OPTIMAL MAPPING OF NEURAL-NETWORK SIMULATIONS ONTO MESSAGE-PASSING MULTICOMPUTERS

BY

LON-CHAN CHU

B.S., National Taiwan University, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

ABSTRACT

In this thesis, optimal mapping of multilayered feed-forward artificial neural networks onto message-passing multicomputers is formulated as an integer programming problem. The objective is to minimize completion time of parallel simulations of neural networks on target multicomputers with respect to expected computation and communication times. This optimization problem is NP-hard, in general, and a branch-and-bound algorithm is described for solving the problem. By observing that computation time is dominant over communication time in most cases, a simplified algorithm with negligible error is described and analyzed. Both static and dynamic mapping algorithms are studied for mapping neural networks onto multicomputers depending on whether workload is static or time-varying. The static algorithm performs the optimal mapping once before the neural-network simulation is started. The dynamic algorithm performs the optimal mapping during the neural-network simulation whenever workload changes significantly. Experimental results for the static mapping algorithm, including those on a network of workstations and Intel iPSC/2 hypercube computers of different cube sizes, are shown and are found to be very close to those predicted by analysis. Experimental results for the dynamic mapping algorithm, including those on 3-processor, 10-processor, 25-processor and 100-processor multicomputers, are obtained by simulations. The simulations are required for controlling and reproducing the workload. Numerical results are presented to demonstrate the advantage of the dynamic mapping algorithm.

ACKNOWLEDGEMENTS

I express my sincere gratitude to all the people who have helped me in the course of my graduate study. My thesis advisor, Professor Benjamin W. Wah, was always available for discussions and encouraged me to explore new ideas. I am grateful to Mark Gooley, Pankaj Mehra, Zhang Zhou, Kumar Ganapathy and all my colleagues in the Center for Reliable and High-performance Computing for all the invaluable suggestions and references.

I thank my wife, Yi-Wen, and my parents for many years of love and support.

Finally, I express my appreciation for the financial supports provided by National Science Foundation Grant MIP 88-10584, National Aeronautics and Space Administration Contracts NCC 2-481, NAG 1-613, and Joint Services Electronics Program N00014-90-J-1270.

TABLE OF CONTENTS

CHAPTER 1.

INTRODUCTION

In this thesis, optimal mapping of multilayered feed-forward artificial neural networks (ANNs) onto message-passing multicomputers is studied. The objective is to minimize completion time of parallel ANN simulations on target multicomputers with respect to expected computation and communication times. Both static and dynamic mapping schemes are studied for mapping neural-network simulations onto multicomputers depending on whether the workload is static or time-varying. The static scheme performs the optimal mapping once before the ANN simulation is started. The dynamic scheme performs the optimal mapping whenever the workload changes significantly. To run experiments on the algorithm for solving the static mapping problems, a program called *NeuMap* was developed. Also, a program called *Dsim* was developed for simulating workload-varying multicomputers to run experiments on the dynamic mapping problems.

During the course of exploring methods for solving the optimal mapping problems, several novel techniques were discovered. The major contributions of this thesis are

(1) integer programming formulation of the optimal mapping of ANNs onto message-passing multicomputers,

(2) constrained task graph for modeling the ANN simulation,

(3) novel strategy for mapping coarse-grained task graphs onto message-passing multicomputers,

(4) novel strategy for mapping the task graphs with many coarse-grained task nodes and few fine-grained task nodes onto message-passing multicomputers,

(5) decomposition of error allowance in the multistage solution algorithm for combinatorial optimization problems,

(6) determination of suitability of an existing multicomputer system for ANN applications, and

(7) implementations of NeuMap and Dsim.

1.1. Motive and Approach

ANNs show strong promise in artificial intelligence applications [13, 14]. However, the technologies of implementing ANNs in hardware are not fully mature, and only simple and medium-scale ANNs can be implemented in VLSI at this time [8]. The flexibility of hardware implementation is limited in the sense that it is not easy to change its topology. ANNs are usually studied by simulations on existing computer systems. These simulations require large amounts of computational time and are very attractive candidates for parallel processing.

There are two approaches to parallel processing of ANN simulations. In the first approach, simulations can be coded in an existing programming language, and a parallelizing compiler extracts the parallelism to the extent possible and restructures the program for parallel execution. Second, the mapping of simulations can be studied at the algorithm level, and parallel simulation algorithms can be developed. In this thesis, the second approach is used because a greater amount of parallelism can be exploited, given the knowledge about the problem and the well-defined nature of the simulations.

The target ANNs studied are multilayered networks trained by the static learning rule, *e.g.*, back-error propagation. Two strategies for coping with nonlayered ANNs are also studied. Larger networks with an arbitrary interconnection are not considered in this thesis because mapping these networks is not solvable under the constraints of computer resources at the present time. The target multicomputers are those in which each processor has local memory and the workload is static or time-varying, instead of the single-user environment in an earlier paper [18]. Other recent additions to this earlier work include study of dynamic mapping algorithms as well as static, and significant new results on and techniques for optimal mapping.

The optimal mapping of learning the weights of an ANN onto a static-workload multicomputer is formulated as an integer programming problem with respect to computation and communication times. Constraints on feasibility, configuration, resource and dependence are considered. To reduce the complexity of solving the mapping problem, the multicomputer system is partitioned into disjoint sets of processors according to the ratio of communication to computation times. Each set of processors represents a conceptual processing resource.

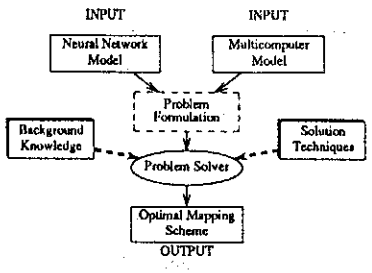

Figure 1.1 : General approach to solving the optimal static mapping.

Based on the limited history of past workloads and limited knowledge of future workloads, it is very difficult to achieve the optimal mapping dynamically when the workload changes. The approach proposed in this thesis approximates the optimal dynamic mapping by scheduling a sequence of optimal mapping, based on the variance of time-varying workloads and the expected time to finish the remaining simulation tasks.

The approach to the optimal mapping of ANN simulations onto static-workload multicomputers is shown in Figure 1.1. As the general optimal static mapping problem is NP-hard, appropriate techniques, such as enumeration and tree search, are considered [6]. Experiments on static-workload and workload-varying message-passing multicomputers are carried out for validating the correctness of the mappings predicted by analysis.

1.2. Related Work

Related work on this problem includes parallel software simulations on multiprocessors, design of generic multicomputers for ANN simulations, and implementations of computers or VLSI chips for ANNs.

H. T. Kung *et al.* map layered ANNs onto a linear array WARP processor with 10 cells [16]. They propose two approaches: network partitioning and data partitioning. The partitioning of an ANN into slices is optimal because the target machine is a ring of processors and the neural network is assumed to be layered. In partitioning data, they use the first nine cells in WARP to perform operations in the forward production phase and the tenth cell for computing weight updates in the backward training phase. That is, one weight update is done for every nine training patterns. They assume that weight updates are usually small, and that consecutive weight updates can be neglected by running several iterations of the simulations with fixed weight values before updating them. In practice, weight updates are not necessarily small and the range of updates is highly application dependent. Moreover, the scheme may not be satisfactory when a large number of cells (or processors) are concerned.

Hwang *et al.* design generic multicomputers suitable for ANN simulations [7, 9]. They discuss design issues on the processing elements and the communication bandwidth requirements, and propose several guidelines for designing generic multicomputers for ANN simulations. However, they use datagram routing, which may result in unpredictable network congestion. The performance of their scheme also depends on the system-supported routing algorithm.

The weight-update process in a multilayered ANN can be considered as a sequence of matrix-vector multiplications. By exploiting this approach, S. Y. Kung *et al.* transform ANN learning to recursive matrix operations, then to a data dependence graph, and finally to a linear systolic array with a fast interconnection network [11, 12]. Active neurons in each layer are evenly distributed to the processing cells of the systolic array, and full resource utilization is obtained in many cases. In fact, this thesis will show that their scheme is optimal when the ANN is layered and the interconnection network is fast. However, they did not consider the case in which the bandwidth is limited and not the same at all links and the processors have different computational capacity. In the latter case, active neurons may not be evenly distributed among all processing cells.

A number of other multiprocessor simulations have been reported. Researchers at Edinburgh simulate ANN learning on a transputer-based Computing Surface with 42 processors [4]. Researchers at Rochester use a 128-node BBN Butterfly multicomputer for simulating ANNs [3].

This thesis addresses some of the deficiencies found in previous studies which either assume a tightly coupled system, such as a systolic array, or present a heuristic mapping algorithm for a set of heterogeneous processors. The integer programming solution presented allows ANN simulations to be carried out optimally on a network of heterogeneous processors. These results are important for designing special-purpose computers for ANN simulations and for determining the suitability of an existing multicomputer system for ANN applications.

1.3. Organization of Thesis

The thesis is organized into eight chapters and five appendices. Chapters 2, 3, and 4 define the model of the ANN, the target multicomputer, and the mapping scheme, respectively. Chapter 5 formulates the mapping problem as an integer programming problem and presents the related constraints. Chapter 6 discusses the solution strategy, techniques and properties. Error decomposition, partitioning algorithms, related theorems, and a branch-and-bound search are described. Chapter 7 describes the experiments on static-workload multicomputers, including those on a 16-node Intel iPSC/2 hypercube computer and on a bus-based network of heterogeneous workstations. Also, it describes the experiments on workload-varying multicomputers, including those on 3-processor, 10-processor, 25-processor and 100-processor multicomputers. Finally, in Chapter 8 the conclusions are drawn.

Brief descriptions and C programs of NeuMap and Dsim with appropriate documentation are shown in Appendices A and B, respectively. Input files they require are also listed.

CHAPTER 2.

MODEL OF ARTIFICIAL NEURAL NETWORKS

In this chapter, the operations of an ANN and its task-graph representation are described, and then the model of the ANN is formalized. The task graph used to represent the ANN operations is slightly different from the traditional one, since certain relations exist between pairs of task nodes, which will be discussed in this chapter.

This model works for the multilayered ANN with a *static learning rule*, which is the one whose learning tasks are static. The one whose learning tasks are time-varying is a *dynamic learning rule*. Certain techniques can be applied to nonlayered ANNs to restructure them into multilayered ANNs. However, the exact optimality of mapping nonlayered ANNs may not be achieved due to error incurred in restructuring. The restructuring techniques and the error will be discussed in this chapter. For the ANN with a dynamic learning rule, the static task graph can not characterize it.

2.1. Basic Operations of ANN

An ANN can be characterized by several major components: a set of neurons, pattern of interconnection, propagation rule, activation rule, output function and learning rule.

A neuron is the basic processing unit, which is characterized by its state, an activation function and an output function, as shown in Figure 2.1. The activation function transforms the input signals associated with their weights and its state value to a new state value. The output function transforms the state value to an output signal.

Neurons can be classified into three types: *input neurons*, *hidden neurons* and *output neurons*. Input neurons receive inputs from the external environment, output neurons send signals to the external environment, and hidden neurons are invisible to the external environment.

The pattern of interconnection determines the dependence of signal flows in a neural network. The propagation rule specifies the formation of the *net input* of a neuron. The activation rule specifies the transformation from the weighted inputs, the global signal (usually used as a

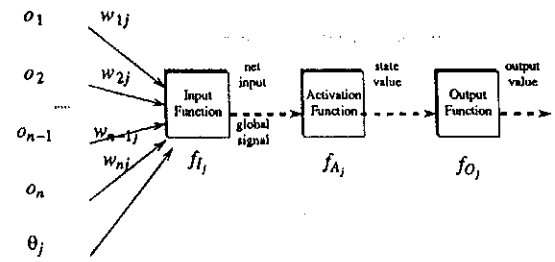

Figure 2.1 : Generic model of a neuron.

threshold) and the current state into a new state. The output function transforms the state of a neuron into an output signal.

The learning rule specifies the mechanism of modifying the strength of a connection. The neural network can learn through incremental modification of connection weights. In general, the modification of a connection weight is a function of four items: (1) the state value of the destination neuron of this connection; (2) the output value of the source neuron of this connection; (3) the current connection weight; and (4) the *teaching input*, which is the expected output value of the destination neuron. For example, the typical back-error propagation (BEP) learning rule is

$$w_{ij}(t+1) = w_{ij}(t) + g\,\delta_j(t)\,o_i(t) + \alpha\,(w_{ij}(t) - w_{ij}(t-1)), \tag{2.1}$$

where g is a gain term, $o_i(t)$ is the output signal of neuron i at time t, α is a momentum term (sometimes set to zero), and $\delta_j(t)$ is an error term for neuron j. If neuron j is an output neuron, then

$$\delta_j(t) = o_j(t)\,(1 - o_j(t))\,(\zeta_j(t) - o_j(t)), \tag{2.2}$$

where $\zeta_j(t)$ is the teaching input to output neuron j. If neuron j is a hidden neuron, then

$$\delta_j(t) = o_j(t)\,(1 - o_j(t))\,\sum_k \delta_k(t)\, w_{jk}(t+1). \tag{2.3}$$

A multilayered neural network can be *clusiered* such that if one neural cluster is connected to another, then all neurons in the first are connected to all neurons in the second. For brevity, in this thesis, a *cluster* is used to refer to a neural cluster. A special case is a fully connected multilayered neural net, which has one cluster in each layer. Note that all neurons in a cluster are *homogeneous* in the sense that they receive the same input signals, perform the same sequence of operations, and send their output signals to the same clusters of neurons.

The operations of an ANN can be divided into two phases: a *production phase* and a *learning phase*. The ANN works by alternating between these two phases. In the production phase, it receives input signals from the external environment and produces output signals to the external environment. In the learning phase, it receives teaching inputs, if they are provided, and modifies the connection weights according to the learning rule.

2.2. Constrained Task Graph

The simulation of an ANN can be represented by a task graph. A traditional task graph consists of *task nodes* and *precedence arcs*. Each task node and each precedence arc , respectively, represent a well-defined task to be performed and a dependence relationship between two task nodes. Consider a task node. The task of its *preceding* node must be performed before its task can be performed. On the contrary, the task of its *succeeding* node cannot be performed until its task is performed.

A task node represents either a production-phase or learning-phase simulation task of a cluster and the precedence arc represents the dependence between two clusters. Without ambiguity, the *cluster* of a task node is referred to the one which the task node is representing. The *size* of a task node is the amount of unit computation to be performed for this node. Note that the task node size depends on the *fan-in* of the cluster if the task node is in the production phase, whereas, the size depends on the *fan-in* and *fan-out* of the cluster if the task node is in the learning phase. The *width* of a precedence arc is the amount of unit communication between two clusters. Note that the task graph is much like the network topology in terms of clusters.

The task graph representing the ANN simulation is called the *constrained task graph* (CTG). A typical example is shown in Figure 2.2. It is slightly different from the traditional one and the differences are (1) the CTG is *vertically symmetric*, and (2) a task node and its symmetry *image* task node correspond to the same cluster in that one of them represents a production-phase task and its image a learning-phase task. A major feature of the CTG is that each task node in the CTG consists of two different types of subtasks, *i.e.*, a computation subtask and a communication subtask.

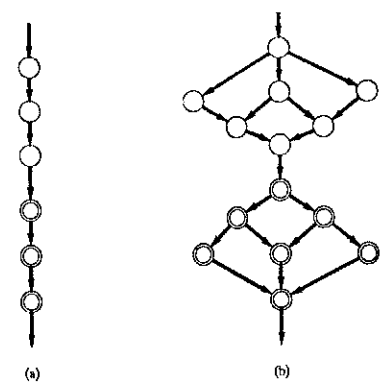

Figure 2.2 : Two examples of CTG: (a) a fully connected ANN and (b) a multilayered ANN.

Formally, each task node i consists of n_i binary *task tuples*, where n_i is the number of neurons in the cluster of task node i. Each task tuple defines the amounts of unit computation and

unit communication of a neuron in its cluster. The task tuples are *indivisible* in the sense that one task tuple cannot be executed across different processors, namely, the task tuple is the basic unit for mapping. Each task tuple contains two items: a computation subtask (*c-subtask*) and a communication (or routing) task (*r-subtask*). A c-subtask is performed only at one processor and a r-subtask is accomplished by migrating data in the multicomputer. The *home node* of a subtask, either c-subtask or r-subtask, is the task node which contains this subtask.

A subtask is *irrevisable* if the size of the subtask cannot be altered during the mapping process. On the contrary, a subtask is *revisable* if the size of the subtask can be changed during the mapping process. The c-subtasks are irrevisable and the r-subtasks are revisable. The r-subtasks are not well-defined *a priori* and can be revised only after all the c-subtasks of its home node and all the c-subtasks of all the succeeding nodes of its home node are mapped. Note that all c-subtasks within a task node are homogeneous, since the computation of all neurons in a cluster is the same. However, the r-subtasks are usually different, since they depend on the interconnection network of the multicomputer and also on the mapping of related c-subtasks. Figure 2.3 shows a task node in which the solid boxes and dotted boxes denote the c-subtasks and r-subtask, respectively.

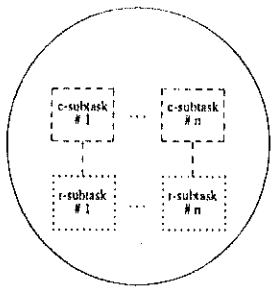

Figure 2.3 : A task node of the CTG.

In this thesis, it is assumed that each neuron is simulated on only one processor and cannot be simulated across processors. As a result, a task node and its image node must have the same mapping of c-subtasks.

The mapping of the ANN simulation can be manipulated on this high-level task graph rather than the low-level detailed ANN operations [18]. One advantage of using the task graph representation is that the details of the ANN operations need not to be gone through. Another advantage is that the task graph representation is independent of the learning rule, as long as the learning rule is static, such that general ANNs with static learning rules can be represented instead of being limited to the BEP [18].

2.3. Formal Model

A *neural network specification*, S_{ANN}, is defined formally as $<NC, KS, LR>$, where *NC* is the neural network configuration, *KS* is the cluster specification, and *LR* is the learning rule. The *neural network configuration NC* is a 4-ary tuple $<K, I_N, K_I, K_O>$, where K is the set of clusters (n_K is the number of clusters), I_N is the interconnection matrix specifying the pattern of interconnection of clusters, K_I is the set of input clusters, and K_O is the set of output clusters. The *cluster specification KS* is a 9-ary tuple $<n, f_I, f_A, f_O, f_J, f_T, \eta_p, \eta_l, \eta_s>$, where n is the number of neurons in this cluster, f_I is the propagation rule of neurons in this cluster in the production phase, f_A is the activation function, f_O is the output function, f_J is the propagation rule in the learning phase, f_T is the learning function, η_p is the amount of computation of a neuron in the production phase, η_l is the amount of computation of a neuron in the learning phase, and η_s is the amount of storage needed for the operations of one neuron. The *learning rule LR* specifies the learning mechanism (*e.g.*, BEP) used in the ANN simulation. Note that if every cluster has the same specification, then one specification is sufficient. Example 2.1 below shows the specification of a fully connected 3-layered ANN.

The *neural network model*, M_{ANN}, is defined formally as $<N_G, I_G>$, where N_G is a set of task nodes and I_G is a matrix indicating the interconnection of task nodes. Let n_Z denote the number of task nodes in the CTG. Each task node z_i can be represented by a ternary tuple $<n_{z_i}, \eta_{z_i}, \eta_{s,z_i}>$, where n_{z_i} is the number of neurons in this task, η_{z_i} is the amount of computation for

each c-subtask, and η_{s,z_i} is the space usage for each neuron in this task. Note that z_i represents the cluster $k_{\phi(i)}$, where $\phi(i) = n_K - |i-(n_K+.5)| + .5$. Recall that n_K is the number of clusters. For example, the task nodes z_1 and z_{2n_K} correspond to the cluster k_1, and z_{n_K} and z_{n_K+1} refer to the cluster k_{n_K}.

Example 2.1. Consider a fully connected 3-layered neural network with 500, 1000, and 200 neurons in layers 1, 2, and 3, respectively. The ANN can be specified as follows.

Neural Network Specification $S_{ANN} = < NC, KS, LR >$

Neural Network Configuration $NC = < K, I_N, K_I, K_O >$:

Neuron Cluster Set $K = \{ k_1, k_2, k_3 \}$, $n_K = 3$.

Interconnection Matrix $I_N = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

Set of Input Neuron Cluster $K_I = \{ k_1 \}$.

Set of Output Neuron Cluster $K_O = \{ k_3 \}$.

Neuron Cluster Specification $KS = < n, f_I, f_A, f_O, f_J, f_T, \eta_p, \eta_l, \eta_s >$:

Propagation Rule f_I : additive, multiplicative.

Activation Function f_A : sigmoid function.

Output Function f_O : identity function.

Propagation Rule f_J : additive, multiplicative.

Training Function f_T : BEP training function with α=0.

Number of Neurons n:

$n_{k_1} = 500$, $n_{k_2} = 1000$, $n_{k_3} = 200$.

Amount of Computation η_p:

$\eta_{p,k_1} = 500\,\eta(f_I) + \eta(f_A) + \eta(f_O)$,

$\eta_{p,k_2} = 500\,\eta(f_I) + \eta(f_A) + \eta(f_O)$,

$\eta_{p,k_3} = 1000\,\eta(f_I) + \eta(f_A) + \eta(f_O)$.

Amount of Computation η_l:

$\eta_{l,k_1} = 500\,\eta(f_J) + 500\,\eta(f_T)$,

$\eta_{l,k_2} = 1000\,\eta(f_J) + 500\,\eta(f_T)$,

$\eta_{l,k_3} = \eta(f_J) + 1000\,\eta(f_T)$.

Amount of Storage η_s:

$\eta_{s,k_1} = 500 + 1$ (word), $\eta_{s,k_2} = 500 + 1$ (words), $\eta_{s,k_3} = 1000 + 1$ (words).

Learning Rule LR : BEP.

Neural Network Model $M_{ANN} = < N_G, I_G >$

Task Node Set $N_G = \{ z_1, z_2, z_3, z_4, z_5, z_6 \}$, $n_Z = 6$.

$n_{z_i} = n_{k_i}$, $\eta_{z_i} = \eta_{p,k_i}$, and $\eta_{s,z_i} = \eta_{s,k_i}$, for i=1,2,3

$n_{z_{i+3}} = n_{k_i}$, $\eta_{z_{i+3}} = \eta_{l,k_i}$, and $\eta_{s,z_{i+3}} = \eta_{s,k_i}$, for i=1,2,3

Interconnection Matrix $I_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

2.4. Coping with Nonlayered ANN

The target ANNs studied in this thesis are multilayered. However, in general, an ANN may be nonlayered. To fit nonlayered ANNs into the mapping problem, restructuring either the ANN or CTG is necessary. The restructuring will be discussed in the following two subsections. The calculation of errors in optimality incurred in the restructuring will be discussed in Section 6.3.

2.4.1. Restructuring a nonlayered ANN into a multilayered ANN

The nonlayered ANN can be restructured into a "multilayered" ANN. After the ANN is restructured, it can be formally modeled in the CTG. However, solving the best restructuring is harder than solving the original optimal mapping problem, since the quality of restructuring cannot be determined until the optimal mapping problem is solved. Consequently, the restructuring must be heuristic and some errors will be incurred. It is desirable that the error bound can be calculated.

A *pure* multilayered ANN has each cluster in a layer communicating only with the clusters in its neighboring layers. However, a nonlayered ANN may not be able to be restructured

into a pure multilayered ANN, since the above definition may not be satisfied. However, the nonlayered ANN can be restructured into *wavefronts* of clusters such that a wavefront of clusters can be active concurrently; namely, there is no dependence among the clusters in a wavefront. Note that a cluster in a wavefront may communicate with certain clusters not in its neighboring wavefronts. Such an ANN consisting of wavefronts is called a *semi-layered* ANN, one in which a cluster may communicate with certain clusters not in its neighboring layers. After the nonlayered ANN is restructured into a semi-layered ANN, it fits the optimal mapping problem. The derivation from optimality incurred by restructuring can be derived based on the ratio of communication to computation times, which will be described in Section 6.3.

2.4.2. Restructuring a constrained task graph

Another way to cope with the nonlayered ANN is restructuring the CTG. This is especially useful to handle singular task nodes in the CTG. A *singular* task node is the one whose size is very limited compared to those for the majority of task nodes in the CTG. Since the singular task node is relatively small, it can be merged into a neighboring task node. The error in optimality incurred by restructuring can be derived based on the ratio of the size of the singular task node to the size of the larger task node into which the singular one is to be merged. The calculation of the error incurred will also be described in Section 6.3. As with restructuring an ANN, the optimal restructuring of a CTG is difficult and impractical. Heuristic restructuring with a guaranteed error bound is sufficient and is adopted in this thesis.

CHAPTER 3.

MODEL OF MULTICOMPUTERS

In this chapter, the basic architecture of multicomputers and the characterization of the time-varying workload are described. Then the model of multicomputer will be formalized.

3.1. Basic Architecture of Multicomputers

A multicomputer is a system consisting of a set of processors each with local memory, a set of communication links which connect the processors, workload descriptors and queuing disciplines. The target multicomputer may have a time-varying workload. The queuing discipline is scheduled by the mapping.

A processor consists of a processing unit, its workload descriptor, its local memory and a set of communication ports through which it can communicate with other processors. A processor may also have an I/O facility for communication with the external environment. The computation power of a processor is characterized by the *execution time per unit computation*, which includes the processing unit activity and memory-access activity. The size of the local memory of each processor is a constraint in the mapping problem.

The interconnection network in most multicomputers can be classified into one of four classes: point-to-point, multistage interconnection networks (MIN), crossbar and bus. The complexity of the routing problem mainly depends on the interconnection network and the traffic on it. The routing problem for a bus is the simplest, while that for a point-to-point network is the hardest. For the second case, it can be simplified if a broadcast mechanism is supported. In this thesis, only the point-to-point and bus interconnection networks are considered, because it is difficult to cover all possible interconnection networks. MIN and crossbar interconnection networks are treated as fully connected point-to-point networks.

The traffic load in the interconnection network can be merged into the workload of processors. This merge is appropriate, since the interconnection network can be treated as a set of resources just like processors and each of them has its own workload. Another reason is that the ANN simulation alternates computation and communication iteratively such that the effects

of communication can be merged with the computation. Hereafter, the workload used in this thesis refers to the real workload in a processor combined with the effect of random traffic within the interconnection network.

A communication link i can be modeled by a setup time τ_{s_i}, a transfer rate r_i, and a set P_{ls_i} of processors it supports. The communication setup time includes that of the physical link and overhead for processing setup. This parameter can be obtained for a real system by measuring the time for sending a null message. The transfer rate indicates the amount of unit data which can be transmitted over a link per unit time. The overhead of processing transmission is included in calculating transmission time. The transmission time τ_{l_i} per unit datum is the reciprocal of the transfer rate. The processors supported by a link are those which communicate with other processors via this link.

Note that overlapped computations and communications may be allowed. If the associated overhead is small, then such overlaps should be exploited in the mapping.

3.2. Stochastic Workloads

In a normal operational environment, several processes may be active in a processor during the simulation such that a processor may not be able to be dedicated to executing the ANN simulation. In other words, the workload will affect the final mapping result and should be treated as a mapping parameter. In an earlier paper [18], the time-varying workload is not considered.

In this thesis, the workload is characterized by the *processor utility*, which is the *percentage* of processor utilization allocated for the ANN simulation. Higher workload means lower processor utility.

To be more specific, the workload ω_i in processor i can be characterized by the processor utility μ_i such that $\omega_{i,q} = 1 / \mu_{i,q}$ in the q-th time quantum. A *time quantum* is a time interval during which the workload can be meaningfully measured. Note that $0 < \mu_{i,q} \leq 1$ and $1 \leq \omega_{i,q}$. The time quantum is selected to be large enough to allow μ_i to be nonzero. This characterization is appropriate, because if the ANN simulation can utilize only μ_i within a quantum, then the expected number of active processes should be the reciprocal of μ_i. Without loss of

generality, in this thesis, the time quantum is set to the time for one iteration of ANN simulation, such that μ_i is always nonzero for all i. For simplicity, the workload is assumed to be static during a time quantum.

In a degenerate case, ω_i is 1 for every processor i. This is the single-user case. Another interesting case has a certain set of processors with heavy workloads, such as those of a file server or network gateway, and other processors have relatively light workloads. Given this information on the workload distribution, the mapping will favor those with lighter workloads. Without such information, all the processors will be treated the same; as a result, the mapping may cause bad performance of the simulation.

3.3. Model of Multicomputers

A *multicomputer model*, M_M, is defined formally as $<MC, PS, LS>$, where MC is the multicomputer configuration, PS is the processor specification, and LS is the link specification.

The *multicomputer configuration* MC is a 5-ary tuple $<P, I_M, L, P_I, P_O>$, where P is the set of processors, I_M is the interconnection matrix specifying the interconnection of processors, L is the set of links, P_I is the set of processors that have input facilities, and P_O is the set of processors that have output facilities.

The *processor specification* PS is a 5-ary tuple $<\tau_c, m, \kappa_{ol}, \tau_o, WL>$, where τ_c is the execution time per unit computation, m is the size of local memory, κ_{ol} is a binary variable specifying overlapped computations and communications, τ_o is the overhead of overlapped operations, and WL is the workload descriptor. If the overlap is allowed, then $\kappa_{ol} = 1$; otherwise, $\kappa_{ol} = 0$.

The workload descriptor WL is a 6-ary tuple $<p_0, p_1, p_2, \delta, b_u, b_l>$, where p_0, p_1, and p_2 are the probabilities that the workload in the next iteration of ANN simulation will remain the same, increase, or decrease, respectively, δ is the slope of change in workload if the workload increases or decreases, and b_u is the upper bound and b_l the lower bound on workloads. Note that $p_0 + p_1 + p_2 = 1$. The procedure of workload generation based on the workload descriptor will be described in Section 7.2.

The *link specification LS* is a ternary tuple $<r, \tau_s, P_{ls}>$, where r is the data transfer rate of this link, τ_s is the corresponding setup time, and P_{ls} is the set of processors supported by this link. Note that only one processor is assumed to be able to transmit a frame on a link at any time. This assumption is reasonable, because if k processors can transmit on link l concurrently, then k *logical* links $l_1, ..., l_k$ can be used instead of the single *physical* link l.

The above definition has to be specified for each different processor and each different link in the multicomputer. Examples 3.1 and 3.2 below illustrate the models of a 16-node hypercube and a network of heterogeneous workstations, respectively.

Example 3.1. Consider the Intel iPSC/2 hypercube computer with 16 homogeneous processors and 32 homogeneous links. Note that the hypercube is in the single-user mode so that the workload is always 1.

Multicomputer Model $M_M = < MC, PS, LS >$

Multicomputer Configuration $MC = < P, I_M, L, P_I, P_O >$

Processor Set $P = \{ p_0, p_1, ..., p_{15} \}$ $n_P = 16$.

Interconnection Matrix $I_M = \begin{bmatrix} A & B & B & C \\ B & A & C & B \\ B & C & A & B \\ C & B & B & A \end{bmatrix}$

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Link Set $L = \{ l_0, l_1, ..., l_{31} \}$, $n_L = 32$.

Input Processor Set $P_I = \{$ Cube-Manager $\}$.

Output Processor Set $P_O = \{$ Cube-Manager $\}$.

Processor Specification $PS = < \tau_c, m, \kappa_{ol}, \tau_o, WL >$.

Execution Time Per Unit Computation $\tau_{c_i} = 1.0$ (ms), $\forall\ i$.

Local Memory Size $m_i = 1$ (M words), $\forall\ i$.

Overlapping Feature $\kappa_{ol_i} = 0$, $\forall\ i$.

Overlapping Overhead $\tau_{o_i} = 0$.

Workload Descriptor $WL = < p_0, p_1, p_2, \delta, b_u, b_l >$

$p_{0_i} = 1, p_{1_i} = 0, p_{2_i} = 0, \delta_i = 0, b_u = 1, b_l = 1$, $\forall\ i$.

Link Specification $LS = < r, \tau_s, P_{ls} >$

Point-to-Point Link:

Data Transfer Rate r_i $(1/\tau_{t_i}) = 0.253$ (word/μs), $\forall\ i$.

Setup Time $\tau_{s_i} = 0.65$ (ms), $\forall\ i$.

Supported Processor Set P_{ls_i} ($i = d'_1\ d'_0\ d_2 d_1 d_0$ and $j = d'_1\ d'_0$) =

$\{ (x, y) \mid x = d_{2j} 0 d_{j-1/0}$ and $y = d_{2j} 1 d_{j-1/0} \}$, $\forall\ i$.

Broadcast:

Data Transfer Rate $r_{bc} = 0.097$ (word/μs).

Setup Time $\tau_{s_{bc}} = 6.5$ (ms).

Example 3.2. Consider a network of three heterogeneous workload-varying workstations connected by an Ethernet bus.

Multicomputer Model $M_M = < MC, PS, LS >$

Multicomputer Configuration $MC = < P, I_M, L, P_I, P_O >$

Processor Set $P = \{ p_0, p_1, p_2 \}$, $n_P = 3$.

Interconnection Matrix $I_M = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$

Link Set $L = \{ l_0 \}$, $n_L = 1$. /* Ethernet Bus */

Input Processor Set $P_I = \{ p_0, p_1, p_2 \}$

Output Processor Set $P_O = \{ p_0, p_1, p_2 \}$

Processor Specification $PS = < \tau_c, m, \kappa_{ol}, \tau_o, WL >$

Execution Time Per Unit Computation τ_c:

$\tau_{c_0} = 28.5$ (ms), $\tau_{c_1} = 25.5$ (ms), $\tau_{c_2} = 16.7$ (ms).

Local Memory Size m (max { local real memory, disk swap space }):

$m_0 = 3$ (M words), $m_1 = 5$ (M words), $m_2 = 10$ (M words).

Overlapping Feature $\kappa_{ol_i} = 0$, $\forall$ i.

Overlapping Overhead $\tau_{o_i} = 0$, $\forall$ i.

Workload Descriptor $WL = < p_0, p_1, p_2, \delta, b_u, b_l >$

$p_{0_0} = 0.24, p_{1_0} = 0.71, p_{2_0} = 0.05, \delta_0 = 0.70, b_{u_0} = 25, b_{l_0} = 1$

$p_{0_1} = 0.13, p_{1_1} = 0.57, p_{2_1} = 0.30, \delta_1 = 0.78, b_{u_1} = 25, b_{l_1} = 1$

$p_{0_2} = 0.29, p_{1_2} = 0.53, p_{2_2} = 0.18, \delta_2 = 0.59, b_{u_2} = 25, b_{l_2} = 1$

Link Specification $LS = < r, \tau_s, P_{ls} >$

Data Transfer Rate r ($1/\tau_t$) = 0.188 (word/μs).

Setup Time τ_s = 108.36 (ms). (one-time cost)

Supported Processor Set $P_{ls} = \{ 0, 1, 2 \}$.

CHAPTER 4.

THE MAPPING SCHEME

A mapping scheme of an ANN onto a multicomputer includes a scheme for assigning neurons (or simulation tasks) to processors and one for routing data across the interconnection network. The assignment of neurons (or c-subtasks) must meet constraints on integrity, feasibility and resources. The routing scheme must meet constraints on dependence, feasibility, resources and configurations. Note that the routing of data cannot be determined until the clusters involved are assigned to processors.

4.1. Formalism of Mapping Scheme

A solution to the mapping problem is called a *mapping scheme* (Φ) and is defined formally as a 4-ary tuple $\Phi(M_{ANN}, M_M, \Phi_A, \Phi_R)$, where M_{ANN} is the ANN model, M_M is the multicomputer model, Φ_A is the *assignment scheme*, and Φ_R is the *routing scheme*. Φ_A and Φ_R are related in that the routing of data cannot be determined until the clusters involved are assigned to processors. All the feasible mapping schemes constitute the mapping space $\Omega(M_{ANN}, M_M)$. The optimal mapping problem is to find a mapping $\Phi^* \in \Omega(M_{ANN}, M_M)$ such that

$$T_{EXEC}(\Phi^*) = \min_{\Phi \in \Omega} T_{EXEC}(\Phi), \tag{4.1}$$

where T_{EXEC} is the completion time for the given mapping.

The assignment scheme Φ_A can be represented as an integer-valued *assignment matrix* A of size n_K-by-n_P. (Recall that n_K is the number of clusters and n_P is the number of processors.) The element a_{ij} of matrix A indicates the number of neurons in cluster i assigned to processor j. When $a_{ij}>0$, processor j is called a *home processor* of cluster i.

Φ_R can be represented as a *routing vector* R of cardinality n_L. Recall that n_L is the number of links. The i-th element of R is a set Ξ_i of 4-ary tuples which keep the statistics of communication on the i-th link. The 4-ary tuple is $<p, f, t_s, t_u>$, where p denotes the processor that transmits frames via this link, f denotes the data frame transferred via this link, t_s denotes the start time for transmission, and t_u denotes the time period for using this link. The timestamp provides information about the use of a link at a particular time. The component of the routing

vector is a set rather than a number because a link can be used at different times by different processors and all these uses must be described. Specifically, a member ξ_{ij} in set Ξ_i means that link i is allocated to processor $p_{\xi_{ij}}$ starting at time $t_{s_{\xi_{ij}}}$ for a period of $t_{u_{\xi_{ij}}}$.

Consider the ANN in Example 2.1 (in Chapter 2) and the multicomputer in Example 3.2 (in Chapter 3). One possible assignment matrix A is as follows.

$$A = \begin{bmatrix} 96 & 157 & 247 \\ 195 & 312 & 493 \\ 39 & 62 & 99 \end{bmatrix}.$$

One possible 4-ary member of the set Ξ_0 is $< p = 0$, frame-from-processor-0, $t_s = 26198\,(\mu s)$, $t_u = 319(\mu s) >$.

4.2. Computation Segment

The computation of the ANN simulation can be broken into several *computation segments* according to the dependence constraints of the ANN. For brevity, a segment is used to refer to a segment. Each segment can start only after its preceding one finishes. A segment is defined by its *entry point* and *exit point*. An entry point of a segment at a particular processor pinpoints the time when this processor receives the first frame from the *preceding* processors. The preceding processors simulate the preceding neurons. An exit point of a segment pinpoints the time when this processor sends the last frame to the succeeding processors. The succeeding processors simulate the succeeding neurons. The overlap of computation and communication in a processor is defined by the interval between the entry point and the *commit point*, which pinpoints the time when this processor finishes receiving the last frames from the preceding processors for the current segment.

Formally, the jth segment at processor i is denoted by $s_{ij} = < s_{1ij}, s_{2ij}, s_{3ij} >$, where s_{1ij}, s_{2ij}, and s_{3ij} are the entry point, commit point, and exit point, respectively. Note that two processors may have different entry and exit points for the same segment. Let h denote the *height* of the ANN, which is the number of clusters encountered along the longest acyclic path from an input cluster to an output cluster. In a multilayered ANN, h is simply the number of layers. Let K_s denote the set of clusters involved in segment s. The maximum number of segments is

$2h$, because h segments are for the production phase, and another h is for the learning phase. Consider the example of the ANN in Example 2.1, where h is equal to 3.

CHAPTER 5.

PROBLEM FORMULATION

The objective function of the mapping problem is the completion time for training the ANN, as a function of computation and communication times. The computation time includes the time for computing neuron output signals in the production phase and the time for updating weights in the learning phase. The communication time includes the time for sending neuron output signals to succeeding processors in the production phase and the time for sending error signals to preceding processors in the learning phase.

The mapping problem can be stated as follows. Given an ANN model and a multicomputer model, find the optimal mapping of the ANN simulation onto the processors such that the completion time of the ANN simulation, including the production and learning phases with respect to computation and communication times, is minimized and that the constraints on feasibility, dependence, resources and configurations are all satisfied.

5.1. Mathematical Formulation

The optimal mapping problem can be formulated mathematically as follows. Given a neural network model M_{ANN} and a multicomputer model M_M, find an assignment matrix A and a routing vector R to achieve the optimal objective function ***OBJ***.

$$OBJ = \min_{A,R} \max_{p \in P} T_{EXEC_p}(A, R), \tag{5.1}$$

such that A and R satisfy the constraints on feasibility, configurations, resources and dependence. The "max" is to find the completion time which is determined by the last processor that finishes the simulation. For brevity, T_{EXEC_p} is used instead of $T_{EXEC_p}(A, R)$ in the following discussion.

T_{EXEC_p} of a processor can be formulated as the sum of the times $T_{EXEC_{pq}}$ for segment q at processor p. Since there are $2h$ segments,

$$T_{EXEC_p} = \sum_{q=1}^{2h} T_{EXEC_{pq}}. \tag{5.2}$$

$T_{EXEC_{pq}}$ should include the computation time $T_{COMP_{pq}}$ and the communication time $T_{COMM_{pq}}$. The computation time of a segment is the sum of execution times of simulations for neurons corresponding to it. That is,

$$T_{COMP_{pq}} = \tau_{c_p} \sum_{i \in K_q} a_{ip} \left(\eta_{p_i}\, \delta_c(q \le h) + \eta_{l_i}\, \delta_c(q > h) \right), \tag{5.3}$$

where δ_c is a binary function such that $\delta_c(z) = 1$ if the predicate z is true; otherwise, it is 0. Recall that K_q is the set of clusters involved in segment q. The communication time $T_{COMM_{pq}}$ is the time interval from the previous exit point to the current commit point. That is,

$$T_{COMM_{pq}} = t_{s_{3pq}} - t_{s_{3pq-1}}. \tag{5.4}$$

Communication can be overlapped with computation after the first frame arrives. The idle time between the previous exit time and the time when the first frame arrives is called the *bubble time* and is denoted by $T_{BUBBLE_{pq}}$. Note that $T_{BUBBLE_{pq}} \le T_{COMM_{pq}}$. Also note that the time when the first frame arrives is equal to the entry time in the overlapped case, and that the entry time is equal to the commit time in the nonoverlapped case. The execution time for segment q at processor p can be written as

$$T_{EXEC_{pq}} = \kappa_{ol_p} T_{BUBBLE_{pq}} + (1 - \kappa_{ol_p}) T_{COMM_{pq}} + T_{COMP_{pq}}. \tag{5.5}$$

By substituting Eq. (5.5) and Eq. (5.2) into Eq. (5.1), the objective function ***OBJ*** can be rewritten as

$$OBJ = \min_{A,R} \max_{p \in P} \sum_{q=1}^{2h} \left\{ \kappa_{ol_p} T_{BUBBLE_{pq}} + (1 - \kappa_{ol_p}) T_{COMM_{pq}} + T_{COMP_{pq}} \right\}. \tag{5.6}$$

The bubble time $T_{BUBBLE_{pq}}$ is equal to the time interval between the exit point of segment $q-1$ and the time when the first useful frame arrives. To determine the arrival time of the first frame, the arrival time of each frame must be known. Let $t_{arr_{kpq}}$ be the arrival time of the frame containing values produced at processor k for segment q at processor p. If $k = p$, then $t_{arr_{kpq}}$ is set to $t_{s_{3pq-1}}$. This means that frames produced by itself arrive exactly at the exit time, which agrees with Eq. (5.8). Then $T_{BUBBLE_{pq}}$ is

$$T_{BUBBLE_{pq}} = \min_{\substack{k \in P \\ k \neq p}} \left\{ t_{arr_{kpq}} - t_{s_{3pq-1}} \right\}. \tag{5.7}$$

The arrival time of each frame depends on the exit point of the previous segment at the source processor and the traffic along the communication path. It can be formulated as

$$t_{arr_{kpq}} = t_{s_{3kq-1}} + T_{PATH}(\Lambda_{kpq}), \tag{5.8}$$

where $T_{PATH}(\Lambda_{kpq})$ is the time needed to send the frame containing values produced at processor k for segment q at processor p along the communication path Λ_{kpq}. This represents the average transmission time along the path from processor k to processor p and the average delay at the intermediate processors due to traffic. This time can be written as

$$T_{PATH}(\Lambda_{kpq}) = \sum_{\substack{u \in \Lambda_{kpq} \\ f_{kpq} = f_{\xi_{ux}} \\ v \text{ is the next link to } u \\ \text{in path } \Lambda_{kpq}}} \left\{ t_{u_{\xi_{ux}}} + T_{DELAY}(f_{kpq}, u, v) \right\}, \tag{5.9}$$

where $T_{DELAY}(f_{kpq}, u, v)$ is the delay time at the processor between link u and v. Recall that $t_{u_{\xi_{ux}}}$ is the link usage time and is defined by $t_{u_{\xi_{ux}}} = \tau_{s_u} + \eta_{f_{kpq}} \tau_{t_u}$. The delay time mainly depends on the traffic on links u and v, and the frames that arrive first will be transmitted first. The delay function can be written as

$$T_{DELAY}(f_{kpq}, u, v) = \sum_{\substack{\xi_{ux} \in \Xi_u \\ \xi_{vy} \in \Xi_v}} \left\{ \left\{ t_{s_{\xi_{vy}}} - (t_{s_{\xi_{ux}}} + t_{u_{\xi_{ux}}}) \right\} \times \delta_0 (f_{kpq} - f_{\xi_{ux}}) \times \delta_0 (f_{kpq} - f_{\xi_{vy}}) \right\}, \tag{5.10}$$

where δ_0 is a *discriminating function* such that $\delta_0(z)=1$ if $z=0$, otherwise $\delta_0(z)=0$.

The entry time $t_{s_{1pq}}$, the commit time $t_{s_{2pq}}$, and the exit time $t_{s_{3pq}}$ are defined, respectively, by Eqs. (5.11), (5.12), and (5.13).

$$t_{s_{1pq}} = \min_{k \in P} t_{arr_{kpq}}. \tag{5.11}$$

$$t_{s_{2pq}} = \max_{k \in P} t_{arr_{kpq}}. \tag{5.12}$$

$$t_{s_{3pq}} = t_{s_{3pq-1}} + T_{EXEC_{pq}}. \tag{5.13}$$

The link start times $t_{s_{\xi_{ux}}}$ and $t_{s_{\xi_{vy}}}$, and the link usage time $t_{u_{\xi_{ux}}}$ in Eq. (5.10) are specified in the mapping scheme.

In summary, the objective function *OBJ* for a mapping scheme can be completely determined by combining Eqs. (5.6) through (5.13).

Consider the multicomputer in Example 3.2. The objective function can be written as

$$OBJ = \min_{A,R} \max_{p \in P} \sum_{q=1}^{2h} \left\{ T_{COMM_{pq}} + T_{COMP_{pq}} \right\}. \tag{5.14}$$

When a mapping scheme is determined, a_{ip}, $t_{s_{2pq}}$, and $t_{s_{3pq}}$ can be determined accordingly. Then, by using Eqs. (5.3) and (5.4), the above objective function can be calculated.

5.2. Constraints

Four groups of constraints must be satisfied: feasibility constraints, configuration constraints, resource constraints and dependence constraints.

The feasibility constraints include the feasibility of assignment (Eq. (C.1a)) and the feasibility of link allocation (Eq. (C.1b)). The feasibility of assignment requires all neurons in each cluster to be assigned to a subset of processors and each neuron to be assigned to exactly one processor. This constraint is involved when the neural assignment is made. Note that the feasibility of assignment is checked only when the production task nodes are mapped, since the assignments for the learning task nodes are constrained to be the same as those for the production task nodes. The feasibility of link allocation requires a communication link not to be allocated more than once during the period when it is used. This constraint is involved when the link allocation is made.

Feasibility Constraints

$$\sum_{j=0}^{n_p-1} a_{ij} = n_i \ \forall \ i = 1, \ldots, K. \tag{C.1a}$$

$$t_{s_{\xi_{ik}}} \text{ not} \in [\, t_{s_{\xi_{ij}}}, t_{s_{\xi_{ij}}} + t_{u_{\xi_{ij}}}] \ \forall \ \xi_{ij} \neq \xi_{ik} \in \Xi_i, \ i = 0, \ldots, L-1. \tag{C.1b}$$

The configuration constraint is the one on transmitting processors (Eq. (C.2)). This constraint requires the processor transmitting a data frame over a link to be a member of the set of processors supported by this link. This constraint is involved when the transmitting processor is granted.

Configuration Constraints

$$p_{\xi_{ij}} \in P_{ls_i}, \ \forall \ \xi_{ij} \in \Xi_i, \ i = 0, \ldots, n_L - 1. \tag{C.2}$$

The resource constraint is the constraint on local memory (Eq. (C.3)), which requires the total amount of space allocated for computation at a processor not to exceed the limit of its local memory.

Resource Constraints

$$\sum_{j=1}^{n_K} a_{ji}\, \eta_{ls_j} \le m_i, \ \forall \ i = 0, \ldots, n_P - 1. \tag{C.3}$$

The dependence constraints include the usage dependence (Eq. (C.4a)) and the production dependence (Eq. (C.4b)). The usage dependence requires an output value of a neuron to be used only after the value has been produced. This constraint is involved when the neuron output value at its home processor is to be transmitted to other processors. The production dependence requires an output value to be produced only after all of its required input signals arrive. This constraint is involved when an output value is to be produced.

Dependence Constraints

$$t_o\left[\pi_p(i,p) = 1\right] < t_o\left[\pi_u(i,p) = 1\right], \ \text{if } a_{ip} > 0 \ \forall \ p = 0, \ldots, n_P - 1, i = 1, \ldots, n_K. \tag{C.4a}$$

$$t_o\left[\pi_a(i,p) = 1\right] < t_o\left[\pi_u(i,p) = 1\right], \ \forall \ p = 0, \ldots, n_P - 1, \ i = 1, \ldots, n_K. \tag{C.4b}$$

where π_p is the production occurrence function, π_u is the usage occurrence function, π_a is the arrival occurrence function, and t_o is the occurrence time function. The occurrence function is 1 if the corresponding event occurs; otherwise, it is 0. For example, $\pi_p(i,p) = 1$ if the signals of cluster i are produced at processor p, otherwise, 0. π_u and π_a are defined accordingly.

For the problem of mapping the ANN in Example 2.1 (in Chapter 2) onto the multicomputer in Example 3.2 (in Chapter 3), constraints (C.1a) can be written into

$$a_{10} + a_{11} + a_{12} = 500,$$

$$a_{20} + a_{21} + a_{22} = 1000,$$

$$a_{30} + a_{31} + a_{32} = 200.$$

Constraint (C.3) can be written into

$$501 \times a_{10} + 501 \times a_{20} + 1001 \times a_{30} \le 3 \times 10^6,$$

$$501 \times a_{11} + 501 \times a_{21} + 1001 \times a_{31} \le 5 \times 10^6,$$

$$501 \times a_{12} + 501 \times a_{22} + 1001 \times a_{32} \le 1 \times 10^7.$$

5.3. Complexity

The integer programming formulation described in Sections 5.1 and 5.2 is nonlinear because the objective function is nonlinear and constraints (C.1b) and (C.2) are nonlinear. To understand the complexity of the formulation, the number of variables used in it is derived first. These variables are due to the elements of assignment matrix A and those of the routing vector R. Let n_A denote the number of variable items in matrix A. n_A is simply equal to the number of elements of matrix A, *i.e.*,

$$n_A = n_K\, n_P. \tag{5.15}$$

The number of elements in vector R is equal to its cardinality, *i.e.*, n_L. However, each element Ξ_i in the routing vector R is itself a set of 4-ary tuples ξ_{ij}. *Three* items in the tuple are variables, *i.e.*, p, f, and t_s. The number of tuples in set Ξ_i depends on the number of routing subproblems ($2\,h$), the number of processors (n_P), and the *diameter* (D_M) of the multicomputer. There are $2\,h$ routing problems because there exists one routing problem between two adjacent layers and another between an IO cluster and the external environment. The diameter of a multicomputer is the maximum length of the shortest path between any pair of processors if each link is of unit length. Let n_R denote the number of variable items in vector R. Then,

$$n_R \le n_{R,\max} = 6\, h\, n_L\, n_P\, D_M. \tag{5.16}$$

The number of variable items is equal to the number of variable items in A and R. Let n_Φ denote the number of variable items in the mapping. Then,

$$n_\Phi = n_A + n_R \le n_{\Phi,\max} = n_K\, n_P + 6\, h\, n_L\, n_P\, D_M \tag{5.17}$$

$n_{\Phi,\max}$ is very large in most cases; however, simplification of the mapping problem with negligible error is possible because the computation time is generally *predominant* over the communication time. This simplification technique will be introduced in Section 6.1.

For the problem of mapping the neural network in Example 2.1 onto the multicomputer in Example 3.2,

$$n_A = 3 \times 3 = 9, \quad \text{and} \quad n_{R,\max} = 6 \times 2 \times 1 \times 3 \times 1 = 36.$$

For the problem of mapping to the multicomputer in Example 3.1,

$$n_A = 3 \times 16 = 48, \quad \text{and} \quad n_{R,\max} = 6 \times 2 \times 32 \times 16 \times 4 = 24576.$$

$n_{R,\max}$ for the 16-node hypercube is very large. However, through the simplification technique, $n_{R,\max}$ will decrease dramatically. For example, if the 16-node hypercube is transformed into two partitions, then

$$n_{R,simplified,\max} = 6 \times 2 \times 1 \times 2 \times 1 = 24.$$

The complexity of the integer programming formulation also depends on the number of possible values that each variable can acquire.

The routing problem can be illustrated as follows. Consider a case in which each processor is associated with a set of frames to be migrated, and each frame is also associated with a set of destination processors. The routing problem is to find a scheme such that the completion time is minimal for migrating every frame from its home processor, which produces this frame, to its destination processors. This routing problem, called the *multiple partial broadcasting* problem, is very hard to solve for large interconnection networks. It is harder than the traditional NP-complete communication problems, such as the optimum communication spanning tree [10] and the minimum broadcast time [5] problems.

The mapping problem formulated degenerates to the traditional precedence constrained scheduling problem if the communication overhead is neglected. The precedence constrained scheduling problem has been proved to be NP-complete by transformation from *3SAT* [17].

5.4 Dynamic Mapping Algorithm

The multicomputer may have time-varying workloads. The mapping of the ANN simulation should be adjusted when the workload characteristic changes. However, it may not be practical to embed the dynamic load balancing handler inside the ANN simulation such that the ANN simulation is always optimally mapped. Instead, an approximate approach is adopted; that is, re-map the ANN simulation whenever the workload changes significantly during its execution and the new mapping scheme is expected to pay back the cost of re-mapping. It may not be cost-effective if the re-mapping occurs too frequently because the re-mapping itself causes some overhead.

The decision to perform re-mapping depends on (1) the *current simulation time* t_{sim} for one iteration of ANN simulation; (2) the *predicted simulation time* $\hat{t}_{sim}$ for one iteration of ANN simulation; (3) the *expected mapping time* $\hat{t}_{map}$ for solving the optimal mapping; (4) the *remaining simulation time* t_{rem} which predicts the time for completing the remaining simulation tasks based on the current simulation time t_{sim}; and (5) the *predicted remaining simulation time* $\hat{t}_{rem}$ which predicts the time for completing the remaining simulation tasks and is calculated as

$$\hat{t}_{rem} = t_{rem} \times \frac{\hat{t}_{sim}}{t_{sim}}. \tag{5.18}$$

Let t_g denote the *re-mapping gain* which is the difference between the remaining simulation time without re-mapping and the new remaining simulation time after re-mapping plus the expected re-mapping time. That is,

$$t_g \triangleq t_{rem} - \hat{t}_{rem} - \hat{t}_{map}. \tag{5.19}$$

The re-mapping should be carried out if there is a positive gain, that is, $t_g > 0$.

After the new optimal mapping is found, the data for neuron states need to be migrated across different processors. Then the ANN simulation is resumed. In the following analysis, the time for migrating data due to new mapping is assumed to be included in the mapping time.

CHAPTER 6.

SOLUTION STRATEGY, TECHNIQUES AND PROPERTIES

The mapping problem can sometimes be simplified with negligible error because computation time generally dominates communication time. This dominance occurs either when the number of neurons in each cluster is large or when communication time is relatively small.

In this chapter, a strategy for solving the optimal mapping problem and a technique of reducing the problem complexity are described. A branch-and-bound search algorithm for finding the optimal mapping with a guaranteed deviation from the optimality is described. Finally, a geometric view to interpret the routing scheme is also discussed. Some old results published in an earlier paper [18] are included for comparison.

6.1. Overall Strategy of Solving Mapping Problem

The mapping problem can be simplified with negligible error because computation time usually dominates communication time, at least within some local set of processors called a *partition*. The routing scheme can be relaxed to a certain heuristic routing instead of the exact optimal routing within each partition. The strategy proposed is first to divide the multicomputer into disjoint partitions and solve the optimal mapping onto these partitions, as shown in Figure 6.1. That is, solve the optimal assignment on partitions, find the optimal inter-partition routing, and then resolve the heuristic intra-partition routing.

This strategy is different from the traditional one which solves the optimal assignment to processors as well as the optimal routing among all processors. Note that the assignment and routing are tightly coupled, regardless of partitions or processors, such that each cannot be fully determined independent of the other. This strategy may cause some error; however, this is negligible and the error bound is guaranteed; that is, the accuracy is guaranteed. These will be discussed in the following subsections.

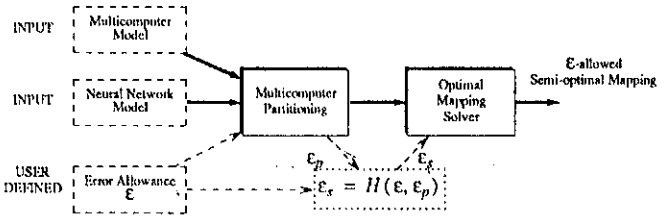

Figure 6.1 : Solution strategy for the optimal mapping problem with guaranteed accuracy.

6.2. Partitioning of Multicomputers

The dominance of computation time is characterized by the ratio of communication to computation times. Before elaborating on the dominance and possible error, some concepts and notations need to be clarified and defined. Important symbols are summarized in Table 6.1. These symbols are explained briefly here.

For a given partitioning of processors (the heuristic partitioning method will be discussed later), the optimal mapping and routing of neurons on the partitions can be found using a branch-and-bound algorithm based on the nonlinear integer programming formulation. As stated before, these two problems are tightly coupled and cannot be solved independently. The computation and intra-partition routing times for cluster i in the optimal case, $t_{iQ}{}^*$, satisfy the following equation.

$$t_{iQ}{}^* = t^*_{c_{iQ}} + t^*_{r_{iQ}}. \tag{6.1}$$

Figure 6.2a illustrates the neural network, the assignment of clusters to partitions, the mapping of clusters within a partition, and the mapping of clusters on the entire multicomputer. In Figure 6.2b, the three processors represent one partition Q. The three blocks on the left represent the three segments for cluster 1, which are processed concurrently by the processors in Q. Note that $t_{c_{4,1}}$ includes all times during which one or more processors are performing

Table 6.1 : Summary of symbols used in lemmas and theorems.

Symbol	Meaning	Symbol	Meaning
Φ_1^*	optimal mapping of clusters onto the given multicomputer	Φ_1	optimal assignment (without considering intra-partition communication delay), optimal inter-partition routing, and heuristic intra-partition routing
$t_{iQ}{}^*$	for Φ_1^*, time interval during which one or more processors are performing computation or communication for cluster i on partition Q, and it is not overlapped with computation of the cluster that follows it	t_{iQ}	for Φ_1, time interval during which one or more processors are performing computations or communications for cluster i on partition Q, and it is not overlapped with computation in the cluster that follows it
$t^*_{c_{iQ}}$	for Φ_1^*, time interval during which one or more processors are performing computation for cluster i on partition Q	$t_{c_{iQ}}$	for Φ_1, time interval during which one or more processors are performing computation for cluster i on partition Q
$t^*_{r_{iQ}}$	for Φ_1^*, time interval during which all processors are performing intra-partition communication for cluster i on partition Q, and it is not overlapped with computation in this cluster or the cluster that follows it	$t_{r_{iQ}}$	for Φ_1, time interval during which all processors are performing intra-partition communication for cluster i on partition Q, and it is not overlapped with computation in this cluster or the cluster that follows it
$t^*_{R_{iQ}}$	for Φ_1^*, same as $t^*_{r_{iQ}}$ except that inter-partition communication time is concerned	$t_{R_{iQ}}$	for Φ_1, same as $t_{r_{iQ}}$ except that inter-partition communication time is concerned
n_{iQ}	number of neurons in cluster i assigned to partition Q	γ_{iQ}	$= t_{r_{iQ}} / t_{c_{iQ}} = t_{comm_{iQ}} / t_{comp_{iQ}}$
$t_{comp_{iQ}}$	$= t_{c_{iQ}} \times \lvert Q \rvert / n_{iQ}$	$t_{comm_{iQ}}$	$= t_{r_{iQ}} \times \lvert Q \rvert / n_{iQ}$
Φ^*_{ref}	optimal mapping of clusters on the reference multicomputer	$T_{ref}(\Phi)$	completion time based on certain mapping Φ on the reference multicomputer
$t^*_{ref,iQ}$	$(= t_{c_{iQ}})$ for Φ^*_{ref}, time interval during which partition Q is performing computations for cluster i	$t^*_{R_{ref,iQ}}$	$(= t_{R_{iQ}})$ for Φ^*_{ref}, communication time not overlapping with computation

computations for cluster 1, and that $t_{r_{4,1}}$ represents the unoverlapped intra-partition communication times between computations in cluster 2 and cluster 4. If overlaps between communications and computations are allowed, $(t_{r_{4,1}} + t_{R_{4,1}})$ represents the minimal interval between the time when the computations of the last segment in cluster 2 are completed and the time when the first computation in one of the segments of cluster 4 can begin. Figure 6.2c shows the timing diagrams for simulating the five clusters in two partitions.

Similarly, the definition of t_{iQ} satisfies the following equation.

$$t_{iQ} = t_{c_{iQ}} + t_{r_{iQ}} = \frac{n_{iQ}}{|Q|} \times \left(t_{comp_{iQ}} + t_{comm_{iQ}} \right). \tag{6.2}$$

In this case, the neurons in a cluster are first allocated by ignoring their communication requirements. It is obvious that an even distribution of neurons according to the computation power of processors in partition Q will result in the minimal completion time $t_{c_{iQ}}$. (A more general result will be proved in Theorem 6.3.)

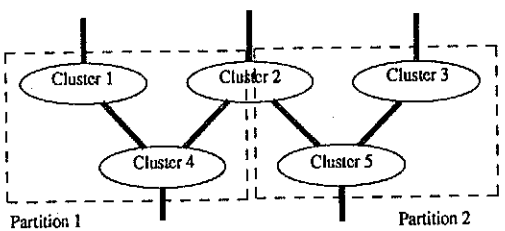

Figure 6.2a : Mapping of 5 clusters on 2 partitions.

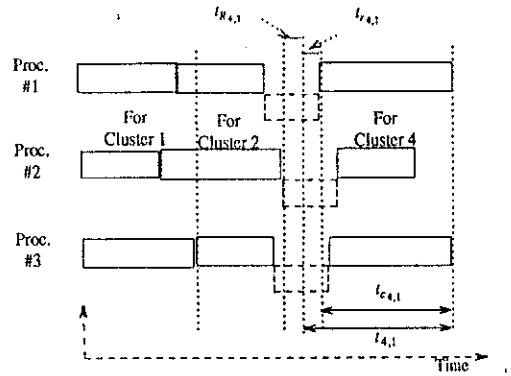

Figure 6.2b : Timing diagram showing mapping within partition 1.

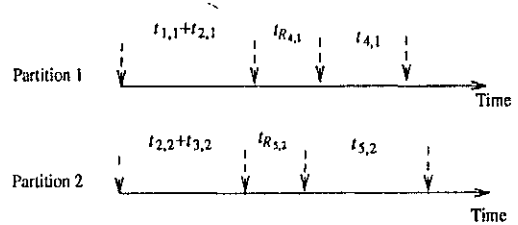

Figure 6.2c : Timing diagram showing overall mapping.

The intra-partition communication time, $t_{r_{iQ}}$, is computed using a heuristic routing scheme. For simplicity, it is assumed that each processor broadcasts its results according to a minimum spanning tree, and that broadcasts of different processors are done sequentially. As a result, there is never any congestion involved in this communication scheme. It is therefore simple to compute $t_{r_{iQ}}$, the interval between the time when the last inter-partition communication in cluster i is completed and the time when the first computation in cluster i begins. Note that $t_{r_{iQ}}$ represents a worst-case communication delay.

Another observation about the definitions in Table 6.1 is that $t_{comp_{iQ}}$ is per-neuron average computation time for cluster i, and that $t_{comm_{iQ}}$ is per-neuron average communication time for cluster i (based on a heuristic routing scheme). Since $t_{c_{iQ}}$ is a lower-bound estimate and $t_{r_{iQ}}$ is an upper-bound estimate, γ_{iQ} consequently represents a *worst-case* communication-to-computation time ratio that can be experienced in partition Q for processing cluster i.

The last observation is that both t_{iQ}^* and t_{iQ} include the execution times in the production and learning phases.

The following lemma and theorem show the upper bound on the error due to a heuristic routing scheme for a given partitioning of processors.

Mapping Heuristic 6.1.

Neurons within a cluster can be mapped by a branch-and-bound algorithm to one or more partitions with the following assumptions:

(a) optimal partitioning of processors is known and does not change,

(b) routing across partitions is optimal (with time $t_{R_{iQ}}^*$), and

(c) routing within partitions is suboptimal (with time $t_{r_{iQ}}$).

Lemma 6.1. Based on Mapping Heuristic 6.1, the difference between t_{iQ} and t_{iQ}^* is bounded from above by t_{iQ}^* times the comm/comp ratio γ_{iQ}. That is,

$$t_{iQ} \le t_{iQ}^* (1 + \gamma_{iQ}). \tag{6.3}$$

Proof. Since t_{iQ}^* is the optimal execution time, the following relation holds

$$t_{c_{iQ}} = \frac{n_{iQ}\, t_{comp_{iQ}}}{|Q|} \le t_{iQ}^* \le t_{iQ}. \tag{6.4}$$

The optimal assignment without considering communication overhead will result in shorter completion time than a mapping in which the communication overhead is included. In the latter case, the assignment is functionally constrained by the routing. Therefore, it can be derived

$$t_{c_{iQ}} \le t_{c_{iQ}}^* = t_{iQ}^* - t_{r_{iQ}}^*. \tag{6.5}$$

Simple algebraic manipulations on Eqs. (6.4) and (6.5) result in the following inequality, which proves the lemma.

$$\begin{aligned} t_{iQ} &\le t_{iQ}^* - t_{r_{iQ}}^* + t_{r_{iQ}} \\ &= t_{iQ}^* \left(1 + \frac{n_{iQ}\, t_{comm_{iQ}} / |Q| - t_{r_{iQ}}^*}{t_{iQ}^*} \right) \\ &\le t_{iQ}^* \left(1 + \frac{n_{iQ}\, t_{comm_{iQ}} / |Q| - t_{r_{iQ}}^*}{n_{iQ}\, t_{comp_{iQ}} / |Q|} \right) \\ &\le t_{iQ}^* \left(1 + \frac{t_{comm_{iQ}}}{t_{comp_{iQ}}} \right) \\ &\le t_{iQ}^* \left(1 + \gamma_{iQ} \right). \end{aligned} \tag{6.6}$$

Q.E.D.

The following theorem generalizes the concept of Lemma 6.1 to the whole multicomputer and ANN, and shows that the error bound can be guaranteed if Mapping Heuristic 6.1 is used.

Theorem 6.1. Consider a multilayered ANN with n_L layers and a multicomputer with $n_{P'}$ disjoint partitions. Assume that every cluster i in every partition Q of processors has comm/comp ratio $\gamma_{iQ} \triangleq t_{comm_{iQ}} / t_{comp_{iQ}}$ no greater than a predefined value ε. Let $T(\Phi_1)$ be the completion time based on Mapping Heuristic 6.1. Then,

$$T(\Phi_1) \le T(\Phi_1^*)(1+\varepsilon). \tag{6.7}$$

Proof. Let K_l be the set of clusters in layer l. The time in each processor allocated to cluster i can be computed by the summation of $t_{iQ}{}^*$ and $t^*_{R_{iQ}}$, for optimal intra-partition routing, and by the summation of t_{iQ} and $t_{R_{iQ}}$, for heuristic intra-partition routing. If a cluster is not allocated to a partition, then its time is zero. The completion time $T(\Phi_1^*)$ of mapping Φ_1^* can be expressed by

$$T(\Phi_1^*) = \max_{Q} \sum_{l=0}^{n_L-1} \sum_{i \in K_l} \left[t^*_{R_{iQ}} + t_{iQ}{}^* \right]. \tag{6.8}$$

The completion time $T(\Phi_1)$ of mapping Φ_1 is

$$T(\Phi_1) = \max_{Q} \sum_{l=0}^{n_L-1} \sum_{i \in K_l} \left[t^*_{R_{iQ}} + t_{iQ} \right]. \tag{6.9}$$

According to Lemma 6.1, $t_{iQ} \le t_{iQ}{}^* (1+\varepsilon)$ holds. Simple algebraic manipulations show that

$$\begin{aligned} T(\Phi_1) &= \max_{Q} \sum_{l=0}^{n_L-1} \sum_{i \in K_l} \left[t^*_{R_{iQ}} + t_{iQ} \right] \\ &\le \max_{Q} \sum_{l=0}^{n_L-1} \sum_{i \in K_l} \left[t^*_{R_{iQ}} + (1+\varepsilon)\, t_{iQ}{}^* \right] \\ &\le \max_{Q} \sum_{l=0}^{n_L-1} \sum_{i \in K_l} \left[1+\varepsilon \right] \left[t^*_{R_{iQ}} + t_{iQ}{}^* \right] \\ &\le T(\Phi_1^*)(1+\varepsilon). \end{aligned} \tag{6.10}$$

Q.E.D.

According to Theorem 6.1, given an error bound ε for the comm/comp ratio, the multicomputer can be divided into several partitions such that the comm/comp ratio of each partition for simulating part of a given cluster is less than the bound ε. Hence, the optimal mapping on the partitioned multicomputer with a heuristic routing scheme within each group will have its completion time no greater than $(1+\varepsilon)$ times the completion time of the optimal mapping on the unpartitioned multicomputer. The maximum of all comm/comp ratios of the partitioned multicomputer is called the *error degree*. A small error degree will result when the numbers of neurons in all clusters are large or when the partitions are small.

One problem with Mapping Heuristic 6.1 is that the assumption (a) of known optimal partitioning is *impractical* since the optimal partitioning is generally unknown. Further, solving the optimal partitioning is much harder in general than solving the optimal mapping problem. In an earlier paper [18], Mapping Heuristic 6.1 is used except the optimal partitioning is relaxed to a heuristic partitioning. Since then, new results show that that the error bound still can be guaranteed even if the assumption (a) in Mapping Heuristic 6.1 is relaxed. This brings up Mapping Heuristic 6.2 based on Heuristic Partitioning Algorithm 6.3.

To prove the guarantee of the error bound, a *conceptual* multicomputer called the *reference multicomputer* is introduced. The reference multicomputer consists of a set of *conceptual* processors, each corresponds to a partition of real processors such that the reference multicomputer is actually the *partitioned* multicomputer, except that there is no intra-partition routing since each partition is treated as a real undividable processor. Note that in this discussion there will be three types of multicomputers: original (unpartitioned), partitioned and reference. Actually, the former two physically refer to the same multicomputer.

The following lemmas and theorem show that the error bound can be guaranteed if Mapping Heuristic 6.2 is used.

Mapping Heuristic 6.2.

Neurons within a cluster can be mapped by a branch-and-bound algorithm to one or more partitions with the following assumptions: (a) routing across partitions is optimal (with time $t^*_{R_{iQ}}$), and (b) routing within partitions is suboptimal (with time $t_{r_{iQ}}$).

Heuristic Partitioning Algorithm 6.3.

1. Select one processor not included in any partition to form a new partition. If all processors have been partitioned, then exit.
2. For a given partition and a processor not included in any other partition, if γ_{iQ} for all processors in this partition (including the newly selected processor) does not exceed the error allowance ε, then include the new processor into this partition. This step is repeated for all partitions already formed and all processors not included in any partition. Go to step 1.

Lemma 6.2. The completion time based on the optimal mapping Φ^*_{ref} on the reference multicomputer is no greater than the completion time based on the optimal mapping Φ^*_1 on the original multicomputer. That is,

$$T_{ref}(\Phi^*_{ref}) \le T(\Phi^*_1). \tag{6.11}$$

Proof. The optimal mapping Φ^*_1 on the original multicomputer is also a *feasible* mapping (ignoring the intra-partition communication) on the reference multicomputer. Since there is no intra-partition communication in the reference multicomputer, we have $T_{ref}(\Phi^*_1) \le T(\Phi^*_1)$. By the definition of optimality of Φ^*_{ref}, we have

$$T_{ref}(\Phi^*_{ref}) \le T_{ref}(\Phi^*_1) \le T(\Phi^*_1). \tag{6.12}$$

Q.E.D.

Lemma 6.3. The difference between t_{iQ} and $t^*_{ref,iQ}$ is bounded from above by $t^*_{ref,iQ}$ times γ_{iQ}. That is,

$$t_{iQ} \le t^*_{ref,iQ}\,(1+\gamma_{iQ}). \tag{6.13}$$

Proof. Since $t^*_{ref,iQ}$ is the optimal completion time on the reference multicomputer, by the definition of $t_{c_{iQ}}$, the following relation holds

$$t_{c_{iQ}} = t^*_{ref,iQ}. \tag{6.14}$$

By definition of t_{iQ},

$$t_{iQ} = t_{c_{iQ}} + t_{r_{iQ}} = t^*_{ref,iQ}\,(1+\frac{t_{r_{iQ}}}{t_{c_{iQ}}}) \le t^*_{ref,iQ}\,(1+\gamma_{iQ}). \tag{6.15}$$

Q.E.D.

Lemma 6.4. Consider a multilayered ANN with n_L layers, and a multicomputer with $n_{P'}$ disjoint partitions, and the reference multicomputer. Assume that every cluster i in every partition Q has comm/comp ratio $\gamma_{iQ} \triangleq t_{comm_{iQ}} / t_{comp_{iQ}}$ no greater than a predefined value ε. Let $T(\Phi_1)$ be the completion time based on Mapping Heuristic 6.2. Then,

$$T(\Phi_1) < T_{ref}(\Phi^*_{ref})\,(1+\varepsilon). \tag{6.16}$$

Proof. Let K_l be the set of clusters in layer l. The completion time $T_{ref}(\Phi^*_{ref})$ on the reference multicomputer can be expressed as

$$T_{ref}(\Phi^*_{ref}) = \max_Q \sum_{l=0}^{n_L-1} \sum_{i\in K_l} \left(t^*_{R_{ref,iQ}} + t^*_{ref,iQ}\right). \tag{6.17}$$

The completion time $T(\Phi_1)$ on the partitioned multicomputer can be expressed as

$$T(\Phi_1) = \max_Q \sum_{l=0}^{n_L-1} \sum_{i\in K_l} \left(t^*_{R_{ref,iQ}} + t_{iQ}\right). \tag{6.18}$$

According to Lemma 6.3, Eq. (6.13) holds. Simple algebraic manipulations show that

$$T(\Phi_1) = \max_Q \sum_{l=0}^{n_L-1} \sum_{i\in K_l} \left(t^*_{R_{ref,iQ}} + t_{iQ}\right)$$

$$\leq \max_{Q} \sum_{l=0}^{n_L-1} \sum_{i \in K_l} \left\{ t^*_{R_{ref,iQ}} + (1+\varepsilon)\, t^*_{ref,iQ} \right\}$$

$$\leq \max_{Q} \sum_{l=0}^{n_L-1} \sum_{i \in K_l} (1+\varepsilon) \left\{ t^*_{R_{ref,iQ}} + t^*_{ref,iQ} \right\}$$

$$\leq T_{ref}(\Phi^*_{ref})(1+\varepsilon). \tag{6.19}$$

Q.E.D.

Theorem 6.2. Consider a multilayered ANN with n_L layers and a multicomputer with n_P disjoint partitions and its reference multicomputer. Assume that every cluster i in every partition Q has comm/comp ratio $\gamma_{iQ} \triangleq t_{comm_{iQ}} / t_{comp_{iQ}}$ no greater than a predefined value ε. Let $T(\Phi_1)$ be the completion time based on Mapping Heuristic 6.2. Then,

$$T(\Phi_1) \leq T(\Phi_1^*)(1+\varepsilon). \tag{6.20}$$

Proof. By Lemmas 6.2 and 6.4,

$$T(\Phi_1) \leq T_{ref}(\Phi^*_{ref})(1+\varepsilon) \leq T(\Phi_1^*)(1+\varepsilon). \tag{6.21}$$

Q.E.D.

Theorem 6.2 shows that the bound of the error to the optimality can still be guaranteed, even if Mapping Heuristic 6.2 and Heuristic Partitioning Algorithm 6.3 are used. In an earlier work [18], the error bound was thought to be unable to be guaranteed. This new result is very important, because it says that the semi-optimal mapping on the partitioned multicomputer can have the different assignment of neurons from the exact optimal mapping. Further, it says that the partitioning does not affect the bound of the error to the optimality, as long as it satisfies the comm/comp ratio. Therefore, any partitioning algorithm will suffice instead of the best partitioning as was previously thought necessary.

The following theorem shows that distributing neurons proportionally within a partition according to the computational power of processors within the partition is optimal.

Theorem 6.3. Assume that n_{iQ} neurons in neural cluster i are assigned to a partition Q. The optimal assignment on Q can be obtained by distributing the n_{iQ} neurons evenly according to the computation power of processors. Processor j completes at approximately $x_{ij}\tau_j + t_j$, where x_{ij} is the number of neurons in cluster i assigned to processor j, τ_j and t_j are, respectively, the execution time of unit computation and the amount of time that processor j is not available for the ANN simulation.

Proof. Since computation time dominates communication time in this partition, only computation time has to be considered in the proof. Let X_{iQ} be the possible mapping of cluster i on Q. The optimal execution time can be written as

$$t_{iQ} = \min_{X_{iQ}} \max_{j \in Q} \{ x_{ij}\tau_j + t_j \}. \tag{6.22}$$

Let z_{ij} ($= x_{ij}\tau_j + t_j$) be the computation time of processor j for cluster i. Then, the completion time of mapping X_{iQ} is

$$C_{<X_{iQ}>} = \max_{j \in Q} \{ x_{ij}\tau_j + t_j \}. \tag{6.23}$$

Assume the assignment as stated in the theorem such that $z_{ij} = C_{<X^*_{iQ}>}$ for every j, where $C_{<X^*_{iQ}>}$ is the completion time of the optimal mapping in Q. Since $\sum_{j \in Q} x_{ij} = n_{iQ}$, $C_{<X^*_{iQ}>}$ can be easily derived as

$$C_{<X^*_{iQ}>} = \frac{n_{iQ} + \sum_{j \in Q} t_j/\tau_j}{\sum_{j \in Q} 1/\tau_j}, \tag{6.24}$$

where $|Q|$ is the cardinality of Q. By assuming another assignment X'_{iQ} such that

$$C_{<X'_{iQ}>} = \max_{j \in Q} \{ x'_{ij}\tau_j + t_j \} \leq C_{<X^*_{iQ}>}, \tag{6.25}$$

then for every j, assignment x'_{ij} satisfies an inequality $x'_{ij}\tau_j \leq C_{<X'_{iQ}>} - t_j$. By summing all x'_{ij},

$$n_{iQ} = \sum_{j \in Q} x'_{ij} = \sum_{j \in Q} \frac{C_{<X'_{iQ}>} - t_j}{\tau_j} < \sum_{j \in Q} \frac{C_{<X^*_{iQ}>} - t_j}{\tau_j} = n_{iQ}. \tag{6.26}$$

A contradiction! Consequently, $C_{<X'_{iQ}>} \geq C_{<X^*_{iQ}>}$ must hold; that is, the optimal execution time is $C_{<X^*_{iQ}>}$.

Q.E.D.

According to Theorem 6.3, x_{ij} can be calculated by using the following equality.

$$x_{ij}\,\tau_j = \left\lfloor \frac{n_{iQ} + \sum_{j \in Q} t_j/\tau_j}{|Q|/\tau_j} - t_j \right\rfloor. \tag{6.27}$$

Note that if $t_j = 0$ for every j, then a uniform distribution according to the computation power of processors in Q follows from Theorem 6.3. Also note that if

$$\frac{n_{iQ} + \sum_{j \in Q} t_j/\tau_j}{|Q|/\tau_j} < t_j \tag{6.28}$$

is true, then the most negative x_{ij} can first be set to zero and x_{ik} can be recomputed for every $k \neq j$ in Q. This process may have to be repeated several times in the worst case.

Corollary 6.1. In a system with homogeneous processors connected by a fast interconnection network (such as a linear systolic array assumed by S. Y. Kung *et al.* [11, 12]), an even distribution of neurons in a cluster to all processing cells results in the minimal completion time of the simulation.

Proof. Since the interconnection network is fast, the computation overhead dominates the communication overhead. According to Theorem 6.2, the entire system can be considered as one partition with negligible error in the optimal mapping. Further, according to Theorem 6.3, neurons should be mapped evenly to all processing elements.

Q.E.D.

The resource parameters of a partition q, including the set Q of processors, can be defined as follows.

$$\frac{1}{\tau_{c_q}} = \sum_{j \in Q} \frac{1}{\tau_{c_j}}, \tag{6.29a}$$

$$m_q = \sum_{j \in Q} m_j, \tag{6.29b}$$

$$\kappa_{ol_q} = 1 \quad \text{if } k_{ol_i} = 1 \text{ for some } i \in Q, \tag{6.29c}$$

$$\tau_{o_q} = \text{average of } \tau_{o_i}. \tag{6.29d}$$

After the partitions are generated, the communication links connecting a partition to other partitions can be grouped into *conceptual* links such that a conceptual link connecting two partitions includes all links connecting a processor in one partition to any processor in the second partition. The parameters of conceptual link λ, consisting of a set Λ of real links, can be defined as follows.

$$\tau_{s_\lambda} = \frac{1}{|\Lambda|} \sum_{i \in \Lambda} \tau_{s_i}, \tag{6.30a}$$

$$\frac{1}{\tau_{t_\lambda}} = \sum_{i \in \Lambda} \frac{1}{\tau_{t_i}}. \tag{6.30b}$$

P_{ts} also needs to be modified accordingly.

The complexity of the mapping problem depends on the number of partitions, the interconnection of multicomputers and the resource parameters. Figure 6.3 shows the mapping times for solving the optimal mapping of ANN FC-1 (which will be described in Chapter 7) onto different numbers of partitions (or processors for $\varepsilon = 0$). Note that the execution time grows exponentially with respect to the number of partitions, since the mapping problem is NP-hard. Figure 6.4 shows that the mapping times for solving the optimal mapping of fully connected ANNs having different numbers of clusters onto a three-partition (or three-processor for $\varepsilon = 0$) multicomputer. It is observed that the mapping time seems to grow exponentially with the number of clusters.

6.3. Decomposition of Error Allowance

The approach adopted to solving the optimal mapping problem consists of two stages: multicomputer partitioning and optimal mapping solver. Each stage can incur certain error degree to reduce the mapping time. The following lemma and theorems show that the total error degree incurred can be calculated by the error degrees incurred in each stage.

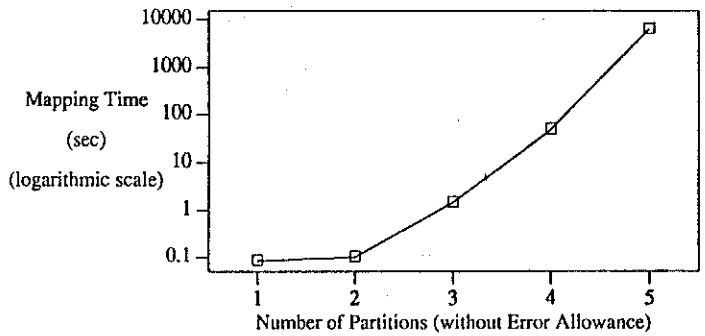

Figure 6.3 : Execution time of solving the optimal mapping of ANN FC-1 onto different numbers of partitions (or processors for $\varepsilon = 0$).

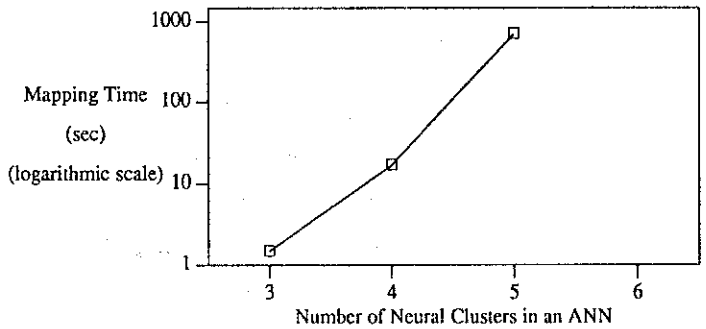

Figure 6.4 : Execution time of solving the optimal mapping of fully connected ANNs having different numbers of clusters onto a three-partition (or three-processor for $\varepsilon = 0$) multicomputer.

Lemma 6.5. Consider a two-stage problem solver, say S_1 and S_2, and each stage incurs a certain error degree, say ε_1 by S_1 and ε_2 by S_2. Then, the total error degree $\varepsilon_{1,2}$ incurred is bounded above by

$$\varepsilon_{1,2} \le \varepsilon_{1,2_{(max)}} = \varepsilon_1 + \varepsilon_2 + \varepsilon_1 \varepsilon_2. \tag{6.31}$$

Proof. Let t^* be the optimal solution value (the completion time based on the optimal mapping). Also let t_1 and t_2 be the solution values after error degrees ε_1 and ε_2, respectively, are incurred. Then, we have

$$t_1 \le t^* (1+\varepsilon_1), \tag{6.32a}$$

$$t_2 \le t_1 (1+\varepsilon_2). \tag{6.32b}$$

By combining the above two equations,

$$t_2 \le t^* (1+\varepsilon_1)(1+\varepsilon_2) = t^* (1+\varepsilon_1+\varepsilon_2+\varepsilon_1 \varepsilon_2). \tag{6.33}$$

Therefore, the maximum of the total error degree is

$$\varepsilon_{1,2_{(max)}} = \varepsilon_1 + \varepsilon_2 + \varepsilon_1 \varepsilon_2. \tag{6.34}$$

The lemma is proved by taking this maximum as its upper bound.

Q.E.D.

Theorem 6.4. Consider an n-stage problem solver, say $S_1, \ldots, S_n$, and each stage incurs certain error degree, say $\varepsilon_1, \ldots, \varepsilon_n$. Then, the total error degree $\varepsilon_{1,n}$ incurred is bounded above by

$$\varepsilon_{1,n} \le \sum_{k=1}^{n} \sum_{\text{for every } P_k \in S_{P_k}} \prod_{i \in P_k} \varepsilon_i. \tag{6.35}$$

where P_k is a permutation $(i_1, \ldots, i_k)$ from $(1, \ldots, n)$ and S_{P_k} is the set of all possible permutations.

Proof. This theorem can be proved by applying Lemma 6.5 iteratively. First, the composite error bound $\varepsilon_{1,2_{(max)}}$ incurred by stages S_1 and S_2 can be calculated by Eq. (6.31) in Lemma 6.5. Next, include stage S_3 and calculate the composite error bound $\varepsilon_{1,3_{(max)}}$ based on $\varepsilon_{1,2_{(max)}}$ and ε_3

as in Eq. (6.31). Iteratively, $\varepsilon_{1,k_{(max)}}$ can be calculated based on $\varepsilon_{1,k-1_{(max)}}$ and ε_k. Finally, $\varepsilon_{1,n_{(max)}}$ can be calculated.

Q.E.D.

The integration of error degrees gives the worst-case upper bound of the total error degree. For the case $n = 3$, the error bound is

$$\varepsilon_{1,3_{(max)}} = \varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_1 \varepsilon_2 + \varepsilon_1 \varepsilon_3 + \varepsilon_2 \varepsilon_3 + \varepsilon_1 \varepsilon_2 \varepsilon_3. \tag{5.36}$$

The transformation of a nonlayered ANN into a semi-layered ANN (mentioned in Section 2.4.1) is a sequence of merges. It can be treated as a sequence of stages, each for a merge. Each merge may incur certain error degree, which is bounded by the comm/comp ratio. By applying the result in Theorem 6.4, the total error degree incurred in the transformation can be calculated. If the error allowance is given, by using the result in Theorem 6.4 the feasibility of a transformation can be checked.

The transformation of the singular task nodes (mentioned in Section 2.4.2) is also a sequence of merges. The error incurred in a merge is bounded by the ratio of execution time of the particular singular task node to that of a large task node into which the singular one is to be merged.

If the total error degree allowed is given and the error degree incurred in multicomputer partitioning is also known, then the error degree allowed in the optimal mapping solver can be set based on the total error degree allowed and the error degree incurred in multicomputer partitioning, as shown in Figure 6.1. The following theorem and corollary show the decomposition of error degree in this approach.

Theorem 6.5. Consider a problem solver consisting of two stages, say S_1 and S_2. If the total error allowance is limited to $\varepsilon_{1,2}$ and the error degree incurred in stage S_1 is ε_1, then the error degree allowed to be incurred in stage S_2 is bounded above by

$$\varepsilon_2 \le \varepsilon_{2_{(max)}} = \frac{\varepsilon_{1,2} - \varepsilon_1}{1 + \varepsilon_1}. \tag{6.37}$$

Proof. To achieve the largest error allowance, let $\varepsilon_{1,2_{(max)}}$ equal $\varepsilon_{1,2}$ in Eq. (6.31). Then,

$$\varepsilon_{2_{(max)}} = \frac{\varepsilon_{1,2_{(max)}} - \varepsilon_1}{1 + \varepsilon_1} = \frac{\varepsilon_{1,2} - \varepsilon_1}{1 + \varepsilon_1}. \tag{6.38}$$

The theorem is proved by taking $\varepsilon_{2_{(max)}}$ as the upper bound.

Q.E.D.

The decomposition of error allowance in the n-stage problem solver can also be done by rearranging the equation in Theorem 6.4. The decomposition of error allowance provides an elegant way to reduce mapping time by scheduling error degrees to each stage based on the gain versus error in each stage.

Corollary 6.2. If the error allowance to the mapping problem is ε and the error degree (due to ignoring negligible communication time) incurred in multicomputer partitioning is ε_p, then the error degree ε_s allowed in the optimal mapping solver is

$$\varepsilon_s = \frac{\varepsilon - \varepsilon_p}{1 + \varepsilon_p}. \tag{6.39}$$

Proof. Because the approach to solving the optimal mapping problem consists of two stages, the result in Theorem 6.5 can be applied. By substituting $\varepsilon = \varepsilon_{1,2}$, $\varepsilon_p = \varepsilon_1$, and $\varepsilon_s = \varepsilon_2$ into Theorem 6.5, this corollary is proved by using the worst-case error allowance for ε_s.

Q.E.D.

6.4. Search Representation

The mapping problem formulated by the nonlinear integer programming can be solved by search. During the search, each node represents either a possible assignment of a certain cluster or a possible routing between two layers. One important feature of this representation is that the search branches on neural clusters rather than on individual neurons.

The branch-and-bound (B&B) algorithm is used as the search method in solving the mapping problem, since the branch-and-bound search is a general form of a variety of famous search methods. Further, some domain knowledge about multicomputer architecture and ANN

operations can help to accelerate the search by narrowing the search space. The B&B is preferred since the domain knowledge can be easily embedded into it.

A branch-and-bound search is characterized by four components: *branching* rule, *selection* rule, *pruning* rule and *termination* rule. The branching rule defines a mechanism for sprouting children. The selection rule defines a mechanism for selecting an existing node to evaluate. The pruning rule defines a mechanism for pruning unpromising nodes. One mechanism for pruning is using *bounding functions*. Each node is associated with a lower bound and an upper bound. A node can be pruned if its lower bound is larger than the incumbent.

A lower bound consists of two parts: *partial* solution cost t_{PS} and *lookahead* lower-bound solution cost t_{LB}. The partial solution cost of a node is a cost computed from the root of the search tree to this node. The lookahead solution cost is an under-estimated cost computed from this node to the optimal one in the subtree rooted at it. The upper bound also consists of two parts: partial solution cost and lookahead *feasible* solution cost t_{LF}. The lookahead feasible solution cost is a cost computed from this node to a feasible solution in the subtree rooted at it.

Let τ_c^* and τ_c' be the execution times per unit computation on the fastest partition and on the slowest partition, respectively. Let τ_t^* and τ_s^* be, respectively, the transmission time per unit communication and the setup time at the fastest link.

Consider a node in the search tree; suppose there are $n_{K'}$ clusters to be assigned. The lookahead lower-bound cost for computation denoted by $t_{C_{LB}}$ can be computed by simulating $n_{K'}$ neurons on all partitions, each of which is as fast as the most powerful partition in the system.

$$t_{C_{LB}} = \frac{\tau_c^* \, n_{K'}}{n_Q} \tag{6.40}$$

where n_Q is the number of partitions.

Let the lookahead feasible cost for computation be denoted by $t_{C_{LF}}$. The worst case is to simulate all neurons sequentially on the most powerful partitions. Note that in this case no communication is required.

$$t_{C_{LF}} = \tau_c^* \, n_{K'}. \tag{6.41}$$

Another possibility of calculating $t_{C_{LF}}$ is that the remaining K' clusters are mapped heuristically

($t_{C_{LF}}'$) plus the worst-case communication ($t_{R_{LF}}$ described later),

$$t_{C_{LF}}' = \frac{\tau_c' \, n_{K'}}{n_Q}. \tag{6.42}$$

Let the lookahead lower-bound cost for communication be $t_{R_{LB}}$. Suppose there are $n_{F'}$ frames to be migrated, say, $f_1, \ldots, f_{n_{F'}}$. To compute the lower bound, these frames are sent over a conceptual interconnection network consisting of n_L links, each has the same communication power as the most powerful link. Further, full utilization of all links is assumed.

$$t_{R_{LB}} = \frac{\tau_t^* \sum_{i=1}^{n_{F'}} \eta_{f_i} h_i + \tau_s^* \sum_{i=1}^{n_{F'}} h_i}{n_L}, \tag{6.43}$$

where h_i is equal to the number of communication hops in migrating frame f_i.

Let the lookahead feasible cost for computation be denoted by $t_{R_{LF}}$. The worst case is to migrate frames one by one by broadcasting a frame to its destination partitions via communication graph G_{f_i} (which will be described in Section 6.5).

$$t_{R_{LF}} = \sum_{i=1}^{n_{F'}} \left(\eta_{f_i} \sum_{j \in G_{f_i}} \tau_{t_j} + \sum_{j \in G_{f_i}} \tau_{s_j} \right). \tag{6.44}$$

Combining the above equations with overlap predicate κ_{oi}, the lower and upper bounds can be formulated as

$$t_{LB} = t_{PS} + \left(t_{R_{LB}} (1 - \kappa_{oi}) + t_{C_{LB}} \right), \tag{6.45}$$

$$t_{UB} = t_{PS} + \min \{ t_{R_{LF}} + t_{C_{LF}}', t_{C_{LF}} \}. \tag{6.46}$$

6.5. Domain Knowledge

Domain knowledge about multicomputer architecture and ANN operations is very useful to focus search around a path to the optimal mapping by exploring promising nodes and pruning unpromising nodes such that the mapping time is reduced.

Frames having a longer trip should be routed in preference to those having a shorter trip, since the former are very likely to become a communication bottleneck. Broadcast is preferred

in a topologically well-structured multicomputer such as hypercube computers, because neurons are very likely to be assigned to the majority of partitions.

When an ANN grows, computation time will become more dominant over communication time. This phenomenon can be illustrated by the following simple calculations. Consider a simple multilayered ANN has L layers and each layer has N neurons. Assume the number of processors in the target multicomputer is P. Computation time for a neuron is $O(N)$. Then, total computation time for a layer is $O(N^2)$. However, the communication for each layer is to migrate N neuron outputs over P processors at most. Then, total communication time for a layer is $O(PN)$. The ratio of computation to communication times is $O(N/P)$. As a result, computation time will become more dominant over communication time for a given multicomputer. This phenomenon is significant since a larger ANN needs less communication requirement. This dominance frequently occurs in that most ANNs are large enough. Actually, this dominance has been employed to reduce the complexity of solving the mapping problem by partitioning the target multicomputer with a negligible error.

The entire multicomputer can be treated as a graph if a partition is a vertex and a communication link is an edge. The set of partitions and links over which a frame f travels forms a *communication graph* (or simply *c-graph*) denoted by G_f. Consider a subgraph G'_f, which is composed of all partitions the frame f heads for and all links connecting all these partitions. Such a subgraph is called the *destination graph* (or simply *d-graph*). The d-graph usually is a spanning tree if no redundant communication occurs. The c-graph is the union of a d-graph and a path from the partition the frame f originates to any processor in the d-graph. The path is usually the shortest path between the source partition and the d-graph. If the source partition is a member of the d-graph, then this path is just null. In a routing problem, the c-graphs of all frames must be solved. The c-graphs may overlap each other, but interleave along a time axis.

CHAPTER 7.

EXPERIMENTAL RESULTS

In this chapter, experimental results on static-workload multicomputers and workload-varying multicomputers are shown. The experiments on static-workload multicomputers include those on a set of these three heterogeneous workstations (with static workloads) connected by Ethernet and those on 16-node, 8-node and 4-node Intel iPSC/2 hypercube computers. The experiments on workload-varying multicomputers include those on a network of these three workstations mentioned above (but with time-varying workloads), and 10-processor, 25-processor and 100-processor multicomputers connected by interconnection networks consisting of high-speed and low-speed communication links. Note that the experiments on time-varying workloads do not include those for iPSC/2 hypercube computers because their workloads are static.

Two programs are implemented for solving the optimal mapping problem. A program called *NeuMap* is the solver which includes a multicomputer partitioner and an optimal mapping solver. Another program called *Dsim* simulates workload-varying multicomputers.

Dsim allows communication on (point-to-point or bus) links as well as broadcasting. During dynamic mapping, Dsim and NeuMap cooperate in a way that Dsim is the client and NeuMap is the server. Whenever Dsim decides (based on the rules described in Section 5.4) to do re-mapping, Dsim calls NeuMap and waits for a new mapping scheme from NeuMap. After Dsim receives the new one, it starts to simulate the ANN simulation. Important symbols used in this chapter to illustrate the experimental results are summarized in Table 7.1.

All experiments use the ANN benchmarks whose parameters are summarized in Table 7.2 (a, b and c). All parameters are measured on a base machine which has the highest computation power among these three workstations. For hypercube computers, due to memory limitation, the sizes of ANNs are reduced to half, namely, the number of neurons in each cluster is reduced to half. For workload-varying multicomputers, including 10 processors, 25 processors and 100 processors, all ANNs are enlarged such that the number of neurons in each cluster is 10 times as much as that listed in Table 7.2a, b and c.

Table 7.1 : Summary of important symbols used in summarizing experimental results.

Symbol	Meaning	Symbol	Meaning
T_{seq}	completion time of one iteration of sequential ANN simulation on the machine which has the least completion time	$T_{\|\|}$	completion time of one iteration of parallel ANN simulation on the multicomputer according to the optimal mapping with a user-specified error
T_{pred}	predicted (by NeuMap) completion time of one iteration of parallel ANN simulation on the multicomputer according to the optimal mapping with a user-specified error	T_{sim}	simulated (on Dsim) completion time of one iteration of parallel ANN simulation on the multicomputer according to the optimal mapping with a user-specified error
S	speedup of parallel ANN simulation ($S \triangleq T_{seq} / T_{\|\|}$)	ε_{user}	error allowance specified by a user
ε_{pred}	difference between $T_{\|\|}$ and T_{pred} ($\varepsilon_{pred} \triangleq \mid T_{pred} - T_{\|\|} \mid / T_{\|\|}$)	ε_{sim}	difference between $T_{\|\|}$ and T_{sim} ($\varepsilon_{sim} \triangleq \mid T_{sim} - T_{\|\|} \mid / T_{\|\|}$)
T_{static}	total simulated (on Dsim) completion time of N_{iter} iterations of ANN simulation plus one mapping time expended in NeuMap	T_{dyn}	total simulated (on Dsim) completion time of N_{iter} iterations of ANN simulation plus the summation of all mapping times expended in NeuMap
g	gain of using dynamic mapping algorithm over static one ($g \triangleq T_{static} / T_{dyn}$)	n_k	number of neurons in cluster k
S_{max}	maximum possible speedup of parallel processing of ANN simulations (if all processors are homogeneous, then it is equal to the number of processors)	g_{max}	maximum possible gain of using dynamic mapping algorithm over static one; it is obtained by performing mapping for each iteration and ignoring mapping overhead
η_S	normalized speedup ($\eta_S \triangleq S / S_{max}$)	η_g	normalized gain ($\eta_g \triangleq g / g_{max}$)
$Prec_k$	set of preceding clusters of cluster k	$Succ_k$	set of succeeding clusters of cluster k
$\eta_{p,k}$	amount of production-phase computation per neuron in cluster k	$\eta_{l,k}$	amount of learning-phase computation per neuron in cluster k

Table 7.2a: Summary of models of these three fully connected multilayered ANNs.

Fully Connected Feed-Forward ANNs											
ANN Topology			FC-1			FC-2			FC-3		
Cluster k	$Prec_k$	$Succ_k$	n_k	$\eta_{p,k}$	$\eta_{l,k}$	n_k	$\eta_{p,k}$	$\eta_{l,k}$	n_k	$\eta_{p,k}$	$\eta_{l,k}$
1	IN	2	500	2.49	2.73	600	2.99	3.28	200	1.06	1.08
2	1	3	1000	2.49	5.31	200	2.99	6.41	1500	1.06	2.15
2	OUT	3	200	4.94	10.63	500	1.05	2.15	200	7.35	15.98

Table 7.2b: Summary of models of these three hybrid multilayered ANNs.

Multilayered Feed-Forward ANNs											
ANN Topology			ML-1			ML-2			ML-3		
Cluster k	$Prec_k$	$Succ_k$	n_k	$\eta_{p,k}$	$\eta_{l,k}$	n_k	$\eta_{p,k}$	$\eta_{l,k}$	n_k	$\eta_{p,k}$	$\eta_{l,k}$
1	IN	2, 3, 4	500	2.53	2.80	200	1.05	1.14	800	4.26	4.60
2	1	5	200	2.53	5.52	300	1.05	2.17	300	4.26	8.80
3	1	5, 6	500	2.53	5.52	500	1.05	2.17	200	4.26	8.80
4	1	6	300	2.53	5.52	200	1.05	2.17	400	4.26	8.80
5	2, 3	7	400	3.56	7.58	600	3.97	8.55	200	2.73	5.34
6	3, 4	7	600	4.13	8.56	400	3.47	7.46	500	3.19	6.43
7	5, 6	OUT	200	5.08	10.69	200	4.94	10.65	400	3.74	7.51

Table 7.2c: Summary of models of these three nonlayered ANNs.

Nonlayered Feed-Forward ANNs											
ANN Topology			NL-1			NL-2			NL-3		
Cluster k	$Prec_k$	$Succ_k$	n_k	$\eta_{p,k}$	$\eta_{l,k}$	n_k	$\eta_{p,k}$	$\eta_{l,k}$	n_k	$\eta_{p,k}$	$\eta_{l,k}$
1	IN	2, 3, 6	300	1.65	1.73	800	4.33	4.95	400	2.01	2.14
2	1	4, 5	800	1.65	3.36	500	4.33	9.02	300	2.01	4.35
3	1	8	600	1.65	3.29	400	4.33	8.75	200	2.01	4.30
4	2	8	500	4.27	8.97	400	2.76	5.46	200	1.52	3.18
5	2	7	700	4.27	8.90	800	2.76	5.39	400	1.52	3.26
6	1	8	400	1.65	3.29	400	4.35	8.75	500	2.01	4.30
7	5	8	600	3.76	7.63	500	4.35	8.51	300	2.02	4.31
8	3, 4, 6, 7	OUT	300	11.32	22.50	200	8.89	18.10	400	5.89	12.86

Table 7.3: Summary of communication parameters used.

Communication Parameters					
Communication Parameter	Workstation	Hypercube Computer			
	Bus	Node-to-Node Link	16-Node Broadcast	8-Node Broadcast	4-Node Broadcast
τ_s (ms)	108.36	0.65	6.5	3.6	2.0
τ_t (μs)	5.33	3.95	103	48	21

The communication parameters used are summarized in Table 7.3 which includes those for a bus connecting these three workstations and those for 4-node, 8-node and 16-node hypercube computers. Communication setup time is obtained by measuring the transmission time of a null communication. Transmission time per word is obtained by applying linear approximation over communication times for different frame sizes. Note that all communication parameters include preprocessing and postprocessing times.

7.1. Experiments on Static-Workload Multicomputers

Static workloads are found in hypercube computers or in a network of workstations under exclusive usage, which change with time very slowly and negligibly. Note that all experiments are measured for one iteration of ANN simulation, since all iterations are identical.

7.1.1. Experiments on static-workload workstations

The target multicomputer is a set of these three heterogeneous workstations connected by an Ethernet, which is specified in Example 3.2 in Section 3.3. Machine 1 has the lowest computation power and machine 3 the highest. Each processor is assumed to use virtual-circuit communication, which has a one-time setup cost. Broadcast using datagrams on Ethernet is not used in the experiments due to the small number of processors. However, it will be useful when the number of processors is large.

The predicted and experimental results are summarized in Table 7.4. The experiments are conducted as follows. The optimal mappings are solved with user-specified error $\varepsilon_{user} = 0$ for FC-1, FC-2 and FC-3, and with $\varepsilon_{user} = 1\%$ for other ANNs. Note that the predicted error ε_{pred} is larger than the user-specified error allowance ε_{user}, even though ε_{pred} is very small, around 1%–2%. The reason is that synchronization and problem-partitioning overheads are not included in the model proposed. Further, unexpected page faults may also cause some overheads. The simulation error ε_{sim} (on Dsim) is also very small; this could be the empirical evidence of correctness of Dsim.

It is observed in Table 7.4 that the speedup efficiency η_S changes slowly with the communication requirements, namely, the speedup efficiency is slightly higher for fully connected

Table 7.4: Summary of predicted and experimental results of simulating the nine ANNs on the three workstations with static workloads.

Experimental Results on the Three Workstations								
ANN	T_{pred} (sec)	$T_{\|}$ (sec)	ε_{pred} (%)	T_{seq} (sec)	S	η_S	T_{sim} (sec)	ε_{sim} (%)
FC-1	100.95	102.22	1.25	225.47	2.23	.995	101.10	1.09
FC-2	54.03	54.83	1.47	121.23	2.21	.986	54.33	0.92
FC-3	74.15	74.95	1.07	167.88	2.23	.995	74.20	1.01
ML-1	193.31	195.42	1.07	435.80	2.23	.995	193.96	0.74
ML-2	139.06	141.02	1.39	310.40	2.20	.982	139.52	1.06
ML-3	221.83	225.63	1.71	485.18	2.15	.960	222.61	1.34
NL-1	318.82	319.67	0.26	704.88	2.21	.987	320.26	0.18
NL-2	344.36	355.88	3.24	748.07	2.10	.938	346.91	2.52
NL-3	150.81	154.87	2.62	338.93	2.19	.978	151.84	1.95

ANNs and slightly lower for nonlayered ANNs, because the former have lower communication requirements and the latter higher.

7.1.2. Experiments on hypercube computers

The 16-node iPSC/2 hypercube computer [1, 2] can be configurated to 16-node, 8-node or 4-node hypercube computers. It provides packet switching for inter-processor communication [15]. Concurrent asynchronous broadcasts allow the network server to route frames more efficiently so that full utilization on communication links can almost be achieved. Note that the broadcast parameters in Table 7.3 are measured under the condition that all processors broadcast concurrently and asynchronously rather than that one processor broadcasts at a time.

The predicted and experimental results of simulating these nine ANNs listed in Table 7.2 on hypercube computers of different cube sizes are summarized in Table 7.5. The experiments are conducted as follows. The optimal mappings are solved with user-specified error $\varepsilon_{user} = 1\%$ for the 4-node cube, and $\varepsilon_{user} = 2\%$ for 8-node and 16-node cubes. As in the workstation case, the predicted error ε_{pred} is larger than the user-specified error ε_{user}, even though ε_{pred} is very small, around 1%~3%. The reason is that synchronization and problem-partitioning overhead is not included in the model proposed.

Table 7.5: Summary of predicted and experimental results of simulating the nine ANNs on the hypercube computers.

Experiment Results on iPSC/2 Hypercube Computer												
ANN	4-Node Hypercube				8-Node Hypercube				16-Node Hypercube			
	T_{pred} (sec)	$T_{\|}$ (sec)	ε_{pred} (%)	S	T_{pred} (sec)	$T_{\|}$ (sec)	ε_{pred} (%)	S	T_{pred} (sec)	$T_{\|}$ (sec)	ε_{pred} (%)	S
FC-1	2.239	2.248	0.40	3.89	1.154	1.170	1.37	7.48	0.645	0.645	0.00	13.57
FC-2	1.177	1.180	0.25	3.91	0.612	0.627	2.39	7.37	0.354	0.355	0.28	13.01
FC-3	1.672	1.667	0.30	3.87	0.880	0.880	0.00	7.34	0.528	0.524	0.76	12.32
ML-1	4.273	4.300	0.63	3.64	2.244	2.302	2.52	6.80	1.329	1.359	2.21	11.52
ML-2	3.154	3.185	0.97	3.60	1.684	1.701	1.00	6.73	1.051	1.033	1.74	11.09
ML-3	4.727	4.799	1.50	3.65	2.465	2.556	3.67	6.86	1.428	1.495	4.48	11.72
NL-1	6.797	6.848	0.74	3.68	3.540	3.585	1.26	7.03	2.044	2.031	0.64	12.41
NL-2	7.228	7.311	1.14	3.67	3.736	3.784	1.27	7.09	2.103	2.150	2.19	12.48
NL-3	3.369	3.382	0.38	3.64	1.798	1.816	0.99	6.78	1.121	1.153	2.78	10.67

It is interesting to observe that the speedup efficiency is higher for smaller cubes. This happens because for a larger cube each node has less computation task and the synchronization and problem-partitioning overhead becomes relatively more significant. Another minor reason is that larger cubes need more time for inter-processor communication. It is also interesting to note that for the same cube size the speedup changes slowly with the communication requirement, as in the workstation case.

A major limitation on using hypercubes for ANN simulations is due to memory space limitation in each processor. The system lacks a virtual-memory facility from each processor to the common secondary memory, and all accesses to the secondary memory must be handled by the Cube Manager. When the number of neurons mapped to each processor is larger than the capacity of its local memory, part of the data must be kept in the Cube Manager. This results in a high volume of traffic between the Cube Manager and the rest of the system.

7.2. Experiments on Workload-Varying Multicomputers

The static mapping algorithm is insufficient to capture time-varying workloads. Instead, the dynamic mapping algorithm is used. The dynamic mapping algorithm repeats the optimal mapping of the ANN simulation whenever the workload changes significantly.

Parallel ANN simulations are performed on Dsim rather than on a real multicomputer as in the previous two experiments. The reason is two-fold.

(1) To illustrate the effect of workloads in ANN simulations, the reproduction of workloads is essential; however, it is very difficult to reproduce workloads in a real multicomputer.

(2) The mapping results have already been empirically verified in the previous two experiments; therefore, it is reasonable to use Dsim, which also is empirically verified in the case of the three workstations.

Each machine is associated with a workload descriptor (defined in Section 3.3) as 6-ary tuple $WL = <p_0, p_1, p_2, \delta, b_u, b_l>$, where p_0, p_1, and p_2 are the probabilities that the workload in the next iteration of ANN simulation will remain the same, increase, and decrease, respectively, δ is the slope of change in workload if the workload increases or decreases, and b_u and b_l are the upper bound and lower bound on workload, respectively. Note that $p_0 + p_1 + p_2 = 1$.

Given a workload descriptor for each machine after the k-th iteration of ANN simulation, the workload ω for the $(k+1)$-th iteration is generated based on the descriptor and ω in the k-th iteration. The procedure of workload generation is shown in Heuristic Workload Generation Algorithm 7.1.

Heuristic Workload Generation Algorithm 7.1.

1. Generate a random variable $v \in [0,1)$.
2. If ω_k is not *saturated*, namely, it is neither equal to b_u nor b_l, then
2.1. If $v < p_0$ then $\omega_{k+1} = \omega_k$,
2.2. Else if $v < p_0 + p_1$ then $\omega_{k+1} = \omega_k + \delta$,
2.3. Else (namely, $v \geq p_0 + p_1$) $\omega_{k+1} = \omega_k - \delta$.
2.4. If $\omega_{k+1} > b_u$, then let $\omega_{k+1} = b_u$.
2.5. If $\omega_{k+1} < b_l$, then let $\omega_{k+1} = b_l$.
3. If ω_k is saturated to the upper bound, namely, $\omega_k = b_u$, then
3.1. If $v < p_0 + p_1$, then $\omega_{k+1} = \omega_k$,
3.2. Else (namely, $v \geq p_0 + p_1$) $\omega_{k+1} = \omega_k - \delta$.
4. If ω_k is saturated to the lower bound, namely, $\omega_k = b_l$, then
4.1. If $v < p_0 + p_2$, then $\omega_{k+1} = \omega_k$,
4.2. Else (namely, $v \geq p_0 + p_2$) $\omega_{k+1} = \omega_k + \delta$.

There may exist better workload generation algorithms in terms of modeling or synthesizing real workloads. The workload generation used is not intended to synthesize or reproduce the past real workloads on real machines. Instead, it is used only for evaluating the dynamic mapping algorithm through its wide spectrum of reproducible random generations of workloads. Table 7.6 shows the workload descriptors ($b_l = 1$ and $b_u = 25$) of WL-1, WL-2 and WL-3 used in the three-processor case. By examining the workloads (not shown here due to space limitation) generated by Heuristic Workload Generation Algorithm 7.1, it was found to be sufficient for this purpose. Note that the generation of workloads is a Markov process, because the new workload is based only on the very previous one rather than the entire history of

workloads. Further, the generation of workloads can be reproduced because the seed to the random generator can be controlled.

Table 7.6: Workload descriptors of WL-1, WL-2 and WL-3 used in the three-processor case.

Summary of Workload Descriptors													
Workload Index	Random Seed	Machine 1				Machine 2				Machine 3			
		p_0	p_1	p_2	δ	p_0	p_1	p_2	δ	p_0	p_1	p_2	δ
WL-1	1	.24	.71	.05	0.70	.13	.57	.30	0.78	.29	.53	.18	0.59
WL-2	2	.15	.51	.34	0.57	.02	.83	.15	0.40	.01	.82	.17	0.54
WL-3	3	.22	.72	.06	0.44	.27	.69	.04	0.69	.15	.70	.15	0.52

The descriptors for all machines in the experiments are generated randomly. The lower bound and upper bound of workloads in the experiments are set to 1 and 25. Note that if the workload is equal to 1, it means that the processor is totally dedicated to the ANN simulation; while if the workload is equal to 25, it means that only 4% of processor utilization is for the ANN simulation.

The simulation results on workload-varying multicomputers, including those on 3-workstation, 10-processor, 25-processor and 100-processor multicomputers, are shown in Table 7.7. The gain g is referred to the speedup of the dynamic mapping algorithm with respect to the static. Specifically, for the duration of N_{iter} iterations, the gain is defined as $g \triangleq T_{dyn} / T_{static}$, where T_{dyn} is total simulated completion time of N_{iter} iterations of ANN simulation plus the sum

Table 7.7: Summary of simulation results on workload-varying multicomputers.

Simulation Results on Workload-Varying Multicomputers with 95% Confidence									
ANN	Workload	3 Processors		10 Processors		25 Processors		100 Processors	
		E[g]	$E[\eta_g]$	E[g]	$E[\eta_g]$	E[g]	$E[\eta_g]$	E[g]	$E[\eta_g]$
FC-1	WL-1	1.46±.13	.987±.001	1.93±.01	.972±.001	2.43±.01	.981±.001	3.43±.15	.915±.018
FC-2	WL-2	2.07±.01	.912±.002	2.24±.01	.970±.001	2.69±.01	.971±.001	4.12±.20	.934±.017
FC-3	WL-3	1.10±.01	.996±.001	2.36±.01	.906±.001	2.83±.01	.987±.001	4.33±.20	.953±.013
ML-1	WL-4	1.42±.01	.978±.001	1.66±.01	.992±.001	2.37±.02	.952±.002	3.92±.18	.892±.019
ML-2	WL-5	1.22±.01	.968±.001	2.09±.01	.960±.001	1.98±.02	.832±.003	4.08±.19	.878±.019
ML-3	WL-6	2.24±.01	.968±.001	2.28±.01	.965±.001	2.25±.02	.954±.001	4.35±.19	.873±.015
NL-1	WL-7	4.17±.02	.959±.001	2.09±.01	.962±.001	2.12±.02	.987±.001	3.46±.16	.903±.019
NL-2	WL-8	2.01±.01	.921±.001	1.79±.01	.947±.001	2.08±.01	.972±.001	3.70±.16	.907±.021
NL-3	WL-9	1.59±.01	.893±.002	2.02±.01	.968±.001	2.22±.01	.945±.001	3.76±.15	.813±.018

of all mapping times expended in the re-mappings, and T_{static} is total simulated completion time of N_{iter} iterations of ANN simulation plus one mapping time.

The expected gains and gain efficiencies shown in Table 7.7 are *95% confidence intervals*. For 3, 10 and 25 processors, in all experiments, each sample has 10 time quantums and each time quantum corresponds to one iteration of ANN simulation. For 3, 10 and 25 processors, the number of samples is 100 each. For 100 processors, the number of samples is 10 only because of its long simulation time.

In Table 7.7, the gain is usually around 1–4, which is not significantly large, because the dynamic mapping algorithm can gain significantly only when the workload in each machine is significantly divergent. For example, if all machines have similar workload behaviors (high or low almost at the same time), then the re-mapping does not gain too much because the relative

computation power of each machine still remains the same with respect to other machines. However, if some machines have very divergent workload behaviors, then re-mapping can gain significantly because the relative computation power of each machine changes.

It is interesting to note that larger multicomputers generally have better gains but worse gain efficiencies. The reason for better gains is that larger multicomputers are likely to have divergent workload behaviors. Worse gain efficiencies occur because re-mapping overhead is relatively higher for larger multicomputers since their simulation times are shorter.

Gain efficiency η_g is referred to the normalized goodness of the dynamic mapping algorithm with respect to the best gain g_{max} which is calculated under the condition that re-mapping is performed for each iteration and the time expended in the re-mapping is not counted. Specifically, $\eta_g \triangleq g / g_{max}$. In Table 7.7, gain efficiencies are very high; this indicates that the dynamic mapping algorithm achieves almost full utilization of resources.

CHAPTER 8.

CONCLUSIONS

In this thesis, the optimal mapping of neurons in the static learning process of the multilayered artificial neural network on the multicomputer system is studied. Processors in the multicomputer may be heterogeneous and with time-varying workloads, and also may be connected by communication links of different speeds. The mapping problem is NP-hard in general. A number of results are developed for simplifying the complexity of the mapping problem. By observing that the computation time usually dominates the communication time in the learning operations within a layer of the neural network, the processors are partitioned into groups such that the error deviation of a heuristic routing scheme from the optimal one can be bounded. Experimental results using a 16-node Intel iPSC hypercube computer and a network of three workstations are shown and are found to be very close to the results predicted analytically. Experimental results on time-varying workloads show that the dynamic mapping algorithm can achieve almost full utilization of computing resources. The results obtained are useful for designing a special-purpose computer for ANN simulations and for determining the suitability of an existing computer system for ANN applications.

REFERENCES

[1] R. Arlauskas, "iPSC/2 system: A second generation hypercube," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988.

[2] P. Close, "The iPSC/2 node architecture," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988.

[3] J. A. Feldman, M. A. Fanty, N. H. Goddard, and K. J. Lynne, "Computing with structured connectionist networks," *Communications of the ACM*, vol. 31, no. 2, pp. 170-187, February 1988.

[4] B. M. Forrest, D. Roweth, N. Stroud, D. J. Wallace, and G. V. Wilson, "Implementing neural network models on parallel computers," *The Computer Journal*, vol. 30, no. 5, pp. 413-419, British Computer Society, Cambridge University Press, 1987.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability*. San Francisco, CA: Freeman, 1979.

[6] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*. New York, NY: John Wiley & Sons, 1972.

[7] J. Ghosh and K. Hwang, "Mapping neural networks onto message-passing multicomputers," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 291-330, 1989.

[8] H. P. Graf , L. D. Jackel, and W. E. Hubbard, "VLSI implementation of a neural network model," *IEEE Computer*, vol. 21 no. 3 pp. 41-49, March 1988.

[9] K. Hwang and J. Ghosh, "Critical issues in mapping neural networks on message-passing multicomputers," *International Symposium on Computer Architecture*, pp. 3-11, ACM/IEEE, 1988.

[10] D. S. Johnson, J. K. Lenstra and A. H. G. Rinnooy Kan, "The complexity of the network design problem," *Networks*, 1978.

[11] S. S. Kung and J. N. Hwang, "Parallel architectures for artificial neural nets," *Proceedings of the International Conference on Systolic Arrays*, pp. 163-174, IEEE, 1988.

[12] S. Y. Kung and J. N. Hwang, "A unified systolic architecture for artificial neural networks," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 358-387, 1989.

[13] R. P. Lippmann, "An introduction to computing with neural nets," *IEEE Acoustics, Speech and Signal Processing Magazine*, pp. 4-22, April 1987.

[14] J. L. McClelland and D. E. Rumelhart, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Cambridge, MA: Bradford Books (MIT Press), 1985.

[15] S. Nugent, "The iPSC/2 direct-connect communication technology," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988.

[16] D. A. Pomerleau, G. S. Gusciora, D. S. Touretzky, and H. T. Kung, "Neural network simulation at Warp speed: How we got 17 million connections per second," *Proceedings of International Conference on Neural Networks*, vol. II, pp. 143-150, IEEE, July 1988.

[17] J. D. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Sciences*, vol. 10, pp. 384-393, 1975.

[18] B. W. Wah and L.-C. Chu, "Efficient mapping of neural networks on multicomputers," *Proceedings of International Conference on Parallel Processing* vol. I, pp. 234-241, Aug. 1990.

APPENDIX A.

NEUMAP PROGRAM

This appendix mainly describes NeuMap and how to use it. Section A.1 describes briefly the algorithm, organization and options in NeuMap. Section A.2 describes how to use it. Finally, Section A.3 lists its source code.

A.1. General Description

NeuMap is a program which maps ANN simulations onto message-passing multicomputers. The scenario in NeuMap is as belows.

1. NeuMap reads in multicomputer and ANN specifications.
2. NeuMap partitions the multicomputer based on a user-specified error allowance. Attributes in the unpartitioned multicomputer are transformed accordingly.
3. NeuMap calculates error allowed in the mapping solver.
4. NeuMap solves optimal mapping with error allowed.

The major components in NeuMap include (1) interface for reading in multicomputer and ANN specifications, (2) partitioning of a multicomputer, and (3) branch-and-bound mapping solver. These components are physically placed in the following directories: include, interface, partition and search. They are quite self-explained. Detailed descriptions for each individual file will be explained in site rather than here because there are too many files.

The compilation of NeuMap is done by a command "make neumap". Other available makefile options are listed and self-explained in the Makefile. Dsim of different modes can also be compiled by it.

Several options are available for reporting statistics and debugging NeuMap:

STATISTICS	Reports all important statistics.
DEBUG	Provides multiple levels of details in debugging information.

A.2. Steps for Using NeuMap

Follow these steps to execute the supplied program.

(1) Go to the directory ~software/chu/ms/bin.

(2) Specify the options described in the previous section for compilation in file Makefile.

(3) Compile NeuMap by typing "make neumap".

(4) Create input files for NeuMap. Two input files are required; one is the specification of a multicomputer and the other is the specification of an ANN. Examples are available in directory ~software/chu/ms/lab/input. The format of input files is described later.

(5) Run the program by typing "neumap mimd-map ann-map error debug-mode > output", where "mimd-map" is an input file name for a multicomputer, "ann-map" is an input file name for an ANN, "error" is the user-specified error allowance, *e.g.*, 0.05, "debug-mode" is the level of detail of debugging information, *e.g.*, 1 for less details and 4 for complete details, and "output" is an output file.

The format of the input file for a multicomputer is described as follows.

(1) Specify the number of processors, the number of links and the diameter of this multicomputer. For example, for a 16-node hypercube, they are

16 32 4

Next, provide some knowledge of the multicomputer such as the topology, if available. For example, for hypercube computers,

HOME

SYM

The former indicates that processors and links are all homogeneous. The latter indicates that the computer is symmetric. For the three workstations, they are neither, then nothing is required to be specified.

(2) Specify the processors. The specification is enclosed by "startproc" and "endproc". Each type of processors is specified in two lines. The first line specifies how many processors belong to this type. The second specifies execution time per unit computation, local memory size in word, IO facility (1 for yes and 0 for no), overlap predicate (1 for yes and 0 for no) and overlap overhead. For example, for a 16-node hypercube, they are

startproc

homo 16

1.0 1000 0 0 0.0

endproc

It means that there are 16 homogeneous processors each having 1.0 millisecond for unit computation, 1000 kilo-words, no IO facility, no overlap and overlap overhead is 0.0. For the network of three workstations, they are

startproc

homo 1

28.5 20000 1 0 0.0

homo 1

25.5 50000 1 0 0.0

homo 1

16.7 50000 1 0 0.0

endproc

(3) Specify the adjacency matrix of the interconnection network. The specification is enclosed by "startadj" and "endadj". If there are P processors, then there will be P lines each having P entries. Each entry indicates the link which connects them. For example, for the three workstations connected by a bus, it can be written as

startadj

-1 0 0

0 -1 0

0 0 -1

endadj

where -1 indicates that there is no such link. For a 16-node hypercube, it is shown in the file ~software/chu/ms/lab/input/ipsc/mimd-map-16 listed in this appendix.

(4) Specify the links. The specification is enclosed by "startlink" and "endlink". Each line in between specifies the setup and transmission times for each transmission. For example, for the three workstations, it is

startlink

homo 1

0.0 0.00533

endlink

Note that the setup time in the three-workstation case is 0 because it is a one-time cost and will not require setup for each transmission. For a 16-node hypercube, it is shown in the file ~software/chu/ms/lab/input/ipsc/mimd-map-16 listed in this appendix.

(5) Specify the processors supported by these links. The specification is enclosed by "startsupport" and "endsupport". Each link requires two lines of specification. The first line specifies the link and the number of supported processors. The second line specifies all of the processors it supports. For example, for the network of three workstations, it is specified as

startsupport

0 3

0 1 2

endsupport

They are read as the link 0 has three supported processors, 0, 1, and 2.

(6) Specify broadcast facility. The specification is enclosed by "startbc" and "endbc". Two lines are required for each broadcast. The first line indicate the broadcast. The second line specifies the scope of processors covered, setup and transmission times. For a 16-node hypercube, it can be written as

startbc

1

0 15 6.5 0.103

endbc

They are read as this multicomputer has one broadcast which covers processors 0, ..., 15. Also, the setup and transmission times are 6.5 and 0.103, respectively.

The format of the input file for an ANN is described as follows.

(1) Specify the type of this ANN and its structure. The first line specifies the type. Two types are covered at the present time. They are FULL (for fully connected multilayered ANN) and HYBRID (for hybrid multilayered ANN). The second line specifies the number of layers and the number of clusters. Then, each layer is described by one line, each specifying the number of clusters in it and what they are. For example, the ML-1 is specified as

HYBRID

4 7

0 1 0

1 3 1 2 3

2 2 4 5

3 1 6

They are read as ML-1 is a hybrid multilayered ANN which has 4 layers and 7 clusters. Layer 0 consists of one cluster and it is cluster 0. Layer 1 consists of three clusters, 1, 2 and 3. Layer 2 consists of two clusters, 4 and 5. Layer 3 consists of one cluster and it is cluster 6.

(2) Specify each cluster one by one. One cluster requires four lines. The first line specifies the cluster, the range of neurons belonging to it, and its IO features. The second line specifies the per-neuron production-phase computation units and communication units, and learning-phase computation units and communication units. The third line specifies the number of the preceding clusters and what they are. The fourth line specifies the number of the succeeding clusters and what they are. For the example of ML-1, they can be written as

0 0 499 1 0

2.53 1. 2.80 1.

0

3 1 2 3

The first line is read as cluster 0 includes neurons 0, ..., 499, and it is an input cluster (the fourth is 1) and not an output cluster (the fifth is 0).

Specifically, examples of multicomputers are in files

~software/chu/ms/lab/input/sun-3/mimd-map

~software/chu/ms/lab/input/ipsc/mimd-map-4

~software/chu/ms/lab/input/ipsc/mimd-map-8

~software/chu/ms/lab/input/ipsc/mimd-map-16

Examples of ANNs for mapping onto workstations are in files

~software/chu/ms/lab/input/full-net/ann-map-[1-6]

~software/chu/ms/lab/input/multi-layer/ann-map-[1-6]

~software/chu/ms/lab/input/non-layered/ann-map-[1-6]

Examples of ANNs for mapping onto hypercube computers are in files

~software/chu/ms/lab/input/ipsc/ann-map-FC[1-3]

~software/chu/ms/lab/input/ipsc/ann-map-ML[1-3]

~software/chu/ms/lab/input/ipsc/ann-map-NL[1-3]

The output is very self-explained and can be easily understood, because all output data are accompanied with clear explanations. Therefore, they are not repeated here.

~software/chu/ms/lab/input/multi-layer/ann-map-1
ML-1 (4 Layers and 7 Clusters)

```
HYBRID
4 7
0 1 0
1 3 1 2 3
2 2 4 5
3 1 6
0 0 499 1 0
2.53 1. 2.80 1.
0
3 1 2 3
1 500 699 0 0
2.53 1. 5.52 1.
1 0
1 4
2 700 1199 0 0
2.53 1. 5.52 1.
1 0
2 4 5
3 1200 1499 0 0
2.53 1. 5.52 1.
1 0
1 5
4 1500 1899 0 0
3.56 1. 7.58 1.
2 1 2
1 6
5 1900 2499 0 0
4.13 1. 8.56 1.
2 2 3
1 6
6 2500 2699 0 1
5.08 1. 10.69 1.
2 4 5
0
```

~software/chu/ms/lab/input/sun-3/mimd-map
3 Workstations Connected by Ethernet


```
3 1 1
startproc
homo 1
28.5 20000 1 0 0.0
homo 1
25.5 50000 1 0 0.0
homo 1
16.7 50000 1 0 0.0
endproc
startadj
-1 0 0
0 -1 0
0 0 -1
endadj
startlink
homo 1
0.0 0.00533
endlink
startsupport
0 3 0 1 2
endsupport
```

~software/chu/ms/lab/input/sun-3/mimd-map
16-Node iPSC/2 Hypercube Computer

```
16 32 4
HOMO
SYM
startproc
homo 16
1.0 1000 0 0 0.0
endproc
startadj
-1 0 1 -1 4 -1 -1 -1 24 -1 -1 -1 -1 -1 -1 -1
0 -1 -1 2 -1 5 -1 -1 -1 25 -1 -1 -1 -1 -1 -1
1 -1 -1 3 -1 -1 6 -1 -1 -1 26 -1 -1 -1 -1 -1
-1 2 3 -1 -1 -1 -1 7 -1 -1 -1 27 -1 -1 -1 -1
4 -1 -1 -1 -1 8 9 -1 -1 -1 -1 -1 28 -1 -1 -1
-1 5 -1 -1 8 -1 -1 10 -1 -1 -1 -1 -1 29 -1 -1
-1 -1 6 -1 9 -1 -1 11 -1 -1 -1 -1 -1 -1 30 -1
-1 -1 -1 7 -1 10 11 -1 -1 -1 -1 -1 -1 -1 -1 31
24 -1 -1 -1 -1 -1 -1 -1 -1 12 13 -1 16 -1 -1 -1
-1 25 -1 -1 -1 -1 -1 -1 12 -1 -1 14 -1 17 -1 -1
-1 -1 26 -1 -1 -1 -1 -1 13 -1 -1 15 -1 -1 18 -1
```

```
-1 -1 -1 27 -1 -1 -1 -1 -1 14 15 -1 -1 -1 -1 19
-1 -1 -1 -1 28 -1 -1 -1 16 -1 -1 -1 -1 20 21 -1
-1 -1 -1 -1 -1 29 -1 -1 -1 17 -1 -1 20 -1 -1 22
-1 -1 -1 -1 -1 -1 30 -1 -1 -1 18 -1 21 -1 -1 23
-1 -1 -1 -1 -1 -1 -1 31 -1 -1 -1 19 -1 22 23 -1
endadj
startlink
homo 32
0.65 0.00395 1
endlink
startsupport
0 2 0 1
1 2 0 2
2 2 1 3
3 2 2 3
4 2 0 4
5 2 1 5
6 2 2 6
7 2 3 7
8 2 4 5
9 2 4 6
10 2 5 7
11 2 6 7
12 2 8 9
13 2 8 10
14 2 9 11
15 2 10 11
16 2 8 12
17 2 9 13
18 2 10 14
19 2 11 15
20 2 12 13
21 2 12 14
22 2 13 15
23 2 14 15
24 2 0 8
25 2 1 9
26 2 2 10
27 2 3 11
28 2 4 12
29 2 5 13
30 2 6 14
31 2 7 15
endsupport
startbc
1
0 15 6.5 0.103
endbc
```


A.3. Program Listing

Makefile

```
# make neumap     ==>   neumap
# make dsim       ==>   dsim
# make gamx       ==>   gmax
# make sim        ==>   sim
# make all        ==>   neumap, dsim, gamx and sim

# Three compilation options:
#       STATISTICS: reports important statistics.
#       DEBUG: provides mutiple levels of details in debugging information.
#       NOLOGO: does not print a logo.

CC = cc
CminorFlags = -DDEBUG -DSTATISTICS -g
#CminorFlags = -DNOLOGO -g
CFlags = -DDMAPPING $(CminorFlags)
LoadFlags = -lm
#LoadFlags = -lm -lg
HOME = ..

INCL = $(HOME)/neumap/include
INTFC = $(HOME)/neumap/interface
PART = $(HOME)/neumap/partition
SEARCH = $(HOME)/neumap/search
DSIM = $(HOME)/dsim
BIN = $(HOME)/bin

MajorHelpFiles = $(INCL)/limits.h $(INCL)/define.h $(INCL)/config.h $(INCL)/macro.h
HelpFiles = $(MajorHelpFiles) $(INCL)/var.h

ObjFiles = include.o interface.o partition.o search.o

DSimFiles = $(DSIM)/sim.h $(DSIM)/var.c $(DSIM)/main.c $(DSIM)/sim.c $(DSIM)/event.c\
                                        $(DSIM)/map.c


# main program
neumap: $(HelpFiles) $(ObjFiles)
        $(CC) $(CFlags) $(ObjFiles) $(LoadFlags) -o neumap

dsim: $(DSimFiles)
        touch .dsim.c; rm .dsim.c; cat $(DSimFiles) > .dsim.c;\
        $(CC) $(CFlags) .dsim.c $(LoadFlags) -o dsim

gmax: $(DSimFiles)
        touch .sim.c; rm .sim.c; cat $(DSimFiles) > .sim.c;\
        $(CC) -DGMAX $(CminorFlags) .sim.c $(LoadFlags) -o gmax
```

```
sim: $(DSimFiles)
        touch .sim.c; rm .sim.c; cat $(DSimFiles) > .sim.c;\
        $(CC) $(CminorFlags) .sim.c $(LoadFlags) -o sim

all: $(HelpFiles) $(ObjFiles) $(DSimFiles)
        $(CC) $(CFlags) $(ObjFiles) $(LoadFlags) -o neumap; \
        touch .dsim.c; rm .dsim.c; cat $(DSimFiles) > .dsim.c;\
        $(CC) $(CFlags) .dsim.c $(LoadFlags) -o dsim

print:
        cd $(HOME); PRINT;

clean:
        rm -f $(BIN)/*.o $(BIN)/core $(BIN)/.*.c $(BIN)/neumap

noso:
        rm -f $(BIN)/.*.c

# include/
include.o: $(MajorHelpFiles) $(INCL)/var.c
        touch .include.c; rm .include.c;\
        cat $(MajorHelpFiles) $(INCL)/var.c > .include.c;\
        $(CC) $(CFlags) -c .include.c -o include.o;

# interface/
interface.o: $(HelpFiles) $(INTFC)/*.c
        touch .interface.c; rm .interface.c;\
        cat $(HelpFiles) $(INTFC)/*.c > .interface.c;\
        $(CC) $(CFlags) -c .interface.c -o interface.o

# partition/
partition.o: $(HelpFiles) $(PART)/*.c
        touch .partition.c; rm .partition.c;\
        cat $(HelpFiles) $(PART)/*.c > .partition.c;\
        $(CC) $(CFlags) -c .partition.c -o partition.o

# search/
search.o: $(HelpFiles) $(SEARCH)/*.c
        touch .search.c; rm .search.c;\
        cat $(HelpFiles) $(SEARCH)/*.c > .search.c;\
        $(CC) $(CFlags) -c .search.c -o search.o
```

INCLUDE Directory

---

include/limits.h

```
#ifndef _neumap_limits_h_
#define _neumap_limits_h_
#include      <limits.h>
#define       HUGE_SHORT      SHRT_MAX        /** 32767 **/
#define       HUGE_INT        INT_MAX         /** 2147483647 **/
#define       HUGE_LONG       LONG_MAX        /** 2147483647 **/
#define       HUGE_FLOAT      FLT_MAX         /** 3.40282346638528860e+38 **/
#define       HUGE_DOUBLE     DBL_MAX         /** 1.7976931348623157e+308 **/
#define       HUGE_DEPTH      HUGE_INT
#endif _neumap_limits_h_
```

---

include/define.h

```
/** Global constants and type declarations are in this file.
** Some of them are quite self-explanatory, but some are not.
** For those not, descriptions are added. **/

#ifndef __define_h_
#define __define_h_
#include      <stdio.h>
#include      <ctype.h>
#include      <math.h>
#include      <sys/types.h>
#include      <sys/times.h>

#definefree(e)          /** Avoid strange deallocation routines in Unix. **/
#defineBUFFER_SIZE              256

typedef enum { NO = 0, YES = 1 }        yesno;
typedef enum { FALSE = 0, TRUE = 1 }    boolean;
typedef enum
{       /** Communication related constants **/
        BC,                     /** Broadcast **/
        PP,                     /** Point-to-point **/
        PCOMM,                  /** Production-phase communication **/
        LCOMM                   /** Learning-phase communication **/
} comm;

typedef enum
{       /** Kinds of neural clusters **/
        HIDDEN_NEURON, /** Hidden units **/
        INPUT_NEURON,   /** Input neural clusters **/
        OUTPUT_NEURON,/** Output neural clusters **/
        IO_NEURON       /** Input/output neural clusters **/
} neutype;

typedef enum
{       /** Kinds of nodes **/
        GOAL_NODE,              /** Goal node **/
```

```
        ROOT_NODE,              /** Root node **/
        PROUT_NODE,             /** Production-phase routing node **/
        LROUT_NODE,             /** Learning-phase routing node **/
        TERM_ASGN_NODE,         /** Terminal assignment node **/
        NTERM_ASGN_NODE         /** Nonterminal assignment node **/
} nodtype;

struct _route__
{       /** Type of route **/
        comm            type;   /** Broadcast and point-to-point **/
        float           error;  /** Error degree **/
        struct _route__ *next;
};
typedef struct _route__                 route;

typedef struct
{       /** Data type for processors **/
        float           t;              /** Computation power **/
        int             mem_size;       /** Local memory size **/
        int             kappa;          /** Overlap feature **/
        float           tkappa;         /** Overhead of overlap **/
        yesno           io;             /** IO feature **/
        int             partitnID;      /** Partition membership **/
} process;

typedef struct
{       /** Data type for links **/
        float           ts, tx;         /** Setup and transmission times **/
        int             idx;            /** Traffic index **/
        int             *sup;           /** Support set **/
        int             nsup;           /** Number of supports **/
        int             chID;           /** Channel ID **/
} link;

typedef struct
{       /** Data type for broadcasts **/
        float           ts, tx;  /** Setup and transmission timess **/
        int             from, to;       /** Range **/
} beast;

typedef struct
{       /** Data type for ANN neural clusters **/
        int             layerID;        /** Layer ID this cluster resides **/
        int             from, to;       /** Range **/
        int             n;              /** Number of neurons, n=to-from+1 **/
        float           pcunit;         /** Units of production-phase computation **/
        float           prunit;         /** Units of production-phase routing **/
        float           lcunit;         /** Units of learning-phase computation **/
        float           lrunit;         /** Units of learning-phase routing **/
        neutype         type;           /** Type of neuron cluster **/
        int             npred, *pred;   /** Predecessors and its number **/
        int             nsucc, *succ;   /** Successors and its number **/
} cluster;
```



```
typedef struct
{       /** Data type for partitions **/
        float           t;              /** Computation power **/
        int             mem_size;       /** Local memory size **/
        int             kappa;          /** Overlap feature **/
        float           tkappa;         /** Overhead of overlap **/
        yesno           io;             /** IO feature **/
        int             n;              /** Size of this partition **/
        int             *member;        /** Member set of this partition **/
} partitn;

typedef struct
{       /** Data type for channels (connecting partitions) **/
        float           ts, tx;         /** Setup and transmission times **/
        int             *sup;           /** Support set **/
        int             nsup;           /** Number of supports **/
        int             n;              /** Size of this channel **/
        int             *member;        /** Member set of this channel **/
} channel;

typedef struct
{       /** Data type for ANN layers **/
        int             n;              /** Number of members **/
        int             *member;        /** Member set of this layer **/
} layer;

struct _node__
{       /** Data type for search nodes **/
        struct _node__*parent, *next;
        float           g_cost;         /** Current actual cost **/
        float           lowb, upb;      /** Lower and upper bounds **/
        int             nk;             /** Cluster if it is an assignment node **/
        int             degree;         /** Branching degree **/
        int             nsprout;        /** Number of children **/
        int             depth;          /** Depth of this node **/
        nodtype         type;           /** Type of node **/
        float           *itime;         /** Array of initial times **/
        float           *ctime;         /** Array of completion times **/
        int             *assign;        /** Assignment vector **/
        route           *route;         /** Routing vector **/
};
typedef struct _node__                  node;

#endif __define_h_
```

---

include/macro.h

```
#ifndef __macro_h_
#define __macro_h_
```

```
#definestreql(x,y)      (! strcmp(x,y))         /** Check equality for two strings **/
#definexcluster(i)      (CLUSTER + i)           /** Array of ANN neural clusters **/
#definexlayer(i)            (LAYER + i)         /** Array of ANN layers **/
#definexprocess(i)      (PROCESS + i)           /** Array of processors **/
#definexpartitn(i)      (PARTITN + i)           /** Array of partitions **/
#definexbcast(i)            (BCAST + i)         /** Array of broadcasts **/
#definexlink(i)             (LINK + i)          /** Array of links **/
#definexchannel(i)      (CHANNEL + i)           /** Array of channels **/

/** Element (i,j) of this ANN interconnection matrix **/
#definexannicm(i,j)                 (ANN_ICM + (i * nCluster) + j)

/** Element (i,j) of this multicomputer interconnection matrix **/
#definexprocessicm(i,j)             (PROCESS_ICM + (i * nProcess) + j)

/** Element (i,j) of this multi-partition interconnection matrix **/
#definexpartitnicm(i,j)             (PARTITN_ICM + (i * nPartitn) + j)

/** Computation speed of this processor **/
#definexspeed(pid)                  (1.0 / xprocess(pid)->t)

/** Assignment of the second (cluster) onto the first (processor) **/
#definexasgnt(pid,nk)               (ASGNT + (pid * nCluster) + nk)

/** Completion time of this search node **/
#definecompletion_time(x)           (x->g_cost)

/** Production-phase computation time **/
#definepart_P_comp_time(n,nk,pk)  (n * xcluster(nk)->pcunit * xpartitn(pk)->t)

/** Learning-phase computation time **/
#definepart_L_comp_time(n,nk,pk)  (n * xcluster(nk)->lcunit * xpartitn(pk)->t)

/** Completion time **/
#defineproc_comp_time(nk,p)         (xcluster(nk)->n * xprocess(p)->t * (xcluster(nk)->pcunit + xcluster(nk)->lcunit))

#endif  __macro_h_
```

INTERFACE Directory

---

interface/main.c

```
main (argc, argv)
/** Enter NeuMap. **/
```



```
        int argc; char *argv[];
{       struct tms t_start, t_end;
        char *mimdfile = "mimd-in", *annfile = "ann-in";

        if (argc >= 2) mimdfile = argv[1];
        if (argc >= 3) annfile = argv[2];
        if (argc >= 4) sscanf (argv[3], "%f", &allowApprox);
        read_MIMD (mimdfile);
        read_ANN (annfile);

#ifdef STATISTICS
        times (&t_start);
#endif
        times (&tsim0);
        partitioning ();
#ifdef STATISTICS
        times (&t_end);
        partitionTime = t_end.tms_utime - t_start.tms_utime;
#endif
        set_mapping_approx ();

#ifdef STATISTICS
        times (&t_start);
#endif
        mapping ();

#if defined(STATISTICS) && defined(MONITOR)
        times (&t_end);
        mappingTime = t_end.tms_utime - t_start.tms_utime;
        printf ("\n\nMapping Profile\n");
        printf ("Partitioning Time = %d\n", partitionTime);
        printf ("Mapping Time = %d\n", mappingTime);
        printf ("Total Exec Time = %d\n", partitionTime + mappingTime);
#endif
}
```

---

interface/mimd.c

```
read_MIMD (filename)
/** Read in a multicomputer specification. **/
        char *filename;
{       FILE *input = fopen (filename, "r");

        getline (inputBuffer, BUFFER_SIZE, input);
        sscanf (inputBuffer, "%d %d %d", &nProcess, &nLink, &diameterMIMD);
        while (getline (inputBuffer, BUFFER_SIZE, input)) {
                sscanf (inputBuffer, "%s", patternTemp);
                if (streql (patternTemp, "startproc")) read_processors (input);
                else if (streql (patternTemp, "startadj")) read_adj_matrix (input);
```

```
            else if (streql (patternTemp, "startlink")) read_links (input);
            else if (streql (patternTemp, "startsupport")) read_supports (input);
            else if (streql (patternTemp, "startbc")) read_broadcasts (input);
            else if (streql (patternTemp, "HOMO")) homoMIMD = 1;
            else if (streql (patternTemp, "SYM")) symMIMD = 1;
        }
        fclose (input);
        if (homoMIMD && ! BCAST) error ("read_MIMD: 'HOMO' & 'BC' Are Supposed To Come Together");
}


read_processors (input)
/** Read in the specifications of processors. **/
        FILE *input;
{       process *p, *q; int nhomo, i;

        if (nProcess) PROCESS = (process *) malloc (nProcess * sizeof (process));
        else error ("read_processors: nProcess Undefined");
        p = PROCESS;
        while (getline (inputBuffer, BUFFER_SIZE, input)) {
                sscanf (inputBuffer, "%s", patternTemp);
                if (streql (patternTemp, "endproc")) break;
                if (streql (patternTemp, "homo")) sscanf (inputBuffer, "%s %d", patternTemp, &nhomo);
                else nhomo = 1;
                if (getline (inputBuffer, BUFFER_SIZE, input))
                        sscanf (inputBuffer, "%f %d %d %d %f", &(p->t),
                                &(p->mem_size), &(p->io), &(p->kappa), &(p->tkappa));
                else error ("read_processors: Incorrect Proc Format");
                for (i = 1; i < nhomo; i++) {
                        (q = p+i)->t = p->t;
                        q->mem_size = p->mem_size; q->io = p->io;
                        q->kappa = p->kappa; q->tkappa = p->tkappa;
                }
                p += nhomo;
        }
        if (p-PROCESS != nProcess) error ("read_processors: Full Processor Spec Are Required");
}


read_adj_matrix (input)
/** Read in a multicomputer interconnection matrix. **/
        FILE *input;
{       int i, j;

        PROCESS_ICM = (int *) malloc (nProcess * nProcess * sizeof (int));
        for (i = 0; i < nProcess; i++)
                for (j = 0; j < nProcess; j++) fscanf (input, "%d", xprocessicm(i,j));
        while (getline (inputBuffer, BUFFER_SIZE, input)) {
                sscanf (inputBuffer, "%s", patternTemp);
                if (streql (patternTemp, "endadj")) break;
        }
}
```



```
read_links (input)
/** Read in the specifications of links. **/
        FILE *input;
{       link *lp, *lq; int nhomo, i;

        if (nLink) LINK = (link *) malloc (nLink * sizeof (link));
        else error ("read_links: nLink Undefined");
        lp = LINK;
        while (getline (inputBuffer, BUFFER_SIZE, input)) {
                sscanf (inputBuffer, "%s", patternTemp);
                if (streql (patternTemp, "endlink")) break;
                if (streql (patternTemp, "homo")) sscanf (inputBuffer, "%s %d", patternTemp, &nhomo);
                else nhomo = 1;
                if (getline (inputBuffer, BUFFER_SIZE, input))
                        sscanf (inputBuffer, "%f %f", &(lp->ts), &(lp->tx));
                else error ("read_links: Incorrect Proc Format");
                for (i = 1; i < nhomo; i++) {
                        (lq = lp+i)->ts = lp->ts;
                        lq->tx = lp->tx;
                }
                lp += nhomo;
        }
        for (i = 0; i < nLink; i++) xlink(i)->chID = -1;
        if (lp-LINK != nLink) error ("read_links: Full Link Spec Are Required");
}


read_supports (input)
/** Read in supporting processors of links. **/
        FILE *input;
{       link *lp; int id, n, i, j;

        for (i = 0; i < nLink; i++) {
                fscanf (input, "%d %d", &id, &n);
                (lp = xlink(id))->nsup = n;
                lp->sup = (int *) malloc (n * sizeof(int));
                for (j = 0; j < n; j++) fscanf (input, "%d", (lp->sup+j));
        }
        skip_input_to (input, "endsupport", "read_supports: 'endsupport' Expected, but EOF");
}


read_broadcasts (input)
/** Read in the specifications of broadcasts in a multicomputer. **/
        FILE *input;
{       bcast *bc; int i;

        getline (inputBuffer, BUFFER_SIZE, input);
        sscanf (inputBuffer, "%d", &nBcast);
        BCAST = (bcast *) malloc (nBcast * sizeof(bcast));
        for (i = 0; i < nBcast; i++) {
                bc = xbcast(i);
                getline (inputBuffer, BUFFER_SIZE, input);
```

```
            sscanf (inputBuffer, "%d %d %f %f", &(bc->from), &(bc->to), &(bc->ts), &(bc->tx));
        }
        skip_input_to (input, "endbc", "read_broadcasts: Full Broadcast Spec Are Required");
}


skip_input_to (input, pattern, errmsg)
/** Check closing commands and skip unnecessary white spaces. **/
        FILE *input;
        char *pattern, *errmsg;
{
        do {    if (getline (inputBuffer, BUFFER_SIZE, input))
                        sscanf(inputBuffer, "%s", patternTemp);
                else error (errmsg);
        } while (! streql (patternTemp, pattern));
}
```

---

interface/ann.c

```
read_ANN (filename)
/** Read in an ANN specification. **/
        char *filename;
{       cluster *nc; int i, j, in, out, n, id, *member;
        FILE *input = fopen (filename, "r");

        /** Read in general attributes of an ANN. **/
        fullANN = hybridANN = 0;
        getline (inputBuffer, BUFFER_SIZE, input);
        sscanf (inputBuffer, "%s", patternTemp);
        if (streql (patternTemp, "FULL")) fullANN = 1;
        else if (streql (patternTemp, "HYBRID")) hybridANN = 1;
        else error ("read_ANN: 'FULL' Or 'HYBRID' Are Expected At The First Line");

        /** Read in the specifications of neural layers. **/
        fscanf (input, "%d %d", &nLayer, &nCluster);
        if (fullANN && nLayer != nCluster)
                error("read_ANN: 'FULL' Means # Of Layers Equals To # Of Clusters");
        LAYER = (layer *) malloc (nLayer * sizeof (layer));
        CLUSTER = (cluster *) malloc (nCluster * sizeof (cluster));
        for (i = 0; i < nLayer; i++) {
                fscanf (input, "%d %d", &id, &n);
                xlayer(id)->n = n;
                xlayer(id)->member = member = (int *) malloc (n * sizeof (int));
                for (j = 0; j < n; j++) {
                        fscanf (input, "%d", member+j);
                        xcluster(*(member+j))->layerID = id;
                }
        }
```



```
        /** Read in the specifications of neural clusters. **/
        for (i = 0; i < nCluster; i++) {
                fscanf (input, "%d", &id);
                nc = xcluster(id);
                fscanf (input, "%d %d", &(nc->from), &(nc->to));
                nNeuron += (nc->n = nc->to - nc->from + 1);

                fscanf (input, "%d %d", &in, &out);
                if (in && out) nc->type = IO_NEURON;
                else if (in) nc->type = INPUT_NEURON;
                else if (out) nc->type = OUTPUT_NEURON;
                else nc->type = HIDDEN_NEURON;

                fscanf (input, "%f %f", &(nc->pcunit), &(nc->prunit));
                fscanf (input, "%f %f", &(nc->lcunit), &(nc->lrunit));

                fscanf (input, "%d", &(nc->npred));
                if (nc->npred > 0) {
                        nc->pred = (int *) malloc (nc->npred * sizeof (int));
                        for (j = 0; j < nc->npred; j++) fscanf (input, "%d", (nc->pred+j));
                }
                else nc->pred = NULL;

                fscanf (input, "%d", &(nc->nsucc));
                if (nc->nsucc > 0) {
                        nc->succ = (int *) malloc (nc->nsucc * sizeof (int));
                        for (j = 0; j < nc->nsucc; j++) fscanf (input, "%d", (nc->succ+j));
                }
                else nc->succ = NULL;
        }
        fclose (input);
}
```

PARTITION Directory

---

partition/entry.c

```
partitioning ()
/** Partition a multicomputer and transform attributes.
 ** The algorithm of partitioning is shown in Chapter 6.
 ** Please refer to it for details. **/
{
#if defined(DEBUG) || defined(MONITOR)
        char *msg1 = "Partitioning processors ...\n";
```

```
        char *msg2 = "No virtual link due to single partition\n";
        char *msg3 = "Virtualizing Links ...\n";
#endif
#ifdef DEBUG
#ifndef MONITOR
        if (debugMode >= 1) printf ("%s", msg1);
#endif
#endif
#ifdef MONITOR
        printf ("%s", msg1);
#endif
        partition_processors ();
#ifdef DEBUG
        if (debugMode >= 1) {
#ifndef MONITOR
                printf ("%s", (nPartitn == 1) ? msg2 : msg3);
#endif
        }
#endif
#ifdef MONITOR
        printf ("%s", (nPartitn == 1) ? msg2 : msg3);
#endif
        virtualize_links ();    /** Find channels. **/
}
```

---

partition/partition.c

```
/** Temporary arrays for partitioning are declared here. **/
int     *_partitn_;             /** Processors in a partition **/
int     *_partitnSize_;         /** Sizes of partitions **/
float   *_compT_;               /** Minimum computation time in each processor **/
float   *_commT_;               /** Maximum communication time in each processor **/
float   *_minCompT_;            /** Minimum computation time in each partition **/
float   *_maxCommT_;            /** Maximum communication time in each partition **/


partition_processors ()
/** Partition a multicomputer. **/
{
        alloc_partition_associates ();
        find_partitions ();
        dealloc_partition_associates ();
        discover_number_of_partitions ();
        discover_partition_membership ();
        sort_partition_members ();
        construct_partition_attributes ();
        sort_partitions ();
        find_extreme_partition_attributes ();
}
```



```
alloc_partition_associates ()
/** Allocate temporary arrays for partitioning. **/
{       int i, j; float t;

        /** Reset partitions. **/
        _partitn_ = (int *) malloc (nProcess * sizeof (int));
        for (i = 0; i < nProcess; i++) *(_partitn_+i) = i;

        /** Alloc auxilary structures. **/
        _partitnSize_ = (int *) malloc (nProcess * sizeof (int));
        _compT_ = (float *) malloc (nProcess * sizeof (float));
        _commT_ = (float *) malloc (nLink * sizeof (float));
        _minCompT_ = (float *) malloc (nProcess * sizeof (float));
        _maxCommT_ = (float *) malloc (nProcess * sizeof (float));

        /** Find minimum completion time (per neuron). **/
        for (i = 0; i < nProcess; i++) {
                *(_partitnSize_+i) = 1;
                *(_compT_+i) = HUGE_FLOAT;
                for (j = 1; j < nCluster; j++) {
                /** enum all poss clusters **/
                        t = proc_comp_time(j,i);
                        /** min compT for clusters **/
                        if (t < *(_compT_+i)) *(_compT_+i) = t;
                }
                *(_minCompT_+i) = *(_compT_+i);
                *(_maxCommT_+i) = 0.0;
        }

        /** Find max communication time (per neuron). **/
        for (i = 0; i < nLink; i++) {
                *(_commT_+i) = 0.0;
                for (j = 1; j < nCluster; j++) {
                /** Enumerate all possible neural clusters. **/
                        t = (xlink(i)->ts + xlink(i)->tx * xcluster(j)->n) / xcluster(j)->n;
                        if (t > *(_commT_+i)) *(_commT_+i) = t;
                }
        }
}


find_partitions ()
/** Find partitions according to the algorithm of partitioning. **/
{       int i, j, k, link_idx, proc_idx, size, diameter;
        float tcomp, tcomp1, tcomp2, tcomm, err;

        /** Find partitions. **/
        for (i = 0; i < nProcess; i++) {
                for (proc_idx = 0; proc_idx < nProcess; proc_idx++) {
                        if ((link_idx = *xprocessicm(i,proc_idx)) >= 0) {
                                if (*(_partitn_+i) != *(_partitn_+proc_idx)) {
                                        tcomp1 = *(_minCompT_+i);
                                        tcomp2 = *(_minCompT_+proc_idx);
```

```
                        tcomp = (tcomp1 < tcomp2) ? tcomp1 : tcomp2;
                        tcomm = *(_maxCommT_+i) + *(_maxCommT_+proc_idx)
                                + *(_commT_+link_idx);
                        size = *(_partitnSize_+i) + *(_partitnSize_+proc_idx);
                        diameter = (diameterMIMD <= size-1) ? diameterMIMD : size-1;
                        err = tcomm * diameter * size;
                        if (err / tcomp <= allowApprox) {
                                if (err > timeError)
                                /** Find max error. **/
                                        timeError = err;
                                if (err / tcomp > partApprox)
                                        partApprox = err / tcomp;
                                for (k = 0; k < nProcess; k++) {
                                        if (*(_partitn_+k) == *(_partitn_+proc_idx)
                                        || *(_partitn_+k) == *(_partitn_+i)) {
                                                *(_partitn_+k) = *(_partitn_+i);
                                                *(_minCompT_+k) = tcomp;
                                                *(_maxCommT_+k) = tcomm;
                                                *(_partitnSize_+k) = size;
                                        }
                                }
                        }
                }
        }
}


dealloc_partition_associates ()
/** Release temporary arrays. **/
{ free (_compT_); free (_commT_); free (_minCompT_); free (_maxCommT_); }


discover_number_of_partitions ()
/** Discover the number of partitions. **/
{       int i, j, size, *id = (int *) malloc (nProcess * sizeof (int));

        for (i = 0; i < nProcess; i++) *(id+i) = -1;
        for (i = 0; i < nProcess; i++) *(id + *(_partitn_+i)) = 1;
        for (i = 0; i < nProcess; i++)
        if (*(id+i) == 1) {
                *(id+i) = nPartitn++;
                for (size = j = 0; j < nProcess; j++)
                if (*(_partitn_+j) == i) size++;
                *(_partitnSize_+ *(id+i)) = size;
        }
        for (i = 0; i < nProcess; i++)
        xprocess(i)->partitnID = *(id + *(_partitn_+i));
        free (id); free (_partitn_);
}
```



```
discover_partition_membership ()
/** Discover members of each partition. **/
{       int i, j, index;
        partitn *pk;

        PARTITN = (partitn *) malloc (nPartitn * sizeof (partitn));
        for (i = 0; i < nPartitn; i++) xpartitn(i)->n = *(_partitnSize_+i);
        for (i = 0; i < nPartitn; i++) {
                (pk = xpartitn(i))->member = (int *) malloc (nProcess * sizeof (int));
                for (index = j = 0; j < nProcess; j++)
                        if (xprocess(j)->partitnID == i) *(pk->member + index++) = j;
                if (pk->n != index) error ("discover_partitn_membership");
        }
        free (_partitnSize_);
}


sort_partition_members ()
/** Sort members inside each partition according to computation power. **/
{       int i, j, k, n, who, *member, itmp; float t, tmin;

        for (i = 0; i < nPartitn; i++) {
                n = xpartitn(i)->n;
                member = xpartitn(i)->member;
                for (j = 0; j < n; j++) {
                        tmin = xprocess(*(member+j))->t;
                        who = j;
                        for (k = j+1; k < n; k++) {
                                t = xprocess(*(member+k))->t;
                                if (tmin > t) { tmin = t; who = k; }
                        }
                        if (who != j) {
                                itmp = *(member+j);
                                *(member+j) = *(member+who);
                                *(member+who) = itmp;
                        }
                }
        }
}


construct_partition_attributes ()
/** Transform attributes of processors into those of partitions. **/
{       int i, j, n; process *p; partitn *pk;

        for (i = 0; i < nPartitn; i++) {
                n = (pk = xpartitn(i))->n;
                pk->t = 0.0; pk->mem_size = 0; pk->kappa = 1; pk->tkappa = 0.0;
                for (j = 0; j < n; j++) {
                        p = xprocess(*(pk->member+j));
                        pk->t += (1.0 / p->t); pk->mem_size += p->mem_size;
                        pk->kappa = pk->kappa && p->kappa; pk->tkappa += p->tkappa;
                        if (p->io) pk->io = 1;
```

```
            }
            pk->t = 1.0 / pk->t; pk->tkappa /= ((float) n);
        }
}

sort_partitions ()
/** Sort partitions according to computation power. **/
{       int i, j, who; float t, tmin;
        partitn *pk, *pktemp = (partitn *) malloc (sizeof (partitn));

        for (i = 0; i < nPartitn; i++) {
                tmin = (pk = xpartitn(i))->t; who = i;
                for (j = i+1; j < nPartitn; j++) {
                        t = xpartitn(j)->t;
                        if (tmin > t) { tmin = t; who = j; }
                }
                if (who != i) {
                        *pktemp = *(xpartitn(i));
                        *(xpartitn(i)) = *(xpartitn(who));
                        *(xpartitn(who)) = *pktemp;
                }
        }
        free (pktemp);
}


find_extreme_partition_attributes ()
/** Find the best and worst attributes. **/
{       int i;

        for (i = 0; i < nPartitn; i++) {
                if (xpartitn(i)->t < bestPartitnT) bestPartitnT = xpartitn(i)->t;
                if (xpartitn(i)->t > worstPartitnT) worstPartitnT = xpartitn(i)->t;
        }
}
```

---

partition/channel.c

```
virtualize_links ()
/** Transform links into channels. **/
{
        if (nPartitn == 1) { nChannel = 0; CHANNEL = NULL; return; }
        CHANNEL = (channel *) malloc (nLink * sizeof (channel));
        discover_partition_supports ();
        discover_partition_adj_matrix ();
        discover_channel_membership ();
        sort_channel_members ();
        construct_channel_attributes ();
        find_extreme_channel_attributes ();
}


discover_partition_supports ()
/** Discover supporting partitions of each link. **/
{       int i, j, k, nsup, *sup, diff, n_part_sup; channel *ch;
        int *part_sup = (int *) malloc (nPartitn * sizeof (int));

        for (i = 0; i < nLink; i++) {
                nsup = xlink(i)->nsup; sup = xlink(i)->sup;

                /** Reset supporting partitions. **/
                for (j = 0; j < nPartitn; j++) *(part_sup+j) = 0;

                /** Set supporting partitions. **/
                for (j = 0; j < nsup; j++) *(part_sup + xprocess(*(sup+j))->partitnID) = 1;

                /** Calculate the number of partitions connected by this link. **/
                for (n_part_sup = j = 0; j < nPartitn; j++) if (*(part_sup+j)) n_part_sup++;
                if (n_part_sup == 1) continue;

                /** Check difference between this channel and existing channels. **/
                for (diff = 1, j = 0; j < nChannel && diff; j++)
                        if (n_part_sup == (ch = xchannel(j))->nsup) {
                                for (k = 0; k < n_part_sup; k++)
                                        if (! *(part_sup + *(ch->sup+k)))
                                        /** Force to exit immediately. **/
                                                k = n_part_sup+1;
                                if (k == n_part_sup) diff = 0;
                        }

                /** This channel is different from all existing channels. **/
                if (diff) {
                        xlink(i)->chID = nChannel;
                        (ch = xchannel(nChannel++))->nsup = n_part_sup;
                        ch->sup = (int *) malloc (n_part_sup * sizeof (int));
                        for (k = j = 0; j < n_part_sup; j++) {
                                while(! *(part_sup+k)) k++;
                                *(ch->sup+j) = k++;
                        }
                }
                /** j-1 is necessary because j was increased by one. **/
                else xlink(i)->chID = j-1;
        }
}


discover_partition_adj_matrix ()
/** Discover a multi-partition adjacent matrix. **/
{       int i, j, k, nsup; partitn *pk; channel *ch;

        PARTITN_ICM = (int *) malloc (nPartitn * nPartitn * sizeof (int));
```

```
        for (i = 0; i < nPartitn; i++) for (j = 0; j < nPartitn; j++) *(xpartitnicm(i,j)) = -1;

        /** Calculate partition ICM. **/
        for (i = 0; i < nChannel; i++) {
                nsup = (ch = xchannel(i))->nsup;
                for (j = 0; j < nsup; j++)
                        for (k = 0; k < nsup; k++)
                                if (j != k) *(xpartitnicm(*(ch->sup+j),*(ch->sup+k))) = i;
        }
}


discover_channel_membership ()
/** Discover members of each channel. **/
{       int i, j, index; channel *ch;

        for (i = 0; i < nChannel; i++) {
                (ch = xchannel(i))->n = 0;
                for (j = 0; j < nLink; j++) if (xlink(j)->chID == i) (ch->n)++;
                ch->member = (int *) malloc (ch->n * sizeof (int));
                for (index = j = 0; j < nLink; j++) if(xlink(j)->chID == i) *(ch->member + index++) = j;
                if (ch->n != index) error ("discover_channel_membership");
        }
}


sort_channel_members ()
/** Sort members inside each channel according to communication power. **/
{       int i, j, k, n, who, itmp, *member; float tmin, t; channel *ch;

        for (i = 0; i < nChannel; i++) {
                n = (ch = xchannel(i))->n; member = ch->member;
                for (j = 0; j < n; j++) {
                        tmin = xlink(*(member+j))->tx; who = j;
                        for (k = j+1; k < n; k++) {
                                t = xlink(*(member+k))->tx;
                                if (tmin > t) { tmin = t; who = k; }
                        }
                        if (who != j) {
                                itmp = *(member+j);
                                *(member+j) = *(member+who);
                                *(member+who) = itmp;
                        }
                }
        }
}


construct_channel_attributes ()
/** Transform attributes of links into those of channels. **/
{       int i, j, n, *member; channel *ch;

        for (i = 0; i < nChannel; i++) {
```



```
                n = (ch = xchannel(i))->n; member = ch->member;
                ch->ts = ch->tx = 0.0;
                for (j = 0; j < n; j++) {
                        ch->ts += xlink(*(member+j))->ts;
                        ch->tx += (1.0 / xlink(*(member+j))->tx);
                }
                ch->ts /= ((float) n); ch->tx = 1.0 / ch->tx;
        }
}


find_extreme_channel_attributes ()
/** Find the best and worst attributes of channels. **/
{       int i; channel *ch;

        for (i = 0; i < nChannel; i++) {
                ch = xchannel(i);
                if (ch->ts < bestChTs) bestChTs = ch->ts;
                if (ch->tx < bestChTx) bestChTx = ch->tx;
                if (ch->ts > worstChTs) worstChTs = ch->ts;
                if (ch->tx > worstChTx) worstChTx = ch->tx;
        }
}


yesno is_partitn_support (partitn_id, ch_id)
/** Check if the second (channel) is supported by the first (partition). **/
        int partitn_id, ch_id;
{       int i, n; channel *ch = xchannel(ch_id);

        for (i = 0; i < ch->nsup; i++) if (partitn_id == *(ch->sup+i)) return YES;
        return NO;
}
```

SEARCH Directory

---

search/abc.c

```
/** This file contains temporary arrays for the search solver. **/
int     *inTempArray;           /** Combination **/
int     *outTempArray;          /** Combination **/
int     *ptrTempArray;          /** Combination **/
float   *timeTempArray;         /** Time related buffers **/
float   *tauTempArray;          /** Time related bin-packing distribution **/
```

```
float   *bTimeTempArray;        /** Time related bounding **/
float   *bTauTempArray;         /** Time related bounding **/
int     *sourceTempArray;       /** Routing **/
int     *destTempArray;         /** Routing **/
int     *diffTempArray;         /** Routing **/
int     *bcTempArray;           /** Broadcast routing **/
int     **dataTempMatrix;       /** Routing **/
#define xdata(i,j)      *(*(dataTempMatrix + i) + j) /** Data matrix **/

/** Averages of parameters of channels **/
float   avgChTs;                /** Average setup time of channels **/
float   avgChTx;                /** Average transmission time of channels **/

/** Bottleneck of parameters of channels **/
float   bottleChTs;
float   bottleChTx;
int     bottleCh;

/** Current broadcast attributes **/
float   bcTs = 0.0;             /** Broadcast transmission time **/
float   bcTx = 0.0;             /** Broadcast setup time **/
```

---

search/entry.c

```
mapping ()
/** Enter the mapping solver. **/
{
#if defined(DEBUG) || defined(MONITOR)
        char *msg1 = "initializing search environment ...\n";
        char *msg2 = "searching optimal mapping ...\n";
#endif
#ifdef DEBUG
#ifndef MONITOR
        if (debugMode >= 1) printf ("%s", msg1);
#endif
#endif
#ifdef MONITOR
        printf ("%s", msg1);
#endif
        init_search ();

#ifdef DEBUG
#ifndef MONITOR
        if (debugMode >= 1) printf ("%s", msg2);
#endif
#endif
#ifdef MONITOR
        printf ("%s", msg2);
#endif
```



```
        search ();

#if defined(STATISTICS) && defined(MONITOR)
        printf ("\n\nSearch Statistics:\n");
        printf ("Number of Nodes Generated = %d\n", nGenerated);
        printf ("Number of Nodes Feasible = %d\n", nFeasible);
        printf ("Number of Nodes Infeasible = %d\n", nInfeasible);
        printf ("Number of Nodes Pruned = %d\n", nPruned);
#endif
}


root_generator ()
/** Generate a root node for the search tree. */
{
        rootNode = allocate_node (NULL, 0.0, -1, -1, ROOT_NODE);
        rootNode->lowb = evaluate_lower_bound (rootNode);
        rootNode->upb = evaluate_upper_bound (rootNode);
        insert (rootNode);
}


init_search ()
/** Initialize search environment. **/
{       int i, j;

        /** Temporary arrays for combination **/
        inTempArray = (int *) malloc (nProcess * sizeof (int));
        outTempArray = (int *) malloc (nProcess * sizeof (int));
        ptrTempArray = (int *) malloc (nProcess * sizeof (int));
        for (i = 0; i < nPartitn; i++) *(inTempArray+i) = i;

        /** Temporary arrays for assignment and routing nodes **/
        timeTempArray = (float *) malloc (nProcess * sizeof (float));
        tauTempArray = (float *) malloc (nProcess * sizeof (float));

        /** Temporary arrays for bounding **/
        bTimeTempArray = (float *) malloc (nProcess * sizeof (float));
        bTauTempArray = (float *) malloc (nProcess * sizeof (float));

        /** Temporary arrays for routing **/
        sourceTempArray = (int *) malloc (nProcess * sizeof (int));
        destTempArray = (int *) malloc (nProcess * sizeof (int));
        diffTempArray = (int *) malloc (nProcess * sizeof (int));
        dataTempMatrix = (int **) malloc (nProcess * sizeof (int *));
        for (i = 0; i < nPartitn; i++) *(dataTempMatrix+i) = (int *) malloc (nProcess * sizeof (int));

        /** Temporary arrays for broadcasting **/
        bcTempArray = (nBcast) ? (int *) malloc (nBcast * sizeof (int)) : NULL;
        avgChTs = avgChTx = 0.0;
        if (nChannel) {
                for (i = 0; i < nChannel; i++) {
                        avgChTs += xchannel(i)->ts;
```

```
                avgChTx += xchannel(i)->tx;
        }
        avgChTs /= ((float) nChannel);
        avgChTx /= ((float) nChannel);

        bottleChTs = xchannel(i)->ts;
        bottleChTx = xchannel(i)->tx;
        bottleCh = 0;
        for (i = 1; i < nChannel; i++)
                if (bottleChTx < xchannel(i)->tx) {
                        bottleCh = i;
                        bottleChTx = xchannel(i)->tx;
                        bottleChTs = xchannel(i)->ts;
                }
    }

    /** Reset the assignment matrix. **/
    ASGNT = (int *) malloc (nProcess * nCluster * sizeof (int));
    for (i = 0; i < nProcess; i++) for (j = 0; j < nCluster; j++) *(xasgnt(i,j)) = 0;

    heap_init ();
}


yesno is_infeasible (x)
/** Check infeasibility of a search node. **/
    node *x;
{
    if (x->type == NTERM_ASGN_NODE && x->degree != -1 && x->degree < 2) return YES;
    return NO;
}


yesno is_feasible (x)
/** Check feasibility of a search node. **/
    node *x;
{ return ((x->type == GOAL_NODE) ? YES : NO); }


float evaluate_cost (p)
/** Evaluate the cost of a search node. **/
    node *p;
{ return p->g_cost; }


#define clear_stack()       sp = -1;
#define push(p)             *(stack + ++sp) = p
#define pop()               (*(stack + sp--))
#define empty_stack()       (sp < 0)

evaluate_solution (fp, x)
/** Given a goal node, trace back up to the root and
 ** push all search nodes onto a stack along the path
```



```
** from this goal node to the root node.
** Eventually, the goal node is at the bottom and
** the root node is at the top. Then, pop node by node
** from the stack to construct a mapping scheme. **/
    FILE *fp; node *x;
{   int i, sp, count = 1, amount; node **stack, *p; float *bp_time;
    FILE *fpsim, *fpmachine = fopen ("map-machine", "w");
#ifdef MONITOR
    float time;
#endif

    if (x == NULL) error ("strange! no solution");
    fpsim = fopen ("map-sim", "w");
    fprintf (fpsim, "%f %f\n", completion_time (x), ((float) (tsim1.tms_utime - tsim0.tms_utime)));

    /** Initialize processors for intra-partition even distribution. **/
    bp_time = (float *) malloc (nProcess * sizeof (float));
    for (i = 0; i < nProcess; i++) *(bp_time+i) = 0.0;

    /** Calculate the number of terminal assignment nodes. **/
    for (p = x; p; p = p->parent) count++;
    if (count <= 1) error ("Strange ! No Solution ? Internal Error !?");

    /** Push all search nodes onto the stack along the path
     ** from this goal node to the root node. **/
    stack = (node **) malloc (count * sizeof (node *));
    clear_stack ();
    for (p = x ; p; p = p->parent) push (p);

    /** Pop node by node to construct a mapping scheme. **/
    while (! empty_stack ()) {
        switch ((p = pop ())->type) {
            case TERM_ASGN_NODE:
                detailed_assign (p, bp_time);
#ifdef MONITOR
                fprintf (fp, "For Cluster %d\n", p->nk);
                fprintf (fp, "Partition-Level Assignment: ");
                fprintf (fp, "{partitn,#neuron}\n");
                for (i = 0; i < nPartitn; i++) fprintf (fp, "{%d,%d}  ", i, *(p->assign+i));
                fprintf (fp, "\n");
                fprintf (fp, "Processor-Level Assignment: ");
                fprintf (fp, "{process,#neuron}\n");
                for (i = 0; i < nProcess; i++) fprintf (fp, "{%d,%d}  ", i, *(xasgnt(i,p->nk)));
                fprintf (fp, "\n");
                fprintf (fp, "node = %d, ctime = %g\n", p, max_float (p->ctime, nPartitn));
#endif
                for (i = 0; i < nProcess; i++) {
                    fprintf (fpmachine, "machine=%d, cluster=%d => ", i, p->nk);
                    fprintf (fpsim, "%d %d  ", i, p->nk);
                    amount = *(xasgnt(i,p->nk));
                    fprintf (fpmachine, "%d\n", amount);
                    fprintf (fpsim, "%f %f %f %f\n",
                                amount * xcluster(p->nk)->pcunit,
```

```
                                        amount * xcluster(p->nk)->prunit,
                                        amount * xcluster(p->nk)->lcunit,
                                        amount * xcluster(p->nk)->lrunit);
                        }
                        break;
                case PROUT_NODE:
#ifdef MONITOR
                        if (p->route) timeError += p->route->error;
                        fprintf (fp, "Routing ...\n");
                        fprintf (fp, "node = %d, ctime = %g\n", p, max_float (p->ctime, nPartitn));
                        break;
                case LROUT_NODE:
                        if (p->route) timeError += p->route->error;
                        fprintf (fp, "Learning+Routing ...\n");
                        fprintf (fp, "node = %d, ctime = %g\n", p, max_float (p->ctime, nPartitn));
                        break;
#endif
                default: break;
            }
        }
        fclose (fpsim); fclose (fpmachine);

#ifdef MONITOR
        time = completion_time (x);
        fprintf (fp, "OPTIMAL MAPPING\n");
        fprintf (fp, "Min Completion Time = %g\n", time);
        fprintf (fp, "With Error = %g\n", timeError / time);
        fprintf (fp, "Approx Allowed = %g\n", allowApprox);
        fprintf (fp, "Approx Partition Introduced = %g\n", partApprox);
        fprintf (fp, "Approx Allocated for Mapping = %g\n", mappingApprox);
#endif
        free (bp_time); free (stack);
}

#undef clear_stack()
#undef push(p)
#undef pop()
#undef empty_stack()


set_mapping_approx ()
/** Calculate an approximation degree for the mapping solver
 ** based on user-specified error allowance and error incurred
 ** in partitioning. **/
{ mappingApprox = (allowApprox - partApprox) / (1.0 + partApprox); }


enter_pruned (x)
/** Mark this search node as pruned. **/
        node *x;
{
#ifdef STATISTICS
        nPruned++;
```



```
#endif
        free_node(x);
}


yesno is_dominated (x) node *x; { return NO; }
/** Check dominance of this search node. **/


detailed_assign (x, bp_time)
/** Solve a detailed mapping scheme at processor level
 ** rather than at partition level **/
        node *x; float bp_time[];
{       int pk;

        for (pk = 0; pk < nPartitn; pk++)
                if (*(x->assign+pk) > 0) detailed_partitn_assign (pk, x->nk, *(x->assign+pk), bp_time);
}


detailed_partitn_assign (pk, nk, amount, bp_time)
/** Auxiliary routine for detailed_assign above,
 ** Even distribution of neurons over processors inside a partition
 ** according to computation power. **/
        int pk, nk, amount;
        float bp_time[];
{       int n = xpartitn(pk)->n, *member = xpartitn(pk)->member;
        int i, diff, pid;
        float ratio = 0.0, inverse = 0.0, num, portion;
        float unit = xcluster(nk)->pcunit + xcluster(nk)->lcunit;

        for (i = 0; i < n; i++) {
                pid = *(member+i);
                *(tauTempArray+pid) = xprocess(pid)->t * unit;
        }

        for (i = 0; i < n; i++) {
                pid = *(member+i);
                ratio += (bp_time[pid] / *(tauTempArray+pid));
                inverse += (1.0 / *(tauTempArray+pid));
        }
        num = (amount + ratio) / inverse;

        for (diff = amount, i = 0; i < n; i++) {
                pid = *(member+i);
                portion = (num - bp_time[pid]) / *(tauTempArray+pid);
                diff -= (*(xasgnt(*(member+i),nk)) = ((int) portion));
        }

        /** make up roundoff error **/
        while (diff >= n)
                for (i = 0; i < n; i++, diff--) (*(xasgnt(*(member+i),nk)))++;
        for (i = 0; i < diff; i++) {
```

```
            pid = *(member+i);
            (*(xasgnt(pid,nk)))++;
            bp_time[pid] += (xprocess(pid)->t * unit);
        }
}
```

search/search.c

```
search ()
/* This routine is actually a branch-and-bound algorithm. */
{       node *x; float cost;
#ifdef MONITOR
#define MONITOR_COUNT 100
        int monitor_count = 0;
#endif
        root_generator ();
        for (x = delete (); x; x = delete ()) {
#ifdef MONITOR
            if (++monitor_count >= MONITOR_COUNT) {
                printf (".");
                monitor_count = 0;
            }
#endif
            if (optimalSolution && x->lowb > gUpperBound || is_dominated (x)) {
#ifdef STATISTICS
                ++nPruned;
#endif
                free_node (x);
            } else

            if (is_feasible (x)) {
#ifdef STATISTICS
                ++nFeasible;
#endif
                cost = evaluate_cost (x);
                if (cost < completionTime) {
                    gUpperBound = completionTime = cost;
                    if (optimalSolution) free_node (optimalSolution);
                    optimalSolution = x;
                }
                else free_node (x);
            } else

            if (is_infeasible (x)) {
#ifdef STATISTICS
                ++nInfeasible;
#endif
                free_node (x);
            } else split (x);
```



```
        }
#ifdef MONITOR
        printf("\n");
#endif
        times (&tsim1);
        evaluate_solution (stdout, optimalSolution);
}
```

search/split.c

```
split (x)
/* Split a search node into several child nodes. */
        node *x;
{       node *p; nodtype type = x->type;
        int nk = x->nk, l, l1, i; float t;

        if (type == PROUT_NODE || type == LROUT_NODE) l1 = (l = nk) + 1;
        else {
                l = xcluster(nk)->layerID;
                if (nk >= nCluster - 1) l1 = l; else l1 = xcluster(nk+1)->layerID;
        }

        switch (type) {
                case GOAL_NODE:
                        error ("split: impossible GOAL_NODE here");
                        break;

                case ROOT_NODE: /** Sprout from a root node. **/
                        for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = 0.0;
                        sprout (x);
                        break;

                case PROUT_NODE: /** Find the next assignment. **/
                        if (l < nLayer - 2) {
                                for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(x->ctime+i);

                                /** interface to next assign node **/
                                for (x->nk = -1, i = 0; i <= l+1; i++) x->nk += xlayer(i)->n;
                                sprout (x);
                                x->nk = l; /** Restore from initialization. **/
                        } else {
                                x->nk = nLayer;
                                for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(x->ctime+i);
                                L_routing (x);
                        }
                        break;

                case LROUT_NODE: /** Solve learning-phase routing. **/
                        if (l > 1) {
```

```
                for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(x->ctime+i);
                L_routing (x);
        } else {
                /** l <= 1 **/
                t = max_float (x->ctime, nPartitn);
                p = allocate_node (x, t, 0, -1, GOAL_NODE);
                L_comp (p);
                p->g_cost = max_float (p->ctime, nPartitn);
                establish_new (p);
        }
        break;

case TERM_ASGN_NODE:
        if ((nk == 0 && fullANN) || (l1 <= 1 && hybridANN)) {
        /** This case is sprouting only and no routing necessary.
         ** For a fully-connected ANN, it is a terminal
         ** assignment node with neural cluster 0.
         ** For a hybrid multilayer ANN, it is a neural cluster
         ** within the first 2 layers. **/
                for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(x->ctime+i);
                sprout (x);
        } else
        if ((nk == nCluster-1) || (l >= 1 && l1 == l+1)) {
        /** Regardless of fully-connected ANN or hybrid
         ** multilayer ANN, this case first occurs at layer 1.
         ** The production-phase routing is from layer 0 to
         ** layer 1. **/
                P_routing (x);
        } else {
                for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(x->ctime+i);
                sprout (x);
        }
        break;

case NTERM_ASGN_NODE:
        /* This situation is same for all kinds of ANNs. */
        for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(x->itime+i);
        expand (x);
        break;

default: /** This is an impossible case. **/
        error ("split: no such split");
        break;
    }
}
```

search/expand.c

```
expand (x)
```



```
/** Expand a node from a nonterminal assignment node (NTERM_ASGN_NODE).
 ** Children are different combinations of assignments from this node
 ** as well as a NTERM_ASGN_NODE child. **/
        node *x;
{       node *new; int i, first_time, degree = x->degree, nk = x->nk;

        if (degree < 2) {       /** This NTERM_ASGN_NODE node is closed. **/
#ifdef STATISTICS
                ++nInfeasible;
#endif
                free_node (x);
                return;
        }

        /** Sprout (subset) full assignment children. **/
        first_time = 1;
        for (i = 0; i < nPartitn; i++) *(inTempArray+i) = i;
        while (get_comb (degree, nPartitn, first_time) == YES) {
                first_time = 0;
                new = allocate_node (x, 0.0, nk, -1, TERM_ASGN_NODE);
                bp_assign (nk, outTempArray, degree, new);
                establish_new (new);
        }

        /** Sprout a nonterminal assignment node. **/
        new = allocate_node (x, x->g_cost, nk, degree-1, NTERM_ASGN_NODE);
        establish_new (new);
        (x->nsprout)++;
        for (i = 0; i < nPartitn; i++) *(new->ctime+i) = *(new->itime+i) = *(timeTempArray+i);
}


sprout (x)
/** A new assignment node is going to sprout its children. It sprouts
 ** a number of TERM_ASGN_NODEs and a NON_TERM_ASGN_NODE node. **/
        node *x;
{       node *new; int nk = x->nk+1, i, j;

        x->nsprout = nPartitn + 2;

        /** Sprout singular children. **/
        for (i = 0; i < nPartitn; i++) {
                new = allocate_node (x, 0.0, nk, -1, TERM_ASGN_NODE);
                for (j = 0; j < nPartitn; j++)
                        if (i == j) *(new->assign+j) = xcluster(nk)->n; else *(new->assign+j) = 0;
                for (j = 0; j < nPartitn; j++) *(new->ctime+j) = *(new->itime+j) = *(timeTempArray+j);
                assign_comp_time (new, nk);
                new->g_cost = max_float (new->ctime, nPartitn);
                establish_new (new);
        }

        /** Sprout full-assignment children. **/
        new = allocate_node (x, 0.0, nk, -1, TERM_ASGN_NODE);
```

```
        bp_assign (nk, inTempArray, nPartitn, new);
        establish_new (new);

        /** Sprout a nonterminal assignment node. **/
        new = allocate_node (x, x->g_cost, nk, nPartitn-1, NTERM_ASGN_NODE);
        establish_new (new);
        for (i = 0; i < nPartitn; i++) *(new->ctime+i) = *(new->itime+i) = *(timeTempArray+i);
}


establish_new (new)
/** Evaluate lower and upper bounds of this new node.
 ** Pruning test is done accordingly.
 ** Insert this node into a heap if it is not pruned. **/
        node *new;
{
        if (new->type == NTERM_ASGN_NODE) {
                new->lowb = new->parent->lowb;
                new->upb = new->parent->upb;
        } else {
                new->lowb = evaluate_lower_bound (new);
                new->upb = evaluate_upper_bound (new);
                if (new->upb < gUpperBound) gUpperBound = new->upb;
        }

        if (optimalSolution && new->lowb > get_expected_opt (gUpperBound)) enter_pruned (new);
        else insert (new);
}


yesno get_comb (n, m, first_time)
/** Find a permutation of n processors selected from m processors. **/
        int n, m, first_time;
{       int i;

        if (first_time) for (i = 0; i < n; i++) ptrTempArray[i] = i;

        /** Consider the special case for symmetric and homogeneous MIMD. **/
        if (homoMIMD && symMIMD) {
                if (first_time) {
                        for (i = 0; i < n; i++) outTempArray[i] = i;
                        return YES;
                }
                return NO;
        }

        /** Now, come to the general cases. **/
        if (ptrTempArray[n-1] < m) {
                for (i = 0; i < n; i++) outTempArray[i] = inTempArray[ptrTempArray[i]];
                for (i = n-1; ++ptrTempArray[i] > m-n+i; i--) if (i <= 0) break;
                for (i = 1; i < n; i++)
                        if (ptrTempArray[i] > m-n+i) ptrTempArray[i] = ptrTempArray[i-1] + 1;
                return YES;
```



```
        }
        return NO;
}


float get_expected_opt (val) float val;
/* Calculate the expected optimality for bounding by approximation. */
{ return (val / (1.0 + mappingApprox)); }
```

search/bound.c

```
float evaluate_lower_bound (x)
/** Evaluate the lower bound of a node.
 ** The algorithm is described in Chapter 6. **/
        node *x;
{       nodtype type = x->type;
        int nk = x->nk, l = x->nk, i;
        float ctime = x->g_cost, tcomp = 0.0, tcomm = 0.0;

        if (nPartitn <= 1) return 0.0;
        switch (type) {
                case GOAL_NODE: break;

                case ROOT_NODE:
                        tcomp = best_PCT (0, nCluster-1) + best_LCT (nCluster-1, 0);
                        tcomm = best_PRT (0, nLayer-2) + best_LRT (nLayer-1, 1);
                        ctime = (tcomp + tcomm) / nPartitn;  /** both nPartitn **/
                        break;

                case PROUT_NODE:
                        tcomp = best_LCT (nCluster-1, 0);
                        tcomm = best_LRT (nLayer-1, 1);
                        if (l < nLayer - 2) {
                                for (nk = i = 0; i < l+2; i++) nk += xlayer(i)->n;
                                tcomp += best_PCT (nk, nCluster-1);
                                tcomm += best_PRT (l+1, nLayer-2);
                        }
                        ctime += ((tcomp + tcomm) / nPartitn);
                        break;

                case LROUT_NODE:
                        if (l > 0) {
                                for (nk = -1, i = 0; i < l; i++) nk += xlayer(i)->n;
                                tcomp = best_LCT (nk, 0);
                                if (l > 1) tcomm = best_LRT (l-1,1);
                                ctime += ((tcomp + tcomm) / nPartitn);
                        }
                        break;
```

```
        case TERM_ASGN_NODE:
                tcomp = best_LCT (nCluster-1, 0);
                tcomm = best_LRT (nLayer-1, 1);
                if (nk < nCluster - 1) tcomp += best_PCT (nk+1, nCluster-1);
                if ((l = xcluster(nk)->layerID) == 0) l = 1;
                tcomm += best_PRT (l-1, nLayer-2);
                ctime += ((tcomp + tcomm) / nPartitn);
                break;

        case NTERM_ASGN_NODE:
                if (x->parent) {
                        if (ctime < x->parent->lowb) ctime = x->parent->lowb;
                }
                else error ("evaluate_lower_bound: impossible condition");
                break;

        default:
                error ("evaluate_lower_bound: no such node type");
                break;
    }

    if (x->parent) if (x->parent->lowb > ctime) ctime = x->parent->lowb;
    return ctime;
}


float evaluate_upper_bound (x)
/** Evaluate the upper bound of a node.
 ** The algorithm is described in Chapter 6. **/
    node *x;
{   nodtype type = x->type;
    int nk = x->nk, l = x->nk, i;
    float ctime = x->g_cost, tcomp = 0.0, tcomm = 0.0;

    if (nPartitn <= 1) return HUGE_FLOAT;
    switch (type) {
        case GOAL_NODE: break;

        case ROOT_NODE:
                tcomp = worst_PCT (0, nCluster-1) + worst_LCT (nCluster-1, 0);
                tcomm = worst_PRT (0, nLayer-2) + worst_LRT (nLayer-1, 1);
                if (nBcast > 0) ctime = tcomp / nPartitn + tcomm;
                else ctime = (tcomp + tcomm * nChannel) / nPartitn;
                tcomp = best_PCT (0, nCluster-1) + best_LCT (nCluster-1, 0);
                if (tcomp < ctime) ctime = tcomp;
                break;

        case PROUT_NODE:
                tcomp = worst_LCT (nCluster-1, 0);
                tcomm = worst_LRT (nLayer-1, 1);
                if (l < nLayer - 2) {
                        for (nk = i = 0; i < l+2; i++) nk += xlayer(i)->n;
                        tcomp += worst_PCT (nk, nCluster-1);
```



```
                        tcomm += worst_PRT (l+1, nLayer-2);
                }
                if (nBcast > 0) ctime += (tcomp / nPartitn + tcomm);
                else ctime += (tcomp + tcomm * nChannel) / nPartitn;
                break;

        case LROUT_NODE:
                if (l > 0) {
                        for (nk = -1, i = 0; i < l; i++) nk += xlayer(i)->n;
                        tcomp = worst_LCT (nk, 0);
                        if (l > 1) tcomm = worst_LRT (l-1,1);
                        if (nBcast > 0) ctime += (tcomp / nPartitn + tcomm);
                        else ctime += (tcomp + tcomm * nChannel) / nPartitn;
                }
                break;

        case TERM_ASGN_NODE:
                tcomp = worst_LCT (nCluster-1, 0);
                tcomm = worst_LRT (nLayer-1, 1);
                if (nk < nCluster - 1) tcomp += worst_PCT (nk+1, nCluster-1);
                if ((l = xcluster(nk)->layerID) == 0) l = 1;
                tcomm += worst_PRT (l-1, nLayer-2);
                if (nBcast > 0) ctime += (tcomp / nPartitn + tcomm);
                else ctime += (tcomp + tcomm * nChannel) / nPartitn;
                break;

        case NTERM_ASGN_NODE:
                if (x->parent) ctime = x->parent->upb;
                else error ("evaluate_upper_bound: impossible condition");
                break;

        default:
                error ("evaluate_upper_bound: no such node type");
                break;
    }
    return ctime;
}


float best_PCT (nk1, nk2)
/** Calculate the best production-phase computation time. **/
    int nk1, nk2;
{   int i; float t = 0.0;

    for (i = nk1; i <= nk2; i++) t += (bestPartitnT * xcluster(i)->n * xcluster(i)->pcunit);
    return t;
}


float best_LCT (nk1, nk2)
/** Calculate the best learning-phase computation time. **/
    int nk1, nk2;
{   int i; float t = 0.0;
```

```
        for (i = nk1; i >= nk2; i--) t += (bestPartitnT * xcluster(i)->n * xcluster(i)->lcunit);
        return t;
}


float best_PRT (l1, l2)
/** Calculate the best production-phase routing time. **/
        int l1, l2;
{       int i, j, nk; float t = 0.0, amount;

        for (i = l1; i <= l2; i++)
                for (j = 0; j < xlayer(i)->n; j++) {
                        nk = *(xlayer(i)->member + j);
                        amount = xcluster(nk)->n * xcluster(nk)->prunit;
                        if (nBcast > 0) {
                                amount /= nProcess;
                                /** BCAST[0] is the primary broadcast. **/
                                t += (xbcast(0)->ts + amount * xbcast(0)->tx);
                        }
                        else t += (bestChTs + amount * bestChTx);
                }
        return t;
}


float best_LRT (l1, l2)
/** Calculate the best learning-phase routing time. **/
        int l1, l2;
{       int i, j, nk; float t = 0.0, amount;

        for (i = l1; i >= l2; i--)
                for (j = 0; j < xlayer(i)->n; j++) {
                        nk = *(xlayer(i)->member + j);
                        amount = xcluster(nk)->n * xcluster(nk)->lrunit;
                        if (nBcast > 0) {
                                amount /= nProcess;
                                /** BCAST[0] is the primary broadcast. **/
                                t += (xbcast(0)->ts + amount * xbcast(0)->tx);
                        }
                        else t += (bestChTs + amount * bestChTx);
                }
        return t;
}


float worst_PCT (nk1, nk2)
/** Calculate the worst production-phase computation time. **/
        int nk1, nk2;
{       int i; float t = 0.0;

        for (i = nk1; i <= nk2; i++) t += (worstPartitnT * xcluster(i)->n * xcluster(i)->pcunit);
        return t;
}
```



```
float worst_LCT (nk1, nk2)
/** Calculate the worst learning-phase computation time. **/
        int nk1, nk2;
{       int i; float t = 0.0;

        for (i = nk1; i >= nk2; i--) t += (worstPartitnT * xcluster(i)->n * xcluster(i)->lcunit);
        return t;
}


float worst_PRT (l1, l2)
/** Calculate the worst production-phase routing time. **/
        int l1, l2;
{       int i, j, nk; float t = 0.0, amount;

        for (i = l1; i <= l2; i++)
                for (j = 0; j < xlayer(i)->n; j++) {
                        nk = *(xlayer(i)->member + j);
                        amount = xcluster(nk)->n * xcluster(nk)->prunit;
                        if (nBcast > 0)
                        /** BCAST[0] is the primary broadcast. **/
                                t += (xbcast(0)->ts + amount * xbcast(0)->tx);
                        else t += (worstChTs + amount * worstChTx);
                }
        return t;
}


float worst_LRT (l1, l2)
/** Calculate the worst learning-phase routing time. **/
        int l1, l2;
{       int i, j, nk; float t = 0.0, amount;

        for (i = l1; i >= l2; i--)
                for (j = 0; j < xlayer(i)->n; j++) {
                        nk = *(xlayer(i)->member + j);
                        amount = xcluster(nk)->n * xcluster(nk)->lrunit;
                        if (nBcast > 0)
                        /** BCAST[0] is the primary broadcast. **/
                                t += (xbcast(0)->ts + amount * xbcast(0)->tx);
                        else t += (worstChTs + amount * worstChTx);
                }
        return t;
}
```

---

search/comp.c

```
assign_comp_time (x, nk)
/* Calculate computation time for assignment nodes. */
```

```
        node *x; int nk;
{       int i, amount;

        for (i = 0; i < nPartitn; i++) {
                amount = *(x->assign+i);
                *(x->ctime+i) += part_P_comp_time (amount, nk, i);
        }
}


L_comp (x)
/** Perform learning-phase computation tasks for a layer. **/
        node *x;
{       int l = x->nk, i, amount; node *p;

        for (p = x->parent; p; p = p->parent)
                if (p->type == TERM_ASGN_NODE)
                        if (xcluster(p->nk)->layerID == l)
                                for (i = 0; i < nPartitn; i++) {
                                        amount = *(p->assign+i);
                                        *(x->ctime+i) += part_L_comp_time (amount, p->nk, i);
                                }
}


bp_assign (nk, pklist, n, x)
/** Assign a neural cluster over a set of partitions according to
 ** computation power and idleness of each partition.
 ** The arrays itime, ctime and g_cost are evaluated accordingly.
 ** The algorithm is described in Chapter 6. **/
        int nk, pklist[], n; node *x;
        /** Parameter n is the number of partitions in list pklist. **/
{       int i, pk, amount;

        for (i = 0; i < nPartitn; i++) *(x->ctime+i) = *(x->itime+i) = *(timeTempArray+i);
        bp_distr (nk, pklist, n, x);
        assign_comp_time (x, nk);
        x->g_cost = max_float (x->ctime, nPartitn);
}


bp_distr (nk, pklist, n, x)
/** Perform a bin-packing distribution based on
 ** the initial start time in node x and
 ** computation power of partitions specified in list pklist.
 ** The algorithm is described in Chapter 6. **/
        int nk, pklist[], n; node *x;
{       int i, j, k, ni, diff; float ratio = 0.0, inverse = 0.0, num, portion;

        for (i = 0; i < nPartitn; i++) *(tauTempArray+i) = 0.0;
        for (i = 0; i < n; i++) {
                k = pklist[i];
                *(tauTempArray+k) = part_P_comp_time (1, nk, k);
        }
        ni = xcluster(nk)->n;
        for (i = 0; i < n; i++) {
                k = pklist[i];
                ratio += (*(x->itime+k) / *(tauTempArray+k));
                inverse += (1.0 / *(tauTempArray+k));
        }

        num = (ni + ratio) / inverse;
        for (i = 0; i < nPartitn; i++) *(x->assign+i) = 0;
        diff = ni;
        for (i = 0; i < n; i++) {
                j = pklist[i];
                portion = (num - *(x->itime+j)) / *(tauTempArray+j);
                diff -= (*(x->assign+j) = ((int) portion));
        }

        /** Make up roundoff error. **/
        while (diff >= n) for (i = 0; i < n; i++) { diff--; (*(x->assign + pklist[i]))++; }
        for (i = 0; i < diff; i++) (*(x->assign+pklist[i]))++;
}
```


scarch/comm.c

```
P_routing (x)
/** Solve production-phase routing from layer l to layer l+1.
 ** The array ctime and the g_cost of node x are updated accordingly,
 ** and the array ctime and g_cost of a new node are updated also. **/
        node *x;
{       int type = x->type, l = xcluster(x->nk)->layerID - 1;
        int i, j, z, n, *member, nk;
        node *new, *p_s, *p_d; route *r, *r0;

        new = allocate_node (x, 0.0, l, -1, PROUT_NODE);
        for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(new->ctime+i);
        for (p_s = x; p_s; p_s = p_s->parent) {
                if (p_s->type == TERM_ASGN_NODE) {
                        if (xcluster(p_s->nk)->layerID == l) {
                                for (i = 0; i < nPartitn; i++) {
                                        *(sourceTempArray+i) = *(p_s->assign+i);
                                        *(destTempArray+i) = 0;
                                }
                                for (p_d = x; p_d; p_d = p_d->parent) {
                                        if (p_d->type == TERM_ASGN_NODE) {
                                                if (is_succ_cluster (p_d->nk, p_s->nk) == YES) {
                                                        for (i = 0; i < nPartitn; i++)
                                                                *(destTempArray+i) =
                                                                *(destTempArray+i) || *(p_d->assign+i);
                                                }
```

```
                        }
                    }
                    (r = routing (p_s->nk, PCOMM))->next = NULL;
                    if (new->route) {
                        for (r0 = new->route; r0->next; r0 = r0->next) ;
                            r0->next = r;
                    }
                    else new->route = r;
                }
            }
        }

        for (i = 0; i < nPartitn; i++) *(new->ctime+i) = *(timeTempArray+i);
        new->g_cost = max_float (new->ctime, nPartitn);
        establish_new (new);
}


L_routing (x)
/** Solve learning-phase routing from layer l to layer l-1.
 ** The array ctime and the g_cost of node x are updated accordingly,
 ** and the array ctime and g_cost of a new node are updated also. **/
        node *x;
{       int type = x->type, l = x->nk - 1, i, j, z, n, *member, nk;
        node *new, *p_s, *p_d; route *r, *r0;

        L_comp ((new = allocate_node (x, 0.0, l, -1, LROUT_NODE)));
        for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = *(new->ctime+i);
        for (p_s = x; p_s; p_s = p_s->parent) {
            if (p_s->type == TERM_ASGN_NODE) {
                if (xcluster(p_s->nk)->layerID == l) {
                    for (i = 0; i < nPartitn; i++) {
                        *(sourceTempArray+i) = *(p_s->assign+i);
                        *(destTempArray+i) = 0;
                    }
                    for (p_d = p_s->parent; p_d; p_d = p_d->parent) {
                        if (p_d->type == TERM_ASGN_NODE) {
                            if (is_pred_cluster (p_d->nk, p_s->nk) == YES) {
                                for (i = 0; i < nPartitn; i++)
                                    *(destTempArray+i) =
                                    *(destTempArray+i) || *(p_d->assign+i);

                                (r = routing (p_s->nk,
                                LCOMM))->next = NULL;
                                if (new->route) {
                                    for (r0 = new->route;
                                    r0->next; r0 = r0->next) ;
                                    r0->next = r;
                                }
                                else new->route = r;
                            }
                        }
                    }
```



```
                }
            }
        }
        for (i = 0; i < nPartitn; i++) *(new->ctime+i) = *(timeTempArray+i);
        new->g_cost = max_float (new->ctime, nPartitn);
        establish_new (new);
}


yesno is_pred_cluster (nk1, nk2)
/* Check if cluster nk1 is one of the predecessors of cluster nk2. **/
        int nk1, nk2;
{       int npred = xcluster(nk2)->npred, *pred = xcluster(nk2)->pred, i;

        for (i = 0; i < npred; i++) if (nk1 == *(pred+i)) return TRUE;
        return FALSE;
}


yesno is_succ_cluster (nk1, nk2)
/* Check if cluster nk1 is one of the successors of cluster nk2. **/
        int nk1, nk2;
{       int nsucc = xcluster(nk2)->nsucc, *succ = xcluster(nk2)->succ, i;

        for (i = 0; i < nsucc; i++) if (nk1 == *(succ+i)) return TRUE;
        return FALSE;
}
```

---

search/route.c

```
route *routing (nk, phase)
/* Find a route. */
        int nk; comm phase;
{       route *r = (route *) malloc (sizeof (route));

        /** Reset routing error. **/
        r->error = routeError = 0.0;
        r->next = NULL;

        /** See if broadcast is applicable or not. **/
        if ((homoMIMD && symMIMD) || (is_subset (sourceTempArray, destTempArray, nPartitn) == YES))
            if (broadcast_route (nk, phase) == YES) { r->type = BC; return r; }

        /** Unfortunately, broadcast is not applicable, point-to-point
         ** routing is used. **/
        r->type = PP;
        link_route (nk, phase);
        r->error = routeError;
        return r;
```

```
}


link_route (nk, phase)
/** Solve a point-to-point routing.
 ** The initial start time is in array timeTempArray.
 ** The completion time will be in array timeTempArray.
 **/
        int nk; comm phase;
{       int i, j, *path;
        float tau1 = 0.0, tau2 = 0.0, tsum = 0.0, tmax = 0.0, rt = 0.0, t, unit;

        /** Calculate the amounts of communication per neuron. **/
        unit = (phase == PCOMM) ? xcluster(nk)->prunit : xcluster(nk)->lrunit;

        /** Set a difference vector between source and destination vectors. */
        for (i = 0; i < nPartitn; i++)
                *(diffTempArray+i) = (*(sourceTempArray+i) > 0 && *(destTempArray+i) == 0) ?
                        *(sourceTempArray+i) : 0;

        if (discover_broadcast (destTempArray) == YES) {
        /** As described in Chapter 6, a communication graph consists of
         ** a path and a broadcast within a destination graph. **/
                /* Find a path. */
                for (i = 0; i < nPartitn; i++) {
                        if (*(diffTempArray+i)) {
                                t = comm_path (i, destTempArray, *(diffTempArray+i) * unit);
                                tsum += t;
                                if (t > rt) rt = t;
                        }
                }
                routeError += (tsum - rt);
                /* Find a broadcast. */
                for (i = 0; i < nPartitn; i++) {
                        if (*(sourceTempArray+i) && *(destTempArray+i))
                                tau1 += (bcTs + *(sourceTempArray+i) * unit * bcTx);

                        if (*(diffTempArray+i))
                                tau2 += (bcTs + *(diffTempArray+i) * unit * bcTx);
                }

                /** Update time in array destTempArray. **/
                for (i = 0; i < nPartitn; i++)
                        if (*(destTempArray+i))
                                *(timeTempArray+i) += (rt > tau1 ? rt+tau2 : tau1+tau2);
        } else {
                /** The case is point-to-point routing only. **/
                path = (int *) malloc (nPartitn * sizeof (int));
                for (i = 0; i < nPartitn; i++) *(path+i) = 0;
                for (i = 0; i < nPartitn; i++)
                        if (*(sourceTempArray+i))
                                for (j = 0; j < nPartitn; j++)
                                        if (*(destTempArray+j)) {
```



```
                                                *(path+j) = 1;
                                                *(timeTempArray+j) +=
                                                comm_path (i, path, *(sourceTempArray+i) * unit);
                                                *(path+j) = 0;
                                        }
                free (path);
        }
}


float comm_path (s, dest, unit)
/** Find a path from the source to a destination. **/
        int s, dest[]; float unit;
{       int i, *path = (int *) malloc (nPartitn * sizeof (int)); float t;

        for (i = 0; i < nPartitn; i++) *(path+i) = 0; *(path+s) = 1;
        t = aux_comm_path (s, dest, unit, path);
        free (path);
        return t;
}


float aux_comm_path (s, dest, unit, path)
/** This routine in fact is an auxilary routine for the above routine.
 ** It recursively finds a path from the source to a destination. **/
        int s, dest[], path[]; float unit;
{       int i, *patharray, *neighbor, n, nbor, found = -1; channel *ch;
        float t, ts = HUGE_FLOAT, tx = HUGE_FLOAT, tmin = HUGE_FLOAT;

        if (dest[s]) return 0.0;
        neighbor = (int *) malloc (nPartitn * sizeof (int));
        for (i = 0; i < nPartitn; i++) *(neighbor+i) = 0;
        n = discover_neighbor (s, neighbor);

        for (i = 0; i < n; i++) {
                nbor = *(neighbor+i);
                if (dest[nbor]) {
                        ch = xchannel(*(xpartitnicm(s,nbor)));
                        if (ch->tx < tx) { found = nbor; ts = ch->ts; tx = ch->tx; }
                }
        }

        if (found >= 0) { path[found] = 1; free (neighbor); return (ts + unit * tx); }

        patharray = (int *) malloc (nPartitn * sizeof (int));
        for (i = 0; i < nPartitn; i++) *(patharray+i) = *(path+i);
        *(patharray+s) = 1;
        tmin = HUGE_FLOAT;
        found = -1;
        for (i = 0; i < n; i++) {
                nbor = *(neighbor+i);
                if (! *(patharray+nbor)) {
                        t = aux_comm_path (nbor, dest, unit, patharray);
```

```
                ch = xchannel(*(xpartitnicm(s,nbor)));
                t += (ch->ts + unit * ch->tx);
                if (t < tmin) { found = nbor; tmin = t; }
            }
        }
        if (found >= 0) path[found] = 1; else error ("aux_comm_path: negative found");
        free (neighbor); free (patharray);
        return tmin;
}


int discover_neighbor (s, neighbor)
/* Discover all immediate neighbors of a partition (or processor). **/
        int s, neighbor[];
{       int count, i;

        for (count = i = 0; i < nPartitn; i++)
                if (xpartitnicm(s,i) || xpartitnicm(i,s)) neighbor[count++] = i;
        return count;
}


yesno discover_broadcast (list)
/* Discover an appropriate broadcast. **/
        int list[]; /** This is a list of partitions (or processors). **/
{       int bc, i, j, in_range, count = 0; bcast *bcp;

        bcTs = bcTx = HUGE_FLOAT;
        for (i = 0; i < nPartitn; i++) if (list[i]) count++;
        if (nBcast) {
                for (bc = 0; bc < nBcast; bc++) *(bcTempArray+bc) = 0;
                for (bc = 0; bc < nBcast; bc++) {
                        bcp = xbcast(bc);
                        in_range = 1;
                        for (i = 0; i < nPartitn; i++)
                                if (list[i])
                                        if (bcp->from <= i && i <= bcp->to) (*(bcTempArray+bc))++;
                                        else in_range = 0;
                        if (in_range) { bcTs = bcp->ts; bcTx = bcp->tx; return YES; }
                        /** only in one group **/
                }

                bc = which_max_int (bcTempArray, nBcast);
                if (*(bcTempArray+bc) == count) {
                /** belong to the same BC region **/
                        bcTs = xbcast(bc)->ts;
                        bcTx = xbcast(bc)->tx;
                        return YES;;
                }
        }
        return NO;
}
```



```
yesno broadcast_route (nk, phase)
/** Find a broadcast route.
 ** The initial start time is in array _timearray.
 ** The completion time will be in array _timearray.
 **/
        int nk; comm phase;
{       int i; float t, sum, unit;

        if (discover_broadcast (destTempArray)) {
                for (sum = 0.0, i = 0; i < nPartitn; i++) {
                /** find dominating frame **/
                        unit = (float) *(sourceTempArray+i);
                        unit /= ((float) xpartitn(i)->n);
                        if (unit > sum) sum = unit;
                }
                unit = (phase == PCOMM) ? xcluster(nk)->prunit : xcluster(nk)->lrunit;
                sum = (phase == PCOMM) ? (sum * unit) : (xcluster(nk)->n * unit);
                t = bcTs + sum * bcTx;

                /** If broadcast is used, all data migrations are assumed
                 ** to be completed at the same time, e.g., in iPSC/2. **/
                for (i = 0; i < nPartitn; i++) *(timeTempArray+i) += t;
                t = max_float (timeTempArray, nPartitn);
                for (i = 0; i < nPartitn; i++) *(timeTempArray+i) = t;
                return YES;
        }
        return NO;
}
```

---

search/memory.c

```
node *pool_mgr = NULL;
node *_alloc_node ()
/** Allocate a search node from the NeuMap-maintained free list. **/
{       node *p; int n, n0, n1, n2, n3, i, *addr;

        n0 = sizeof (node) / sizeof (int);
        n1 = nPartitn * sizeof (int) / sizeof (int);
        n2 = nPartitn * sizeof (float) / sizeof (int);
        n3 = nPartitn * sizeof (float) / sizeof (int);
        n = n0 + n1 + n2 + n3;

        if (pool_mgr == NULL) {
                pool_mgr = (node *) malloc (256 * n * sizeof (int));
                for (i = 0; i < 256; i++) {
                        addr = (int *) pool_mgr + i * n;
                        p = (node *) addr;
                        addr = (int *) pool_mgr + (i+1) * n;
                        p->next = (node *) addr;
```

```
            }
            addr = (int *) pool_mgr + 255 * n;
            p = (node *) addr;
            p->next = NULL;
        }
        pool_mgr = (p = pool_mgr)->next;
        addr = (int *) p;
        p->assign = addr + n0;
        p->itime = (float *) (addr + n0 + n1);
        p->ctime = (float *) (addr + n0 + n1 + n2);
        return p;
}


node *allocate_node (parentp, g_cost, nk, degree, type)
/** Allocate a search node. Certain fields are initialized. **/
        node *parentp; float g_cost; int nk, degree, type;
{       int i; node *p = _alloc_node ();
#ifdef STATISTICS
        ++nGenerated;
#endif
        p->parent = parentp; p->next = NULL; p->lowb = g_cost;
        p->upb = HUGE_FLOAT; p->g_cost = g_cost; p->nk = nk; p->route = NULL;
        p->degree = degree; p->nsprout = 0; p->type = type;
        if (parentp) p->depth = parentp->depth + 1; else p->depth = 0;
        for (i = 0; i < nPartitn; i++) if (type == TERM_ASGN_NODE) *(p->assign+i) = 0;
        if (parentp)
                for (i = 0; i < nPartitn; i++) *(p->ctime+i) = *(p->itime+i) = *(parentp->ctime+i);
        else for (i = 0; i < nPartitn; i++) *(p->ctime+i) = *(p->itime+i) = 0.0;
        if (parentp) (parentp->nsprout)++;
        return p;
}


free_node (x)
/* Release a search node to the NeuMap-maintained free list. */
        node *x;
{       int i;

        if (x->nsprout > 0) return;
        if (x->parent) if ((--(x->parent->nsprout)) <= 0) free_node (x->parent);
        x->next = pool_mgr;
        pool_mgr = x;
}


free_list (list)
/* Release a list of search nodes. */
        node *list;
{       node *p, *p0;

        for (p = list; p; ) { p0 = p->next; free_node (p); p = p0; }
}
```



```
#define getkeyHeap(x) (x->lowb)

insert (x)
/** Insert a node into a priority heap. **/
        node *x;
{       hnode *p;

        /** create a heap node **/
        (p = heap_node_alloc ())->info = x;
        p->key = getkeyHeap (x);
        heap_insert (p);
}


node *delete ()
/** Delete a node from a priority heap. **/
{       hnode *p; node *q = NULL;

        if (heap_root) { p = heap_delete (); q = p->info; heap_node_release (p); }
        return q;
}

#undef getkeyHeap(x)
```

---

search/etc.c

```
int getline (line, limit, input)
/** Read in a line. **/
        char line[]; int limit; FILE *input;
{       int i;

        if (fgets (line, limit, input)) {
                for (i = 0; i < limit && line[i] != ' ' && line[i] != '\n'; i++) ;
                if (line[i] == '\n') line[i] = ' ';
                return 1;
        }
        else return 0;
}


error (s)
/** Report an error and exit NeuMap. **/
        char s[];
{       char msg[200];

        sprintf (msg, "***** ERROR ***** : %s\n", s);
        perror (msg);
        exit (0);
}
```

```
yesno is_input_cluster (k)
/** Check if cluster x is an input cluster. **/
        int k;
{       neutype type = xcluster(k)->type;

        return ((type == INPUT_NEURON || type == IO_NEURON) ? YES : NO);
}


yesno is_output_cluster (k)
/** Check if cluster x is an output cluster. **/
        int k;
{       neutype type = xcluster(k)->type;

        return ((type == OUTPUT_NEURON || type == IO_NEURON) ? YES : NO);
}


yesno is_2nd_layer_cluster (k)
/** Check if cluster x is in the second layer. **/
        int k;
{       int n = xlayer(1)->n, *member = xlayer(1)->member, i;

        for (i = 0; i < n; i++) if (*(member+i) == k) return YES;
        return NO;
}


yesno is_subset (s1,s2,n)
/** If s1 <= s2, return YES. Otherwise, return NO. **/
        int s1[], s2[], n;      /** Parameter n is the length. **/
{       int i;

        for (i = 0; i < n; i++) if (s1[i] && (! s2[i])) return NO;
        return YES;
}


float max_float (list, n)
/** Find an maximum out of a list of n floating-point numbers. **/
        float list[]; int n;
{       int i; float the_max = - HUGE_FLOAT;

        for (i = 0; i < n; i++) if(*(list+i) > the_max) the_max = *(list+i);
        return the_max;
}


float min_float (list, n)
/** Find an minimum out of a list of n floating-point numbers. **/
        float list[]; int n;
{       int i; float the_min = HUGE_FLOAT;
```



```
        for (i = 0; i < n; i++) if (*(list+i) < the_min) the_min = *(list+i);
        return the_min;
}


int which_max_int (list, n)
/** Find which entry is holding the maximum integer. **/
        int list[], n;
{       int i, who = 0, the_max = list[0];

        for (i = 1; i < n; i++) if (list[i] > the_max) { the_max = list[i]; who = i; }
        return who;
}
```

---

search/heap.c

```
/** This file contains routines maintaining a dynamic linked heap.
 ** These routines are not documented because they will not affect
 ** the correctness of NeuMap if they are replaced by another
 ** priority queue, eg. a single-linked list to the extreme case. **/

#define HEAP_ALLOC_SIZE                 1024

typedef float           keytype;
struct _hnode__
{
        keytype                 key;
        node            *info;
        struct _hnode__  *parent;
        struct _hnode__  *left_child;
        struct _hnode__  *right_child;
};
typedef         struct _hnode__         hnode;

hnode  *heap_manager = NULL;    /** List head of a free list of heap nodes **/
hnode  *heap_root = NULL;  /** Root of this heap **/
hnode  *heap_last = NULL;  /** Last node of this heap **/

enum lrhand { LEFT_HAND = 1, RIGHT_HAND = 2 } whichHeapTemp;

heap_node_release (p) hnode *p; { p->parent = heap_manager; heap_manager = p; }


heap_release (p)
        hnode *p;
{       node *q;

        if (!p) return;
        if (p->left_child) heap_release (p->left_child);
```

```
        if (p->right_child) heap_release (p->right_child);
        q = p->info;
        q->next = activeList;
        activeList = q;
        heap_node_release (p);
}


hnode *heap_node_alloc ()
{       hnode *p; int i;

        /** If the pool of free heap nodes is empty, then
         ** allocate 1000 heap nodes at a time to avoid spreadout of
         ** heap nodes to a wide range of virtual-memory pages. **/
        if (! heap_manager) {
                p = heap_manager = (hnode *)
                        malloc (HEAP_ALLOC_SIZE * sizeof (hnode));
                for (i = 0; i < HEAP_ALLOC_SIZE-1; i++) (p+i)->parent = p+i+1;
                (p+HEAP_ALLOC_SIZE-1)->parent = NULL;
        }
        heap_manager = (p = heap_manager)->parent;
        p->parent = p->left_child = p->right_child = NULL;
        p->info = NULL;
        return p;
}


hnode *locate_next (p)
        hnode *p;
{
        /** If the heap contains only the root, then the next is
         ** the left child of the root. **/
        if (p == heap_root) { whichHeapTemp = LEFT_HAND; return p; }

        /** If the heap_last is the left_child of its parent, then
         ** the next is the right_child of its parent. **/
        if (p == p->parent->left_child) { whichHeapTemp = RIGHT_HAND; return p->parent; }

        /** Trace upward until p is the left_child of its parent. **/
        for (p = p->parent; p != heap_root; p = p->parent)
                if (p == p->parent->left_child) {
                        /** Trace downward, first to the right then to the
                         ** left, and all the way to the terminal one. **/
                        for (p = p->parent->right_child; p->left_child;
                                        p = p->left_child) ;
                        whichHeapTemp = LEFT_HAND;
                        return p;
                }

        /** Trace downward to the left from the heap root. **/
        while (p->left_child) p = p->left_child;
        whichHeapTemp = LEFT_HAND;
        return p;
```



```
}


hnode *locate_prev (p, prev_root)
        hnode *p, *prev_root;
{
        /** If the heap contains only one node, then just let it be
         ** empty after the deletion. **/
        if (p == heap_root) return NULL;

        /** If the heap_last is the right_child of its parent, then
         ** the prev is the left_child of its parent. **/
        if (p == p->parent->right_child) return p->parent->left_child;

        /** Trace upward until p is the right_child of its parent. **/
        for (p = p->parent; p != prev_root; p = p->parent)
                if (p == p->parent->right_child) {
                        for (p = p->parent->left_child; p->right_child; p = p->right_child);
                        return p;
                }

        /** Trace downward to the right from the heap root. **/
        while (p->right_child) p = p->right_child;
        return p;
}


heap_insert (p)
        hnode *p;
{       hnode *q;

        /** If the heap is empty, just modify heap_root and heap_last. **/
        if (! heap_last) { heap_root = heap_last = p; return; }

        /** Locate which node is the one immediately preceding to heap_last. **/
        q = locate_next (heap_last);

        /** Update new heap_last. **/
        (heap_last = p)->parent = q;

        /** Determine this new node p should be connected to left_child
         ** or to right_child. **/
        if (whichHeapTemp == LEFT_HAND) q->left_child = p; else q->right_child = p;

        /** Maintain the heap property. A bias is introduced to the tie
         ** in which the newer one is prefered. **/
        while (p->key <= q->key) {
                exchange (q, p);
                /** check if root is met **/
                if (! (q = (p = q)->parent)) break;
        }
}
```

```
hnode *heap_delete ()
{       hnode *to_be_returned = heap_root, *p;

        /** If there is only one node in this heap, then just return it
         ** and reset heap_last and heap_root to NULL. **/
        if (heap_last == heap_root) {
                heap_last = heap_root = NULL;
                return to_be_returned;
        }

        /** After deletion, only the root exists. **/
        if (heap_last == heap_root->left_child) {
                (heap_root = heap_last)->parent = NULL;
                return to_be_returned;
        }

        heap_last = locate_prev ((p = heap_last), to_be_returned);
        if (p == p->parent->left_child) p->parent->left_child = NULL;
        else p->parent->right_child = NULL;

        /** Let p be the heap root. **/
        p->parent = NULL;
        p->left_child = heap_root->left_child;
        p->right_child = heap_root->right_child;
        if (p->left_child) p->left_child->parent = p;
        if (p->right_child) p->right_child->parent = p;
        heap_root = p;

        while (p->left_child) {
                if (p->key <= p->left_child->key) {
                        if (! p->right_child) return to_be_returned;
                        if (p->key <= p->right_child->key) return to_be_returned;
                        exchange (p, p->right_child);
                        p = p->right_child;
                } else
                if (! p->right_child) {
                        exchange (p, p->left_child);
                        p = p->left_child;
                } else
                if (p->left_child->key <= p->right_child->key) {
                        exchange (p, p->left_child);
                        p = p->left_child;
                } else {
                        exchange (p, p->right_child);
                        p = p->right_child;
                }
        }
        return to_be_returned;
}


exchange (parent, child)
        hnode *parent, *child;
```



```
{       keytype key_tmp; node *info_tmp;

        key_tmp = parent->key;
        parent->key = child->key;
        child->key = key_tmp;

        info_tmp = parent->info;
        parent->info = child->info;
        child->info = info_tmp;
}


heap_init ()
{
        heap_release (heap_root);
        heap_root = heap_last = NULL;
        free_list (activeList);
        activeList = NULL;
}
```

APPENDIX B.

DSIM PROGRAM

This appendix mainly describes Dsim and how to use it. Section B.1 describes briefly the algorithm, organization and options in NeuMap. Section B.2 describes how to use it. Finally, Section B.3 lists its source code.

B.1. General Description

Dsim is a program which simulates parallel ANN simulations on a uniprocessor. The scenario in Dsim is as belows.

1. Dsim reads in multicomputer and ANN specifications.
2. Dsim randomly generates workload descriptors for all processors.
3. Dsim randomly generates the workload for each processor based on its descriptor.
4. Dsim reports a multicomputer specification (including workload) and an ANN specification to NeuMap and calls it to solve the optimal mapping.
5. Dsim waits until the optimal mapping is solved, and then reads in the new mapping scheme.
6. Dsim starts to simulate the ANN parallel simulation, until the current iteration is completed.
7. Dsim randomly generates a new workload for each processor.
8. Dsim checks the gain of calling NeuMap again. If positive, then go to Step 4. Otherwise, go to Step 6. This process is repeated until all required iterations are completed.

The major components in Dsim include (1) interface for reading in multicomputer and ANN specifications and mapping schemes, (2) generator of workload and its descriptor, (3) event manager, (4) resource manager, and (5) mapping manager. These components are physically decomposed into the following files: sim.h, var.c, main.c, sim.c, event.c, map.c and load.c. They are briefly explained here.

sim.h All constant declarations and type definitions are included in it.

var.c All important global variables are declared in it.

main.c Trigger routines for Dsim, including IO routines and workload generator, are included in it. Several handy utilities are also included in it.

sim.c Simulation routines, including resource and event managers, are in it.

event.c General utilities for handling events are in it, including an event queue manager and an event space manager.

map.c Routines for mapping management, including decision-making for calling NeuMap and handshaking with NeuMap, are in it.

load.c Performance modes and interdependency of source files in Dsim are specified in it.

For different purposes, Dsim can be configurated into three modes:

DMAPPING Simulates the ANN and calls NeuMap for a new mapping schemes if there is a gain (for measuring T_{dyn}).

GMAX Simulates the ANN and calls NeuMap for a new mapping scheme at every iteration (for calculating g_{max}).

SIM Calls NeuMap only once and then simulates the ANN (for measuring T_{static}).

Further, several options are available for monitoring execution progress of and debugging Dsim.

STATISTICS Reports all important statistics.

MONITOR Reports execution progress of Dsim and the control thread between Dsim and NeuMap,

DEBUG Provides multiple levels of details in debugging information.

B.2. Steps for Using Dsim

Follow these steps to execute the supplied program.

(1) Go to the directory ~software/chu/ms/bin.

(2) Specify the options described in the previous section for compilation in Makefile.

(3) Compile Dsim by typing "make dsim" Other options are available, like Gmax by typing "make gmax" and Sim by typing "make sim"

(4) Create input files for Dsim. Four input files are required; two are the specifications of a multicomputer and the other two are the specifications of an ANN. Examples are available in directory ~software/chu/ms/lab/input. The formats of input files are described later. The input file names are required to be specified in the file FILES. For example, the file names in FILES normally are

mimd-sim

mimd-slot

mimd-map

ann-sim

ann-map

map-sim

The file mimd-sim specifies a multicomputer for Dsim. The file mimd-slot is used by Dsim to generate a file mimd-map which NeuMap can understand. The file ann-sim specifies an ANN for Dsim. The file ann-map is used by NeuMap and Dsim does not use it. Finally, the file map-sim is the pipe used between Dsim and NeuMap.

(5) Run the program by typing "dsim num-quanta rand-seed error debug-mode" where "num-quanta" is the number of iterations of ANN simulation, "rand-seed" is the seed for a random workload generator, "error" is the user-specified error allowance, *e.g.*, 0.05, and "debug-mode" is the level of details of debugging information, *e.g.*, 1 for fewer details and 4 for complete details. Three output files are available. One is "wl-param" which specifies the workload descriptors used. Another is "wl-pf" which specifies the workloads used. The other is "dsimtime-pf" which contains the results produced by Dsim.

The format of file mimd-sim is described as follows.

(1) Specify the number of broadcasts in the first line. Then, specify the number of processors and the number of links. For the network of three workstations, they are

0

3 1

They are read as there is no broadcast (0 in the first line) and 3 machines and 1 communication link. For the case of 25 processors, there is 1 broadcast, then the specification is

1 2.5 0.025 0 24

It is read as the broadcast 1 has setup time 2.5 (ms) and transmission time per word is 0.025(ms) and it covers processors from 0 to 24.

(2) Specify the number of homogeneous processors and their computation power in terms of execution time per unit computation. For the network of three workstations, they are

1 28.5

1 25.5

1 16.7

They are read as one processor has power 28.5 and another one has 25.5 and the third one has 16.7.

(3) Specify the number of homogeneous links and setup and transmission times. For the network of three workstations, they are

1 0.0 0.00533

(4) Specify the interconnection in terms of gateway. If there are P processors, then $P \times P$ lines are required. Each processor requires P lines. A line for processor i specifies the number of destination processors which it must go through the specified gateway to reach, the link to the gateway, and the gateway. For example of the network of three workstations, for processor 0

1 0 0

1 0 1

1 0 2

The first line specifies the destination processor 0, the link to the gateway is 0, and the gateway is processor 0. Note that this line actually is useless and is only for padding, because it is meaningless to send frames from a processor to itself. The second line specifies the destination processor 1, the link to the gateway is 0, and the gateway is 1. Because the gateway is the destination processor, it implies there is a direct link connecting processors 0 and 1. The third line is for destination processor 2. For a more interesting example, the case of 10 processors is considered here. For processor 0

1 0 0

1 0 1

1 0 2

1 0 3

1 0 4

5 2 5

The first five lines are read as before. The interesting line is the sixth. It is read as processors from 5 to 9 (consecutive 5 processors having the same specifications), the link to the gateway is link 2, and the gateway is processor 5.

The format of file ann-sim is described as follows.

(1) Specify the number of clusters, the starting cluster in the input layer, the starting cluster in the second layer and the starting cluster in the output layer. For the example of ML-1, they are

7 0 3 6

They are read as ML-1 has 7 clusters, its input layer starts from cluster 0, its second layer starts from cluster 3, and its output layer starts from cluster 6.

(2) Specify the clusters. Each cluster requires 3 lines. The first line specifies the cluster and the number of neurons in this cluster. The second line specifies the number of preceding clusters and what they are. The third line specifies the number of succeeding clusters and what they are. For the example of ML-1, the specification for cluster 0 is

0 500

0

3 1 2 3

They are read as cluster 0 has 500 neurons, it has no preceding clusters (0 in the second line), and it has 3 succeeding clusters and they are clusters 1, 2 and 3.

The file mimd-slot is exactly the same as the file mimd-map used in NeuMap, except the execution times pre unit computation for all processors are parameterized such that Dsim can change them according to workloads. The resulting file is mimd-map which is used by NeuMap. The file ann-map is exactly the same as the file ann-map used in NeuMap and Dsim does not use it. The file map-sim is the pipe used between NeuMap and Dsim. The output file dsimtime-pf reports the performance in a line for every 10 quanta. The format is the current quantum, cumulative simulation time, simulation time for this quantum, predicted simulation time, cumulative mapping time and latest mapping time.

~software/chu/ms/lab/input/sun-3/mimd-sim

```
0
3 1
1 28.5
1 25.5
1 16.7
1 0.0 0.00533
1 0 0
1 0 1
1 0 2
1 0 0
1 0 1
1 0 2
1 0 0
1 0 1
1 0 2
```

~software/chu/ms/lab/input/sun-3/mimd-slot

```
3 1 1
startproc
homo 1
$0 20000 1 0 0.0
homo 1
$1 50000 1 0 0.0
homo 1
$2 50000 1 0 0.0
endproc
startadj
-1 0 0
0 -1 0
0 0 -1
endadj
startlink
homo 1
0.0 0.00533
endlink
startsupport
0 3 0 1 2
endsupport
```

~software/chu/ms/lab/input/multi-layer/ann-sim-1

```
7 0 3 6
0  500
0
3  1 2 3
1  200
1  0
1  4
2  500
1  0
2  4 5
3  300
1  0
1  5
4  400
2  1 2
1  6
5  600
2  2 3
1  6
6  200
2  4 5
0
```


B.3. Program Listing

dsim/load.c

```
#define STATISTICS
#define MONITOR
#define DMAPPING

#include "sim.h"
#include "var.c"
#include "main.c"
#include "sim.c"
#include "event.c"
#include "map.c"
```

dsim/sim.h

```
#ifndef _DSIM_H__
#define _DSIM_H__

#include    <stdio.h>
#include    <math.h>

#defineNUM_WL_PARAM             7
#defineMAX_FNAME_LENGTH         30
#defineNOP                  630427
#defineMACHINE_WEIGHT           7.2317

#defineWL_IN_TREND_LOWB         0.5
#defineWL_IN_TREND_UPB          0.9
#defineWL_OUT_TREND_LOWB        0.0
#defineWL_OUT_TREND_UPB         (1.0 - intrend)
#defineWL_STRAIGHT              (1.0 - intrend - outtrend)
#defineWL_XMIN_SLOPE            0.2
#defineWL_XMAX_SLOPE            1.0
#defineWL_YMIN_SLOPE            0.1
#defineWL_YMAX_SLOPE            0.5
#defineWL_UPPER_BOUND           25.0
#defineWL_LOWER_BOUND           1.0

typedef enum
{       /** Kinds of events **/
        DONT_CARE,      /** This event does not care. **/
        COMP,           /** Computation **/
        NEXT,           /** Computation is next. **/
```

```
    COMM,               /** Communication **/
    ARRIVE,             /** Arrival of a frame **/
    BCAST,              /** Broadcast **/
    BCAST_ARRIVE,       /** Arrival due to broadcast **/
    PRODUCTION,         /** Production-phase task **/
    LEARNING,           /** Learning-phase task **/
    IDLE,               /** Machine is idle. **/
    BUSY                /** Machine is busy. **/
} simtype;

typedef enum
{   /** Kinds of workload descriptor parameters **/
    IN_TREND = 0,       /** In-trend parameter **/
    OUT_TREND = 1,      /** Out-of-trend parameter **/
    STRAIGHT = 2,       /** Go-straight parameter **/
    SLOPE = 3,          /** Slope of change of workload **/
    UPB = 4,            /** Upper bound of workload **/
    LOWB = 5,           /** Lower bound of workload **/
    TREND = 6           /** Trend of workload, 1 up, 0 straight, -1 down **/
} wltype;

typedef     float                       simtime;
typedef enum  { FALSE = 0, TRUE = 1 }   boolean;
typedef struct { int l, m; }            icnet;

struct _event_
{   /** Data type for events **/
    simtype     type;
    simtime     time;
    int         m0, m1, m2, m3;     /** These are machines. **/
    int         kid;                /** Cluster ID **/
    simtype     phase;              /** Production or learning **/
    struct _event_ *next;
};
typedef     struct _event_          event;

typedef struct
{   /** Data type for links **/
    float       ts, tx;
    simtype     status;
    event       *waitQ;
} link;

typedef struct
{   /** Data type for mapping schemes **/
    float   pcomp, pcomm;   /** Amounts of production-phase tasks **/
    float   lcomp, lcomm;   /** Amounts of learning-phase tasks **/
} mapping;

typedef struct
{   /** Data type for machines **/
    simtype     status;             /** Busy status or idle status **/
    simtype     phase;              /** Production or learning **/
```



```
    int         kid;                /** Currently simulated cluster **/
    float       tc, noload_tc;      /** Execution time per unit computation **/
    float       wlparam[NUM_WL_PARAM];  /** Workload descriptor **/
    mapping     *MAP;               /** Mapping scheme **/
} machine;

typedef struct
{   /** Data type for ANN neural clusters **/
    int         num;                /** Number of neurons **/
    int         num_pred, num_succ; /** Number of neighbors **/
    int         *pred, *succ;       /** Neighbors **/
} cluster;

/** Array of machines **/
#definexMACHINE(m)      (MACHINE + m)

/** Array of links **/
#definexLINK(l)         (LINK + l)

/** Access to workload descriptors **/
#definexWL(m,which)     (xMACHINE(m)->wlparam[which])

/** Access to mapping schemes **/
#definexMAP(m)          (xMACHINE(m)->MAP)

/** Array of neural clusters **/
#definexCLUSTER(k)      (CLUSTER + k)

/** Interconnection matrix **/
#definexICNET(i,j)      (ICNET + i * NumMachines + j)

/** Interconnection links **/
#definexICLINK(i,j)     (xLINK(xICNET(i,j)->l))

/** Array of machine status **/
#definexMSA(m)          (*(MSA + m))

/** Data matrix **/
#definexDATA(m1,m2,k)   (*(DATA + m1 * NumMachines * NumClusters + m2 * NumClusters + k))

#definis_input_layer(kid)       (kid <= InClusterTh)
#definis_2nd_input_layer(kid)   (kid <= In2ndClusterTh)
#definis_output_layer(kid)      (kid >= OutClusterTh)

#endif _DSIM_H__
```

---

dsim/var.c

```
/** This file contains important global variables. **/
```

```
int     NumClusters = 0;        /** Number of ANN neural clusters **/
int     NumMachines = 0;        /** Number of processors **/
int     NumLinks = 0;           /** Number of communication links **/
int     NumBcasts = 0;          /** Number of broadcasts **/
int     NumQuantumns = -1;      /** Number of simulation iterations **/
int     Quantumn = 0;           /** Iteration timer **/
long    NumEvents = 0;          /** Number of events created **/

int     RandomSeed = 1;         /** Random seed to a random generator **/
float   ErrorAllowance = 0.0;   /** User-specified error allowance **/
char    *MIMD_Sim = NULL;       /** Input filename pointer **/
char    *MIMD_Slot = NULL;      /** Input filename pointer **/
char    *MIMD_Map = NULL;       /** Input filename pointer **/
char    *ANN_Sim = NULL;        /** Input filename pointer **/
char    *ANN_Map = NULL;        /** Input filename pointer **/
char    *MAP_Sim = NULL;        /** Input filename pointer **/

int     InClusterTh = 0;        /** Input cluster threshold **/
int     In2ndClusterTh = 0;     /** 2nd input cluster threshold **/
int     OutClusterTh = 0;       /** Output cluster threshold **/

float   BcastTs = 0.0;          /** Broadcast setup time **/
float   BcastTx = 0.0;          /** Broadcast transmission time **/
int     BcastFrom = 0;          /** Broadcast coverage **/
int     BcastTo = 0;            /** Broadcast coverage **/

machine *MACHINE = NULL;        /** Array of machines **/
link    *LINK = NULL;           /** Array of links **/
icnet   *ICNET = NULL;          /** Interconnection net **/
cluster *CLUSTER = NULL;        /** Array of clusters **/
simtime SimTime = 0.0;          /** Simulation time **/
simtime AvgSimTime = 0.0;       /** Average simulation time **/
simtime PredSimTime = 0.0;      /** Predicted simulation time **/
simtime MapTime = 0.0;          /** Mapping time **/
simtime CST = 0.0;              /** Cumulative simulation time **/
simtime CMT = 0.0;              /** Cumulative mapping time **/
event   *EventQ = NULL;         /** Head of an event queue **/
int     *MSA = NULL;            /** Machine status array **/
int     *DATA = NULL;           /** 3-dimensional data matrix */
```

dsim/main.c

```
/** command: dsim  #quantumns random-seed error debug **/

main (argc, argv)
        int argc;
        char *argv[];
{       FILE *fp, *fopen ();
```



```
        MIMD_Sim =  (char *) malloc (MAX_FNAME_LENGTH * sizeof (char));
        MIMD_Slot = (char *) malloc (MAX_FNAME_LENGTH * sizeof (char));
        MIMD_Map =  (char *) malloc (MAX_FNAME_LENGTH * sizeof (char));
        ANN_Sim =   (char *) malloc (MAX_FNAME_LENGTH * sizeof (char));
        ANN_Map =   (char *) malloc (MAX_FNAME_LENGTH * sizeof (char));
        MAP_Sim =   (char *) malloc (MAX_FNAME_LENGTH * sizeof (char));

        fp = fopen ("FILES", "r");
        if (! get_line (MIMD_Sim,  MAX_FNAME_LENGTH, fp)) error ("MIMD_Sim",  NOP);
        if (! get_line (MIMD_Slot, MAX_FNAME_LENGTH, fp)) error ("MIMD_Slot", NOP);
        if (! get_line (MIMD_Map,  MAX_FNAME_LENGTH, fp)) error ("MIMD_Map",  NOP);
        if (! get_line (ANN_Sim,   MAX_FNAME_LENGTH, fp)) error ("ANN_Sim",   NOP);
        if (! get_line (ANN_Map,   MAX_FNAME_LENGTH, fp)) error ("ANN_Map",   NOP);
        if (! get_line (MAP_Sim,   MAX_FNAME_LENGTH, fp)) error ("ANN_Sim",   NOP);
        fclose (fp);

        if (argc >= 2) if (*(argv[1]) != '*') NumQuantumns = atoi (argv[1]);
        if (argc >= 3) if (*(argv[2]) != '*') RandomSeed = atoi (argv[2]);
        if (argc >= 4) if (*(argv[3]) != '*') sscanf (argv[3], "%f", &ErrorAllowance);

        read_cluster ();
        read_machine ();
        if (ErrorAllowance >= 0.0) { init_sim(); sim (); } else workload_only ();
}


read_machine ()
/** Read in the specifications of machines. **/
{       FILE *fp; int i, j, k, count, num, m, l, sum = 0;
        float tc, ts, tx, comp, comm, intrend, outtrend; char fname[30];

        fp = fopen (MIMD_Sim, "r");
        /** Read in the specification of mahcine. **/
        fscanf (fp, "%d", &NumBcasts);
        if (NumBcasts) fscanf (fp, "%f %f %d %d", &BcastTs, &BcastTx, &BcastFrom, &BcastTo);
        fscanf (fp, "%d %d", &NumMachines, &NumLinks);
        MACHINE = (machine *) malloc (NumMachines * sizeof (machine));
        LINK = (link *) malloc (NumLinks * sizeof (link));
        ICNET = (icnet *) malloc (NumMachines * NumMachines * sizeof (icnet));
        for (i = 0; i < NumMachines; i++)
                xMAP(i) = (mapping *) malloc (NumClusters * sizeof (mapping));

        /** Read in execution time per unit computation. **/
        for (i = 0; i < NumMachines;) {
                fscanf (fp, "%d %f", &num, &tc);
                for (j = 0; j < num; j++, i++) xMACHINE(i)->noload_tc = tc;
        }

        /** Read in the specifications of links. **/
        for (i = 0; i < NumLinks;) {
                fscanf (fp, "%d %f %f", &num, &ts, &tx);
                for (j = 0; j < num; j++, i++) {
                        xLINK(i)->ts = ts; xLINK(i)->tx = tx;
```

```
                xLINK(i)->status = IDLE; xLINK(i)->waitQ = NULL;
        }
    }

    /** Read in specification of an interconnection net. **/
    for (k = 0; k < NumMachines * NumMachines;) {
        fscanf (fp, "%d %d %d", &num, &l, &m);
        for (count = 0; count < num; count++, k++) {
            i = k / NumMachines; j = k % NumMachines;
            if (i == j) { xICNET(i,j)->l = -1; xICNET(i,j)->m = i; }
            else { xICNET(i,j)->l = l; xICNET(i,j)->m = m; }
        }
    }
    fclose (fp);

    /** Generate workload descriptors. **/
    srand (RandomSeed);
    for (i = 0; i < NumMachines; i++) {
        intrend = gen_random_range (WL_IN_TREND_LOWB, WL_IN_TREND_UPB);
        outtrend = gen_random_range (WL_OUT_TREND_LOWB, WL_OUT_TREND_UPB);
        xWL(i,IN_TREND) = intrend;
        xWL(i,OUT_TREND) = outtrend;
        xWL(i,STRAIGHT) = WL_STRAIGHT;
        xWL(i,SLOPE) = gen_random_range (WL_YMIN_SLOPE, WL_YMAX_SLOPE)
            / gen_random_range (WL_XMIN_SLOPE, WL_XMAX_SLOPE);
        xWL(i,LOWB) = WL_LOWER_BOUND;
        xWL(i,UPB) = WL_UPPER_BOUND;
        xWL(i,TREND) = gen_random_int (-1, 1);
    }
    srand (RandomSeed);
}


read_cluster ()
/** Read in ANN neural clusters **/
{   FILE *fp; int kid, i, j, n, k;

    fp = fopen (ANN_Sim, "r");
    fscanf (fp, "%d %d %d %d", &NumClusters, &InClusterTh, &In2ndClusterTh, &OutClusterTh);
    CLUSTER = (cluster *) malloc (NumClusters * sizeof (cluster));
    for (i = 0; i < NumClusters; i++) {
        fscanf (fp, "%d %d", &kid, &n);
        xCLUSTER(kid)->num = n;
        fscanf (fp, "%d", &n);
        xCLUSTER(kid)->num_pred = n;
        if (n > 0) {
            xCLUSTER(kid)->pred = (int *) malloc (n * sizeof (int));
            for (j = 0; j < n; j++) {
                fscanf (fp, "%d", &k);
                *(xCLUSTER(kid)->pred + j) = k;
            }
        }
        fscanf (fp, "%d", &n);
```



```
        xCLUSTER(kid)->num_succ = n;
        if (n > 0) {
            xCLUSTER(kid)->succ = (int *) malloc (n * sizeof (int));
            for (j = 0; j < n; j++) {
                fscanf (fp, "%d", &k);
                *(xCLUSTER(kid)->succ + j) = k;
            }
        }
    }
}


gen_new_workload ()
/** Generate new workload based on its workload descriptor. **/
{   int i; float noload_tc, wl, tc, p;
    float intrend, outtrend, straight, slope, lowb, upb, up, down;

    for (i = 0; i < NumMachines; i++) {
        intrend = xWL(i,IN_TREND);
        outtrend = xWL(i,OUT_TREND);
        straight = xWL(i,STRAIGHT);
        slope = xWL(i,SLOPE);
        lowb = xWL(i,LOWB);
        upb = xWL(i,UPB);
        wl = (tc = xMACHINE(i)->tc) / (noload_tc = xMACHINE(i)->noload_tc);

        if (wl == lowb) { up = intrend + outtrend; down = 0.0; } else
        if (wl == upb) { up = 0.0; down = intrend + outtrend; }
        else {
            switch (xWL(i,TREND)) {
                case -1: /** down **/
                    up = outtrend; down = intrend; break;
                case 0: /** straight **/
                    up = intrend; down = outtrend; break;
                case 1: /** up **/
                    up = intrend; down = outtrend; break;
                default:
                    error ("gen_new_workload: no such trend");
                    break;
            }
        }

        p = gen_random_float ();
        if (p < up) { wl += slope; xWL(i,TREND) = 1; } else
        if (p < up + down) { wl -= slope; xWL(i,TREND) = -1; }
        else { xWL(i,TREND) = 0; }

        if (wl > upb) wl = upb; else
        if (wl < lowb) wl = lowb;
        xMACHINE(i)->tc = wl * noload_tc;
    }
}
```

```
workload_only ()
/** Investigate workload only. **/
{       int i; char fname[30]; FILE *fp;
#defineTRANSIENT_PHASE          500
        sprintf (fname, "WL-%d-%d", NumMachines, RandomSeed);
        fp = fopen (fname, "w");
        for (Quantumn = 0; Quantumn < NumQuantumns + TRANSIENT_PHASE; Quantumn++) {
                gen_new_workload ();
                if (Quantumn >= TRANSIENT_PHASE) {
                        fprintf (fp, "%d  ", Quantumn);
                        for (i = 0; i < NumMachines; i++)
                                fprintf (fp, " %f", xMACHINE(i)->tc / xMACHINE(i)->noload_tc);
                        fprintf (fp, "\n");
                }
        }
#undef TRANSIENT_PHASE
}


int gen_random_int (left, right)
/** Generate a random integer over [left,right]. **/
        int left, right;
{ return (left + ((int) (gen_random_float() * (right - left + 1)))); }


float gen_random_float ()
/** Generate a random floating number over [0.0,1.0). **/
{ return (((float) rand()) / ((float) (0x7fffffff))); }


float gen_random_range (left, right)
/** Generate a random floating number over [left,right). **/
        float left, right;
{ return (left + gen_random_float() * (right - left)); }


error (str, val)
/** Report an error and exit. **/
        char *str; int val;
{       char buf[200];

        if (val == NOP) sprintf (buf, "*** ERROR *** => %s", str);
        else sprintf (buf, "*** ERROR *** => %s %d", str, val);
        printf ("%s\n", buf);
        exit (0);
}


char *get_line (line, length, fp)
/** Read in a line. **/
        char line[]; int length; FILE *fp;
{       char *c, *p = fgets (line, length, fp);
```



```
        for (c = line; *c; c++) if (*c == '\n') { *c = NULL; break; }
        return p;
}
```

---

dsin/sim.c

```
init_sim ()
/** Initialize simulation environment. **/
{
        MSA = (int *) malloc (NumMachines * sizeof (int));
        DATA = (int *) malloc (NumMachines * NumMachines * NumClusters * sizeof (int));
}


sim ()
/** Simulate parallel ANN simulations. **/
{       event *e;

        for (Quantumn = 0; Quantumn < NumQuantumns; Quantumn++) {
                gen_new_workload ();
#ifdef DMAPPING
                if (need_remapping ()) remapping ();
#elif GMAX
                remapping ();
#else
                if (Quantumn == 0) remapping ();
#endif
                pre_sim ();
                schedule_startup_events ();

                while ((e = get_event ())) {
                        if (SimTime <= e->time) {
                                SimTime = e->time;
                                switch (e->type) {
                                        case COMP: sim_comp (e); break;
                                        case NEXT: sim_next (e); break;
                                        case COMM: sim_comm (e); break;
                                        case ARRIVE: sim_arrive (e); break;
                                        case BCAST: sim_bcast (e); break;
                                        case BCAST_ARRIVE: sim_bcast_arrive (e); break;
                                        default: break;
                                }
                        }
                }
                post_sim ();
        }
}
```

```
pre_sim ()
/** Do preprocessing for simulation. **/
{       int m0, m3, k, l;

        SimTime = (simtime) 0; EventQ = NULL;
        for (m0 = 0; m0 < NumMachines; m0++) xMSA(m0) = 0;
        for (m0 = 0; m0 < NumMachines; m0++)
                for (m3 = 0; m3 < NumMachines; m3++)
                        for (k = 0; k < NumMachines; k++) xDATA(m0,m3,k) = 0;

        for (l = 0; l < NumLinks; l++) { xLINK(l)->waitQ = NULL; xLINK(l)->status = IDLE; }
}


post_sim ()
/** Do postprocessing for simulation. **/
{       FILE *fp; int m;

        CST += SimTime;
#ifdef STATISTICS
        /** Report workload parameters. **/
        if (Quantumn == 0) {
                fp = fopen ("wl-param", "w");
                for (m = 0; m < NumMachines; m++)
                        fprintf (fp, "%f %f %f %f %f %f\n",
                        xWL(m,STRAIGHT), xWL(m,IN_TREND), xWL(m,OUT_TREND),
                        xWL(m,SLOPE), xWL(m,LOWB), xWL(m,UPB));
                fclose (fp);
        }

        /** Report workload. **/
        fp = fopen ("wl-pf", "a");
        fprintf (fp, "%d  ", Quantumn);
        for (m = 0; m < NumMachines; m++) fprintf (fp, "%f  ", xMACHINE(m)->tc);
        fprintf (fp, "\n");
        fclose (fp);
#endif
        /** Print simulation time (simtime) in unit of millisecond. **/
#ifdef DMAPPING
        fp = fopen ("dsimtime-pf", "a");
#elif GMAX
        fp = fopen ("gmax-pf", "a");
#else
        fp = fopen ("simtime-pf", "a");
#endif
        if (Quantumn % 10 == 0)
                fprintf (fp, "%d    %g  %g  %g    %g  %g\n",
                        Quantumn, CST, SimTime, PredSimTime, CMT, MapTime);
        fclose (fp);
}


schedule_startup_events ()
```



```
/** Create startup events. **/
{       event *e; machine *mp; int m;

        for (m = 0; m < NumMachines; m++) {
                mp = xMACHINE(m); mp->kid = 0;
                mp->phase = PRODUCTION; mp->status = IDLE;
                create_event (COMP, SimTime, m, -1, -1 ,-1, 0, PRODUCTION);
        }
}


sim_comp (e)
/** Simulate computation. **/
        event *e;
{       int m0 = e->m0; simtime t; simtype phase; event *new_e;

        xMACHINE(e->m0)->status = BUSY;
        calc_comp_time (&t, m0, e->kid, e->phase);
        recreate_event (e, NEXT, SimTime + t, m0, -1, -1, -1, e->kid, e->phase);
}


sim_comm (e)
/** Simulate communication. **/
        event *e;
{       int m0 = e->m0, m1 = e->m1, m2, m3 = e->m3; simtime t;

        m2 = xICNET(m1,m3)->m;
        calc_comm_time (&t, m1, m2, e->kid, e->phase);
        schedule_comm (t, m0, m1, m2, m3, e->kid, e->phase);
        free_event (e);
}


sim_bcast (e)
/** Simulate broadcast. **/
        event *e;
{       simtime t;

        calc_bcast_time (&t, e->m0, e->kid, e->phase);
        recreate_event (e, BCAST_ARRIVE, SimTime + t, e->m0, -1, -1, -1, e->kid, e->phase);
}


sim_arrive (e)
/** Simulate arrival of a frame. **/
        event *e;
{       int m0 = e->m0, m1 = e->m1, m2 = e->m2, m3 = e->m3, l;
        machine *m3p = xMACHINE(e->m3); simtime t;

        l = xICNET(m1,m2)->l;
        xLINK(l)->status = IDLE;
        if (m2 == m3) {
```

```
            xDATA(m0,m3,e->kid) = 1;

            /** If the target machine is idle now, try to trigger it. **/
            if (m3p->status == IDLE) {
                if (is_ready_to_sim (m3))
                create_event (COMP, e->time, m3, -1, -1, -1, m3p->kid, m3p->phase);
            }
        } else {
            m1 = m2;
            m2 = xICNET(m1,m3)->m;
            calc_comm_time (&t, m1, m2, e->kid, e->phase);
            schedule_comm (t, m0, m1, m2, m3, e->kid, e->phase);
        }
        free_event (e);
        wakeup_comm (l);
}


sim_bcast_arrive (e)
/** Simulate arrival of a frame due to broadcast. **/
        event *e;
{       int m; machine *mp;

        for (m = BcastFrom; m <= BcastTo; m++) xDATA(e->m0,m,e->kid) = 1;
        /** If the target machine is idle now, try to trigger it. **/
        for (m = BcastFrom; m <= BcastTo; m++) {
            mp = xMACHINE(m);
            if (mp->status == IDLE && is_ready_to_sim (m))
                create_event (COMP, e->time, m, -1, -1, -1, mp->kid, mp->phase);
        }
        free_event (e);
}


sim_next (e)
/** Simulate ready-for-computation. **/
        event *e;
{       int m0 = e->m0, m3, n, *arr;
        float comp; machine *m0p = xMACHINE(e->m0);

        /** Send outputs. **/
        if (get_succ_machines (MSA, &n, &arr, e->kid, e->phase)) {
            if (NumBcasts && is_in_bcast_range (m0, MSA, NumMachines)) {
                create_event (BCAST, SimTime, m0, -1, -1, -1, e->kid, e->phase);
            } else {
                for (m3 = 0; m3 < NumMachines; m3++) {
                    if (xMSA(m3)) {
                        if (m0 == m3) xDATA(m0,m3,e->kid) = 1;
                        else create_event (COMM, SimTime, m0, m0, -1, m3, e->kid, e->phase);
                    }
                }
            }
        }
```



```
        m0p->status = IDLE;
        do {
            if (get_next_cluster (m0p->kid, m0p->phase, &(m0p->kid), &(m0p->phase)) == FALSE) {
            /** No more computation is scheduled. **/
                free_event (e);
                return;
            }
            comp = (m0p->phase == PRODUCTION) ?
            (xMAP(m0) + m0p->kid)->pcomp : (xMAP(m0) + m0p->kid)->lcomp;
        } while (comp <= 0.0);
        if (is_ready_to_sim (m0))
            recreate_event (e, COMP, SimTime, m0, -1, -1, -1, m0p->kid, m0p->phase);
        else free_event (e);
}


schedule_comm (t, m0, m1, m2, m3, kid, phase)
/** Schedule communication events. **/
        simtime t; int m0, m1, m2, m3, kid; simtype phase;
{       event *waitQ = xICLINK(m1,m2)->waitQ, *p, *e;

        e = alloc_event (ARRIVE, t, m0, m1, m2, m3, kid, phase);
        if (waitQ) { for (p = waitQ; p->next; p = p->next) ; p->next = e; }
        else xICLINK(m1,m2)->waitQ = e;
        wakeup_comm (xICNET(m1,m2)->l);
}


wakeup_comm (l)
/** Wake up a queued communication event. **/
        int l;
{       event *e;

        if (l < 0) error ("wakeup_comm: no such link", l);
        if (xLINK(l)->status == IDLE && xLINK(l)->waitQ) {
            xLINK(l)->status = BUSY;
            xLINK(l)->waitQ = (e = xLINK(l)->waitQ)->next;
            e->time += SimTime;
            schedule_event (e);
        }
}


boolean is_ready_to_sim (m)
/** Check if this machine is idle and has enough inputs to start. **/
        int m;
{       int i, j, n, *arr; machine *mp = xMACHINE(m);

        if (  (is_input_layer (mp->kid) && mp->phase == PRODUCTION)
            || (is_output_layer (mp->kid) && mp->phase == LEARNING))
            return TRUE;
        if (get_pred_machines (MSA, &n, &arr, mp->kid, mp->phase) == FALSE) return FALSE;
        for (i = 0; i < NumMachines; i++)
```

```
            if (xMSA(i))
                for (j = 0; j < n; j++)
                    if (xDATA(i,m,*(arr+j)) == 0) return FALSE;
    return TRUE;
}


boolean get_next_cluster (kid, phase, new_kidp, new_phasep)
/** Find what is the next neural cluster to simulate. **/
    int kid, *new_kidp; simtype phase, *new_phasep;
{
    switch (phase) {
        case PRODUCTION:
            if (kid >= OutClusterTh)
            { *new_phasep = LEARNING; *new_kidp = kid; }
            else { *new_phasep = PRODUCTION; *new_kidp = kid + 1; }
            break;
        case LEARNING:
            if (kid <= InClusterTh) return FALSE;
            *new_phasep = LEARNING;
            *new_kidp = kid - 1;
            break;
        default:
            return FALSE;
            break;
    }
    return TRUE;
}


boolean get_pred_clusters (kid, phase, n, arr)
/** Find the predecessors of this neural cluster. **/
    int kid; simtype phase; int *n, **arr;
{
    *n = 0; *arr = NULL;
    switch (phase) {
        case PRODUCTION:
            if (is_input_layer (kid)) return FALSE;
            *n = xCLUSTER(kid)->num_pred;
            *arr = xCLUSTER(kid)->pred;
            break;
        case LEARNING:
            if (is_output_layer (kid)) return FALSE;
            *n = xCLUSTER(kid)->num_succ;
            *arr = xCLUSTER(kid)->succ;
            break;
        default:
            return FALSE;
            break;
    }
    return TRUE;
}
```



```
boolean get_succ_clusters (kid, phase, n, arr)
/** Find the successors of this neural cluster. **/
    int kid; simtype phase; int *n, **arr;
{
    *n = 0; *arr = NULL;
    switch (phase) {
        case PRODUCTION:
            if (is_output_layer (kid)) return FALSE;
            *n = xCLUSTER(kid)->num_succ;
            *arr = xCLUSTER(kid)->succ;
            break;
        case LEARNING:
            if (is_2nd_input_layer (kid)) return FALSE;
            *n = xCLUSTER(kid)->num_pred;
            *arr = xCLUSTER(kid)->pred;
            break;
        default:
            return FALSE;
            break;
    }
    if (*n <= 0) return FALSE;
    return TRUE;
}


boolean get_pred_machines (msa, n, arr, kid, phase)
/** Find the machines corresponding to the predecessors of
 ** this cluster. **/
    int msa[], *n, **arr, kid; simtype phase;
{   int i, k, m;

    for (m = 0; m < NumMachines; m++) msa[m] = 0;
    if (get_pred_clusters (kid, phase, n, arr) == FALSE) return FALSE;
    if (*n <= 0) return FALSE;
    for (m = 0; m < NumMachines; m++)
        for (i = 0; i < *n; i++) {
            k = *(*arr + i);
            if ((msa[m] = ((xMAP(m) + k)->pcomp > 0.0) ? 1 : 0)) break;
        }
    return TRUE;
}


boolean get_succ_machines (msa, n, arr, kid, phase)
/** Find the machines corresponding to the successors of
 ** this cluster. **/
    int msa[], *n, **arr, kid; simtype phase;
{   int i, m, k;

    for (m = 0; m < NumMachines; m++) msa[m] = 0;
    if (get_succ_clusters (kid, phase, n, arr) == FALSE) return FALSE;
    if (*n <= 0) return FALSE;
    for (m = 0; m < NumMachines; m++)
```

```
        for (i = 0; i < *n; i++) {
            k = *(*arr + i);
            if ((msa[m] = ((xMAP(m) + k)->pcomp > 0.0) ? 1 : 0)) break;
        }
    return TRUE;
}


calc_comp_time (tp, m, kid, phase)
/** Calculate the computation time of cluster kid on machine m. **/
    simtime *tp; int m, kid; simtype phase;
{   float comp = 0.0; *tp = (simtime) 0;

    comp = (phase == PRODUCTION) ? (xMAP(m) + kid)->pcomp : (xMAP(m) + kid)->lcomp;
    if (comp > 0.0) *tp = comp * xMACHINE(m)->tc;
}


calc_comm_time (tp, m1, m2, kid, phase)
/** Calculate the communication time for cluster kid
 ** between machines m1 and m2. **/
    simtime *tp; int m1, m2, kid; simtype phase;
{   float comm = 0.0; *tp = (simtime) 0;

    if (m1 != m2)
        comm = (phase == PRODUCTION) ? (xMAP(m1) + kid)->pcomm : (xMAP(m1) + kid)->lcomm;
    if (comm > 0.0) *tp += (xICLINK(m1,m2)->ts + comm * xICLINK(m1,m2)->tx);
}


calc_bcast_time (tp, m, kid, phase)
/** Calculate the broadcast time for cluster kid from machine m. **/
    simtime *tp; int m, kid; simtype phase;
{   float comm = (phase == PRODUCTION) ? (xMAP(m) + kid)->pcomm : (xMAP(m) + kid)->lcomm;
    *tp = BcastTs + comm * BcastTx;
}


boolean is_in_bcast_range (s, msa, n)
/** Check if machine s and destinations in array msa are all covered by
 ** this broadcast. **/
    int s, msa[], n;
{   int i;

    if (s < BcastFrom && s > BcastTo) return FALSE;
    for (i = 0; i < n; i++) if (msa[i] < BcastFrom && msa[i] > BcastTo) return FALSE;
    return TRUE;
}
```



```
event *get_event ()
/** Remove an event from the event queue and return it. **/
{   event *e;

    if (EventQ) EventQ = (e = EventQ)->next; else e = NULL;
    return e;
}


create_event (type, time, m0, m1, m2, m3, kid, phase)
/** Create a new event and insert it into the event queue. **/
    simtype type; simtime time; int m0, m1, m2, m3, kid; simtype phase;
{ schedule_event (alloc_event (type, time, m0, m1, m2, m3, kid, phase)); }


recreate_event (e, type, time, m0, m1, m2, m3, kid, phase)
/** Re-use event e and insert it into the event queue. **/
    event *e; simtype type; simtime time;
    int m0, m1, m2, m3, kid; simtype phase;
{
    reuse_event (e, type, time, m0, m1, m2, m3, kid, phase);
    schedule_event (e);
}


schedule_event (e)
/** Insert event e into the event queue. **/
    event *e;
{   event *p0, *p;

    if (EventQ) {
        for (p0 = p = EventQ; p; p = (p0 = p)->next)
            if (e->time < p->time) {
                if (p0 == p) (EventQ = e)->next = p;
                else (p0->next = e)->next = p;
                return;
            }
        (p0->next = e)->next = NULL;
    }
    else (EventQ = e)->next = NULL;
}


event *_e_pool_mgr = NULL;
event *_alloc_event ()
/** Allocate a free space of an event from the free list manager. **/
{   event *e; int i;
```

```
    if (_e_pool_mgr == NULL) {
        _e_pool_mgr = (event *) malloc (1000 * sizeof (event));
        for (i = 0; i < 1000; i++) (_e_pool_mgr+i)->next = _e_pool_mgr + i + 1;
        (_e_pool_mgr+999)->next = NULL;
    }
    _e_pool_mgr = (e = _e_pool_mgr)->next;
    return e;
}

free_event (e) event *e; { e->next = _e_pool_mgr; _e_pool_mgr = e; }

event *alloc_event (type, time, m0, m1, m2, m3, kid, phase)
/** Allocate an event with proper parameters filled. **/
    simtype type; simtime time;
    int m0, m1, m2, m3, kid; simtype phase;
{   event *e = _alloc_event ();

    NumEvents++;
    e->next = NULL; e->type = type; e->time = time;
    e->m0 = m0; e->m1 = m1; e->m2 = m2; e->m3 = m3;
    e->kid = kid; e->phase = phase;
    return e;
}

reuse_event (e, type, time, m0, m1, m2, m3, kid, phase)
/** Re-use the event e by re-feeding parameters. **/
    event *e; simtype type; simtime time;
    int m0, m1, m2, m3, kid; simtype phase;
{
    e->next = NULL; e->type = type; e->time = time;
    e->m0 = m0; e->m1 = m1; e->m2 = m2; e->m3 = m3;
    e->kid = kid; e->phase = phase;
}
```

---

dsim/map.c

```
boolean need_remapping ()
/** Check if re-mapping is advantageous. **/
{   int q_rem = NumQuantumns - Quantumn;

    if (Quantumn == 0) return TRUE;
    /** Average from the first to the previous one **/
    AvgSimTime = (AvgSimTime * (Quantumn - 1) + SimTime) / Quantumn;
    return ((q_rem * (SimTime - AvgSimTime) >= MapTime) ? TRUE : FALSE);
}
```



```
remapping ()
/** Decide to call NeuMap for a new mapping scheme.
 ** Handshake with NeuMap. **/
{   int d = 0, delta = 0.0; FILE *fp; char cmd[100];

    construct_MIMD_Map ();
    do {    /** Remove old results. **/
        system("rm map-sim");
        /** Call NeuMap to solve optimal mapping. **/
        sprintf (cmd, "neumap %s %s %f %d", MIMD_Map, ANN_Map, ErrorAllowance + delta, d);
#ifdef MONITOR
        printf ("EXECUTING COMMAND ==> %s\n", cmd);
#endif
        system (cmd);
        if ((fp = fopen (MAP_Sim, "r"))) { fclose (fp); break; } else delta += 0.1;
    } while (1);
    read_MAP_Sim ();
}

construct_MIMD_Map ()
/** Construct the specification of a multicomputer such that the
 ** specification can be understood by NeuMap. **/
{   int m; FILE *fpr, *fpw; char *c, buf[500];

    /** read MIMD_Slot and generate MIMD_Map **/
    fpr = fopen (MIMD_Slot, "r");
    fpw = fopen (MIMD_Map, "w");
    while (fgets (buf, 500, fpr)) {
        for (c = buf; *c; c++)
            if (*c == '$') {
                c++; sscanf (c, "%d", &m);
                fprintf (fpw, " %f ", xMACHINE(m)->tc);
                while (*c >= '0' && *c <= '9') c++;
            } else fprintf (fpw, "%c", *c);
    }
    fclose (fpr); fclose (fpw);
}

read_MAP_Sim ()
/** Read in a new mapping scheme. **/
{   FILE *fp; int m, nk, i, j; float comp, comm;

    /** Read in a new mapping scheme. **/
    fp = fopen (MAP_Sim, "r");
    fscanf (fp, "%f %f", &PredSimTime, &MapTime);
    MapTime *= (MACHINE_WEIGHT * 1000.0 / 60.0);
    CMT += MapTime;
    for (i = 0; i < NumMachines; i++) {
        for (j = 0; j < NumClusters; j++) {
            fscanf (fp, "%d %d", &m, &nk);
            fscanf (fp, "%f %f", &comp, &comm);
```

```
                (xMAP(m) + nk)->pcomp = comp;
                (xMAP(m) + nk)->pcomm = comm;
                fscanf (fp, "%f %f", &comp, &comm);
                (xMAP(m) + nk)->lcomp = comp;
                (xMAP(m) + nk)->lcomm = comm;
        }
    }
    fclose (fp);
}
```


APPENDIX C.

PARALLEL ANN SIMULATIONS

This appendix mainly describes parallel ANN simulations on a 16-node iPSC/2 hypercube computer and a network of three workstations, and describes how to use them. Section C.1 describes briefly their organizations. Section C.2 describes how to use them. Finally, Section C.3 lists the programs.

C.1. General Description

A parallel ANN simulation is written based on the mapping result of NeuMap. Two simulation programs are described here. One is on the 16-node iPSC/2 hypercube computer. The other is on the network of three workstations.

For the one on a 16-node iPSC/2 hypercube computer, it includes the following files:

host.c For a program to be able to run on an iPSC/2 hypercube, there must have a cooperating routine which is run on the Cube Manager. This cooperating routine is mainly used to load and start the program on each participating node.

node.c This is the program which is run on each participating node. It includes the main simulation body.

map.h The declarations of a mapping scheme onto a hypercube are included in it.

def.h Several constants particular for a hypercube computer and some debugging shorthands are included in it.

comp.h Important computation-related shorthands are included in it.

comm.h Important communication-related shorthands are included in it.

For the one on the network of three workstations, it includes the following files:

node.c It includes the main simulation body.

comm.c The utilities for stream-based communication are in it.

comp.h Important computation-related shorthands are included in it.

comm.h Important communication-related shorthands are included in it.

C.2. Steps for Using Simulations

The simulations on hypercubes for different ANNs are stored in different directories. For fully connected multilayered ANNs, they are in ~software/chu/ms/ipsc/full-net. For hybrid multilayered ANNs, they are in ~software/chu/ms/ipsc/multi-layer. For nonlayered ANNs, they are in ~software/chu/ms/ipsc/non-layered. Here, take ML-1 as an example. Follow these steps to execute the supplied simulation of ML-1 on a 16-node iPSC/2 hypercube computer.

(1) Go to the directory ~software/chu/ms/ipsc/multi-layer.

(2) Compile the simulation program by typing "make all" which will generate executable files *host* and *node*.

(4) Create a 16-node cube by typing "getcube -t 16m4" which allocates a 16-node hypercube in which each node has 4 mega-byte memory.

(5) Run the program by typing "host"

The simulations on the network of three workstations for different ANNs are also stored in different directories. For fully connected multilayered ANNs, they are in ~software/chu/ms/sun-3/full-net. For hybrid multilayered ANNs, they are in ~software/chu/ms/sun-3/multi-layer. For nonlayered ANNs, they are in ~software/chu/ms/sun-3/non-layered. Here, take ML-1 as an example. Follow these steps to execute the supplied simulation of ML-1 on the network of three workstations.

(1) Go to the directory ~software/chu/ms/sun-3/multi-layer.

(2) Compile the simulation program by typing "make node-1" which will generate an executable file *node-1*.

(3) Run the program node-1 on three workstations by typing "node-1" and supply a node identifier for each workstation, for example, 0 for workstation 0, 1 for workstation 1, and 2 for workstation 2.

C.3. Listing of Parallel ANN Simulations on a 16-node iPSC/2 Hypercube

~software/chu/ms/ipsc/multi-layer/Makefile.c

```
CFLAGS = -O
LDFLAGS = -O

HLIB=/usr/ipsc/lib/chost.a -host /lib/libm.a
NLIB=/usr/ipsc/lib/Llibcnode.a -node  /lib/libm.a

all: host node

NODE_OBJECTS=   node.o
HOST_OBJECTS=   host.o

node:  ${NODE_OBJECTS}
        cc -o node ${NODE_OBJECTS} ${NLIB}

host:   ${HOST_OBJECTS}
        cc -o host ${HOST_OBJECTS} ${HLIB}

clean:
        rm -f *.o host node
```

~software/chu/ms/ipsc/multi-layer/host.c

```
#include        "../src/def.h"
#include        "map.h"

char            debug[100];

main ()
{       /** This is the host process at the host node. **/
        long start_time, exec_time, ack, t_buffer[3], time_buffer[NODES][2];
        FILE *outfp, *fopen(); int i, n, iter;

        /** Set the host process's ID to HOST_PID. **/
        setpid (HOST_PID);

        /** Load the executable file node to process NODE_PID on each node. **/
        load ("node", ALL_NODE, NODE_PID);

        /** Send input signals to all nodes. **/
        for (iter = 0; iter < ITER; iter++) {
                csend (INPUT_TYPE, inbuf, nbytes * NUM_INPUT, ALL_NODE, NODE_PID);
                csend (TEACH_TYPE, tbuf, nbytes * NUM_OUTPUT, ALL_NODE, NODE_PID);
```

```
    }
}
```

~software/chu/ms/ipsc/src/def.h

```
#include    <stdio.h>
#include    <cube.h>
#include    <math.h>

#defineHOST_PID         100
#defineNODE_PID         1
#defineALL_NODE         -1
#defineALL_PID          -1

#defineEND_TYPE         0
#defineINPUT_TYPE       1
#defineOUTPUT_TYPE      2
#defineTEACH_TYPE       3
#defineERROR_TYPE       4
#defineACK_TYPE         5
#defineDEBUG_TYPE       6

#defineITER             1
```

~software/chu/ms/ipsc/multi-layer/map.h

```
/** ANN Model **/

#defineN1               250
#defineN2               100
#defineN3               250
#defineN4               150
#defineN5               200
#defineN6               300
#defineN7               100

#defineN12              (N1 + N2)
#defineN13              (N12 + N3)
#defineN14              (N13 + N4)
#defineN15              (N14 + N5)
#defineN16              (N15 + N6)
#defineN17              (N16 + N7)

#defineNEURONS          (N17)
#defineCLUSTERS         7
```



```
#defineLAYERS           4

#define NUM_INPUT       (N1)
#define NUM_OUTPUT      (N7)

int     nk[CLUSTERS] = { N1, N2, N3, N4, N5, N6, N7 };
int     nkfrom[CLUSTERS] = { 0, N1, N12, N13, N14, N15, N16 };
int     nkto[CLUSTERS] = { N1-1, N12-1, N13-1, N14-1, N15-1, N16-1, N17-1 };

float   w01[N1][N1], w12[N1][N2], w13[N1][N3], w14[N1][N4], w25[N2][N5];
float   w35[N3][N5], w36[N3][N6], w46[N4][N6], w57[N5][N7], w67[N6][N7];

/** Multicomputer Model **/

#ifdef  SEQUENTIAL
#defineNODES            1
#else
#defineNODES            16
#endif

/** Mapping **/

int     asgn[NODES][CLUSTERS] =
{
#ifdef  SEQUENTIAL
        { 250, 100, 250, 150, 200, 300, 100 }
#else
        {  16,  7,  15,  8,  12,  18,  5 },
        {  16,  7,  15,  8,  12,  18,  5 },
        {  16,  7,  15,  8,  12,  18,  5 },
        {  16,  7,  15,  8,  12,  18,  5 },
        {  16,  6,  16,  9,  12,  18,  6 },
        {  16,  6,  16,  9,  12,  18,  6 },
        {  16,  6,  16,  9,  12,  18,  6 },
        {  16,  6,  16,  9,  12,  18,  6 },
        {  16,  6,  15,  10,  13,  19,  5 },
        {  16,  6,  15,  10,  13,  19,  5 },
        {  15,  6,  16,  11,  13,  19,  7 },
        {  15,  6,  16,  11,  13,  19,  7 },
        {  15,  6,  16,  10,  13,  20,  8 },
        {  15,  6,  16,  10,  13,  20,  8 },
        {  15,  6,  16,  10,  13,  20,  8 },
        {  15,  6,  16,  10,  13,  20,  8 },
#endif
};

struct { int from, to; } map[NODES][CLUSTERS];

/** Simulation **/

#defineITER             1

float   out[NEURONS];           /** Outputs of neurons **/
```

```
float   delta[NEURONS];         /** Partial errors of neurons **/
float   theta[NEURONS];         /** Threshold **/

float   inbuf[NEURONS+2];       /** Input buffer **/
float   outbuf[NEURONS+2];      /** Output buffer **/
float   tbuf[NEURONS+2];        /** Teaching input buffer **/

float   gain;
int     my_id;
int     my_count[CLUSTERS];

/** Shorthands:
 ** to avoid intractability in prediction of comp time for exp,
 ** the comp time is set to be fixed. **/

#define float_exp(a)    ((float) exp ((double) (0.5)))
#define sigmoid(a)      (1.0 / (1.0 + float_exp (a)))
#define nbytes          (sizeof (float) / sizeof (char))
#define span(s,e)       (e - s)
```

~software/chu/ms/ipsc/src/comp.h

```
#define iPSC2_INIT()                                                    \
        my_node = mynode (); my_pid = mypid ();  host = myhost ();      \
        time_buffer[0] = my_node; outbuf[0] = (float) my_node;          \
        for (k = 0; k < CLUSTERS; k++) {                                \
                for (count = j = 0; j < my_node; j++) count += asgnt[j][k];  \
                my_count[k] = count;                                    \
        }

#define PROD_COMP_0(kid)                                                \
        count = 1;                                                      \
        for (i = map[my_node][kid-1].from; i <= map[my_node][kid-1].to; i++) {
                net = theta[i]; ii = i - nkfrom[kid-1];

#define PROD_COMP_1(kid)        outbuf[count++] = out[i] = sigmoid(net); }

#define COMP_NET_INPUT(pred,succ,w_matrix,source,offset)                \
        for (k = 0; k < nk[pred-1]; k++) net += (w_matrix [k][ii] * source [k+offset]);

#define MEASURE_0(kid,phase)    start_time = mclock ();

#define MEASURE_1(kid,phase)                                            \
        end_time = mclock (); tspan = (float) span (start_time, end_time);  \
        num = (float) (map[my_node][kid-1].to - map[my_node][kid-1].from + 1);  \
        printf ("my_node=%d  ==>  ", my_node);                          \
        printf ("%s[%d]=%f0, ((phase == 'P') ? "pcunit" : "lcunit"), kid, tspan / num);  \
        tsum += (tspan / num); tcompsum += tspan;
```



```
#define LEARN_UPDATE(pred,succ,w_matrix,is_output_cluster)              \
        if (is_output_cluster)                                          \
                for (i = map[my_node][succ-1].from; i <= map[my_node][succ-1].to; i++)  \
                        delta[i] = out[i] * (1.0 - out[i]) * (tbuf[i] - out[i]);  \
        else                                                            \
                for (i = map[my_node][succ-1].from; i <= map[my_node][succ-1].to; i++)  \
                        delta[i] -= out[i];                             \
        for (i = map[my_node][succ-1].from; i <= map[my_node][succ-1].to; i++) {  \
                ii = i - nkfrom[succ-1];                                \
                for (k = 0; k < nk[pred-1]; k++) w_matrix [k][ii] += (gain * delta[i]);  \
        }

#define LEARN_ERROR(pred,succ,w_matrix,is_output_cluster)               \
        for (i = nkfrom[pred-1]; i <= nkto[pred-1]; i++) {              \
                delta[i] = 0.0; ii = i - nkfrom[pred-1];                \
                for (j = map[my_node][succ-1].from - nkfrom[succ-1];    \
                        j <= map[my_node][succ-1].to - nkfrom[succ-1]; j++)  \
                delta[i] += (delta[j+nkfrom[succ-1]] * w_matrix [ii][j]); outbuf[count++] = delta[i];  \
        }

#define INIT_WEIGHT(pred,succ,w_matrix)                                 \
        for (j = map[my_node][succ-1].from - nkfrom[succ-1];            \
                j <= map[my_node][succ-1].to - nkfrom[succ-1]; j++)     \
        for (i = 0; i < nk[pred-1]; i++) w_matrix [i][j] = 0.5;

#define TEACH_INPUT()                                                   \
        crecv (INPUT_TYPE, inbuf, nbytes * NUM_INPUT);                  \
        crecv (TEACH_TYPE, tbuf, nbytes * NUM_OUTPUT);
```

~software/chu/ms/ipsc/src/comm.h

```
#ifdef STATISTICS
#define PROD_COMM(_pred,_succ)
#else
#define PROD_COMM(_pred,_succ)                                          \
        isend (OUTPUT_TYPE, outbuf, nbytes * (asgnt[my_node][_pred-1] + 1), ALL_NODE, NODE_PID);  \
        for (i = 1; i < NODES; i++) {                                   \
                crecv (OUTPUT_TYPE, outbuf, nbytes * (nk[_pred-1] + 1));  \
                this_node = (int) outbuf[0]; count = 1;                 \
                for (k = map[this_node][_pred-1].from; k <= map[this_node][_pred-1].to; k++)  \
                        out[k] = outbuf[count++];                       \
        }
#endif

#ifdef STATISTICS
#define LEARN_COMM(_pred,_succ)
#else
#define LEARN_COMM(_pred,_succ)                                         \
        isend (ERROR_TYPE, outbuf, nbytes * nk[_succ-1], ALL_NODE, NODE_PID);  \
```

```
        for (i = 1; i < NODES; i++) {                                   \
                crecv (ERROR_TYPE, outbuf, nbytes * nk[_succ-1]);       \
                count = my_count[_succ-1];                              \
                for (k = map[my_node][_succ-1].from; k <= map[my_node][_succ-1].to; k++)  \
                        delta[k] += outbuf[count++];                    \
        }
#endif
```

~software/chu/ms/ipsc/multi-layer/node.c

```
#include        "../src/def.h"
#include        "map.h"
#include        "../src/comp.h"
#include        "../src/comm.h"

main (argc, argv)
        int argc; char *argv[];
{       /** This is a node process. **/
        long my_node, my_pid, host, start_time, end_time, time_buffer[5];
        int i, ii, j, k, count, iter, this_node;
        float net, tspan, num, tsum = 0.0, tcompsum = 0.0;

        iPSC2_INIT ()
        init_ANN_sim (my_node);
        TEACH_INPUT ()

        time_buffer[1] = mclock ();
        /** The following is main operations of an ANN. **/
        for (iter = 0; iter < ITER; iter++) {
        /** Start the PRODUCTION phase. **/

#ifdef STATISTICS
        MEASURE_0 (1, 'P')
#endif
        /** Produce the outputs of neurons in cluster 1. **/
        PROD_COMP_0 (1)
                COMP_NET_INPUT (1, 1, w01, inbuf, 0)
        PROD_COMP_1 (1)
#ifdef STATISTICS
        MEASURE_1 (1, 'P')
#endif

#ifdef STATISTICS
        MEASURE_0 (2, 'P')
#endif
#ifndef SEQUENTIAL
        PROD_COMM (1, 2)
#endif
        /** Produce the outputs of neurons in cluster 2. **/
```



```
        PROD_COMP_0 (2)
                COMP_NET_INPUT (1, 2, w12, out, 0)
        PROD_COMP_1 (2)
#ifdef STATISTICS
        MEASURE_1 (2, 'P')
#endif

#ifdef STATISTICS
        MEASURE_0 (3, 'P')
#endif
        /** Cluster 1's outputs are already available, then
         ** produce outputs of neurons in cluster 3. **/
        PROD_COMP_0 (3)
                COMP_NET_INPUT (1, 3, w13, out, 0)
        PROD_COMP_1 (3)
#ifdef STATISTICS
        MEASURE_1 (3, 'P')
#endif

#ifdef STATISTICS
        MEASURE_0 (4, 'P')
#endif
        /** Cluster 1's outputs are already available, then
         ** produce outputs of neurons in cluster 4. **/
        PROD_COMP_0 (4)
                COMP_NET_INPUT (1, 4, w14, out, 0)
        PROD_COMP_1 (4)
#ifdef STATISTICS
        MEASURE_1 (4, 'P')
#endif

#ifdef STATISTICS
        MEASURE_0 (5, 'P')
#endif
#ifndef SEQUENTIAL
        PROD_COMM (2, 5)
        PROD_COMM (3, 5)
#endif
        /** Produce the outputs of neurons in cluster 5. **/
        PROD_COMP_0 (5)
                COMP_NET_INPUT (2, 5, w25, out, N1)
                COMP_NET_INPUT (3, 5, w35, out, N12)
        PROD_COMP_1 (5)
#ifdef STATISTICS
        MEASURE_1 (5, 'P')
#endif

#ifdef STATISTICS
        MEASURE_0 (6, 'P')
#endif
#ifndef SEQUENTIAL
        /** Cluster 3's outputs are already available. **/
        PROD_COMM (4, 6)
```

```
#endif
        /** Produce the outputs of neurons in cluster 6. **/
        PROD_COMP_0 (6)
                COMP_NET_INPUT (3, 6, w36, out, N12)
                COMP_NET_INPUT (4, 6, w46, out, N13)
        PROD_COMP_1 (6)
#ifdef STATISTICS
        MEASURE_1 (6, 'P')
#endif

#ifdef STATISTICS
        MEASURE_0 (7, 'P')
#endif
#ifndef SEQUENTIAL
        PROD_COMM (5, 7)
        PROD_COMM (6, 7)
#endif
        /** Produce the outputs of neurons in cluster 7. **/
        PROD_COMP_0 (7)
                COMP_NET_INPUT (5, 7, w57, out, N14)
                COMP_NET_INPUT (6, 7, w67, out, N15)
        PROD_COMP_1 (7)
#ifdef STATISTICS
        MEASURE_1 (7, 'P')
#endif

        /** Start the LEARNING phase. **/

#ifdef STATISTICS
        MEASURE_0 (7, 'L')
#endif
        /** Update the weight matrix between clusters 5, 6 and 7. **/
        LEARN_UPDATE (5, 7, w57, 1)
        LEARN_UPDATE (6, 7, w67, 1)
        count = 0;
        LEARN_ERROR (5, 7, w57, 1)
        LEARN_ERROR (6, 7, w67, 1)
#ifdef STATISTICS
        MEASURE_1 (7, 'L')
#endif

#ifdef STATISTICS
        MEASURE_0 (6, 'L')
#endif
#ifndef SEQUENTIAL
        LEARN_COMM (5, 7)
        LEARN_COMM (6, 7)
#endif
        /** Update the weight matrix between clusters 3, 4 and 6. **/
        LEARN_UPDATE (3, 6, w36, 0)
        LEARN_UPDATE (4, 6, w46, 0)
        count = 0;
        LEARN_ERROR (3, 6, w36, 0)
```



```
        LEARN_ERROR (4, 6, w46, 0)
#ifdef STATISTICS
        MEASURE_1 (6, 'L')
#endif

#ifdef STATISTICS
        MEASURE_0 (5, 'L')
#endif
#ifndef SEQUENTIAL
        LEARN_COMM (3, 6)
        LEARN_COMM (4, 6)
#endif
        /** Update the weight matrix between clusters 2, 3 and 5. **/
        LEARN_UPDATE (2, 5, w25, 0)
        LEARN_UPDATE (3, 5, w35, 0)
        count = 0;
        LEARN_ERROR (2, 5, w25, 0)
        LEARN_ERROR (3, 5, w35, 0)
#ifdef STATISTICS
        MEASURE_1 (5, 'L')
#endif

#ifdef STATISTICS
        MEASURE_0 (4, 'L')
#endif
#ifndef SEQUENTIAL
        LEARN_COMM (2, 5)
        LEARN_COMM (3, 5)
#endif
        /** Update the weight matrix between clusters 1 and 4. **/
        LEARN_UPDATE (1, 4, w14, 0)
        count = 0;
        LEARN_ERROR (1, 4, w14, 0)
#ifdef STATISTICS
        MEASURE_1 (4, 'L')
#endif

#ifdef STATISTICS
        MEASURE_0 (3, 'L')
#endif
#ifndef SEQUENTIAL
        LEARN_COMM (1, 4)
#endif
        /** Update the weight matrix between clusters 1 and 3. **/
        LEARN_UPDATE (1, 3, w13, 0)
        count = 0;
        LEARN_ERROR (1, 3, w13, 0)
#ifdef STATISTICS
        MEASURE_1 (3, 'L')
#endif

#ifdef STATISTICS
        MEASURE_0 (2, 'L')
```

```
#endif
#ifndef SEQUENTIAL
        LEARN_COMM (1, 3)
#endif
        /** Update the weight matrix between clusters 1 and 2. **/
        LEARN_UPDATE (1, 2, w12, 0)
        count = 0;
        LEARN_ERROR (1, 2, w12, 0)
#ifdef STATISTICS
        MEASURE_1 (2, 'L')
#endif

#ifdef STATISTICS
        MEASURE_0 (1, 'L')
#endif
#ifndef SEQUENTIAL
        LEARN_COMM (1, 2)
#endif
        /** Update the weight matrix between INPUT and 1. **/
        LEARN_UPDATE (1, 1, w01, 0)
#ifdef STATISTICS
        MEASURE_1 (1, 'L')
#endif

#ifdef STATISTICS
        printf ("my_node=%d  ==>  tsum=%f\0, my_node, tsum);
#endif
        /** This is the end of one ITERATION. **/
        }

        time_buffer[2] = mclock ();
#ifndef STATISTICS
        tspan = (float) span (time_buffer[1], time_buffer[2]);
        printf ("my_node=%d  ==>  ", my_node);
        printf ("Iter=%d,  ComplTime=%g(ms),  ComplTime/Iter=%g(ms)\0, ITER, tspan, tspan / ITER);
#endif
}


init_ANN_sim (my_node)
        long my_node;
{       int i, j;

        /** initialize the gain term. **/
        gain = 0.7;

        /** Set up the assignment scheme. **/
        for (j = 0; j < CLUSTERS; j++) {
                map[0][j].from = nkfrom[j];
                map[0][j].to = map[0][j].from + asgnt[0][j] - 1;
                for (i = 1; i < NODES; i++) {
                        map[i][j].from = map[i-1][j].to + 1;
                        map[i][j].to = map[i][j].from + asgnt[i][j] - 1;
```



```
                }
        }

        INIT_WEIGHT (1, 1, w01)
        INIT_WEIGHT (1, 2, w12)
        INIT_WEIGHT (1, 3, w13)
        INIT_WEIGHT (1, 4, w14)
        INIT_WEIGHT (2, 5, w25)
        INIT_WEIGHT (3, 5, w35)
        INIT_WEIGHT (3, 6, w36)
        INIT_WEIGHT (4, 6, w46)
        INIT_WEIGHT (5, 7, w57)
        INIT_WEIGHT (6, 7, w67)

        for (i = 0; i < NEURONS; i++) {
                theta[i] = 0.5;         /** Initialize thresholds. **/
                inbuf[i] = 1.0;         /** Initialize inbuf. **/
                outbuf[i] = 1.0;        /** Initialize outbuf. **/
                tbuf[i] = 1.0;          /** Initialize tbuf. **/
        }
}
```

C.4. Listing of Parallel ANN Simulations on a Network of Three Workstations.

~software/chu/ms/sun-3/multi-layer/Makefile

```
comp-1:     map.h-1 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-1 node.c > .temp.c; cc -DSTATISTICS .temp.c -lm -o comp-1

comp-2:     map.h-2 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-2 node.c > .temp.c; cc -DSTATISTICS .temp.c -lm -o comp-2

comp-3:     map.h-3 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-3 node.c > .temp.c; cc -DSTATISTICS .temp.c -lm -o comp-3

seq-1:  map.h-1 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-1 node.c > .temp.c; cc -DSEQUENTIAL .temp.c -lm -o seq-1

seq-2:  map.h-2 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-2 node.c > .temp.c; cc -DSEQUENTIAL .temp.c -lm -o seq-2

seq-3:  map.h-3 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-3 node.c > .temp.c; cc -DSEQUENTIAL .temp.c -lm -o seq-3

node-1:     map.h-1 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-1 node.c > .temp.c; cc .temp.c -lm -o node-1

node-2:     map.h-2 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-2 node.c > .temp.c; cc .temp.c -lm -o node-2

node-3:     map.h-1 ../src/comp.h ../src/comm.h node.c
        touch .temp.c; rm .temp.c; cat map.h-3 node.c > .temp.c; cc .temp.c -lm -o node-3
```

~software/chu/ms/sun-3/multi-layer/map.h-1

```
/** ANN Model **/

#defineN1               500
#defineN2               200
#defineN3               500
#defineN4               300
#defineN5               400
#defineN6               600
#defineN7               200

#defineN12              (N1 + N2)
#defineN13              (N12 + N3)
#defineN14              (N13 + N4)
#defineN15              (N14 + N5)
#defineN16              (N15 + N6)
#defineN17              (N16 + N7)

#defineNEURONS          (N17)
#defineCLUSTERS         7
#defineLAYERS           4

int     nk[CLUSTERS] = { N1, N2, N3, N4, N5, N6, N7 };
int     nkfrom[CLUSTERS] = { 0, N1, N12, N13, N14, N15, N16 };
int     nkto[CLUSTERS] = { N1-1, N12-1, N13-1, N14-1, N15-1, N16-1, N17-1 };

float   w01[N1][N1], w12[N1][N2], w13[N1][N3], w14[N1][N4], w25[N2][N5];
float   w35[N3][N5], w36[N3][N6], w46[N4][N6], w57[N5][N7], w67[N6][N7];

/** Multicomputer Model **/

#ifdef  SEQUENTIAL
#defineNODES            1
#else
#defineNODES            3
#endif

/** Mapping **/

int     asgnt[NODES][CLUSTERS] =
{
#ifdef  SEQUENTIAL
        { 500, 200, 500, 300, 400, 600, 200 }
#else
        { 130,  52, 131,  79, 105, 158,  53 },
        { 146,  59, 146,  87, 117, 175,  58 },
        { 224,  89, 223, 134, 178, 267,  89 }
#endif
};

struct { int from, to; } map[NODES][CLUSTERS];

/** Simulation **/

#defineITER             1

float   out[NEURONS];           /** Outputs of neurons **/
float   delta[NEURONS];         /** Partial errors of neurons **/
float   theta[NEURONS];         /** Threshold **/

float   inbuf[NEURONS+2];       /** Input buffer **/
float   outbuf[NEURONS+2];      /** Output buffer **/
float   tbuf[NEURONS+2];        /** Teaching input buffer **/

float   gain;
int     my_id;
typedef struct tms      systime;
```

```
/** Shorthands:
** to avoid intractability in prediction of comp time for exp,
** the comp time is set to be fixed. **/

#definefloat_exp(a)    ((float) exp ((double) (0.5)))
#definesigmoid(a)      (1.0 / (1.0 + float_exp (a)))
#definenbytes          (sizeof (float) / sizeof (char))
#define span(s,e)      (e.tms_utime - s.tms_utime + e.tms_stime - s.tms_stime)
```

~software/chu/ms/sun-3/src/comp.h

```
#definePROD_COMP_0(kid)                                                    \
        count = 0;                                                         \
        for (i = map[my_id][kid-1].from; i <= map[my_id][kid-1].to; i++) {  \
                net = theta[i]; ii = i - nkfrom[kid-1];

#define PROD_COMP_1(kid) outbuf[count++] = out[i] = sigmoid(net); }

#defineCOMP_NET_INPUT(pred,succ,w_matrix,source,offset)                     \
        for (k = 0; k < nk[pred-1]; k++) net += (w_matrix [k][ii] * source [k+offset]);

#defineMEASURE_0(kid,phase)     times (&start_time);

#defineMEASURE_1(kid,phase)                                                \
        times (&end_time); tspan = (float) span (start_time,end_time);     \
        num = (float) (map[my_id][kid-1].to - map[my_id][kid-1].from + 1);  \
        printf ("%s[%d]=%f0, ((phase == 'P') ? "pcunit" : "lcunit"), kid, tspan / num);  \
        tsum += (tspan / num);

#define LEARN_UPDATE(pred,succ,w_matrix,is_output_cluster)                 \
        if (is_output_cluster)                                             \
                for (i = map[my_id][succ-1].from; i <= map[my_id][succ-1].to; i++)  \
                        delta[i] = out[i] * (1.0 - out[i]) * (tbuf[i] - out[i]);    \
        else                                                               \
                for (i = map[my_id][succ-1].from; i <= map[my_id][succ-1].to; i++)  \
                        delta[i] -= out[i];                                \
        for (i = map[my_id][succ-1].from; i <= map[my_id][succ-1].to; i++) {  \
                ii = i - nkfrom[succ-1];                                   \
                for (k = 0; k < nk[pred-1]; k++) w_matrix [k][ii] += (gain * delta[i]);  \
        }

#define LEARN_ERROR(pred,succ,w_matrix,is_output_cluster)                  \
        for (i = nkfrom[pred-1]; i <= nkto[pred-1]; i++) {                 \
                delta[i] = 0.0; ii = i - nkfrom[pred-1];                   \
                for (j = map[my_id][succ-1].from - nkfrom[succ-1];         \
                        j <= map[my_id][succ-1].to - nkfrom[succ-1]; j++)  \
                        delta[i] += (delta[j+nkfrom[succ-1]] * w_matrix [ii][j]);  \
                outbuf[count++] = delta[i];                                \
        }
```



```
#defineINIT_WEIGHT(pred,succ,w_matrix)                                     \
        for (j = map[my_id][succ-1].from - nkfrom[succ-1];                 \
                j <= map[my_id][succ-1].to - nkfrom[succ-1]; j++)          \
                for (i = 0; i < nk[pred-1]; i++) w_matrix [i][j] = 0.5;
```

~software/chu/ms/sun-3/src/comp.h

```
#defineP_SEND_AND_RECV(_to,_from,_pred,_succ)                              \
        _sock = send_sock[_to];                                            \
        SEND (send_sock[_to], (char *) outbuf, nbytes * asgnt[my_id][_pred-1]);  \
        send_sock[_to] = _sock;                                            \
        RECV (msgsock[_from], (char *)  inbuf, nbytes * asgnt[_from][_pred-1]);  \
        for (count = 0, k = map[_from][_pred-1].from; k <= map[_from][_pred-1].to; k++)  \
                out[k] = inbuf[count++];

#definePROD_COMM(_pred,_succ)                                              \
        switch (my_id) {                                                   \
                case 0: P_SEND_AND_RECV(1,2,_pred,_succ)                   \
                        P_SEND_AND_RECV(2,1,_pred,_succ)                   \
                        break;                                             \
                case 1: P_SEND_AND_RECV(2,0,_pred,_succ)                   \
                        P_SEND_AND_RECV(0,2,_pred,_succ)                   \
                        break;                                             \
                case 2: P_SEND_AND_RECV(0,1,_pred,_succ)                   \
                        P_SEND_AND_RECV(1,0,_pred,_succ)                   \
                        break;                                             \
                }

#defineL_SEND_AND_RECV(_to,_from,_pred,_succ)                              \
        _sock = send_sock[_to];                                            \
        SEND (send_sock[_to], (char *) outbuf, nbytes * nk[_succ-1]);      \
        send_sock[_to] = _sock;                                            \
        RECV (msgsock[_from], (char *) inbuf, nbytes * nk[_succ-1]);       \
        for (count = i = 0; i < my_id; i++) count += asgnt[i][_succ-1];    \
        for (k = map[my_id][_succ-1].from; k <= map[my_id][_succ-1].to; k++) delta[k] += outbuf[count++];

#defineLEARN_COMM(_pred,_succ)                                             \
        switch (my_id) {                                                   \
                case 0: L_SEND_AND_RECV(1,2,_pred,_succ)                   \
                        L_SEND_AND_RECV(2,1,_pred,_succ)                   \
                        break;                                             \
                case 1: L_SEND_AND_RECV(2,0,_pred,_succ)                   \
                        L_SEND_AND_RECV(0,2,_pred,_succ)                   \
                        break;                                             \
                case 2: L_SEND_AND_RECV(0,1,_pred,_succ)                   \
                        L_SEND_AND_RECV(1,0,_pred,_succ)                   \
                        break;                                             \
        }
```

~software/chu/ms/sun-3/src/comm.c

```
#include	<stdio.h>
#include	<math.h>
#include	<sys/types.h>
#include	<sys/times.h>
#include	<sys/socket.h>
#include	<netinet/in.h>
#include	<stropts.h>
#include	<netdb.h>
#include	"../src/socket.h"

int		send_sock[NODES+1];
int		_send_sock[NODES+1];
int		recv_sock[NODES+1];
int		msgsock[NODES+1];
struct	hostent		*hp, *gethostbyname ();
struct	sockaddr_in	send_server[NODES];
struct	sockaddr_in	recv_server[NODES];

init_vckt (my_id)
	int my_id;
{	int i;

	switch (my_id) {
		case 0:		/** This is for SUSIE. **/
			CREATE_RECV (&recv_sock[2], &recv_server[2],
				SEND_elaine_susie_PORT,
				RECV_elaine_susie_PORT,
				"elaine", &msgsock[2]);
			CREATE_SEND (&_send_sock[1], &send_server[1],
				SEND_susie_rhea_PORT,
				RECV_susie_rhea_PORT, "rhea");
			CREATE_SEND (&_send_sock[2], &send_server[2],
				SEND_susie_elaine_PORT,
				RECV_susie_elaine_PORT, "elaine");
			CREATE_RECV (&recv_sock[1], &recv_server[1],
				SEND_rhea_susie_PORT,
				RECV_rhea_susie_PORT,
				"rhea", &msgsock[1]);
			break;

		case 1:		/** This is for RHEA. **/
			CREATE_SEND (&_send_sock[2], &send_server[2],
				SEND_rhea_elaine_PORT,
				RECV_rhea_elaine_PORT, "elaine");
			CREATE_RECV (&recv_sock[0], &recv_server[0],
				SEND_susie_rhea_PORT,
				RECV_susie_rhea_PORT,
```



```
				"susie", &msgsock[0]);
			CREATE_RECV (&recv_sock[2], &recv_server[2],
				SEND_elaine_rhea_PORT,
				RECV_elaine_rhea_PORT,
				"elaine", &msgsock[2]);
			CREATE_SEND (&_send_sock[0], &send_server[0],
				SEND_rhea_susie_PORT,
				RECV_rhea_susie_PORT, "susie");
			break;
		case 2:		/** This is for ELAINE. **/
			CREATE_RECV (&recv_sock[1], &recv_server[1],
				SEND_rhea_elaine_PORT,
				RECV_rhea_elaine_PORT,
				"rhea", &msgsock[1]);
			CREATE_SEND (&_send_sock[0], &send_server[0],
				SEND_elaine_susie_PORT,
				RECV_elaine_susie_PORT, "susie");
			CREATE_RECV (&recv_sock[0], &recv_server[0],
				SEND_susie_elaine_PORT,
				RECV_susie_elaine_PORT,
				"susie", &msgsock[0]);
			CREATE_SEND (&_send_sock[1], &send_server[1],
				SEND_elaine_rhea_PORT,
				RECV_elaine_rhea_PORT, "rhea");
			break;
		default: break;
	}

	for (i = 0; i < NODES; i++)
		if (my_id != i) {
			send_sock[i] = _send_sock[i];
			ioctl (msgsock[i], I_SRDOPT, RMSGD);
		}
}

close_vckt (my_id)
	int my_id;
{	int i;

	for (i = 0; i < NODES; i++)
		if (my_id != i) { close (send_sock[i]); close (msgsock[i]); close (recv_sock[i]); }
}

SEND (sock, buf, len)
/** Send a frame. **/
	int sock; char *buf; int len;
{	int status = write (sock, buf, len);

	if (status < 0) perror ("write a frame");
}

RECV (msgsock, buf, len)
/** Read a frame. **/
```

```
        int msgsock; char *buf; int len;
{       int rval = read (msgsock, buf, len);

        if (rval < 0) perror ("read a frame");
        if (rval == 0) printf ("close this connection0);
}

CREATE_SEND (send_sock, send_server, send_port, recv_port, dest_host)
/** This routine is based on the one in Sun Manual for network programming. **/
        int *send_sock; struct sockaddr_in *send_server;
        int send_port, recv_port; char dest_host[];
{       int status;

        *send_sock = socket (AF_INET, SOCK_STREAM, 0);
        send_server->sin_family = AF_INET;
        send_server->sin_addr.s_addr = INADDR_ANY;
        send_server->sin_port = recv_port;
        status = bind (*send_sock, send_server, sizeof (*send_server));
        if (status < 0) perror ("binding fails");
        hp = gethostbyname (dest_host);
        bcopy ((char *) hp->h_addr, (char *) &(send_server->sin_addr), hp->h_length);
        connect (*send_sock, send_server, sizeof (*send_server));
}

CREATE_RECV (recv_sock, recv_server, send_port, recv_port, dest_host, msgsock)
/** This routine is based on the one in Sun Manual for network programming. **/
        int *recv_sock; struct sockaddr_in *recv_server;
        int send_port, recv_port; char dest_host[]; int *msgsock;
{       int status;

        *recv_sock = socket (AF_INET, SOCK_STREAM, 0);
        recv_server->sin_family = AF_INET;
        recv_server->sin_addr.s_addr = INADDR_ANY;
        recv_server->sin_port = recv_port;
        status = bind (*recv_sock, recv_server, sizeof (* recv_server));
        if (status < 0) perror ("binding fails");
        listen (*recv_sock, 5);
        *msgsock = accept (*recv_sock, (struct sockaddr *) 0, (int *) 0);
}
```

~software/chu/ms/sun-3/multi-layer/node.c

```
#include    "../src/comp.h"
#include    "../src/comm.h"
#include    "../src/comm.c"

main (argc, argv)
        int argc; char *argv[];
{       /** This is a node process. **/
```



```
        int _sock, i, ii, j, k, count, iter, this_node;
        float net, tspan, num, tsum = 0.0; long ack;
        systime start_time, end_time, t_sim_0, t_sim_1;

        my_id = atoi (argv[1]);
        printf ("my_id = %d0, my_id);
#ifndef SEQUENTIAL
        init_vckt (my_id);
#endif
        init_ANN_sim ();

        times (&t_sim_0);
        /** This is the main operations of an ANN. **/
        for (iter = 0; iter < ITER; iter++) {

        /** Start the PRODUCTION phase. **/

#ifdef STATISTICS
        MEASURE_0 (1,'P')
#endif
        /** Produce the outputs of neurons in cluster 1. **/
        PROD_COMP_0 (1)
                COMP_NET_INPUT (1, 1, w01, inbuf, 0)
        PROD_COMP_1 (1)
#ifdef STATISTICS
        MEASURE_1 (1,'P')
#endif

#ifndef SEQUENTIAL
        PROD_COMM (1, 2)
#endif
#ifdef STATISTICS
        MEASURE_0 (2,'P')
#endif
        /** Produce the outputs of neurons in cluster 2. **/
        PROD_COMP_0 (2)
                COMP_NET_INPUT (1, 2, w12, out, 0)
        PROD_COMP_1 (2)
#ifdef STATISTICS
        MEASURE_1 (2,'P')
#endif

        /** Cluster 1's outputs are already available. **/
#ifdef STATISTICS
        MEASURE_0 (3,'P')
#endif
        /** Produce the outputs of neurons in cluster 3. **/
        PROD_COMP_0 (3)
                COMP_NET_INPUT (1, 3, w13, out, 0)
        PROD_COMP_1 (3)
#ifdef STATISTICS
        MEASURE_1 (3,'P')
#endif
```

```
        /** Cluster 1's outputs are already available. **/
#ifdef STATISTICS
        MEASURE_0 (4,'P')
#endif
        /** Produce the outputs of neurons in cluster 4. **/
        PROD_COMP_0 (4)
                COMP_NET_INPUT (1, 4, w14, out, 0)
        PROD_COMP_1 (4)
#ifdef STATISTICS
        MEASURE_1 (4,'P')
#endif

#ifndef SEQUENTIAL
        PROD_COMM (2, 5)
        PROD_COMM (3, 5)
#endif
#ifdef STATISTICS
        MEASURE_0 (5,'P')
#endif
        /** Produce the outputs of neurons in cluster 5. **/
        PROD_COMP_0 (5)
                COMP_NET_INPUT (2, 5, w25, out, N1)
                COMP_NET_INPUT (3, 5, w35, out, N12)
        PROD_COMP_1 (5)
#ifdef STATISTICS
        MEASURE_1 (5,'P')
#endif

#ifndef SEQUENTIAL
        /** Cluster 3's outputs are already available. **/
        PROD_COMM (4, 6)
#endif
#ifdef STATISTICS
        MEASURE_0 (6,'P')
#endif
        /** Produce the outputs of neurons in cluster 6. **/
        PROD_COMP_0 (6)
                COMP_NET_INPUT (3, 6, w36, out, N12)
                COMP_NET_INPUT (4, 6, w46, out, N13)
        PROD_COMP_1 (6)
#ifdef STATISTICS
        MEASURE_1 (6,'P')
#endif

#ifndef SEQUENTIAL
        PROD_COMM (5, 7)
        PROD_COMM (6, 7)
#endif
#ifdef STATISTICS
        MEASURE_0 (7,'P')
#endif
        /** Produce the outputs of neurons in cluster 7. **/
        PROD_COMP_0 (7)
```



```
                COMP_NET_INPUT (5, 7, w57, out, N14)
                COMP_NET_INPUT (6, 7, w67, out, N15)
        PROD_COMP_1 (7)
#ifdef STATISTICS
        MEASURE_1 (7,'P')
#endif

        /** Start the TRAINING phase. **/

#ifdef STATISTICS
        MEASURE_0 (7,'L')
#endif
        /** Update the weight matrix between clusters 5, 6 and 7. **/
        LEARN_UPDATE (5, 7, w57, 1)
        LEARN_UPDATE (6, 7, w67, 1)
        count = 0;
        LEARN_ERROR (5, 7, w57, 1)
        LEARN_ERROR (6, 7, w67, 1)
#ifdef STATISTICS
        MEASURE_1 (7,'L')
#endif

#ifndef SEQUENTIAL
        LEARN_COMM (5, 7)
        LEARN_COMM (6, 7)
#endif
#ifdef STATISTICS
        MEASURE_0 (6,'L')
#endif
        /** Update the weight matrix between clusters 3, 4 and 6. **/
        LEARN_UPDATE (3, 6, w36, 0)
        LEARN_UPDATE (4, 6, w46, 0)
        count = 0;
        LEARN_ERROR (3, 6, w36, 0)
        LEARN_ERROR (4, 6, w46, 0)
#ifdef STATISTICS
        MEASURE_1 (6,'L')
#endif

#ifndef SEQUENTIAL
        LEARN_COMM (3, 6)
        LEARN_COMM (4, 6)
#endif
#ifdef STATISTICS
        MEASURE_0 (5,'L')
#endif
        /** Update the weight matrix between clusters 2, 3 and 5. **/
        LEARN_UPDATE (2, 5, w25, 0)
        LEARN_UPDATE (3, 5, w35, 0)
        count = 0;
        LEARN_ERROR (2, 5, w25, 0)
        LEARN_ERROR (3, 5, w35, 0)
#ifdef STATISTICS
```

```
        MEASURE_1 (5,'L')
#endif

#ifndef SEQUENTIAL
        LEARN_COMM (2, 5)
        LEARN_COMM (3, 5)
#endif
#ifdef STATISTICS
        MEASURE_0 (4,'L')
#endif
        /** Update the weight matrix between clusters 1 and 4. **/
        LEARN_UPDATE (1, 4, w14, 0)
        count = 0;
        LEARN_ERROR (1, 4, w14, 0)
#ifdef STATISTICS
        MEASURE_1 (4,'L')
#endif

#ifndef SEQUENTIAL
        LEARN_COMM (1, 4)
#endif
#ifdef STATISTICS
        MEASURE_0 (3,'L')
#endif
        /** Update the weight matrix between clusters 1 and 3. **/
        LEARN_UPDATE (1, 3, w13, 0)
        count = 0;
        LEARN_ERROR (1, 3, w13, 0)
#ifdef STATISTICS
        MEASURE_1 (3,'L')
#endif

#ifndef SEQUENTIAL
        LEARN_COMM (1, 3)
#endif
#ifdef STATISTICS
        MEASURE_0 (2,'L')
#endif
        /** Update the weight matrix between clusters 1 and 2. **/
        LEARN_UPDATE (1, 2, w12, 0)
        count = 0;
        LEARN_ERROR (1, 2, w12, 0)
#ifdef STATISTICS
        MEASURE_1 (2,'L')
#endif

#ifndef SEQUENTIAL
        LEARN_COMM (1, 2)
#endif
#ifdef STATISTICS
        MEASURE_0 (1,'L')
#endif
        /** Update the weight matrix between INPUT and 1. **/
```



```
        LEARN_UPDATE (1, 1, w01, 0)
#ifdef STATISTICS
        MEASURE_1 (1,'L')
#endif

#ifdef STATISTICS
        printf ("tsum = %f0, tsum);
#endif
        /** This is the end of one ITERATION. **/
        }

        times (&t_sim_1);
        tspan = (float) span (t_sim_0, t_sim_1);
        tspan *= (1000.0 / 60.0);
        printf ("Iter=%d, Total Compl Time=%g, Compl Time per Iter=%g0, ITER, tspan, tspan / ITER);
}

init_ANN_sim ()
{       int i, j;

        /** Initialize the gain term. **/
        gain = 0.7;

        /** Set up the mapping scheme. **/
        for (j = 0; j < CLUSTERS; j++) {
                map[0][j].from = nkfrom[j]; map[0][j].to = map[0][j].from + asgnt[0][j] - 1;
                for (i = 1; i < NODES; i++) {
                        map[i][j].from = map[i-1][j].to + 1;
                        map[i][j].to = map[i][j].from + asgnt[i][j] - 1;
                }
        }

        INIT_WEIGHT (1, 1, w01)
        INIT_WEIGHT (1, 2, w12)
        INIT_WEIGHT (1, 3, w13)
        INIT_WEIGHT (1, 4, w14)
        INIT_WEIGHT (2, 5, w25)
        INIT_WEIGHT (3, 5, w35)
        INIT_WEIGHT (3, 6, w36)
        INIT_WEIGHT (4, 6, w46)
        INIT_WEIGHT (5, 7, w57)
        INIT_WEIGHT (6, 7, w67)

        for (i = 0; i < NEURONS; i++) {
                theta[i] = 0.5;         /** Initialize thresholds. **/
                inbuf[i] = 1.0;         /** Initialize inbuf. **/
                outbuf[i] = 1.0;        /** Initialize outbuf. **/
                tbuf[i] = 1.0;          /** Initialize tbuf. **/
        }
}
```