Programming Environment for a High-Performance Parallel Supercomputer with Intelligent Communication

A. GUNZINGER, B. BÄUMLE, M. FREY, M. KLEBL, M. KOCHEISEN, P. KOHLER, R. MOREL, U. MÜLLER, AND M. ROSENTHAL

Electronics Laboratory, Swiss Federal Institute of Technology (ETH), 8092 Zurich, Switzerland; e-mail: gunzinger@ife.ee.ethz.ch

ABSTRACT

At the Electronics Laboratory of the Swiss Federal Institute of Technology (ETH) in Zürich, the high-performance parallel supercomputer MUSIC (MUlti processor System with Intelligent Communication) has been developed. As applications like neural network simulation and molecular dynamics show, the Electronics Laboratory supercomputer is absolutely on par with those of conventional supercomputers, but electric power requirements are reduced by a factor of 1,000, weight is reduced by a factor of 400, and price is reduced by a factor of 100. Software development is a key issue of such parallel systems. This article focuses on the programming environment of the MUSIC system and on its applications.

1 INTRODUCTION

Parallel computers based on standard microprocessors have proven that for many compute-intensive applications they are able to reach performances comparable to those of classical supercomputers at a much lower cost. Tasks like digital signal processing, the training of neural nets, and simulations in physics and chemistry have a great potential for parallel processing, i.e., they can be divided into several processes that run independently in parallel on different processors of a parallel or a distributed computer. The major limiting factors for the attainable speedup of a multiprocessor system against a single-processor machine are the serial part of the program that cannot be parallelized (Amdahl's Law) and the time lost communicating data among the processes. While nothing can be done about the first, the second can be minimized if the system offers fast, low latency communication among the processing elements (PEs) and the programming model really makes use of the available bandwidth. This includes that the programming environment allows simple generation of efficient parallel code.

Different architectures have been proposed by developers [3, 17–19]. Many systems show a rather poor speedup for applications with low data locality. The goal of the MUSIC [7, 13] project was to design and build a parallel supercomputer system with an improved speedup behavior and to demonstrate its performance with real-world applications. This article focuses on the programming environment of the MUSIC system and on its application.

Received November 1994
Revised September 1995

Scientific Programming, Vol. 5, pp. 25–32 (1996)
CCC 1058-9244/96/010025-08

2 SYSTEM CONCEPT

The goal of building parallel systems is to increase the performance by using several processors working in parallel. In digital signal processing and numerical simulation the most simple computing model for a parallel processors is SPMD (single program multiple data) which means that each processor executes the same program on different data. The main problem in this computing model is the exchange of data among the different PEs. To support this computing model, global broadcast is used: All information produced by one single PE is transferred to all other PEs. The ones interested in that data make a local copy of it. Figure 1 illustrates a typical situation: At the beginning each PE has a local copy of the complete input data set and computes a part of the output data set (in this figure a matrix multiplication is shown). The subsequent communication phase redistributes the data so that each PE has all information necessary to continue with its next processing step. As it can be seen in Figure 1 in the case of a large number of processors, communication administration can become the most critical part. Simulations have shown that this communication administration is even more critical than the network bandwidth.

Here, the concept of intelligent communication (IC) starts: The administration of communication is implemented totally in hardware and runs at the same speed as the communication network itself; the communication becomes *intelligent*. This concept makes better use of the bandwidth of any network applied to complex communication schemes; in such applications 95% of the peak communication rate could be measured.

It was the goal of this project to demonstrate the correctness of this concept.

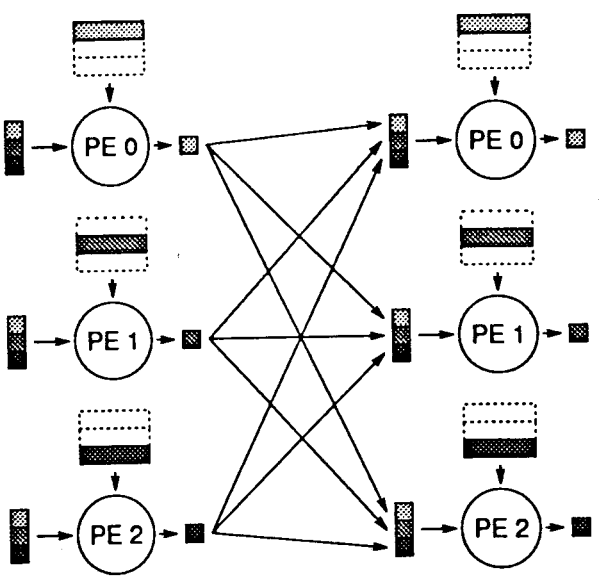

FIGURE 1 Data distribution and collection phases in the MUSIC system.

2.1 System Hardware

Figure 2 gives an overview of the hardware of the MUSIC system. Each board contains three PEs. Each PE consists of a DSP (96002 by Motorola), 2.8 MByte video RAM (VRAM), 0.25..1 MByte static RAM (SRAM), and a communication controller logic cell array (LCA, XILINX XC3090). Because the communication network uses the VRAM's serial port for the communication, its activity affects normal data processing on the DSPs only slightly. Each board also contains a manager (INMOS T805 transputer), connected to the host interface of the DSPs. It is responsible for up- and down-loading of data and code, time measurements, and the dynamic adaption of processor loads. The managers of different boards are connected by their transputer links and form a standard transputer network. For the fast data throughput required by applications such as real-time image processing, special input/output boards can be added. The MUSIC system is connected to the host computer (SUN, PC, MAC) by a transputer link or SCSI connection. The host computer has access to mass storage and user terminals and is responsible for managing the complete MUSIC system from the user's point of view. The communication network is a pipelined ring bus, operating at a 15-MHz clock rate. Its width is 40 bits: 32 data bits and 8 token bits. The tokens contain the identification of the transmitting PE. The IC is implemented in a distributed fashion: Each PE has its own communication controller implemented in a programmable gate array. The IC–controller controls the access from/to the VRAM of the processor to/from the network.

The DSPs run at a 40-MHz clock rate and have a peak performance of 60 MFlops. Notice that the DSP clock rate is independent of the communication clock rate. Up to 21 boards or 63 PEs fit into a standard 19-inch rack, resulting in a 3.8-GFlops system. The power consumption of such a system is less than 800 W (a standard supercomputer uses 400 kW), its weight is 40 kg (the weight of a conventional supercomputer is in the range of 16

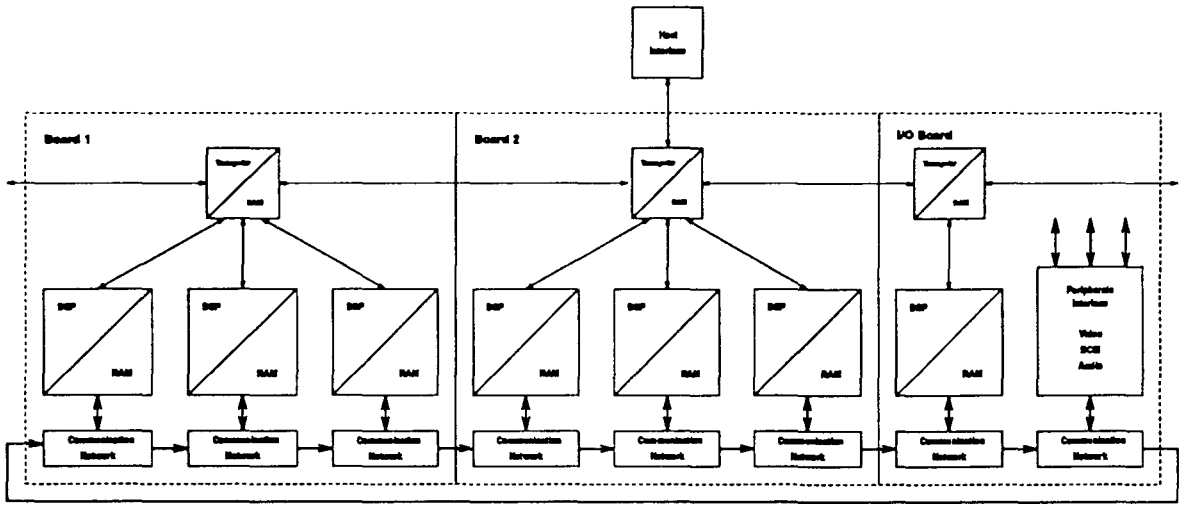

FIGURE 2 Block diagram of the MUSIC system.

tons), and its price is in the range of several thousand dollars (vs. several million dollars for a classical supercomputer).

The MUSIC system also offers different high-speed input/output interfaces such as for audio, video, radar, high-speed network, and hard disks (SCSI). The mobility of the MUSIC system, its low power requirement, and its fast I/O allows it directly at the place of the application, e.g., in a plane, in a car, or for oil exploration.

2.2 System Software

The system software of MUSIC especially supports the implementation of data-parallel algorithms. This means that the same program (algorithm) is executed on several PEs in parallel, but each program is executed on a different data set and produces a different part of the resulting data block. This computing model is also called SPMD. Between two iteration steps a redistribution of the data takes place. Data-parallel iteration steps are often naturally inherent to simulation applications as in physics, chemistry, linear algebra, and neural networks.

The user has to write data-parallel code for one single DSP using C or assembly language for time critical parts in the program (Fig. 3). To write such a parallel program is not essentially more complicated than an implementation on a single processing environment. The main difference is that the data-parallel code must be able to produce only a subset of the resulting data block. Just three functions are needed to control the communication network: `Init_comm()` initializes the communication network for the redistribution of a particular data block. The user program gets all the parameters it needs to run an SPMD program (array sizes, dimensions, and buffer pointers); `Data_ready()` informs the communication network that a new data subset is ready to be transported to other PEs; and `Wait_data()` waits until all expected data values have arrived and are ready to be used for the following computation steps.

The partitioning of the data is basically arbitrary, but the usual way is to tell the operating system only along which axis of a multidimensional data block ($x, y, z \ldots$) the distributions have to be carried out. The actual partitioning scheme is then determined by the operating system at runtime according to the data size and the number of PEs. This means the operating systems on every PE know how many PEs are available, the size of the data array, and in which dimensions this array has to be distributed. According to this, it sets

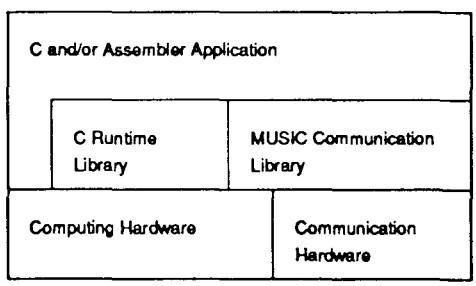

FIGURE 3 Software structure using standard C.

the parameters for the following communication. Because all PEs elements are using the same algorithm to calculate its own data part, no overlap will occur unless otherwise defined. The data partitioning and communication are fully hardware supported for one, two, and three-dimensional data sets. Higher dimensions can be implemented in software.

The standard partitioning method of the operating system is to subdivide the data sets into equally sized pieces. This method works well if the processing time is independent of the data values. Another implemented technique is *dynamic load balancing*. In this case the operating system computes the data distribution according to the computation times of the different PEs in the previous iteration step. In the present solution it is assumed that the computationally intensive areas of a data set differ little between two consecutive iteration steps. The implementation of other load-balancing paradigms is conceivable.

Some languages such as high-performance Fortran (HPF) and high-performance C (HPC) support data-parallel programming. Data distribution and redistribution functions from these languages, the key features for data-parallel programming with HPF and HPC, are directly supported by IC. This makes the implementation of such compilers very easy. No mapping to message passing or writing of new functions has to be done. Figure 4 shows the software structure for HPC. This compiler has been implemented at the Electronics Laboratory [9]. Comparing HPC with a C program with additional parallel functions, most data-parallel applications written in HPC run as fast as in standard C; however, implementation in HPC is much easier.

2.3 Program Example

In the MUSIC system parallelization is done in the data space. This means each PE holds its needed part of the total data and computes its part of the result. After this computation the results are globally communicated and each PE makes a copy of the result data needed for the next calculation. To demonstrate the implementation of a parallel program to the MUSIC system, a simple matrix–vector multiplication is used:

$$x(t+1) = A \cdot x(t)$$

First, the global dimensions of vector x are entered in a `comm_def_t` structure (in this case the order of the vector is 400). The operating system function `Complete_prod_window()` calculates which part of the result data will be on which PE and enters these values (which are different for each PE) in the structure. We use `X_DISTR` here to get a distribution of the data into blocks of similar size. We then have to allocate memory for the number of elements that will be locally on this PE.

```
comm_def_t cd;

cd.dim.x = 400;
cd.dim.y = 1;
cd.dim.z = 1;
cd.elem_size = 1;
Complete_prod_window (&cd, X_DISTR);
Complete_cons_window (&cd, ALL_DISTR,
                      0, 0, 0);
x_old = dmalloc(400, MT_CONS);
x_new = dmalloc(cd.prod.nelements,
                                MT_PROD);
```

Each PE has to calculate only its amount of data, up to the upper local boundary `cd.prod.part.x`. After the calculation, its part of `x_new` has to be communicated and all PEs get the complete result vector. This is done by the following 3-system functions.

The programmer has to look at the problem from a local point of view.

```
MINT iteration(void)
{
  int i, k;
  MINT sum;

  for(i=0; i<cd.prod.part.x; i++)
  {
  sum = 0;
  for (k=0; k<400; k++)
    sum += a[i*400+k]*x_old[k];
  x_new[i]  = sum;
 }
```

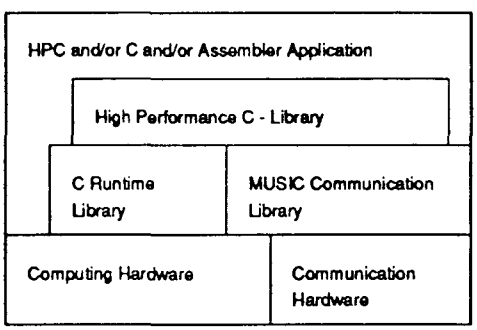

FIGURE 4 Software structure using HPC.

```
  Init_comm(&cd, x_new, x_old,
            COMM_NORM);
  Data_ready(ALL_DATA);
  Wait_data(ALL_DATA);
  return 0;
}
```

HPC[9] is an extension to the C programming language that simplifies data-parallel program ming in distributed memory systems. It supplies constructs that make it easy to distribute data on several PEs and that ease access to this distributed data. In HPC, algorithms can be written from a global point of view in contrast to traditional MUSIC programming which is done from a local point of view. This simplifies application development and makes it easy to migrate software to other data-parallel architectures. To illustrate this approach we show the same example written in HPC.

```
par MINT x_old[400] @ [];
par MINT x_new[400] @ [block];
```

The arrays are declared as par (parallel) variables, which means they are distributed on all PEs. The kind of distribution is stated after the @ character; e.g., array x_new is distributed in a manner that each PE holds a block (of equal size) of the array. Array x_old is not distributed, which means all PEs own a complete copy of it.

The k loop in this version is written with a global index (0 to 400) as it would be in an algorithm for a single processor system. The only difference of HPC to such a program is the first loop in this example. forall is a for loop that can be executed in parallel on every PE. The forall loop is restricted by the on_owner statement to ensure each PE only computes its part of the distributed data. In HPC, the programmer looks at array indices from a global point of view just as if programming a single processor system.

Communication of the distributed vector x_new to x_old is done by just assigning x_new[] to x_old[].

```
par MINT iteration(void)
{
  int i, k;
  MINT sum;

  forall(i=0; i<400; i++)
    on_owner(x_new[i])
    {
        sum = 0;
        for (k=0; k<400; k++)
          sum += a[i][k]*x_old[k];
        x_new[i] = sum;
    }   x_old[] = x_new[];
  return 0;
}
```

Note, the program for both applications is totally independent of the number of PEs; this means, that after compilation a program can run on a 4, 12, or 60 PE system without any changes in the program code. The program loader adjusts the different local counter variables to the right value.

3 APPLICATIONS

Applications of the MUSIC computer include signal processing (audio, video, RADAR), computer graphics, neural networks, and simulations in chemistry and physics.

3.1 Neural Networks

Back-propagation is a very popular algorithm for the learning of layered feed forward neural networks (*multilayer perceptrons*) [16]. Each of the PEs of the MUSIC system computes a subset of the output vector of a specific layer of the neural net. These subsets are collected and a copy of the complete vector is distributed by the communication network to serve as input for the computation of the following layer. To avoid the communication of the updated weights, which would lead to communication saturation very easily, two different weight subsets for the forward and the backward propagation are stored and updated individually on every PE[10].

The measured results and a comparison to other computer systems are given in Table 1. The most critical part in neural net is learning, therefore the performance is measured in MCUPS (million connection updates per second). It tells how many weights may be updated in a second.

On a MUSIC-21 we get 330 MCUPS; this corresponds to 1,408 algorithmic MFlops. However, because the weight update is computed twice, actually 1,870 MFlops are executed. This is about 50% of the peak performance (3.8 GFlops).

We would like to emphasize that the MUSIC system has been designed to be used in research work and therefore has a very high degree of flexibility. That means that it allows almost any modification on the neural network structure and learn-

Table 1. Comparison of Back-Propagation Implementations

System	No. of PEs	Backprop [MCUPS]	Continuous Weight Update
PC (80486, 50 MHz)*	1	0.47	Yes
Sun (Sparcstation 10)*	1	1.1	Yes
Alpha station (150 MHz)*	1	3.2	Yes
Transputer T800 [11]	64	9.9	—
Warp [15]	10	17.0	No
CM-2	64K	40.0	No
CRAY Y-MP C90	1	65.6	Yes
CM-5	512	78.0	No
RAP	40	106	Yes
NEC SX-3**	1	130.0	Yes
MUSIC-21*	63	330	Yes
GF11	356	901	No

* Based on our own measurements.

** Presented by N. Koike of NEC at the 1992 Second ETH-NEC Joint Workshop on Supercomputing (no published reference available).

ing algorithm. Other implementations are much more restricted in this point. The IBM GF11 implementation (900 MCUPS), for instance, parallelizes over the training set. This method only allows batch learning (no immediate weight update), which has the effect that the learning convergence is in many cases much slower.

Convolutional nets are partially connected neural nets with shared weights. Because such nets have a very complicated communication structure, implementation on a parallel computer becomes very difficult. On the MUSIC system convolutional nets were implemented successfully [12].

3.2 Molecular Dynamics

The program MD-Atom is used for time-discrete simulations of the dynamics of atomic fluids. The basic concept of this algorithm is the computation of all partial forces to a single atom generated by all other atoms. This force is considered to be constant in a single time step (10^{-15} to 10^{-14} s) and with Newton's equations of motion a new position is computed.

For each iteration step the following operations are executed: (1) compute the distance between each atom and all other atoms; (2) if the distance is smaller than a cut-off radius R_c, compute the pair force (only atoms in the near neighborhood have an influence); (3) compute sum of all pair forces and make position update [14].

As a benchmark, two models with 125 atoms and 1,000 iterations, and 1000 atoms and 100 iterations, respectively have been chosen [2, 5, 6].

In the MUSIC implementation, the positions are broadcast to all PEs. Each PE does the position update for its share of atoms and broadcasts the new positions again among all PEs. The MUSIC implementation is written in assembly language, as no compiler was available at the time of writing.

To compare the performance of MD-Atom on MUSIC with other supercomputers, an implementation on the NEC SX-3, one on the CRAY YMP, and one on the Sun-4 (IPX) was done by the Laboratory of Physical Chemistry, Swiss Federal Institute of Technology [6, 8]. These implementations were written in Fortran and have been optimized for the respective hardware. As far as we know, these are the fastest implementations of MD-Atom on supercomputers.

The measured results of this comparison are given in Tables 1 and 2. The programs of the Cray

Table 2. Executing Time of MD-Atom on Different Computer Systems

	Model 1 (125/1,000)	Model 2 (1,000/100)
Sun IPX	118	643
CRAY Y-MP	3.7	12.2
NEC SX-3	1.4	4.4
MUSIC-10 (30 PEs)	1.3	3.8
MUSIC-20 (60 PEs)	0.91	2.02

and the NEC supercomputer are using one processor. An implementation for more than one processor has, as far as we know, not been realized yet. As can be seen, the MUSIC system is the fastest in both cases. For the 1,000 atom system running on MUSIC-20, an algorithmic performance of about 500 MFlops was obtained; the MUSIC-20 actually executed 1,163 MFlops.

Apart from the direct computation of distances, other acceleration techniques like pair list are also available on the MUSIC now. The benchmarking of these methods for other supercomputers will be done in the near future.

3.3 EEG Analysis

The quantitative analysis of the human sleep electroencephalogram (EEG) has provided new insights into the processes underlying sleep regulation and given rise to formal mathematical models of sleep regulation. The complexity of the EEG can be estimated by calculating the correlation dimension. This represents a novel approach to exploring the dynamics of sleep and the processes underlying its regulations. Due to the large number of calculations required, only selected short segments (4 to 164 s) of the sleep EEG could be analyzed so far. By using the MUSIC system, whole night EEGs (480 min) of 11 persons were analyzed. A MUSIC system with 21 processors is able to do the calculations in real time [1].

3.4 Plasma Physics

The one-dimensional Particle-In-Cell & Monte Carlo collision code XPDP1 is used to model radio-frequency argon-plasma discharges. The code runs faster on the MUSIC system than on a CRAY Y-MP. The low cost of the MUSIC system allows a 24-h per-day use and the simulation results are available one order of magnitude quicker than with a supercomputer shared with other users. Very good agreement is found between simulation results and measurements done in an experimental argon discharge [4].

4 CONCLUSION

The goal of this project, to build a parallel supercomputer and to demonstrate its performance in real-world applications, could be attained. The low power consumption and small size make it possible to put such a computer on the scientist's desk or use it in many mobile applications. The programming environment allows an easy implementation of data-parallel algorithms achieving a very high performance.

REFERENCES

[1] P. Achermann, R. Hartmann, A. Gunzinger, W. Guggenbühl, and A. A. Borbély, "Correlation dimension of the human sleep electroencephalogram: cyclic changes in the course of the night," *Eur. J. Neurosci.* vol. 6, pp. 497–500.

[2] M. P. Allen and D. J. Tildesley, *Computer Simulations of Liquids*. New York: Oxford University Press, 1987.

[3] M. Annaratone, E. Arnold, T. Gross, H. T. Kung, M. Lam, O. Menzilicioglu, and J. A. Webb, "The WARP computer: architecture, implementation and performance," *IEEE Trans. Comput.*, vol. C-36, no. 12, pp. 1523–1538, Dec. 1987.

[4] M. Fifaz, B. Bäumle, A. Howling, L. Ruegsegger, and W. Schwarzenbach, "Parallel simulation of radio-frequency discharges," in *6th Joint EPS-APS International Conference on Physics Computing*, Lugano, Switzerland, August 1994, pp. 5–8.

[5] W. F. van Gunsteren and H. J. C. Berendsen, "Molecular dynamics computer simulations: methodology, applications and perspectives in chemistry," *Angewandte Chemie*, vol. 29, pp. 992–1023, 1990.

[6] W. F. van Gunsteren, H. J. C. Berendsen, F. Colonna, D. Perahia, J. P. Hollenberg, and D. Lellouch," On searching neighbours in computer simulations of macromolecular systems," *J. Comp. Chem.*, vol. 5, no. 3, pp. 272–279, 1984.

[7] A. Gunzinger, U. Müller, and H. Vonder Mühll, "Architecture and realization of a multi signal processor system," in *Berlin '91*, A. Aliphas, Ed. DSP Associates, 1991, pp. 242–249.

[8] A. Gunzinger, U. A. Müller, W. Scott, B. Bäumle, P. Kohler, H. V. Mühll, F. Müller-Plathe, W. F. van Gunsteren, and W. Guggenbühl, "Achieving supercomputer performance with a DSP array processor, in *Supercomputing '92*, IEEE/ACM, IEEE Computer Society Press, 1992, pp. 543–550.

[9] P. Kalberer and R. Morel, *Parallelisierender Compiler für SPMD-Rechner*. ETH Zürich: Institut für Elektronik, 1994.

[10] W.-M. Lin, V. K. Prasanna, and K. Wojtek Przytula, Algorithmic mapping of neural network models onto parallel simd machines," *IEEE Trans. Computers*, vol. 40, pp. 1390–1401, December 1991.

[11] H. Mühlbein and K. Wolf, "Neural network simu-

lation on parallel computers." *Parallel Computing-89*, in D. J. Evans, G. R. Joubert, and Frans J. Peters, Eds. Amsterdam: North Holland, 1990, pp. 365–374.

[12] U. A. Müller, "Parallel training of convolutional nets with partial baches." *Adv. Neural Information Processing Systems* (NIPS-7) (submitted).

[13] U. A. Müller, B. Bäumle, P. Kohler, A. Gunzinger, and W. Guggenbühl, "Achieving supercomputer performance for neural net simulation with an array of digital signal processors." *IEEE Micro*, vol. 12, pp. 55–65, 1992.

[14] F. Müller-Plathe," Parallelising a molecular dynamics algorithm on a multi-processor workstation." *Computer Phys. Communications*, vol. 61, pp. 285–293, 1990.

[15] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung, "Neural network simulation at warp speed: How we got 17 million connections per second." in *IEEE Int. Conf. Neural Networks*, 1988, p. II-143.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representation by error propagation." in *Parallel Distributet Processing: Explorations in the Microstructure of Cognition*, D. E. Rumelhart and J. L. McClelland, Eds., vol. 1. Cambridge, MA: Bradford Books, 1986, pp. 318–362.

[17] R. R. Shively and L. J. Wu, "Application and packaging of the AT&T DSP3 parallel signal processor." in *Digital Signal Processing-g1*, V. Cappellini and A. G. Constantinides, Eds. New York: Elsevier, 1991.

[18] M. Witbrock and M. Zagha, "An implementation of backpropagation learning on GF11, a large SIMD parallel computer," *Parallel Computing*, vol. 14, pp. 329–346, 1990.

[19] X. Zhang, M. Mckenna, J. P. Mesirov, and D. L. Waltz, "An efficient implementation of the backpropagation algorithm on the connection machine cm-2," in *Advances in Neural Information Processing Systems* (*NIPS-89*), David S. Touretzky, Ed. San Mateo, CA: Morgan Kaufman Publishers, 1990, pp. 801–809.