published in: *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, G.R. Joubert et al. (Editors), Elsevier, 2004.

A comparison of OpenMP and MPI for neural network simulations on a SunFire 6800

Alfred Strey[a]

[a]Department of Neural Information Processing
University of Ulm, D-89069 Ulm, Germany

This paper discusses several possibilities for the parallel implementation of a two-layer artificial neural network on a Symmetric Multiprocessor (SMP). Thread-parallel implementations based on OpenMP and process-parallel implementations based on the MPI communication library are compared. Different data and work partitioning strategies are investigated and the performance of all implementations is evaluated on a SunFire 6800.

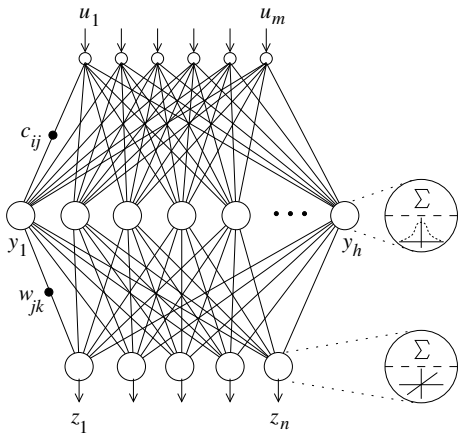
1. Motivation

Since the publication of the computation-intensive error back-propagation training algorithm in 1986, many researchers studied the parallel simulation of artificial neural networks (see e.g. [11] for a good overview). The focus was first on *neuron-parallel* implementations that map the calculations of the neurons representing the main building blocks of neural networks onto different processors. Unfortunately, this biologically realistic implementation strategy suffers from a very high communication/computation ratio. That's why later the *pattern-parallel* simulation was proposed which replicates the neural network in each processor and distributes the training set instead. Here each processor adapts all neural network parameters according to the locally available patterns. Only after each training epoch, the accumulated local updates must be distributed to all processors to compute the new global updates. Thereby the communication/computation ratio can be reduced drastically. However this pattern-parallel implementation requires an epoch-based learning algorithm that modifies the behavior of the original algorithm. Furthermore it is available only for several neural network models. Thus the neuron-parallel implementation is favored by many neural network researchers despite of its higher communication costs.

Symmetric Multiprocessors (SMPs) represent an important parallel computer class. Here several CPUs and memories are closely coupled by a system bus or by a fast interconnect (e.g. a crossbar). Today SMPs are used as stand-alone parallel systems with up to about 64 CPUs or as high-performance compute nodes of clusters. SMPs offer a short latency (a few μs) and a very high memory bandwidth (several 100 Mbyte/s). Thus they seem to be well suited also for a communication-intensive *neuron-parallel* neural network implementation. However no performance analysis of neural network simulations on current SMPs has already been published. So in this paper the results of an experimental study about neuron-parallel implementations of a radial basis function (RBF) network on a SunFire 6800 are presented. Furthermore, two different important parallel programming paradigms that are available for current SMPs are compared: OpenMP for the generation of parallel threads [6] and the communication library MPI.

2. The RBF network

The RBF network represents a typical artificial neural network model suitable for many approximation or classification tasks [8]. It consists of two neuron layers with different functionality. The RBF neurons in the first layer are fully connected to all input nodes (see Fig. 1). Here neuron j computes the squared Euclidean distance x_j between an input vector $\mathbf{u}$ and the weight vector $\mathbf{c}_j$ (represented by the jth column of the weight matrix C). A radial symmetric function f (typically a Gaussian function) is applied to x_j and the resulting outputs y_j of the RBF neurons are communicated via weighted links w_{jk} to the linear neurons of the output layer where the sum z_k is calculated.



RBF training algorithm:

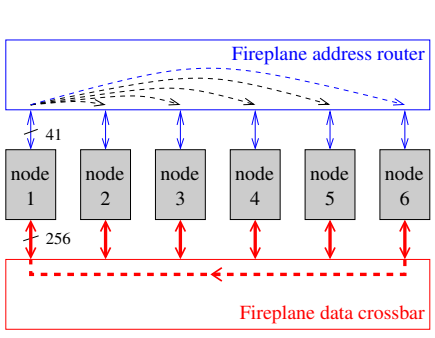
$$x_j = \sum_{i=1}^{m} (u_i - c_{ij})^2 \quad (1)$$
$$y_j = f(x_j) = e^{-x_j/2\sigma_j^2} = e^{-x_j s_j} \quad (2)$$
$$z_k = \sum_{j=1}^{h} y_j w_{jk} \quad (3)$$
$$\delta_k^{(z)} = t_k - z_k \quad (4)$$
$$\delta_j^{(y)} = \sum_{k=1}^{n} \delta_k w_{jk} \quad (5)$$
$$s_j = s_j - \eta_s x_j y_j \delta_j^{(y)} \quad (6)$$
$$w_{jk} = w_{jk} + \eta_w y_j \delta_k^{(z)} \quad (7)$$
$$c_{ij} = c_{ij} + \eta_c (u_i - c_{ij}) \delta_j^{(y)} y_j s_j \quad (8)$$

Figure 1. Architecture and training algorithm of a m-h-n RBF network (with m input nodes, h RBF nodes and n linear output nodes)

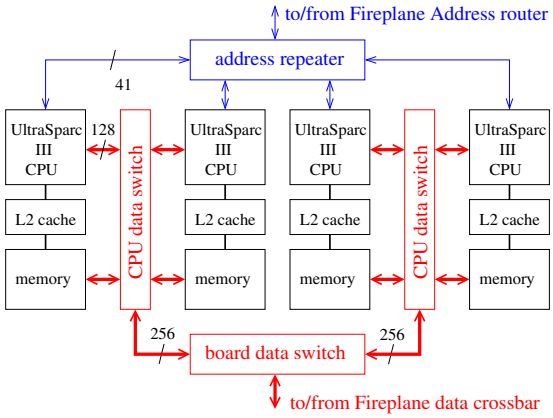
After a proper initialization of the weights, the network is trained by a gradient-descent training algorithm that adapts simultaneously all weights c_{ij}, w_{jk} and σ_j (the center coordinates, heights and widths of the Gaussian bells) according to the error at the network outputs [10]. The complete RBF training algorithm is listed in the right half of Fig. 1.

3. Architecture of the SunFire 6800

All implementations are evaluated on a SunFire 6800 that represents a fast shared-memory architecture with a broadcast-based cache-coherency scheme. It contains 24 UltraSparc III CPUs operating at 900 MHz. The CPUs are connected by a special hierarchical crossbar-based network called *FirePlane* built by 256-bit data lines and 41-bit address lines. A fast address router (see Fig. 2A) replicates each address sent by one CPU in up to 15 clock cycles to all other CPUs that compare it with the addresses of the lines in the local cache [3]. Cache coherency is achieved by a MOESI protocol (i.e. a MESI protocol with an additional *Owned* state). In case of a cache hit the CPU that owns the requested data in its cache sends the data block via the hierarchical data crossbar network to the requesting CPU. The network bandwidth is up to 9.6 Gbyte/s. Each node is built of four CPUs that are arranged in two identical SMPs (see Fig. 2B). Two neighbor CPUs and their local memories are connected by a *CPU data switch*, two neighbor SMPs are connected by a *board data switch*. The memory latency (for accessing a 64-byte data block) is about 200ns, the maximum memory bandwidth is 2.4 Gbyte/s.



A) system architecture (I/O nodes not shown)



B) architecture of a compute node

Figure 2. Architecture of the SunFire 6800 (simplified)

4. Partitioning of the RBF network

From a programmer's point a view, the SunFire 6800 can be considered either as a distributed-memory or a shared-memory architecture. In the former case, the user must explicitly map all data structures to the distributed memory. In the latter case, the user is not explicitly concerned with the data distribution. However he must identify all data variables that are private for each thread.

As already mentioned in Section 1, a neuron-parallel implementation of the RBF network is preferred and analyzed here. Therefore, the two weight matrices C and W and the vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, $\boldsymbol{\delta}^{(z)}$ and $\boldsymbol{\delta}^{(y)}$ describing specific neuron signals should be distributed over all p compute nodes in an appropriate way. Several possibilities have been suggested to distribute the variables of a two-layer neural network:

A) The so-called *partial sum method* (see Fig. 3A) was proposed in 1988 by Pomerleau et al. [9]. Here the rows of both matrices C and W are distributed over all p nodes in the same way: Each node holds an equal number of h/p columns from C and n/p columns from W. It computes the h/p local components of $\mathbf{x}$ and $\mathbf{y}$. For the calculation of the n/p local elements of $\mathbf{z}$ and $\boldsymbol{\delta}^{(z)}$, each node must first gather all components of the distributed vector $\mathbf{y}$ by an *allgather* operation. To adapt the weights of the matrix C, the error $\boldsymbol{\delta}^{(y)}$ in the hidden layer must be calculated first. The i-th element of $\boldsymbol{\delta}^{(y)}$ is basically a dot product of the vector $\boldsymbol{\delta}^{(z)}$ with the row i of the matrix W. However the elements of each row are distributed over all p nodes. To avoid an expensive redistribution of the elements, each node computes first a partial sum $\sum_k \delta_k^{(z)} w_{jk}$. Thereafter all total sums $\boldsymbol{\delta}^{(y)}$ can be computed by an *allreduce* operation. It requires the communication of ph elements, the calculation of h sums and the distribution of the sums to all p nodes. All remaining operations can be implemented locally.

B) To eliminate the costly *allreduce* operation, Kerckhoff proposed the *recomputation method* [4]. Here the matrix W is stored twice: n/p columns and h/p rows of W are mapped onto each node. Thus besides of the matrix W also the transposed matrix W^T is distributed over all p nodes (see Fig. 3B). In consequence, the computation of $\boldsymbol{\delta}^{(y)}$ can be performed locally without any communication, if $\boldsymbol{\delta}^{(z)}$ is available in each node. In total, two *allgather* operations are required here to gather the components of the vectors y and $\boldsymbol{\delta}^{(z)}$ in all nodes. Compared to the partial sum method, the communication requirements are reduced especially for neural networks with large hidden layers. However the update of the weights w_{jk} according to Eq. (7) of Fig. 1 must be calculated for both W and W^T, leading to $3hn/p$ additional arithmetic operations.

C) In the *simplified method* shown in Fig. 3C the rows of the matrix C and the *columns* of the matrix W are distributed over all nodes. Here only one communication operation is required: The vector z that must be available in each node is computed by an *allreduce* operation. All other operations can be performed locally.

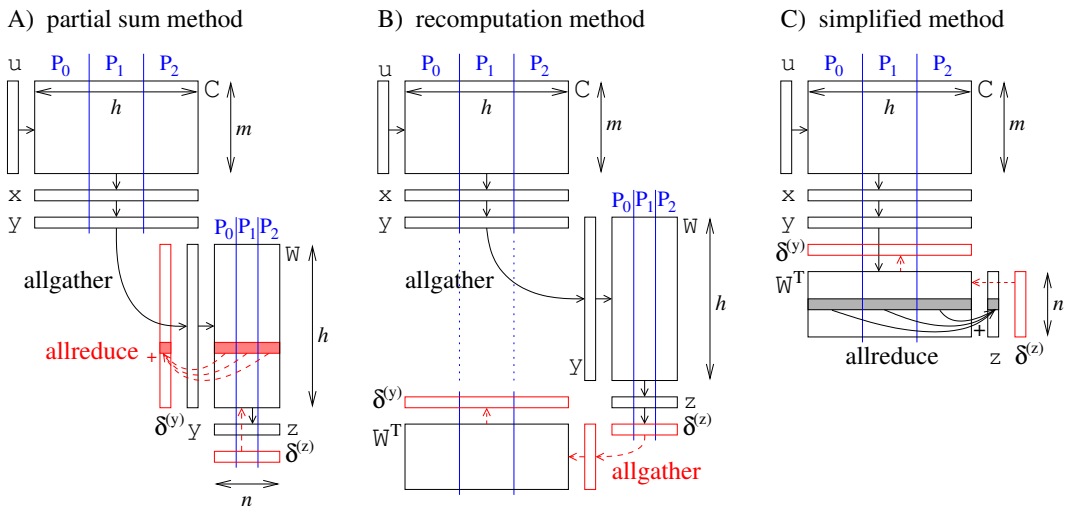


Figure 3. Three different data partitioning methods for an m-h-n RBF network

5. Implementation

The RBF network was implemented in C (Sun Forte 7 C compiler) with Sun MPI (HPC Cluster Tools 4). Compiler optimizations were switched on (`-fast -xarch=v8plusb -xtarget=ultra3`). According to the three partitioning methods described in Section 4, three codes were generated and tested. The OpenMP implementation (Sun OpenMP from Forte Developer 7) of the RBF network was based on a fast sequential C code that was enhanced with OpenMP pragmas. Here also three versions were generated that reflect the different partitioning strategies. Although OpenMP does not allow a manual distribution of data elements, the `for` directives were inserted before the main loop constructs in such a way that the work is shared among the generated threads according to Figures 3A, 3B, and 3C. In addition, the original code was modified to reflect the transposed matrix W^T for the simplified and the recomputation method. To reduce the overhead for creating the threads in each iteration, all threads were generated only once at the beginning. Furthermore, the `nowait` attribute was inserted wherever the implicit barrier synchronization at the end of a parallel `for` loop was superfluous.

In both cases the performance was measured on a SunFire 6800 running the operating system Solaris 9. The number of parallel processes or threads and the size of the RBF network were varied.

6. Performance Analysis

For all experiments the speedup related to the optimized sequential version (without any OpenMP directives or MPI function calls) was calculated. Fig. 4 displays the speedup achieved by OpenMP-based and MPI-based parallel implementations for small and large RBF networks. It can be seen that the performance strongly depends on the selected data partitioning strategy:

- For all OpenMP implementations the *recomputation method* delivers mostly a higher performance than the *simplified method*. Only with one or two threads the recomputation method is slower because of its higher computational effort. The *partial sum method* is extremely slow: the speedup always remains below 1.
- For MPI implementations, the *simplified method* usually leads to the best performance. Especially for large neural networks an approximately linear speedup can be gained. The difference between the three methods is by far less significant than with OpenMP.
- MPI implementations are mostly faster than OpenMP implementations.

The lower performance of the OpenMP implementations was unexpected, because the SunFire 6800 is a symmetric multiprocessor and OpenMP is claimed to be a programming interface for such architectures. Also the advantage of the *recomputation method* that requires a lot of redundant arithmetic operations was surprising. Therefore, a detailed performance analysis was undertaken to

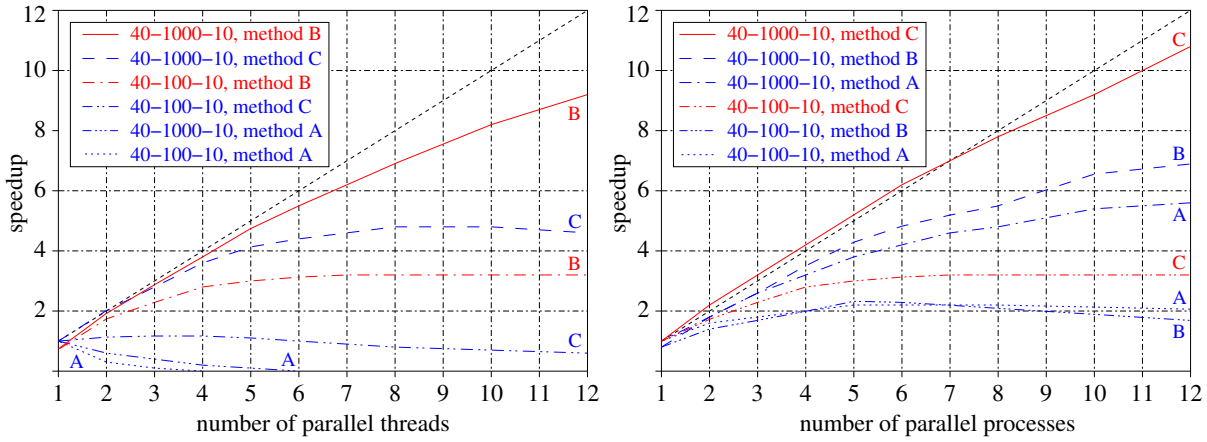


Figure 4. Speedup measured on a SunFire 6800 for the simulation of an *m-h-n* RBF network with OpenMP (left) and MPI (right)

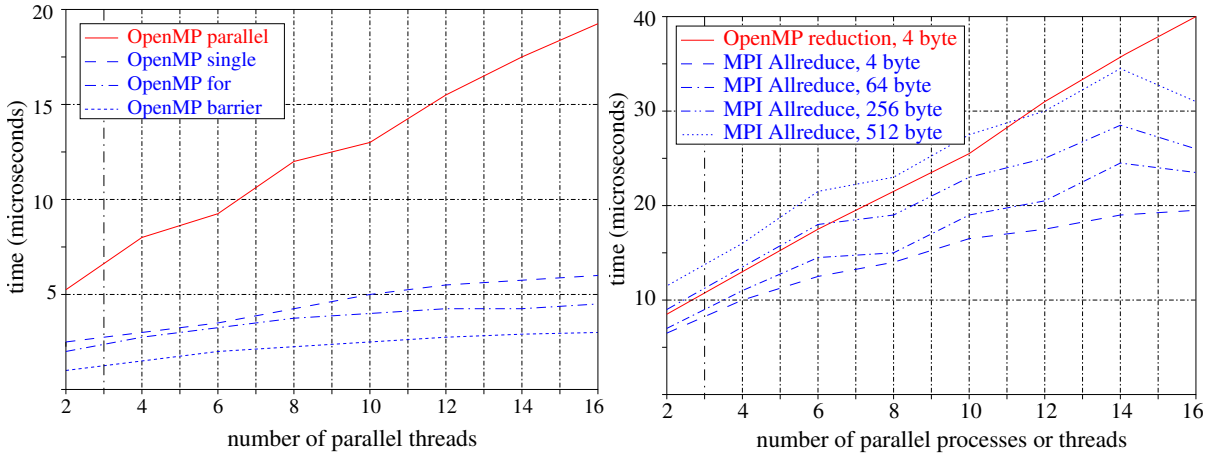


Figure 5. Synchronization time for several OpenMP directives (left) and execution time for a reduction with OpenMP and MPI (right)

find out the performance bottlenecks. To study the OpenMP overhead for process synchronization and scheduling, the OpenMP microbenchmark was applied [2]. Some results are shown in Fig. 5 (left). It can be seen that the synchronization overhead for must OpenMP directives is low and rather independent of the number of threads. The costly `parallel` directive is applied only once at the beginning. The performance of the collective MPI operations was analyzed by the Pallas MPI benchmark suite [7]. From Fig. 5 (right) it becomes evident that on the SunFire 6800 the MPI function `Allreduce` is by far more efficient than its OpenMP counterpart. Furthermore, the function `Allreduce` also allows the reduction of arrays by a single call, whereas the OpenMP `reduction` clause is – according to the OpenMP specification of C/C++ version 1.0 – not available for arrays [6]. Therefore the *recomputation method* representing the only data partitioning method that requires absolutely no reduction achieves the highest performance.

To avoid the costly OpenMP `reduction` clause, several encoding alternatives with other OpenMP directives have been implemented and analyzed. The best performance was achieved by calculating first *all* local sums in each thread and then adding the local sums to the total sums in *one* OpenMP

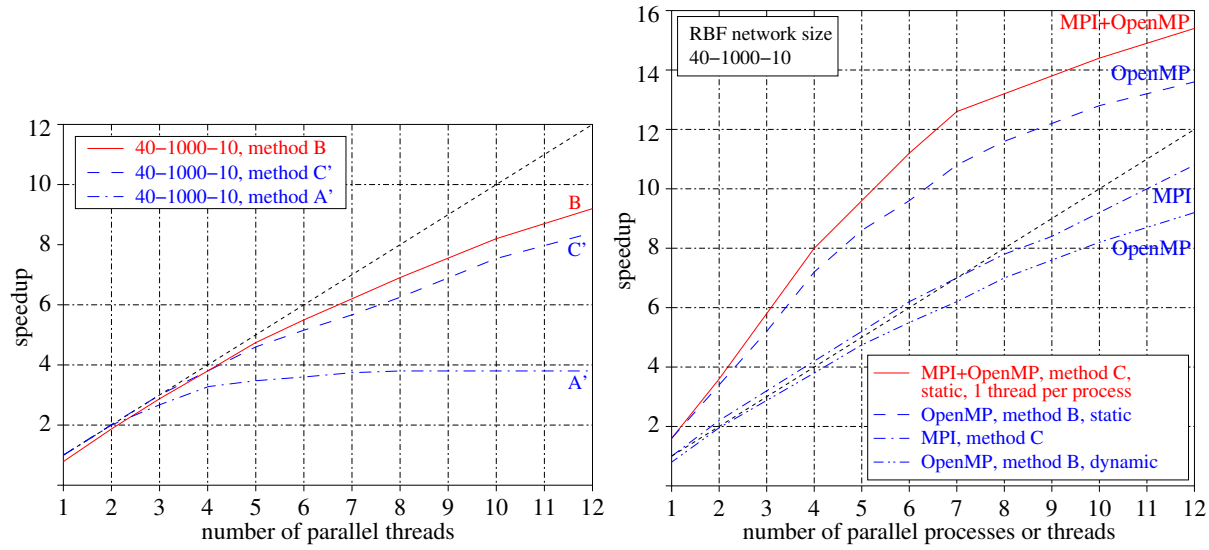


Figure 6. Performance of several implementation alternatives: using the `critical` directive instead of the `reduction` clause (left), using static instead of dynamic arrays (right)

`critical` section. Fig. 6 (left) shows the performance of the accordingly modified methods A' and C'. They are faster than the original versions A and C based on the `reduction` clause (compare Fig. 4). Now the simplified method C' is for up to 4 threads even slightly faster than the recomputation method B. However for more than 4 threads the recomputation method still gains a higher performance.

Replacing the dynamically allocated data arrays by statically declared arrays of known size has lead to some surprising results: Whereas for the sequential reference code and for all MPI-based implementations the performance remained unchanged, the OpenMP-based implementations ran significantly faster (even if only one thread was employed). Even a superlinear speedup can be achieved here (see the dashed curve in Fig. 6 right). This effect is caused by the Sun C compiler that can better exploit the OpenMP directives in loops with static arrays to generate a faster code by using enhanced loop optimizations.

As final experiment, OpenMP directives were inserted also in the MPI-based implementation (with static data arrays) and the number of threads per process was varied. The upper curve in Fig. 6 (right) shows that the mixed MPI/OpenMP implementation delivered a higher performance than the pure MPI or OpenMP solutions. However generating two or more threads per process proved to be not advantageous here.

7. Conclusion

It was shown that the SunFire 6800 as a typical SMP is basically well suited for the communication-intensive neuron-parallel implementation of artificial neural networks. However non-trivial mappings of the neural network data structures and several resulting code modifications are required to achieve a high performance. Some strange artefacts with OpenMP for the Sun C compiler were presented. The MPI-based implementations turned out to be mostly slightly faster than the OpenMP-based implementations. This corresponds to the results of some related studies, that also compare MPI and OpenMP on shared-memory architectures, but for other applications (e.g. [1] [5]). The reason for the lower speedup of OpenMP-based implementation seems to be the unavoidable false cache sharing of data arrays that are frequently modified by several threads. This will be a topic of future investigation.

REFERENCES

[1] M. Bane, R. Keller, M. Pettipher, and I. Smith. A Comparison of MPI and OpenMP Implementations of a Finite Element Analysis Code. In *Proceedings of Cray User Group Summit (CUG 2000), May 22-26*, 2000.

[2] J.M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP, Lund, Sweden*, pages 99–105, 1999.

[3] A. E. Charlesworth. The Sun Fireplane System Interconnect. In *Proceedings of Supercomputing*, 2001.

[4] E.J.H. Kerckhoffs, F.W. Wedman, and E.E.E. Frietman. Speeding up backpropagation training on a hypercube computer. *Neurocomputing*, 4:43–63, 1992.

[5] G. Krawezik and F. Cappello. Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. In *Proceedings of the 15. ACM Symposium on Parallel Algorithms (SPAA 2003)*, pages 118–127, 2003.

[6] OpenMP C/C++ Specification Version 1.0. OpenMP Architecture Review Board, available from `http://www.openmp.org`, 1998.

[7] Pallas MPI Benchmark Suite Version 2.2. Pallas GmbH, Brühl (Germany), available from `http://www.pallas.com/e/products/pmb`.

[8] T. Poggio and F. Girosi. Networks for approximation and learning. *Proceedings of the IEEE*, 78:1481–1497, 1990.

[9] 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 *Proc. 1988 IEEE International Conference on Neural Networks*, pages 143–150, San Diego, Ca., 1988.

[10] F. Schwenker, H.K. Kestler, and G. Palm. Three Learning Phases for Radial Basis Function Networks. *Neural Networks*, 14:439–458, 2001.

[11] N.B. Šerbedžija. Simulating Artificial Neural Networks on Parallel Architectures. *Computer*, 29(3):56–63, 1996.