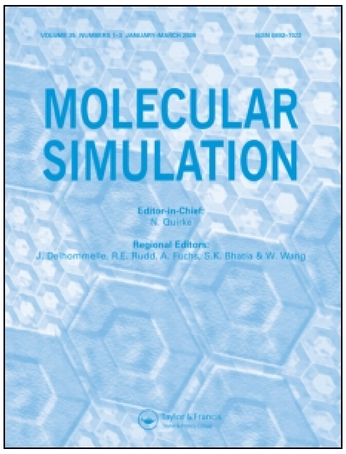

Molecular Simulation
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713644482

Computational Chemistry on Workstation Clusters: Parallel Programming for Molecular Dynamics and Neural Networks

Stephen Fleischman[a]
[a] Convex Computer Corporation, Richardson, Texas, USA

To cite this Article Fleischman, Stephen(1995) 'Computational Chemistry on Workstation Clusters: Parallel Programming for Molecular Dynamics and Neural Networks', Molecular Simulation, 14: 4, 209 — 233
To link to this Article: DOI: 10.1080/08927029508022019
URL: http://dx.doi.org/10.1080/08927029508022019

Molecular Simulation, 1995, Vol. 14, pp. 209–233
Reprints available directly from the publisher
Photocopying permitted by license only

COMPUTATIONAL CHEMISTRY ON WORKSTATION CLUSTERS: PARALLEL PROGRAMMING FOR MOLECULAR DYNAMICS AND NEURAL NETWORKS

STEPHEN FLEISCHMAN

Convex Computer Corporation, 3000 Waterview Parkway, Richardson, Texas 75083 USA. (sfleisch@convex.com)

(Received October 1993, accepted July 1994)

The use of RISC workstation clusters to obtain supercomputer level performance with the use of course-grained message passing parallelism is described. Two types of computational chemistry applications are discussed: molecular dynamics and neural networks.

Several molecular dynamics programs have been parallelized and have shown very good improvements in computational speed. For a 14000 atom Myoglobin dynamics calculation, we have obtained a 6 times acceleration with the program CHARMM on a eight node HP735 cluster using FDDI (Fiber Data Distributed Interface). A feed forward backpropagation neural network was developed that uses a parallelized 'molecular dynamics-like' algorithm. It shows good convergence behavior and very efficient parallel speedup when applied to the problem of protein secondary structure prediction.

KEY WORDS: Molecular dynamics, CHARMM, neural networks, parallel

INTRODUCTION

Clusters of RISC workstations are proving to be a very effective way to obtain high performance computing. By using powerful RISC workstation nodes such as the Hewlett Packard (HP) 735 and high speed communication technology such as FDDI (Fiber Data Distributed Interface), cluster computing can provide supercomputer level performance through the use of coarse grained parallel programming. This report will focus on two application areas in computational chemistry: molecular dynamics and neural networks (for use in protein structure prediction).

There have been widespread interest in using workstation clusters as parallel computers in the computational chemistry community (and for that matter, in scientific computing in general) [1]. They offer easy expandability, configurational flexibility and relatively low cost while potentially providing high levels of computational performance.

The Hardware

The workstations in a cluster are typically connected to each other by some form of local network. For problems in computational chemistry, one generally wants the highest bandwidth and lowest latency interconnection possible. Various hardware suppliers plan to introduce specialized communications products in the near future. However, in this report we will focus on the use of a standard interface: FDDI. This interconnect provides a total bandwidth of approximately 10 megabytes/second. We will show that this is sufficient to obtain a very useful level of parallel performance.

All calculations were carried out of a Convex Meta Series cluster consisting of 8 HP Model 735 workstations connected via FDDI. The PA-RISC processor runs at 99 MHz and has a peak (i.e. never to be achieved) 150 MFLOP rating.

The Software

To obtain parallel performance from workstation clusters, one must, usually, treat them as distributed memory parallel computers. Each workstation node has its own local memory and data is exchange between nodes using message passing [2].

There are available a number of subroutine libraries that provide message passing communication between networked computers. We have elected to use the PVM (Parallel Virtual Machine) library from Oakridge National Laboratories [3]. This package works on a large number of different hardware platforms (the network of computers can be heterogeneous) and is supported by a number of computer manufacturers. The standard library uses TCP/IP sockets to communicate between Unix based compters. It has the further advantage of providing mechanisms to allow for the incorporation of hardware specific communications drivers in the future. The work reported here used the TCP/IP sockets-based routines.

None of the work discussed in this report is machine specific, except, of course, for the actual performance results.

The first section discusses the adaptation of molecular dynamics codes to workstation clusters and shows the resulting performance that is obntained. The following section will show how similar programming techniques can be used in a different area, artificial neural networks, which have been used for protein secondary structure prediction.

MOLECULAR MECHANICS AND DYNAMICS

The Molecular Dynamics Method

The simulation of systems of molecules and atoms using the molecular dynamics (MD) technique is effected by integrating the classical equations of motion to obtain their time dependent position, $\vec{r}$, and velocities, $\vec{v}$:

$$\mathbf{M}\ddot{\vec{r}} = -\vec{\nabla}V = \vec{f} \tag{1}$$

The interaction potential energy between the atoms, V, is differentiated to obtain the forces, f on each atom or molecule (**M** is the diagonal mass matrix). Generally, the equations of motion must be integrated numerically. One commonly used and effective method is the 'leap-frog' algorithm [4–6]:

$$\vec{r}(t+\delta t) = \vec{r}(t) + \delta t\vec{v}(t+\tfrac{1}{2}\delta t) \tag{2a}$$

$$\vec{v}(t+\tfrac{1}{2}\delta t) = \delta t\vec{v}(t-\tfrac{1}{2}\delta t) + \delta t\mathbf{M}^{-1}\vec{f}(t) \tag{2b}$$

The atomic position for each time-step are updated from the half-time-step velocities. The new half-time-step velocities are then obtained from the calculation of the forces on the atoms, $\vec{f}\,(t)$.

Typically, a classical model potential energy function, such as the one below:

$$V = \sum^{\text{Bonds}} k_b(r - r_o)^2 + \sum^{\text{Angles}} k_\theta(\theta - \theta_o)^2 + \sum^{\text{torsions}} V_n(1 - n\cos\phi)$$

$$+ \sum_{i}^{n} \sum_{j>i}^{n} q_i q_j r_{12}^{-1} + \varepsilon_{ij}\left[\left(\frac{r_{ij}^*}{r_{ij}}\right)^{12} - 2\left(\frac{r_{ij}^*}{r_{ij}}\right)^{6}\right] \tag{3}$$

The first three terms in equation 3 represent the internal coordinate contribution to the forces and are computed from fixed lists. The number of such interactions tend to scale linearly with the number of atoms. The last two terms represent the non-bonded interaction between atoms. The number of such interactions scale as $O(N^2)$ and are generally responsible at about 90% of the compute time. The exponential growth in computational time of this component can be limited by the use of a distance based cut-off and a periodically updated neighbor-list [7].

A typical program involves the following steps:

- One time step-up. Read in coordinates, force-field parameters, time-step, etc.
- Dynamics loop, For each time step:
  - calculate neightbor-list if necessary
  - calculate forces and energy
  - update positions and velocites
  - apply temperature and presure scaling
  - accumulate properties for statistical averaging

Parallel performance is achieved by partitioning among the nodes all the interactions contributing to the atomic forces and potential energy. In the next section we will describe how this has been done effectively for a workstation cluster with a relatively simple MD program. This program uses only the last term in equation 3, the Lennard-Jones pair interaction potential to simulate a box of argon atoms. The simplicity of this program provides us with a straightforward means in which to implement and test various parallel programming approaches while still affording a realistic prediction of the behaviors of more general and complex programs. Following this description, the results obtained with one such general MD program, CHARMM [8], will be presented and shown to yield similar performance enhancements.

Parallelizing a Molecular Dynamics Program for Workstation Clusters

As mentioned above, we started our investigations into workstation cluster programming by modifying a simple MD program. The original sequential version was derived from FORTRAN subroutines that accompany the book "Computer Simulations of Liquids" by M. P. Allen and D. J. Tildesley [6], which provides one of the best available overviews to the field.

This simple program is limited to the simulation of Argon atoms for which it contains a built in set of Lennard-Jones (LJ) parameters. The LJ interaction is the only considered in the force calculation. The program employs a distance based cut-off and periodically generates a neighbor list to limit the time spent in the computationally intensive pair interaction calculation. Periodic boundary conditions are used in the minimum image convention [6].

Temperature control is maintained through the use of a Berendsen type thermostat [6, 9]. This technique involves the scaling of the velocities at each time-step and requires that the instantaneous temperature be calculated at each time-step and therefore the kinetic energy, from which it is derived, must be globally summed between all of the nodes.

We initially began with a swquential version of the program (i.e. not parallelized). The actual modifications to the program were relatively few. Four components of the program had to be modified:

1. The input and output (I/O) code.
2. The neighbor list update routine.
3. The force calculation routine.
4. The 'leap-frog' integration routine.

By far the most time consuming and tedious part of the conversion effort was handling the I/O.

There is a large body of literature on parallel computer MD algorithms (see for example references [10] and [11] and the references therein). We chose the straightforward and efficient 'array replication' technique [12, 13].

Each node (processor) possesses a complete set of force arrays and storage for energy, temperature and virial terms. The interactions are partitioned amongst the nodes so that at the end of the force/energy calculation, each node possesses a set of partially complete forces and energy terms. These must then be combined between the processors. This approach has been shown to work well for molecular dynamics on distributed memory parallel computers such as the Intel Delta with up to 512 processors [13]. It has been found to work well with workstation clusters where thenumber of nodes is generally relatively few (typically 8 to 16). The main limitation of this approach is that there is a fixed memory cost proportional to the number of particles that must be available on each and every node. Therefore, it is likely that for very large size molecular systems (e.g., 100,000 atoms) and for machines with greater than 1024 processors, other methods such as the domain decomposition techniques will be necessary [14].

The 'SPMD' (Single Program Multiple Data) [3] programming model has been used here. An identical copy of the executable runs on each node of the cluster with each node being an equal peer (i.e. no master controlling the slave nodes). Sequential code that cannot be parallelized is redundantly computed on each node. It is often cheaper, computationally, to do this rather than have one node compute and send the results and the others block and receive. The workload is partitioned among the nodes and the results communicated between them.

Setup and I/O

The initial setup can be encapsulated in a single routine (PVMFBEGIN in listing 1). In the current version of PVM, the user will run a copy of the program on one node. It will call the PVM routines necessary to start up the copies on the other nodes and to distribute cluster configuration information.

One designated node does the input and output (typically node 0). Every read statement in the original program was modified with a conditional test so that node 0 would do the input

```
PROGRAM FROGGY
Data Declarations
Initialize pvm. This routines contains various
startup functions.
CALL PVMBERGIN (MYNODE, NNODE, TIDS, STATUS)
IF (MYNODE. EQ. 0) THEN
  Read in paramters, coordinates, etc.
   READ (CNUNIT, '(A)') TITLE
  send the input data to the other processes.
  CALL PYMFINITSEND (PVMRAW, STATUS)
  CALL PVMFPACK (INTEGER4, NATOM, 1, 1, STATUS)
  ...
  CALL PVMFPACK (REAL8, RZ, NATOM, 1, STATUS)
  CALL PVMFMCAST (NNODE-1, TIDS (1), 0, STATUS)
ELSE
  CALL PVMFRECV (TIDS (0). 0, STATUS)
  CALL PVMFUNPACK (INTEGER4, NATOM, 1, 1, STATUS)
  ...
  CALL PVMFUNPACK (REAL8, RZ, NATOM, 1, STATUS)
ENDIF
```

Listing 1: Code fragment showing setup and example of the input modifications.

and send the data to the remaining nodes which would do a call to a receive routines to obtain that data (see listing 1). This modification turns out to be the most time consuming part of the conversion of the original program to the message passing model used for parallelization on a cluster.

Potential energy and force calculation

The non-bonded pairwise force calculation is typically responsible for 90% or more of the total CPU time. The intrinsically N^2 nature of this contribution to the forces is reduced by applying a cut-off distance to all pair interactions. Periodically, a neighbor list is generated that contains all unique pairs of atoms that are within the specified cut-off distance. This computationally intensive task is done infrequently, often every 10–20 time steps.

It is during the neighbor list generation step that the computational work is distributed across the nodes. Load balancing is one of the two most significant factors affecting parallel efficiency (the other is computation/communication rate (see below). The all-pairs algorithm [11] was used (see listing 2). The list was update approximately every 10 steps based on the maximum displacement of any one particle since the last update. The only significant modification to the sequential version was changing the outer loop on each node to start at a unique atom and then deal with every NNODE*th* atom. The original loop went from 1 to NATOM-1.

This change produces nearly equal partitioning of the work amongst the nodes for this homogeneous problem. We have also seen that this partitioning as well as incrementing the inner loop works well for bio-molecules [13]. It is not difficult to introduce dynamic load balancing at this point for systems that are subject to significant density fluctuations. Each node maintains its own neighbor list with a portion of the interacions . At every time step, the individual nodes calculate the non-bond energy and forces from their lists.

The force routine (listing 3) underwent some additional modification. There was a minor modification to the outer loop of the routine to match that in the non-bond list

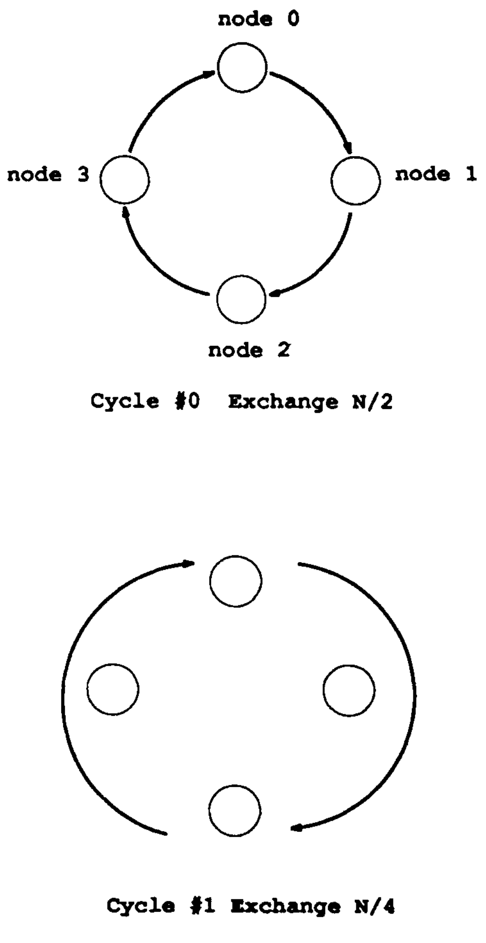

Figure 1 A Distributed communication algorithm. The diagram shows the exchange pattern for 4 nodes. In the first cycle each node sends half of its data to the next node and receives half from the previous node. In the next cycle each node sends one fourth of the data to node i + 2 and receives from i − 2.

structure. The only other major modification is the most crucial: the global combination of the force arrays and energy terms.

We implemented a parallelized, distributed, global combination algorithm [2, 3, 13, 15], using PVM that is used to combine the partially complete force arrays and potential energy values on each node. The global combination runs in parallel, and when finished each node ends up having the complete forces for one unique approximately N/P (N is the size of the number of atoms and P is the number of nodes) and portion of its atoms. The dynamics integrator must be modified to handle this (see below).

Figure 1 contains a schematic diagram of the data flow in this algorithm; where the cluster is viewed to have a ring configuration. Upon completing the force calculation during a time-step, each node calls the routine and executes $\log_2 P$ cycles of communication. On the first cycle each node transmits $\frac{1}{2}$ of its data to *node* + 1 and receives and equal amount from node-1. On the next cycle it sends $\frac{1}{4}$ of its data to *node* + 2 and receives an equal amount from node-1. On the next cycle it sends $\frac{1}{4}$ of its data to *node* + 2 and receives from *node* − 2 and so forth until the last cycle where it exchanges $1/P$ of its data with *nodes* $\pm (\log_2 P - 1)$.

```
   SUBROUTINE UPDATE (RCUT, RLIST, NATOM, STEP, MYNODE, NNODE)
   data declarations
   RLSTSQ = RLIST*RLIST
   Call SAVE (NATOM)
     NLIST = 0
       DO I = MYNODE + 1, NATOM - 1, NNODE
POINT1 (I) = NLIST + 1
        RXI = RX (I)
     ...
       DO J = I + 1, NATOM
       RXIJ = RXI - RX (J)
       ...
       RXIJ = RXIJ - ANINT (RXIJ)
       ...
      RIJSQ = RXIJ*RXIJ + RYIJ*RYIJ + RZIJ*RZIJ
        IF (RIJSQ .LT. RLSTSQ) THEN
     NLIST = NLIST + 1
LIST (NLIST) = J
        ENDIF
      ENDDO
POINT2 (I) = NLIST
   ENDDO
POINT1 (NATOM) = NLIST + 1
POINT2 (NATOM) = NLIST
   RETURN
   END
```

Listing 2: Code fragment showing parallelization of the neighbor list.

```
  SUBROUTINE FORCE (RCUT, RLIST, NATOM, SIGMA, V, W, FX, FY, FZ,
    @ STEP, TIDS, MYNODE, NNODE, IPARPT)
  Data Declarations
  Initialize forces, energy and virial
    DO I = MYNODE + 1, NATOM - 1, NNODE
   JBEG = POINT1 (I)
   JEND = POINT2 (I)
    IF (JBEG .LE. JEND) THEN
     RXI = RX (I)
      ...
     FXI = FX (I)
      ...
DO  JNAB = JBEG, JEND
        Compute forces, potential energy and virial for atoms, j, interacting with atom,
        i and accumulate the forces.
     FXI = FXI + FXIJ
       ...
   FX (J) = FX (J) - FXIJ
       ...
      ENDDO
   FX (I) = FXI
      ...
      ENDIF
  ENDDO
  Distributed combination.
  CALL PVMFGDSUM (MYNODE, TIDS, NNODE, 4, FX, XTEMP, IPARPT, ERR)
  ...
  Global combination.
  CALL PVMFGBLSUM (MYNODE, TIDS, NNODE, 4, V, XTEMP, 1, STATUS)
  CALL PVMFGBLSUM (MYNODE, TIDS, NNODE, 4, W, XTEMP, 1, STATUS)
  RETURN
  END
```

Listing 3: Parallelized force routine code fragment.

```
  Atoms for this processor.
  ASTART = 1 + IPARPT (INODE)
    AFINI = IPARPT(INODE + 1)
      DO I = ASTART, AFINI
     RX (I) = RX (I) + UPX (I)*DT
     RY (I) = RY (I) + UPY (I)*DT
     RZ (I) = RZ (I) + UPZ (I)*DT
   UMX (I) = UPX (I)
   UMY (I) = UPY (I)
   UMZ (I) = UPZ (I)
    ENDDO
```

```
*Do a distributed broadcast of the updated coordinates.*
CALL PVMFDBCAST (MYNODE, TIDS, NNODE, 5, RX, IPARPT, STATUS)
...
```

Listing 4: Parallelized position update loop in integrator.

If the method of inter-node communication permits each node to simultaneously send and receive messages, the combination time is proportional to $(\frac{N}{2} + \frac{N}{4} + \cdots + \frac{N}{P})$ (where N is the size of the data array and P represents the number of nodes). With the currently used FDDI interconnect, where simultaneous sends and receives are not possible, time proportional to $N \log_2 P$ is actually obtained, still much better than the non-parallel time proportional to $NP - 1$ (see Table 1).

The combination time is mainly dependent on the communication rate between nodes and largely determines the parallel efficiency of the program. This, of course, is the second, of the two most significant factors affecting parallel efficiency (the first one being load-balance). The computation/communication rate will largely determine the scalability of an algorithm on a given parallel computer and will often control the choice of appropriate algorithm [10].

The Molecular dynamics integrator

The integrator uses the atomic forces to update the atomic positions and velocities. This part of the program also runs in parallel, since a particular node only updates the positions and velocities of it's atoms (see above). The atomic positions must be exchanged between the nodes, since the force calculation requires the complete set. A parallelized, distributed, global broadcast algorithm has been implemented which is effectively the inverse of the combination algorithm discussed above. The partial kinetic energy values are calculated during the velocity updating and must be combined between the nodes. This quantity is required for constant temperature MD. *The code for all of the necessary global broadcast and summation routines is given in the appendix.*

Table 1 Profile of 16384 Argon Atom MD Calculation.

Number of Nodes	*Total Dynamics Time*	*Global Combination/ Broadcast*		*Force Evaluation*	*Neighbor List Generation*	*Parallel Speedup*
1	277	0.0		32	244	1.00
2	142	3.0	(1.0)[3]	16	122	1.96
4	76	6.5	(2.2)	8	61	3.70
8	45	10.5	(3.3)	4.4	31	6.17

[1]Elapsed wall clock times in seconds for 20 steps of dynamics.
[2]Parallelized using PVM.
[3]Relative time for communication in parentheses. Roughly $O(\log_2 P)$.

Parallel performance

Table 1 summarizes the parallel performance on a test case involving 16384 argon atoms. It also gives a profile of where the time was spent in the various parts of the program. The pair interaction cut-off was approximately 11 Å(3.2σ). The total dynamics time is the quantity used to compute the parallel speedup relative to one node. The global combination and broadcast time incorporates all of the time spent in force accumulation; energy term comdination and coordinate broadcasting. Theforce evaluation time does not include the distributed global combination.

Table 1 reveals that the major portion of the compute time was spent in calculating the neighbor list. This is due to the use of the all-pairs method which is $O(N^2)$. This is largely responsible for the excellent 6 times speedup (relative to 1 node) that we have obtained for an 8 node cluster.

A more efficient linked cell algorithm [6] was incorporated into the program to replace the all-pairs routine. The linked cell algorithm was used to generate a neighbor list. The force routine was unchanged. For this test case, the pair-list generation ran 4 times faster than the all-pairs. When the neighbor list time in table 1 is factored by this number, the parallel speedups relative to 1 node decrease from 1.96, 3.9 and 6.2, with the all-pairs method, to 1.9, 3.1, and 4.3, with the linked cell approach, for 2, 4 and 8 nodes respectively.

As mentioned above, the FDDI interconnect does not provide the capability to allow a node to simultaneously send and receive. Table 1 reveals the effect of this by showing that the relative communication time grows at roughly $N \log_2 P$ instead of N. This is a factor of 3 for 8 nodes. Also note that when the more efficient linked cell algorithm is used, the communication dominated combination and broadcast are responsible for 46% of the total execution time.

Times are in minutes for 100 steps of dynamics. The system contained a total of 14,026 atoms (Myoglobin surrounded by 3830 water molecules). A 12 Å non-bond list and a 10 Å energy cut-off were applied. An 8 node HP735 workstation cluster connected by FDDI was used.

Parallelizing a More General Purpose Program: CHARMM

The program, CHARMM [8] is a general purpose molecular dynamics and mechanics program with widespread use in both industry and academia. Unlike the simple program described in the previous section, all of the interactions in equation 3 are calculated as well as some additional terms. Brooks and Hodoscek [13] originally adapted CHARMM to run on various distributed memory parallel computers such as the Intel Delta using message passing parallel programming techniques as well as on workstation clusters. They kindly provided us with a copy of this program into which we inserted our PVM based communication, global combination and broadcast routines (see the previous section).

There is one main additional consideration in parallelizing CHARMM as compared to the simple MD program: the internal coordinate forces. The calculation of internal coordinate forces and energy is list driven. There are separate interaction lists for the bond angle, torsion and out-of-plane forces. To distribute the work evenly, for each list a node does n/p interactions of a loop over the interactions; where n is the number of interactions and p is the number of nodes.

Table 2 shows the performance results for a 14,026 atom MD calculation that simulates the protein Myoglobin in a 8 Å droplet of water. The details of the calculation are given in the table caption. 8 nodes was 6 times faster that 1 which is very close to the results obtained for the simple Argon simulation program. This is approximately twice the speed of a 1 processor Cray YMP [13].

NEURAL NETWORKS

Method

We have adapted the backpropagation type of artificial neural network (ANN) to run in parallel on a workstation cluster, using programming techniques that are very similar to those used to parallelize the molecular dynamics codes. The resulting program was applied to the problem of protein secondary structure prediction to determine the efficacy of workstation clusters when applied to real-world problems. The actual calculation breaks no new ground in the area of structure prediction.

The *multi-layer feedforward back-propagate* (BP) network [16, 17] consists of an array of interconnected artificial neurons operating in parallel (see Figure 2). Using supervised training (see below), the strengths of the connections can be adjusted so that the network learns the correct responses to a training set of input patterns. If properly constructed and trained, the BP network will be able to properly classify incomplete and noisy patterns that were not in the original training set (i.e. generalize).

The individual neurons are each represented by a non-linear activation function, often the sigmoidal logistic function:

$$o_j = 1(1 + e^{-net_j}) \tag{4}$$

Table 2 Molecular dynamics Simulation of Myoglobin in water, using CHARMM.

# of nodes	*Elapsed Time (Seconds)*	*Parallel Speedup*
1	20.05	1.00
2	10.05	1.99
4	5.38	3.73
8	3.33	6.02

Times are in **minutes** for 100 steps of dynamics. The system contained a total of 14,026 atoms (Myoglobin surrounded by 3830 water molecules). A 12 Å non-bond list and a 10 Å energy cut-off were applied. An 8 node HP735 workstation cluster connected by FDDE was used.

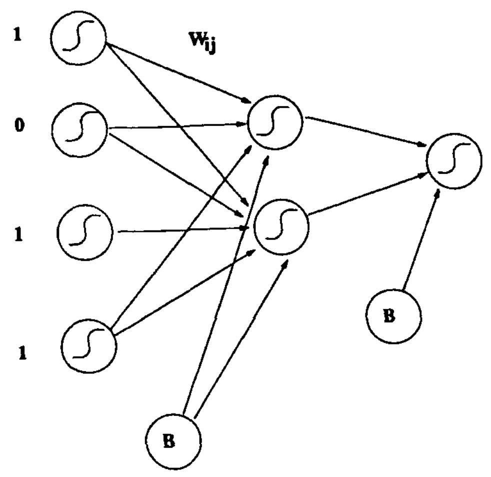

Figure 2 A schematic diagram of a feedforward backpropagation neural network. The circles represent the nodes: the arrows the interconnecting weights

where net_j is the weighted sum of the neuron's inputs minus a threshold bias:

$$net_j = \sum_{i=0}^{n_i - 1} w_{ij} i_i - b_j \tag{5}$$

The terms i_i represent the inputs to the nodes and $w_{i,j}$ corresponds to weights that provide the excitatory and inhibitory strengths of the connections between the nodes. The weights and the threshold biases are the adjustable parameters in this model. The activation of a neuron is its output.

A three layer network was used consisting of an input layer, whose activations (outputs) are the same as its inputs; one hidden layer, to permit second order correlations; and the output layer, whose activations represent the final response of the network to a particular input pattern. We have chosen to encode these inputs as binary values (0, 1).

Supervised training is effected by presenting each pattern in a training set as input and comparing the resulting output layer activations to the desired target values. The weights and the threshold biases are adjusted to minimize a function of this difference, often the mean square error:

$$E = \frac{1}{2np} \sum_{p=1}^{P} \sum_{j=0}^{n-1} (t_{pj} - o_{pj})^2 = \frac{1}{2n} \sum_{p=1}^{P} E_p \tag{6}$$

where n is the number of output neurons; t the target output; o the actual output neuron activation; p is the number of training patterns and n is the number of output neurons. The negative gradient of the error function with respect to the weights is calculated:

$$-\partial E_{p/\partial w_{ij}} = i_i \delta_j \tag{7}$$

For the output layer, δ_j is:

$$\delta_j = o_j(1 - o_j)(t_j - o_j) \tag{8}$$

For the hidden layer(s), chain rule yields:

$$\delta_j = o_j(1 - o_j)\sum_k \delta_k w_{jk} \tag{9}$$

The negative gradient is used to optimize the weights. Most frequently some variation of the steepest descent minimizer has been used [17, 18], though other minimizers such as the conjugate gradient have also been employed [17].

We have found that an alternative approach also works well; the negative gradient can be used in a dynamics-like "leap-frog" integrator (see section?). The total mean square error for a given training set is conservative with respect to the weights. The weights correspond to the atoms of the MD calculation with their values being equivalent to their positions. The masses are arbitrary and can be adjusted to speed convergence. The system can be coupled to a thermostat to obtain termperature control. We used the Berendsen approach [9]. This, along with the inclusion of a minimizer, permits simulated protocols to be employed as a means to overcome the local minima problem [17].

Note that in this approach weight updating is done after a complete presentation of the training set. The alternative is to update after each pattern is presented [19]. The former, in our experience, tends to converge faster and permits the use of the dynamics scheme and efficient cluster parallelization.

A similar approach to parallelization for the network training was taken as was used for molecular dynamics. The gradient arrays are duplicated on each cluster node; the patterns of the training set is partitioned between them. Once per iteration, the partial gradients are combined using the distributed algorithm. Each node updates its set of weights and exchanges them using the distributed broadcast algorithm. This method is efficient only if the training set is large relative to the number of weights. This is the case for the protein secondary structure problem.

A finer grained parallelization technique, of which there are several, would not be effective using the relatively low communication bandwidth of the networked workstation cluster. To fully exploit the parallelism of the neural network as nature does will require custom built computer chips will incredibly large numbers of connections.

Application

The protein secondary structure prediction problem has been addressed by a number of researchers [19–23]. The goal is to predict the type of secondary structure (α-helix, β-sheet, randaom coil) to which each amino acid in a protein sequence belongs. i.e., given the primary structure predict the secondary. In this example we have restricted the experiment to predicting whether or not an amino acid residue is in an α-helix.

The training set was generated by applying the DSSP secondary structure prediction program of Kabsch and Sander [24] to a set of Brookhaven Protein Data Bank crystal coordinate sets. We incorporated the first 64 proteins used by Qian and Sejnowski [19] into the training set. This resulted in a total of 13,895 patterns, one for each residue. Afterwards, a 4521 residue (15 protein) testing set was used to validate the trained network. This set is essentially the first testing set in Qian and Sejnowski's paper [19].

For some of the proteins, more recent crystal structures were used to derive the secondary structure and we used all protein chains.

At the input layer the primary sequences were presented to the network using a windowing technique [19] (see Figure 3). That is, each input pattern consisted of n residues from the primary sequences of the training set. The output of the network was the predicted secondary structure of the middle residue of the window (so n must be odd). Each subsequent pattern was obtained by sliding the window down one residue. Each residue was represented as a 21 bit pattern (one bit per input neuron). Each of the 20 amino acid types was encoded as 20 zero bits and one 1 bit in a unique position. The twentyfirst bit represented a blank to pad the beginning and ends of chains. In this calculation, we used a window size of 17 so that the input layer consisted of $17 \times 21 = 357$ neurons. One neuron was used for the output layer. Ideally, it would register 1 for a helix and 0 for not a helix. In reality, we took $\geqslant .5$ to be a helix and $< .5$ to be not a helix. Performance, in terms of convergence or predictive accuracy, did not improve by using a two neuron output layer.

The hidden layer was made up of two neurons. This is considerably fewer than the number used by other workers [19, 21, 22]. However, it has been shown that the network can be trained on this problem with a small hidden layer [20] and in fact that the lack of one has only small effect on the predictive performance [19, 23]. This implies that the network is extracting mostly first order features from the input (second and higher order would mean that two or more particular residues must be in a given window). Increasing the number of hidden layer neurons can speed up convergence of the training set at the cost of inhibiting the ability of the network to learn general rules for classification [17].

Some modifications were made to the neural network model described above: Superior convergence speed was found when instead of calculating the derivatives of the mean square error (equation 6) the following function suggested by van Ooyen and Nienhus [25] was used:

$$E = \frac{1}{2np} \sum_{p=1}^{P} \sum_{j=0}^{n-1} (t_j \mathrm{In} o_j + (1 - t_j)\mathrm{In}(1 - o_j)) \tag{10}$$

For which the derivative is:

$$\partial E/\partial w_{ij} = (o_j - t_j)i_i \tag{11}$$

instead of,

$$\partial E/\partial w_{ij} = (o_j - t_j)o_j(1 - o_j)i_i \tag{12}$$

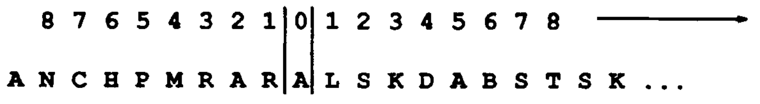

Figure 3 A diagrammatic representation of the input windowing. Each letter represents an amino acid residue. A sequence of 17 amino acids are presented to the network inputs at one time. The window is then moved down one residue to produce the next input pattern (see text)

The $o_j(1 - o_j)$ term in equation 12 causes the derivatives to tend towards zero when the output values are close to their extrema, 0 and 1. Thus the error $t_j - o_j$ could be large without the effect being propagated to the derivatives and, for that reason, the weights.

A further modification was to rescale the derivatives involving the weights between the input and the hidden layers in an 8:1 ratio with respect to the derivatives involving the weights between the hidden and the output layer. This was suggested by Rigler *et al.* [26], as a means to compensate for the reduction in the magnitude of the derivatives with each layer going back from the output layer due to the application of the chain rule, used to obtain them. In the dynamics algorithm this was accomplished by setting the appropriate masses to 1/8 instead of the original value arbitrarily chosen to be 1.

The use of both of the aforementioned modifications enhanced convergence significantly and also avoided the collapse into a false minima (no helices) that appears to have a very wide basin of attraction. Increasing the number of hidden layer neurons up to 20 did not improve convergence.

Results

The use of the dynamics algorithm to update the weights resulted in a convergence with the training set to a mean square error per pattern of 0.06 in 1200 iterations. The reference temperature for the thermostat was set to 0.0001 for the first 100 steps; 0.001 for the next hundred steps and back to 0.0001 for the final 1000 steps. Further conjugate gradient minimization failed to improve upon this.

Note the convergence in spite of the lack of minimization. Interestingly, if a time step was used that was small enough to maintain the equivalent of energy conservation, using the sum of the "Kinetic error" and total mean square error terms, essentially no movement took place on the error surface (with the thermostat turned off, of course). This was not a problem, however, since when the thermostat was applied the system tended to be dissipative and the total mean square error rapidly decreased.The non-conservative behavior may be due to the fact that feedforward neural network error hypersurfaces tend to possess large flat palins in many directions punctuated by steep ravines in a few. These steep ravines would cause some degrees of freedom to have much higher frequency modes than the others.

The traditional steepest descent minimizer was also applied to this problem. The algorithm had two adjustable parameters: the step size (learning rate) and a scaling factor for a damping term based on the previous change in the weights (the momentum term) [16, 18]. Even with hand optimization of these parameters we were not able to match the convergence of the dynamics approach. That is not to say that one of the many variations of the steepest descent method that have appeared in the neural network literature could not match this performance (see for example the adaptive parameter techniques of Tollenaere [27] and Jacobs [28]). Performance of these variations on different problems tends to be rather spotty.

With this size problem the parallel efficiency of the replicated array algorithm on a four node HP735/FDDI workstation cluster was excellent as can be seen in Table 3. There was a 3.6 times speedup with 4 nodes. The parallel efficiency is helped by the fact that there are only 722 interconnections, and thus, derivatives to be communicated

Table 3 Parallel Performance of the Neural Network.

# of nodes	*Training time (seconds)*[a]	*Speedup*
1	113.12	1.0
2	60.68	1.9
4	31.50	3.6

[a]Elapsed wallclock time in seconds.
Protein secondary structure prediction. 50 steps of "dynamics"; weights updated once per interaction.

between the nodes, and that the number of training patterns, 13895, is so much larger than that. Thus the computation/communication ratio is very high.

At the end of training the network would predict 80% of the helix residues correctly and 84% of the non-helix *for the training set proteins*. For non-homologous training set selected by Qian and Sejnowski, the network had a 55% correct prediction rate for helices and 73% for non-helices.

A better indication of predictive ability is the correlation coefficient C_α [29]:

$$C_\alpha = \frac{p_\alpha n_\alpha - u_\alpha o_\alpha}{[(n_\alpha + u_\alpha)(n_\alpha + o_\alpha)(p_\alpha + u_\alpha)(p_\alpha + o_\alpha)]^{1/2}} \tag{13}$$

Where p_α is the number of correct helix predictions; n_α is the number of correct non-helix predictions; u_α is the number of misses and o_α is the number of overpredictions (false positives). $C_\alpha = .31$ for the testing set as compared to Qian and Sejnowski [19] who obtained .35 – 0.41, having trained on a largest set of proteins. This indicates that our dynamics algorithm is converging to reasonable minima on the error surface.

CONCLUSION

In this paper we have shown how workstation clusters can be used to achieve significant levels of computational performance using distributed memory parallel programming techniques. The size of the problems must be fairly large however due to the relatively low bandwidth of the network interface. Even so, the problems presented here have not been of unrealistic size. The 14,000 atoms of the Myoglobin calculation encompasses a single enzyme in a very thin shell of water (8 Å).

We have seen that the computation time/communication time ratio dictates the parallel performance in these applications. The relatively low bandwidth of currently available network interfaces in the main limiting factor. Higher bandwidth (and lower latency) network inerconnection technology will be offered by hardware suppliers in the near future which should greatly improve the parallel performance and make it feasible to increase the number of workstation nodes employed in a single calculation.

Future work will be in the application of domain decomposition techniques. It should be possible to decrease the amount of inter-node communication for large problems.

Program Availability

The simple argon MD program and the neural network package are available from the author upon request. Send e-mail to sfleisch @ convex.com. PVM version 3.1 or higher is required and can be obtained from the developers. Send e-mail to netlib @ ornl.gov or anonymous ftp from cs.utk.edu.

Acknowledgements

The author would like to thank the organizers of the 1993 CCP5 Annual Meeting for their efforts. Convex Computer Corporation (this author's employer) provided the computer resources and other support (salary) for this work. The other members of the computational chemistry group. David Mullally and Murray Smigel, contributed with useful discussions and proof-reading assistance.

APPENDIX: SOURCE CODE FOR GLOBAL BROADCAST AND SUMMATION ROUTINES

The following routine does the global distributed summation.

```
      subroutine pvmfgdsum(node, tids, nproc, msgtype, x, temp, iparpt, err)
      imlicit none
      include 'fpvm3.h'
      integer node, nproc, tids(0:nproc – 1)
      integer msgtype
      real*8 x (0: *), temp(0: *)
      integer iparpt(0:nproc)
      logical err
C---
      integer from, to, tom, ncycles, i, k, mid
      integer c, nc, ns, enode, status, bufid
      logical is-even, first-rcv
      real*8 log2
      real*8 ceil
      external ceil

      log2 = log(2.0D0)
      err = .false.
      if(nproc .le. 1) return

      ncycles = int(ceil(log(dble(nproc))/log2))
      mid = 2**ncycles/2
      do k = 0, ncycles – 1
C---     handle send preparation first
         ns = 0
         call pvmfinitsend(PVMRAW, bufid)
```

```
            do c = 0, nproc − 1
                enode = mod(node-c, nproc)
                if(enode .lt. 0) enode = nproc + enode
                if(enode .ge. mid) then
                  to = enode − mid
                  if(to .ge. 0 .and. to .lt. mid) then
C---            load the buffer
                nc = iparpt(c + 1) − iparpt(c)
                if(nc .gt. 0) then
                  call pvmfpack(REAL, x (iparpt(c)), nc, 1, status)
                  if(status .ne. PvmOk) then
                    err = .true.
                              return
                            endif
                          ns = ns + nc
                        endif
                      endif
                    endif
                  endo
C---            do we send now or do the receive first?
                if (ns.gt.0) then
                  to = node − mid
                  if (to.1t.0) to = nproc + to
                  tom = to/mid
                  is_even = (tom/2)*2.eq.tom
                  if (is_even) then
                    call pvmfsend(tids(to), msgtype, status)
                    if (status.ne.PvmOk) then
                      err = .true.
                      return
                    endif
                  endif
                endif
                first_rcv = .true.
                do c = 0, nproc-1
                  enode = mod(node-c, nproc)
                  if (enode.lt.mid) then
                    from = enode + mid
                    if (from .ge. mid .and. from .lt. nproc) then
                      nc = iparpt(c + 1)-iparpt(c)
                      if(nc.gt.0) then
                        if(first_rcv) then
                          first_rcv = .false.
                          from = mod(node + mid, nproc)
                          call pvmfrecv(tids(from, msgtype, status)
                          if(status.lt.0) then
```

```
                                err = .true.
                                return
                              endif
                            endif
                            call pvmfunpack(REAL8, temp(iparpt(c)), nc, 1, status)
                            if(status.ne.PvmOk) then
                              err = .true.
                              return
                              endif
                              do i = iparpt(c), iparpt(c + 1) - 1
                                  x(i) = x(i) + temp(i)
                            enddo
                          endif
                        endif
                      endif
                    enddo
C---                if send was not done before receive do it here
                    if(ns .gt. 0 .and. .not. is_even) then
                      call pvmfsend(tids(to), msgtype, status)
                      if(status .ne. PvmOk) then
                            err = .true.
                            return
                          endif
                        endif
                        mid = mid/2
                      enddo
                      return
                      end
```

The following routine does the global distributed broadcast.

```
  subroutine pvmfdbcast(node, tids, nproc, msgtype, x, iparpt, err)
  implicit none
  include 'fpvm3.h'
  integer node, nproc, tids, (0:nproc - 1)
  integer mstype
  real*8 x(0:*)
  integer iparpt(0:nproc)
  logical err

C---
  integer twok, twoki, kmax, k
  integer to, from, dest, destm, source
  logical is_even
  real*8 log2
  logical first_snd, first_rcv
```

```
      integer enode, c, nc, ns,status
      real*8 ceil
      external ceil

      err = .false.
      if(nproc .eq. 1) return
      log2 = log(2.0DO)

      Kmas = int(ceil(log(dble(nproc))/log2))

      do k = 0, kmax - 1

C---              2^(kmax - (k + 1))
                  twoki = 2**(kmax - (k + 1))
C---              2^(kmax - k)
                  twok = twok1 + twok1

C---              first do the send
                  ns = 0
                  first_snd = .true.
                  call pvmfinitsend(PVMRAW, status)
        do c = 0, nproc - 1
          enode = mod(node - c, nproc)
          if(enode .lt. 0) enode = nproc + enode
          if(mod(enode, twok) .eq. 0) then
            to = enode + twok1
            if((mod(enode, twok) .eq. 0) .and. (to .lt. nproc)) then
              if(first_snd) then
                first_snd = .false.
                dest = mod(to + c, nproc)
              endif
              nc = iparpt(c + 1) - iparpt(c)
              if(nc.gt. 0) then
                call pvmfpack (REAL8, x(iparpt(c)), nc, 1, status)
                if(status .ne. PvmOk) goto 9999
              endif
              ns = ns + nc
            endif
          endif
        enddo
C---              send now or later?
                    if(ns .gt. 0) then
                      destm = dest/twok
                      is_even = ((destm/2)*2 .eq. destm)
                      if(is_even) then
                        call pvmfsend(tids(dest), msgtype, status)
                  if(status.ne. PvmOk) goto 9999
               endif
            endif
C---        now do the receive
            first_rcv = .true.
            do c = 0, nproc - 1
               enode = mod(node - c, nproc)
               if(enode.lt. 0) enode = nproc + enode
               if((mod(enode, twok).ne. 0).and.(mod(enode, twok1).eq. 0)) then
                  nc = iparpt(c + 1) - iparpt(c)
                  if(nc.gt. 0) then
                     if(first_rcv) then
                        first_rcv = .false.
                        from = enode - twok1
                        source = mod(from + c, nproc)
                        call pvmfrecv(tids(source), msgtype, status)
                        if(status.lt. 0) goto 9999
                     endif
                     c          a          l          l
         pvmfunpack(REAL8, x(iparpt(c)), nc, 1, status)        if(status.ne.
         PvmOk) goto 9999
                  endif
               endif
            enddo
            if(ns.gt. 0.and..not. is_even) then
               call pvmfsend(tids(dest), msgtype, status)
               if(status.ne. PvmOk) goto 9999
            endif
         enddo

   return
9999  err = .true.
   return
   end
```

The following routine does the global summation and re-broadcast.

```
   subroutine pvmfgblsum(node, tids, nproc, msgtype, x, temp, nval, err)
   implicit none
   include 'fpvm3.h'
   integer nproc
   integer tids(0:nproc - 1), node, msgtype, nval
   real*8 x(0:nval - 1), temp(0:nval - 1)
   logical err

   logical is_send, is_recv
```

```
integer from, to, ncycles, status
integer i, k, mid
real*8 log2, twok, twok1
real*8 ceil
external ceil

err = .false.
if(nproc .le. 1) return

log2 = log(2.0D0)

ncycles = int(ceil(log(dble(nproc))/log2))
mid = (2**ncycles)/2
do k = 0, ncycles - 1
   if(node .ge. mid) then
      to = node - mid
      if(to .ge. 0 .and. to .lt. mid) then
         call pvmfinitsend(PVMRAW, status)
         call pvmfpack(REAL8, x, nval, 1, status)
         if(status .ne. PvmOk) then
            err = .true.
            return
         endif
         call pvmfsend(tids(to), msgtype, status)
         if(status .ne. PvmOk) then
            err = .true.
            return
         endif
      endif
   else
      from = node + mid
      if(from .ge. mid .and. from .lt. nproc) then
         call pvmfrecv(tids(from), msgtype, status)
         if(status .lt. 0) then
            err = .true.
            return
         endif
         call pvmfunpack(REAL8, temp, nval, 1, status)
         if(status .ne. PvmOk) then
            err = .true.
            return
         endif
         do i = 0, nval - 1
            x(i) = x(i) + temp(i)
         enddo
      endif
      endif
      mid = mid/2
    enddo
C                broadcast:
                 do k = 0, ncycles - 1
C---             /* 2^(ncycles - (k + 1)) */
       twok1 = 2**(ncycles - (k + 1))
C---             /* 2^(ncycles - k) */
       twok = twok1 + twok1

       if(mod(node, twok).eq. 0) then
          to = node + twok1
          is_send = to .lt. nproc
          is_recv = .false.
       else
          is_recv = mod(node, twok1).eq. 0
          is_send = .false
       endif
       if(is_recv) then
          from = node - twok1
          call pvmfrecv(tids(from), msgtype, status)
          if(status .lt. 0) then
             err = .true.
             return
          endif
          call pvmfunpack(REAL8, x, nval, 1, status)
          if(status .ne. PvmOk) then
             err = .true.
             return
          endif
       else if(is_send) then
          call pvmfinitsend(PVMRAW, status)
          call pvmfpack(REAL8, x, nval, 1, status)
          if(status .ne. PvmOk) then
             err = .true.
             return
          endif
          call pvmfsend(tids(to), msgtype, status)
          if(status .ne. PvmOk) then
             err = .true.
             return
          endif
       endif
    enddo
    return
    end
```

The following function returns the next higher integer of the real number x.

```
      real*8 function ceil(x)
c---  CEILING FUNCTION
      implicit none
      real*8 x

      integer ix
      ix = int(x)
      if(x .gt. float(ix)) ix = ix + 1
      ceil = dble(ix)
      return
      end
```

References

[1] This is evidenced by the presentations given at meetings such as the annual Cluster Computing Workshop at Florida State University and the PVM User's Group meeting.

[2] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors, volume 1,* Prentice Hall, Englewood Cliffs, N.J., 1988, ch. 14.

[3] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek and V.S. Sunderam, "A user's guide to PVM: Parallel virtual machine.", Technical Report ORNL-TM-11826, Oak Ridge National Laboratory, 1991.

[4] R.W. Hockney, "The potential calculation and some applications", *Methods Comput. Phys.,* **9**, 136 (1970).

[5] D. Potter, *Computational Physics,* Wiley, New York, 1972.

[6] M.P. Allen and D.J. Tildesley, *Computer Simulation of Liquids,* Oxford University Press, Inc., Oxford, 1987.

[7] L. Verlet, "Computer 'experiments' on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules.", *Phys. Rev.,* **159**, 98 (1967).

[8] B.R. Brooks, R.E. Bruccolerie, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus, "CHARMM", *J. Comp. Chem.,* **4**, 187 (1983).

[9] H.J.C. Berendsen, J.P.M. Postma, W.F. Van Gunsterin, A. Di Nola and J.R. Haak, "Molecular dynamics with coupling to an external bath", *J. Chem. Phys.,* **81**, 3684 (1984).

[10] D. Fincham, "Parallel computers and molecular simulation", *Molecular Simulation,* **1**, 309 (1987).

[11] D. Fincham and P.J. Mitchell, "Multicomputer molecular dynamics using distributed neighbor lists", *Molecular Simulation,* **7**, 135 (1991).

[12] W. Smith, C. Dean, D. Fincham and D. Tildesley, "DL_POLY A molecular simulation package", *CCP5 Newsletter,* **35**, 15 (1992).

[13] B.R. Brooks and M. Hodoscek, "Parallelization for CHARMM for MIMD machines", *Chemical Design and Automation News,* **7**, 16 (1992).

[14] D. Brown, J.H.R. Clarke, M. Okuda and T. Ymazaki, "A domain decomposition parallelization strategy for molecular dynamics simulations on distributed memory machines", *Computer Physics Communications,* **74**, 67 (1993).

[15] R.A. van de Geijn, "Lapack working note 29", Technical Report CS-91-138, University of Tennessee, 1991.

[16] D. Rumelhart and J. McClelland, *Parallel Distributed Processing,* MIT Press, Cambridge, MA., 1986.

[17] T. Masters, *Practical Neural Network Recipes in C++,* Academic Press, Inc., Boston, 1993, pp. 80–199.

[18] J.A. Freeman and D.M. Skapura, *Neural Networks Algorithms, Applications and Programming Techniques,* Addison-Wesley Publishing Company, Reading, MA, 1991.

[19] N. Qian and T.J. Sejnowski, "Predicting secondary structure of globular proteins using neural network models", *J. Mol. Biol.,* **202**, 865 (1988).

[20] L.H. Holley and M. Karplus, "Protein secondary structure prediction with a neural network", *Proc, Natl. Acad. Sci. USA,* **86**, 152 (1989).

[21] H. Bohr, J. Bohr, S. Brunak, R.M.J. Cotterill, B. Lautrup, L. Norskov, O.H. Olsen and S.B. Petersen, "Protein secondary structure and homology by neural networks", *FEBS Letters*, **241**, 223 (1988).

[22] B. Rost and C. Sander, "Prediction of protein secondary structure at better than 70% accuracy", *J. Mol. Bio.*, **232**, 0 (1993).

[23] M.J. McGregor, T.P. Flores and M.J.E. Sternberg, "Prediction of β-turns in proteins using neural networks", *Protein Engineering*, **2**, 521 (1989).

[24] W. Kabsch and C. Sander, "Dictionary of protein scondary-structure: Pattern recognition of hydrogen bonded and geometrical features", *Biopolymers*, **22**, 2577 (1983).

[25] A. van Ooyen and B. Nienhuis, "Improving the convergence of the black-propagation algorithm", *Nerual Networks*, **5**, 465 (1992).

[26] A.K. Rigler, J.M. Irvine and T.P. Vogl, "Rescaling of variables in back propagation learning", *Neural Networks*, **4**, 225 (1991).

[27] T. Tollenenaere, "Supersab: Fast adaptive back propagation with good scaling properties", *Neural Networks*, **3**, 561 (1990).

[28] R.A. Jacobs, "Increased rates of convergence through learning rate adaptation", *Neural Networks*, **1**, 295 (1988).

[29] B.W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme.", *Biochim. Biophys. Acta*, **405**, 442 (1975).