tree organization. In *SPIE/SPSE Symposium on Electronic Images*, 1990.

[28] M. Misra. Implementation of neural networks on parallel architectures. In *Tech. Report 295.* Dept. of EE Systems hi, Univ. of Soutern California, 1992.

[29] J. Ghosh and K. Hwang. Mapping neural networks onto message passing multicomputers. *Jr. Parallel and Distributed Computing*, April 1989.

[30] Anshul Gupta and Vipin Kumar. On the scalability of Matrix Multiplication Algorithms on parallel computers. Technical Report TR 91-54, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1991. Revised, Sep. 1992.

[31] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation*, pages 50–55. Prentice-Hall Inc, 1989.

[32] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249 – 1268, September 1989.

[33] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Trans. on Computers*, pages 867–872, 1988.

[34] J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In *Int. Conf. on Parallel Processing*, pages 713–716, 1987.

[35] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. Technical report, TR-91-18, Computer Science Department, University of Minnesota, June 1991. A short version of the paper appears in the Proceedings of the 1991 International Conference on Supercomputing, Germany, and as an invited paper in the Proc. of 29th Annual Allerton Conference on Communuication, Control and Computing, Urbana,IL, October 1991.

[36] Anshul Gupta and Vipin Kumar. On the scalability of fft on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 1993 (to appear). available as a technical report TR 90-20, Computer Science Department, University of Minnesota.

[37] Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem. *Journal of Parallel and Distributed Processing (special issue on massively parallel computation)*, 13(2):124–138, October 1991. A short version appears in the Proceedings of the International Conference on Parallel Processing, 1990.

[38] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition.* Springer-Verlag, New York, 1990.

[39] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. Scalability of parallel sorting on mesh multicomputers. *International Journal of Parallel Programming*, 20 (2), 1991. A short version of this paper appears in the Proceedings of the Fifth International Parallel Processing Symposium, 1991.

[40] David J Kuck. A survey of parallel machine organization and programming. *ACM Computing Survey*, 9(1), March 1977.

Appendix A Backpropagation Algorithm

The details of BP are shown in Figure 6. Symbol f denotes the non-linear sigmoidal function used to compute the activation of each node. The thresholds associated with the nodes of the neural network in this description of BP have been ignored. Furthermore, we describe the set training regime in this algorithm. Per pattern training regimes would require weight adjustments in step 3 to be inside the foreach loop.

procedure **BP**(var network, learning_patterns, target, limit)

begin

#iteration := 0;

REPEAT

foreach learning pattern do

begin

/* STEP1 - *Forward Phase:* */

For $j = 1$ to J do /* do for each layer j */ ....(1)

For $i = (I-1)j$ to $(Ij-1)$ do /* for each node, i, in layer j, calculate activation value, a_i^j */

$a_i^j = \mathrm{f}(\sum_{k=0}^{I-1} w_{k,i}^{j-1,j} * a_k^{j-1})$

/* STEP2 - *Backward Phase:* */

/* *STEP2a - calculate delta value* */

For $j = (J-1)$ down to 1 do /* do for each layer j except $(J-1)$ */ ....(2)

For $i = (j-1)I$ to $(jI-1)$ do /* for each node, i, in layer j, calculate delta value, δ_i^j */

if $(j = (J-1))$ then $\delta_i^j = [t_i^j - a_i^l] * [a_i^l * (1 - a_i^l)]$;

else $\delta_i^j = \left[\sum_{k=0}^{I-1} \delta_k^{j+1} * w_{i,k}^{j,j+1}\right] * [a_i^j * (1 - a_i^j)]$;

/* *STEP2b - calculate weight changes* */

For $j = (J-1)$ down to 1 do /* do for each layer j except 0 */ ....(3)

For $i = I(j-2)$ to $I(j-1)-1$ do /* for each node, i, in layer j, calculate weight change, $\Delta w_{i,k}^{j-1,j}$ */

For $k = I(j-1)$ to $Ij-1$ do

$\Delta w_{i,k}^{j-1,j} = \eta * \delta_i^j * a_k^{j-1}$;

$Sum\Delta w_{i,k}^{j-1,j} \leftarrow Sum\Delta w_{i,k}^{j-1,j} + \Delta w_{i,k}^{j-1,j}$ /* accumulate per pattern weight changes */

end;

/* STEP3 - *Change Weight:* */

For $j = (J-1)$ down to 1 do /* do for each layer j except 0 */

For $i = I(j-2)$ to $I(j-1)-1$ do /* for each node, i, in layer j, calculate new weight */

For $k = I(j-1)$ to $Ij-1$ do

$w_{i,k}^{j-1,j} \leftarrow w_{i,k}^{j-1,j} + \frac{Sum\Delta w_{i,k}^{j-1,j}}{cardinality(learning\_patterns)}$;

#iteration := #iteration + 1;

error_of_fit := compute_error();

UNTIL (error_of_fit < target) or (#iteration > limit);

end;

Figure 6: The Backpropagation Algorithm

Appendix B Detailed Cost Model for Checkboarding

A simple model of the parallel run-time of the checkerboarding scheme assumes that computation in the BP algorithm is equally divided among all processors. In contrast, the detailed model accounts for the following two kinds of computations which are not divided equally among the processors:

- **Computation of the sigmoidal functions:** Sigmoidal functions computations are needed to compute the node activations. Let t_{c2} denote the time to compute a sigmoidal function. The serial runtime for this computation is $LJIt_{c2}$ and thus the revised serial runtime T^R_{ser} is:

  $$T^R_{ser} = LJI^2 t_c + LJIt_{c2}$$

  These computations for I nodes are divided among the $\sqrt{P}$ diagonal nodes only. The parallel run-time to compute sigmoidal function is $LJ\frac{I}{\sqrt{P}}t_{c2}$.

- **Weighted summation over a tree:** The checkerboarding scheme imposes balanced trees over the processors in each row and each column of the processor grid. These trees are used to compute the weighted sum to compute activation and error vectors. The processors at the leaves of the tree do not participate in this computation. In contrast, the diagonal processor (i.e. root of the tree) has to carry out additions for the $\frac{I}{\sqrt{P}}$ nodes in each of the $\log(\sqrt{P})$ levels of the tree. Let t_{c3} denote the time for per node computation, related to summation. The parallel run-time to compute summation is $LJI\frac{\log P}{\sqrt{P}}t_{c3}$ for the forward and backward phases.

The revised parallel time of the checkerboarding scheme T^R_{par}, is:

$$T^R_{par} = (LJ\frac{I^2}{P}t_c + LJ\frac{I}{\sqrt{P}}t_{c2} + LJI\frac{\log P}{\sqrt{P}}t_{c3}) + 2LJ\log Pt_s + 2LJI\frac{\log P}{\sqrt{P}}t_w$$

and the speedup is $\frac{T^R_{seq}}{T^R_{par}}$.

In our implementation of checkerboarding on nCUBE2, t_{c2} is approximately 60 micro-seconds and t_{c3} is approximately 0.4 micro-seconds. With addition of floating point processors, these numbers will become smaller. For large networks, the detailed model reduces to the simple model used for the theoretical analysis of checkerboarding, for the following reasons. The coefficient of t_{c2} becomes insignificant with respect to the t_c as the network size (I) increases. This term can be ignored for a large network without compromising accuracy. In addition, the coefficient of t_{c3} is proportional to the coefficient of t_w. Thus, the cost associated with t_{c3} can be merged with the variable cost of sending messages.

Appendix C Derivation of Dominance Region of Hybrid Scheme

The t_w of hybrid scheme is smaller than the t_w term of partitioning if the following condition is met:

$$\begin{array}{llcl} & 2JI^2\log(P) & > & 2JI^2\frac{\log(P_1)}{P_c} + 2L_cJI\frac{\log(P_c)}{\sqrt{P_c}} \\ or & I\log(P_1P_c) & > & I\frac{\log(P_1)}{P_c} + L_c\frac{\log(P_c)}{\sqrt{P_c}} \\ or & I\log(P_1) + I\log(P_c) & > & I\frac{\log(P_1)}{P_c} + L_c\frac{\log(P_c)}{\sqrt{P_c}} \end{array}$$

The inequality can be simplified to $I\log(P_c) > L_c\frac{\log(P_c)}{\sqrt{P_c}}$ since

$$I\log(P_1)\ is\ always\ \geq\ I\frac{\log(P_1)}{P_c}$$

Therefore, for the hybrid to do better than the pure pattern partitioning scheme [9],

$$\begin{array}{lccl} & I & > & \frac{L_c}{\sqrt{P_c}} \\ or & I & > & \frac{L}{P_1\sqrt{P_c}} \\ or & L & < & IP_1\sqrt{P_c} \end{array}$$

Thus we expect the hybrid scheme to outperform pure pattern partitioning for recognition problems, provided $P_1\sqrt{P_c} > \frac{L}{I}$, $P_1 \neq P$.

The t_w term of the hybrid scheme is smaller than the t_w term of the pure checkerboarding scheme if the following condition is met:

$$\begin{array}{llcl} & 2LJI\frac{\log(P)}{\sqrt{P}} & > & 2L_cJI\frac{\log(P_c)}{\sqrt{P_c}} + 2JI^2\frac{\log(P_1)}{P_c} \\ or & L\frac{\log(P)}{\sqrt{P}} & > & L_c\frac{\log(P_c)}{\sqrt{P_c}} + I\frac{\log(P_1)}{P_c} \\ or & L\log(P) & > & L_c\sqrt{P}\frac{\log(P_c)}{\sqrt{P_c}} + I\sqrt{P}\frac{\log(P_1)}{P_c} \\ or & L\log(P_1) + L\log(P_c) & > & L_c\sqrt{P}\frac{\log(P_c)}{\sqrt{P_c}} + I\sqrt{P}\frac{\log(P_1)}{P_c} \end{array}$$

The inequality can be simplified to :

$$\begin{array}{llcl} & L\log(P_1) & > & I\sqrt{P}\frac{\log(P_1)}{P_c} \\ or & L & > & I\frac{\sqrt{P_1}}{\sqrt{P_c}} \end{array}$$

since

$$\begin{array}{lcl} L\log(P_c) & > & L_c\sqrt{P}\frac{\log(P_1)}{P_c} \\ & > & \frac{L}{P_1}\frac{\sqrt{P}}{\sqrt{P_c}}\log(P_c) \\ & > & L\log(P_c)\frac{\sqrt{P}}{P_1\sqrt{P_c}} \\ & > & L\log(P_c)\frac{1}{\sqrt{P_1}}; \Leftarrow\ always\ true\ for\ P_1 > 1 \end{array}$$

Thus, we expect the hybrid scheme to outperform checkerboarding if a large number of processors are available or if the network is small, or if the learning sample set is large.

[9] The actual bound may be a bit looser because of the way we are computing the constraints: ie for $A+B > C+D$, one interpretation could be that $A > C$, and $B > D$, and another $(A - C) > (D - B)$. While the latter is more accurate, we are using the first method to simplify the discussion. Actually, if the latter case is used, the conditions are in the hybrid scheme's favor.

Appendix D Theoretical Analysis of Non-uniform Networks

To simplify the algebraic analysis, we will compare the speedups of the alternative network partitioning schemes under the constraints of high processor utilization. High processor utilization refers to the level of processor utilization attained by these schemes for the case of uniform networks. The algebraic analysis can be generalized to other cases with added complexity for considering the trade-off between speedup and processor utilization.

The total cost of one iteration of checkerboarding can be approximated as $\frac{LW}{P}t_c + 2JL\log(P)t_s + 2RL\log(P)t_w$, where $W = I_0I_1 + I_1I_2 + \cdots + I_{J-1}I_J$, and $R \cong \lceil\frac{I_1}{P}\rceil + \cdots + \lceil\frac{I_J}{P}\rceil$. The cost model for an iteration of vertical sectioning scheme training can similarly be generalized to $\frac{LW}{P}t_c + 2JLPt_s + 2SLt_w$, where $S \cong P[\lceil\frac{I_1}{P}\rceil + \cdots + \lceil\frac{I_J}{P}\rceil]$. Note that the t_s and t_w terms are smaller for checkerboarding. The t_w for checkerboarding is better for $\frac{R}{S}\log(P) < 1$. This condition is true if $\frac{\log(P)}{\sqrt{P}} < 1$; i.e. $P > 16$. Thus all cost components of checkerboarding are strictly better than those of vertical sectioning, for a large number of processors. Overall, checkerboarding will perform strictly better than vertical sectioning scheme.

The hybrid checkerboarding and pattern partitioning schemes can be extended to non-uniform networks. The pattern partitioning scheme needs no modifications for a non-uniform network. The hybrid scheme needs modifications in its component related to checkerboarding. The cost for the various schemes is provided in table 9.

Scheme	Cost
Vertical Sectioning	$\frac{LW}{P}t_c + 2JLPt_s + 2SLt_w$
Checkerboarding	$\frac{LW}{P}t_c + 2JL\log(P)t_s + 2LR\log(P)t_w$
Pattern Partitioning	$\frac{LW}{P}t_c + 2\log(P)t_s + 2W\log(P)t_w$
Hybrid	$\frac{LW}{P}t_c + t_s[2JL_c\log(P_c) + 2\log(P_1)] + t_w[2L_cR\log(P_c) + 2W\log(P_1)]$
	where W = number of weights in the network

Table 9: Cost model for non-uniform networks.

.

The region of dominance of the three schemes does not differ very much from the case of uniform networks. Checkerboarding is preferred if $LR \ll W$, since the t_w term of pattern partitioning becomes worse. Similarly, pattern partitioning is preferred if $W \ll LR$. If $\sqrt{P}$ divides I_k for all layers, then $R = \frac{I_1+I_2+\cdots+I_J}{\sqrt{P}}$ can be used to simplify these expressions. Checkerboarding will be preferred if $L \ll \frac{W}{R} = \frac{\#weights}{\#nodes}\sqrt{P}$, which will be the case in recognition problems. The hybrid scheme will be better than pure pattern partitioning for $W > \frac{L}{P_1}R$. The hybrid scheme will be better than the checkerboarding scheme for $L > \frac{W}{RP_c}$, or for $W < LRP_c$. This can be derived via algebraic manipulations similar to those used in Appendix C. These conditions reduce to corresponding conditions on a uniform network by substituting $W = JI^2$ and $R = \frac{JI}{\sqrt{P}}$.

Appendix E Specification of Checkerboarding Algorithm Iterations

The checkerboarding algorithm iterations is specified in a programming language like notation in this appendix along with a detailed figure 7 illustrating the communication pattern in the forward propagation phase. The description specifies a set of processes, mapping of processes to processors, division of various data-structures onto processes, and the basic steps of checkerboarding for a set training regime. The outermost sequential for loop over patterns can be simplified to simulate a per-pattern training regime.

```
PROCEDURE parallel_BP_iteration( I, J, P : integer);
/* I nodes per layer, J layers + input layer 0, P processors */
assert(sqrtP = floor(sqrt(P)); assert(rootP = sqrtP-1);
assert(lSize = ceiling(I/sqrtP) - 1); /* #nodes/layer/diagonal processor - 1 */
TYPE    rcSet = 0 .. rootP; layerSet = 0 .. J; Iset = 0 .. (I - 1); lSizeSet = 0 .. lSize;
        local_vector = array[lSizeSet] of real;
VAR i: Iset; j : layerSet; row, col : rcSet;
  prcs: array [rcSet,rcSet] of process map [row, col] to processor[exp(2,sqrtP)*row + col];
  act, err: array [Iset,layerSet ] of real map[i,j] to prcs[i mod sqrtP, i mod sqrtP];
  delW, W: array [Iset,Iset,layerSet] of real map[i1,i2,j] to prcs[i1 mod sqrtP, i2 mod sqrtP];
  sum: array [rcSet,rcSet,lSizeSet ] of real map[i1,i2,lSizeSet] to prcs[i1, i2];
BEGIN
 parallel for(row, col in rcSet) do init( row, col, i, j);
 for (pattern in patternSet) do
 begin
   /* Forward Propagation */
   for j := 1 to J do begin
     parallel for col in rcSet do one-to-allBroadcast(prcs[col,col], prcs[rcSet,col], lActSet(col,j));
     parallel for( row , col in rcSet ) do local_weighted_sum(row, col, j);
     parallel for (row in rcSet) do all-to-oneAccumulate(prcs[row,row], prcs[row,rcSet], lSumSet(row,rcSet,j+1));
     parallel for (row in rcSet) do compute_activations(row, j);
   end;
   /* Error Back Propagation */
   parallel for (col in rcSet) do compute_output_error(col, J);
   for j := (J - 1) to 1 do begin
     parallel for (row in rcSet) do one-to-allBroadcast(prcs[row,row], prcs[row,rcSet], lErrSet(row,j));
     parallel for (row , col in rcSet ) do local_error_sum(row, col, j);
     parallel for (col in rcSet) do all-to-oneAccumulate(prcs[col,col], prcs[rcSet,col], lSumSet(rcSet,col,j-1));
     parallel for (col in rcSet) do compute_non_output_error(col, j);
   end;
    parallel for (row , col in rcSet ) do compute_delta(row, col, j); /* Deltas */
 end;
  parallel for (row , col in rcSet ) do adjust_weights(row, col, j);
END;
```

We use a set of P processes arranged in a logical square grid of size $\sqrt{P} \times \sqrt{P}$ to create a one-to-one mapping between processes and processors. The processes are mapped onto hypercube such that rows and columns of logical process grids are mapped to a set of processors, forming a hypercube. The activation vector and error vector elements are divided among the processors on the principal diagonal of the logical grid. The weight matrix is partitioned among the processes in a checkerboard manner.

Forward propagation is divided into four steps per layer shown in the code as well in illustrated in figure 7. Diagonal processes broadcast the local activation vector elements (lActSet(col ,j)) from layer j to other processors in their row. Each process then computes a set of weighted sums towards computing the activation elements mapped to the diagonal processors in its column. The weighted sum is accumulated at diagonal processes along columns to provide a total weighted sum (lSumSet(row,rcSet,j+1)) for computing the activation elements in the next layer. Finally, the diagonal processors apply sigmoidal functions to the weighted sum. By repeating these steps for each layer, forward propagation is completed.

The Backpropagation of error is carried out in a similar fashion. Deltas are computed on all processes in parallel. Each process has the relevant activation vector elements and error vector elements to compute the delta for the local weights. The deltas are accumulated over patterns in the outermost sequential for loop over patterns. Weights are adjusted at the end of the iteration.

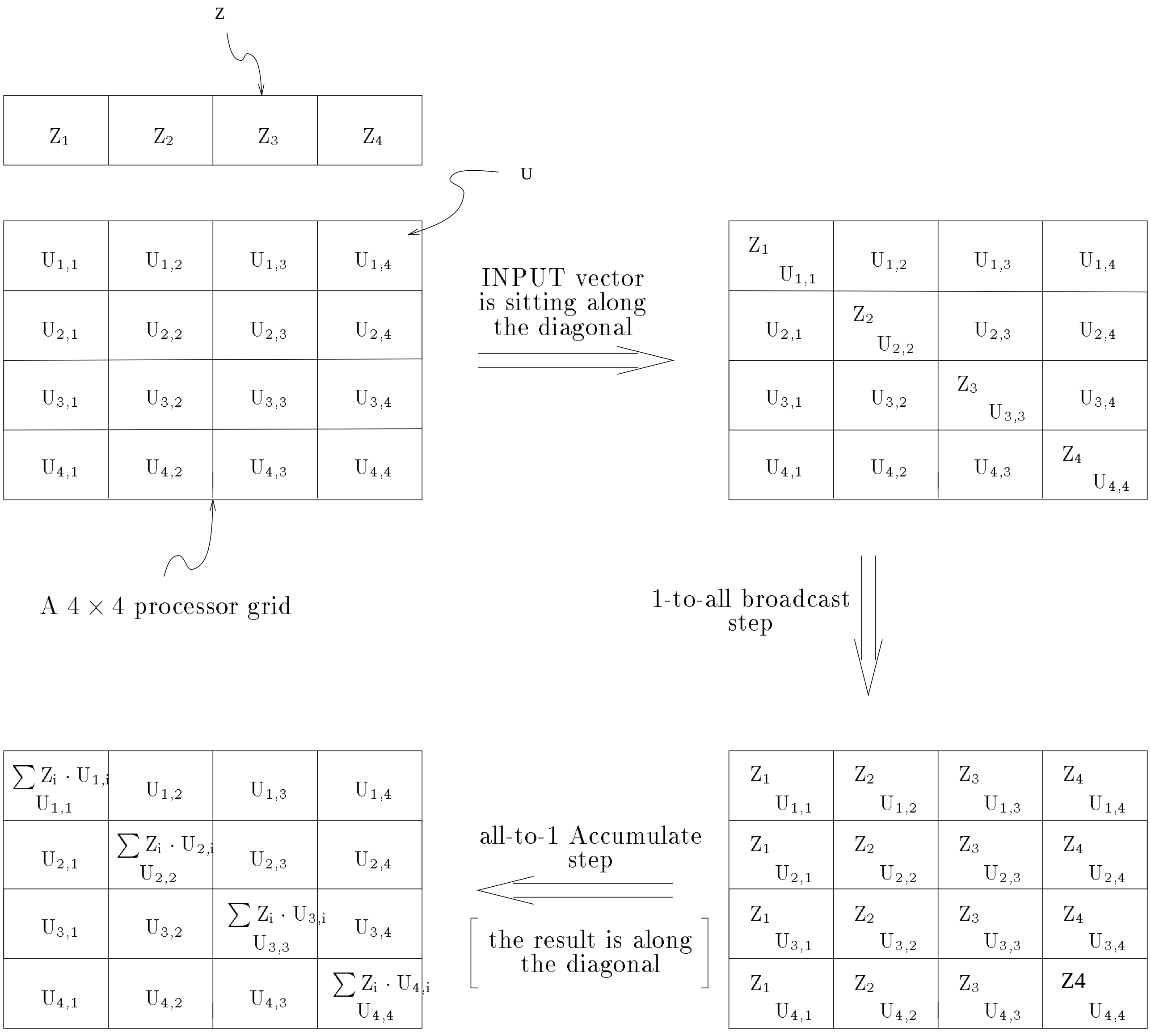

Figure 7: The communication pattern and computation steps for a 4×4 matrix and 4×1 vector