Chapter 9

Digital Simulation of Spiking Neural Networks

A. Jahnke, U. Roth and T. Schönauer

9.1 Introduction

The motivation of digital simulation of artificial neural networks is as diverse as the background of the researchers. Biologists, mathematicians, physicists, psychologists, computer scientists or engineers, all contribute to the field of neural networks. In one case, a very complex model of a single neuron has to be simulated. In another case, a large network of usually simpler neurons must be taken care of. Due to programmability, digital hardware offers a high degree of flexibility and provides a platform for simulations on neuron level as well as on network level. In opposite to signals in nature, digital signals are discrete in value and in time. Therefore, for digital simulations a suitable neuron model needs to be derived. Hence, at first we examine the digital representations of spiking neuron models in subsection 9.2. Then, subsection 9.3 gives an overview of programming environments. The subsequent subsections 9.5-9.7 address the speed-up of pulse-coded neural network simulations by algorithmic improvements and by taking advantage of high-performance hardware.

If simulation time is critical, the first step is to increase the efficiency of the simulation algorithm. Several concepts to reduce simulation time are introduced in subsection 9.4. Especially large networks, very complex models and/or real-time requirements necessitate high-performance hardware. A fair amount of dedicated digital and analog hardware for neural networks has been developed in the past years [Ienne, 1997]. Analog hardware profits from many analogies of neurobiology and the physics of semiconductors and allows the building of very compact, low-power sensor-integrated systems. Analog VLSI neural networks are discussed in chapter 3. The strength of digital hardware is based on reprogrammability and uncomplicated data memories as well as a continuously increasing speed and density of logical circuits. Supercomputers represent high-performance general-purpose capabilities, but are constrained by costs and programming complexity. Dedicated digital hardware exploits the characteristics of types or classes of neural networks and thereby allows an implementation with a better ratio of performance to costs. An essential approach of high-performance digital hardware is to parallelize. Hence, subsection 9.5 describes how networks can be mapped on a parallel computer. Subsection 9.6 finally

evaluates the performance of some digital hardware platforms for the simulation of spiking neural networks.

9.2 Implementation Issues of Pulse-Coded Neural Networks

A detailed overview of neuron models was given in Chapter 1 and Chapter 2 with models ranging from simple integrate-and-fire models to very complex compartmental models such as the Hodgkin-Huxley model. In Section 1.2, a generic Spike Response Model was defined, which represents an intermediate description level. In the current chapter we focus on such types of models. The mathematical description, as already given in Section 1.2.1, is summarized here for convenience:

$$u_i(t) = \sum_{t_i^{(f)} \epsilon \mathcal{F}_i} \eta_i \left(t - t_i^{(f)}\right) + \sum_{j \epsilon \Gamma_i} \sum_{t_j^{(f)} \epsilon \mathcal{F}_j} \varepsilon_{ij} \left(t - t_j^{(f)}\right) \tag{9.1}$$

$$\mathcal{F}_i = \{t_i^{(f)};\ 1 \leq f \leq n\} = \{t \mid u_i(t) = \vartheta\}. \tag{9.2}$$

$$\Gamma_i = \{j \mid j \text{ presynaptic to } i\}. \tag{9.3}$$

where
- $u_i(t)$: membrane potential of neuron i.
- $t_i^{(f)}, t_j^{(f)}$: firing time of neuron i and neuron j, respectively.
- $\mathcal{F}_i$: set of all firing times of neuron i.
- ϑ: if threshold ϑ is reached by $u_i(t)$, neuron i emits a spike.
- Γ_i: set of neurons j presynaptic to i.
- $\eta_i(\cdot)$: kernel to model the refractory period.
- $\epsilon_{ij}(\cdot)$: kernel to model the postsynaptic potential of neuron i induced by a spike of neuron j.

For the sake of a better illustration, we now choose simple filters for the kernels η_i and ϵ_{ij} to construct an example model neuron. We assume that the refractoriness of the neuron is modeled by the kernel definition

$$\eta_i(s) = -\eta_0 \exp(-\frac{s}{\tau_i})\mathcal{H}(s) \tag{9.4}$$

where $s = \left(t - t_i^{(f)}\right)$, τ is the decay time constant, $\mathcal{H}$ represents the Heavyside step function and the constant η_0 resembles the amplitude of the relative refractoriness. The response to presynaptic spikes we model with the kernel

$$\epsilon_{ij}(s) = w_{ij} \exp\left(-\frac{s}{\tau_{ij}}\right) \mathcal{H}(s) \tag{9.5}$$

where τ_{ij} is another decay time constant and w_{ij} is the synaptical strength. Now, the expression

$$\sum_{t_j^{(f)} \epsilon \mathcal{F}_j} \exp(-\frac{s}{\tau_{ij}})\mathcal{H}(s) \tag{9.6}$$

can be interpreted as a leaky integrator, which accumulates incoming spikes from the neuron j and decreases them in time according to their time of arrival and the decay constant τ_{ij}. Correspondingly, in the expression

$$-\eta_0 \sum_{t_i^{(f)} \epsilon \mathcal{F}_i} \exp(-\frac{s}{\tau_i})\mathcal{H}(s) \tag{9.7}$$

the sum can also be represented by a leaky integrator, which accumulates spikes fed back from the neuron i and decays them in time with the time constant τ_i. The example neuron is

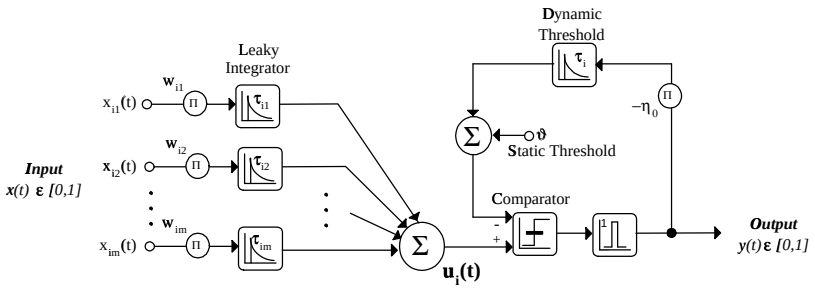

Figure 9.1: Example model neuron with leaky-integrators

illustrated graphically in Fig. 9.1. At any time, a spike might arrive at the input or might be generated at the output: the model is time-continuous. For digital signals on the other hand, both time and amplitude are discrete. Therefore a discrete-time model needs to be derived and its values need to be digitized with a certain resolution.

9.2.1 Discrete-time simulation

To compute a network in discrete-time, a continuous-time period is divided in intervals of a constant duration T. The period T is commonly referred to as time slice, time slot, bin or time step. Within a time slice the new state of the network is calculated. That means all neurons are computed based on the inputs they receive (e.g. spikes generated during the previous time slice) and their internal state variables. The result of this computation is a new state of the network including output spikes generated within this time slice. They become input to postsynaptic neurons during the following time slice. Fig. 9.2 shows a discrete-time representation of the

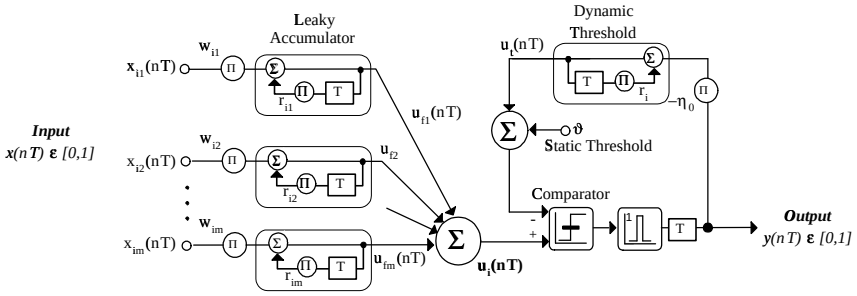

Figure 9.2: Example model neuron in discrete-time with first-order recursive digital filters

example neuron model. Compared to Fig. 9.1, the leaky integrators are now substituted by

first-order recursive digital filters with a relaxation factor relating to the time constant τ

$$r = \exp(-\frac{T}{\tau}) \tag{9.8}$$

yielding a discrete-time recursive relation for the membrane potential $u_i(n)$

$$u_i(n) = -\{\eta_0 y(n) + r_i u_{ti}(n-1) + \vartheta\} + \sum_{j\epsilon\Gamma_i}\{w_{ij}x_{ij}(n) + r_{ij}u_{fij}(n-1)\} \tag{9.9}$$

where u_{fij} denotes the output of each filter element in the input branches of the model neuron and u_{ti} represents the output of the filter in the threshold loop (see Fig. 9.2).

A period T needs to be chosen in such a way, that the behavior of the neural networks is resolved sufficiently in time while not wasting computational resources by computing steps which are too fine. A commonly chosen value for a time slice is 1ms. This interval represents approximately the duration of an action potential. Therefore, a simulation is regarded a real-time simulation, if a time-slice is computed within 1ms or less.

In order to reduce the computational requirements, the model neuron in Fig. 9.2 can be simplified as already mentioned in section 1.2.3.1. The simplification is to assume that several input branches x_{ij} decay with the same time constant τ_{ij}. In this case the summing of these input branches can be performed before the leaky integration. Thereby several synapses represented by the input branches x_{ij} share one filter. This simplified example model neuron will be considered in the following sections.

9.2.2 Requisite arithmetic precision

For simulations on workstations, floating-point representation is a natural choice. Such computers are equipped with fast floating-point units, and so the computation with values in integer or fixed-point representation does not yield an advantage. The situation is quite different for neurocomputers and dedicated digital neuro chips. These systems almost never provide a floating point unit, because it would require significant design effort and a large chip area. So the question arises what the requisite precision for computation with spiking neurons is [Roth et al., 1995]. Regarding the representation with fixed-point numbers, we have to know the requisite precision $i.f$, where i represents the bit length of the integer part and f the bit length of the fraction part. The representation using fixed-point numbers compared to integer numbers has the major advantage that we can change the scaling by shifting the decimal point without changing the total word length $i+f$. The maximum representable value $v_{lim} = 2^i - 2^{-f}$ will be called limitation value. A quantity exceeding this value will be set to the limitation value. The minimum representable number $q = 2^{-f}$ is called quantization step. Quantities smaller than the quantization step will be truncated. In order to determine the requisite arithmetic precision, we need a lower bound for the limitation value and an upper bound for the quantization step. We will focus our discussion on the values of the filter functions. The requisite resolution of other values like membrane potentials or synaptic strength can be derived from the results.

A criterion for the lower bound of the limitation value is easily established. Proper network function demands that the limitation value for the output value of a filter element is larger than the maximum quantity u_{max} this output value may reach:

$$v_{lim} > 2^n \text{ and } n = \text{ceil}\,(ld(u_{max})) \tag{9.10}$$

where ceil() yields the next greater integer number. For a stationary input sequence u_{max} can be computed for our kernel function as the limit value of the sum over a geometric series:

$$u_{max} = \frac{w}{1 - r^k} \tag{9.11}$$

where k denotes the mean time interval between two spikes, and w the mean synaptic strength averaged over time and the various inputs of a filter element.

Computation with truncation has two effects: weights with smaller magnitude are added (weight quantization) and the values of the potentials are lowered after each multiplication with the relaxation factor (arithmetic error). Thus, output values of the filter elements are decaying at a faster rate and arrive at zero, when their value falls below q. As a criterion for the upper bound of the quantization step q we demand, that a received spike should at least be able to have an influence over a distinct time. We will call this time interval k_{inf}, measured in discrete time steps T. Regarding for example a network with a background oscillation as described in Section 1.1.3.2, k_{inf} would be the period of this oscillation.

In the following we will show how the upper bound can be computed for our kernel function ε_{ij}. Our criterion for the upper bound of the quantization step q demands that a weighted and k_{inf}-times relaxed spike is still larger than q:

$$q \;<\; \text{trunc}(w_{min} r^{k_{inf}}) \tag{9.12}$$

where trunc() denotes computation with truncation, and w_{min} the minimum weight value.

At the arrival time of a spike, the synaptic strength is lowered by a quantization error e_0.

$$n = 0 \;:\; \text{trunc}(w) = w - e_0 \tag{9.13}$$

The next time step lowers the output value of the filter function by arithmetic error e_1

$$n = 1 \;:\; \text{trunc}(wr) = r\,(w - e_0) - e_1) \tag{9.14}$$

and so on. Now we will assume that the maximum possible error q occurs at each time step and compute the accumulated error as the limit value of the sum over a geometric series:

$$\text{trunc}\left(w_{min} r^{k_{inf}}\right) = r^{k_{inf}} w_{min} - q\left(\frac{1 - r^{k_{inf}+1}}{1 - r}\right) \tag{9.15}$$

Substituting the right-hand term in equation 9.12 yields the upper bound for the quantization step q:

$$q < w_{min}\left(\frac{r^{k_{inf}} - r^{k_{inf}+1}}{2 - r - r^{k_{inf}+1}}\right) \tag{9.16}$$

For other kernel functions the upper and lower bound can be derived in a similar fashion.

9.2.3 Basic procedures of network computation

During each time slice the new state of the network is computed. One way of dividing the computation of a time slice into phases is:

1.) **Input-Phase** ▷ execute *InputFunction* for all neurons:
$pi_{ij}(n) = w_{ij} \cdot x_{ij}(n)$, $po_i(n) = \eta_0 \cdot y(n)$

2.) **Filter-Phase** ▷ execute *FilterFunction* for all neurons:

$fi_{ij}(n) = pi_{ij}(n) + r_{ij} \cdot u_{fij}(n-1)\}$
$fo_i = po_i(n) + r_i \cdot u_{ti}(n-1)$

3.) **Output-Phase** ▷ execute *OutputFunction* for all neurons:
- ∘ compute membrane potential $u_i(n)$: $u_i(n) = -\{fo_i + \vartheta\} + \sum_{j \epsilon \Gamma_i} fi_{ij}$
- ∘ emit spike, if $u_i(n)$ exceeds threshold

4.) **Learning-Phase** ▷ adjust weights according to learning rule

These phases represent a way of structuring the computation of a new state of the network. They will be of importance in the following sections when examining simulation strategies. Since the implementation of learning depends on the specific learning algorithm, for most of the following general considerations, we omit the Learning Phase.

9.3 Programming Environment

Once a model suitable for a digital simulation has been derived, a software environment and a hardware platform needs to be chosen. The programming of neural networks might be done with programming languages like *C*, *C++* or *Fortran*, mathematic software tools such as *MATLAB* [MATLAB, 1995] or *mathematica* [Freeman, 1994], or even with neural network simulators such as *GENESIS*, *PDP++* or *SNNS*. These different programming environments are now discussed in turn.

The use of a programming language is the most flexible approach. Usually it leads to the most efficient implementation in terms of computation speed too. A graphical user interface and visualization tools can be incorporated with *X11* library functions or tool kits like *Tcl/Tk*. Tool kits ease significantly the development of graphical extensions. Some of them e.g. *Tcl/Tk* even allow implementations which are portable to different hardware platforms and operating systems. Separating the implementation into computational expensive tasks, still coded in a programming language like *C*, and graphical tasks ensures the benefit of a fast implementation. With *Java*, an unified implementation is possible avoiding interface issues, which are sometimes difficult to tackle. However, until a compiler for *Java* is available, performance degradation compared to compiled implementation has to be accepted. All these approaches have in common that they require significant programming skills and programming effort. One usually has to start from scratch and code reusability is only possible to a certain extent. Up to now there are no class libraries available for spiking neurons, only for standard model neurons (e.g. [Blum, 1992]).

Standard mathematic software tools also offer a great amount of flexibility while supplying many built-in mathematical functions and visualization features. They allow thereby convenient programming and simulation of neural networks. However, up to now *MATLAB* or *mathematica* provide a tool-box only for simulations of standard model neurons. Spiking neurons have to be assembled with blocks for kernel functions and so on. Regarding execution times, mathematic software tools are only useful for the simulation of small scale networks or the evaluation of prototype implementations.

Neural network simulators can help to reduce the effort of programming a network to an even greater extent. They are often equipped with a graphical user interface and visualization tools tailored for neural networks. Usually they allow the entering of a network structure graphically or on a high-level, abstract syntax consisting of e.g. objects (neurons, connection-types) and topology (layer, connections) definitions. The abstract, modular syntax allows not only faster programming but also a better comprehensibility and transparency of the code simplifying

debugging and changes in the code. Unfortunately, available neural network simulators are not well suited for the simulation of spiking neurons. The underlying model neurons either do not contain temporal behavior (e.g. *SNNS* or *PDP++*) or the model is too detailed regarding spiking neurons (e.g. *GENESIS*). So the simulation of spiking neurons is not possible with neural simulators for standard model neurons. Incorporating model neurons with temporal behavior into these simulators usually requires significant programming efforts and might lead to an inefficient implementation.

Simulators for compartmental models like *GENESIS* are useful for the detailed simulation of single neurons or small ensembles of neurons. *PGENESIS*, a parallel version of *GENESIS* allows simulation of a few up to thousand neurons on cluster of workstations or supercomputers. In the field of spiking neurons *GENESIS* is mostly used for the comparison of the behavior of detailed bio-physical models and the simplified spiking neurons. The simulation of large-scale networks of spiking neurons requires a significant amount of unnecessary modeling effort and is also not efficient regarding computation times.

In the following we consider different implementations of spiking neurons abstractly. The study of different algorithms will be done with pseudo-code.

9.4 Concepts of Efficient Simulation

Implementing large networks of spiking neurons a huge number of operations and/or memory accesses has to be executed on a digital computer. TS timesteps of a network with N neurons, each with F filters with S synapses each, should be simulated. $F \cdot S$ is the number of connections C per neuron. A straight-forward approach A_1 is shown in the following algorithm:

```
1 foreach timestep in TS do
2   foreach neuron in N do
3     foreach filter in F do
4       foreach synapse in S do
5         InputFunction()
6       FilterFunction()
7     OutputFunction()
```

The number of operations N_{OP} -assuming one operation per $Function()$- and the number of memory accesses N_{MA} could be estimated as follows:

$N_{OP} \approx TS \cdot N \cdot F \cdot S$

$N_{MA} \approx 2 \cdot TS \cdot N \cdot F \cdot S$

It should be noted that $N_{OP} \leq N_{MA}$ is due to read/write operations of the filter value and indicates the I/O-boundness of our algorithm. This point is crucial for the implementation of spiking neurons on digital signal processors which are optimized for the computation of compute-bounded problems ($N_{OP} \gg N_{MA}$).

The network activity a (average number AN of spikes per TS / N) is usually quite low in spiking networks. A very efficient communication scheme for such networks is the event-list protocol [Lazarro et al., 1993]. Only the addresses of those neurons which fire in the present time slice –in the following called active neurons– are registered in a spike event-list. This information is distributed by spikes via the connections C for inputs to all connected neurons in algorithm A_2:

```
1 foreach timestep in TS do
2   foreach active_neuron in AN do
3     foreach connection in C do
4       InputFunction()
5   foreach neuron in N do
6     foreach filter in F do
7       FilterFunction()
9     OutputFunction()
```

The number of operations and memory accesses for A_2 could be estimated as follows:

$N_{OP} \approx TS \cdot N \cdot (F + a \cdot C)$

$N_{MA} \approx 2 \cdot TS \cdot N \cdot (F + a \cdot C)$

A high percentage of filters will be negligible due to the low network activity which means they could be neglected for the computation of a time slice without any change in the results. Concerning a filter state f, we can define a certain threshold Δf. Filters with an absolute value less than Δf should be *negligible* filters while the other filters are *non-negligible*. This is implemented in algorithm A_3:

```
1 foreach timestep in TS do
2   foreach active_neuron in AN do
3     foreach connection in C do
4       InputFunction()
5   foreach neuron in N do
6     foreach filter in F_nn do
7       FilterFunction()
8       | f | ≤ Δf ⇒ filter is negligible
9     OutputFunction()
```

F_{nn} is the number of non-negligible filters. The dependency of F_{nn} from the network activity is denoted by a function $f_{nn}(a)$. We observed in our simulations $f_{nn}(x) \approx 10 \cdot x$, but this might change for different network structures and parameters. The number of operations and memory accesses for A_3 could be estimated as follows:

$N_{OP} \approx TS \cdot (f_{nn}(a) \cdot N \cdot F + a \cdot C)$

$N_{MA} \approx 2 \cdot TS \cdot (f_{nn}(a) \cdot N \cdot F + a \cdot C)$

Until now we did not take into account the memory structure required to store the network connectivity. Typically, the connectivity of a neural network would be stored as a weight matrix $\mathcal{W}$. This is well suited for fully-connected networks where $C = N$. Hence, the required memory size is N^2. However, spiking neural networks are more sparse than fully connected ($C \ll N$) and the weight matrix will mostly be filled with zeros. Besides the waste of memory space the zeros slow down the simulation, because the following loop over N (and not over C!) has to be excuted for the Input Phase instead of the lines 2-4 in A_3:

```
foreach active_neuron in AN do
  foreach connection in N do
    if connection ≠ 0 then
      InputFunction()
```

So, using A_2 or A_3 together with a weight matrix is not efficient for sparse networks where $C \ll N$. A more efficient method for the representation of sparse connectivity is the use of lists, one for each neuron n_i [Hartmann et al., 1995]. Each list is accessed by the neuron's base

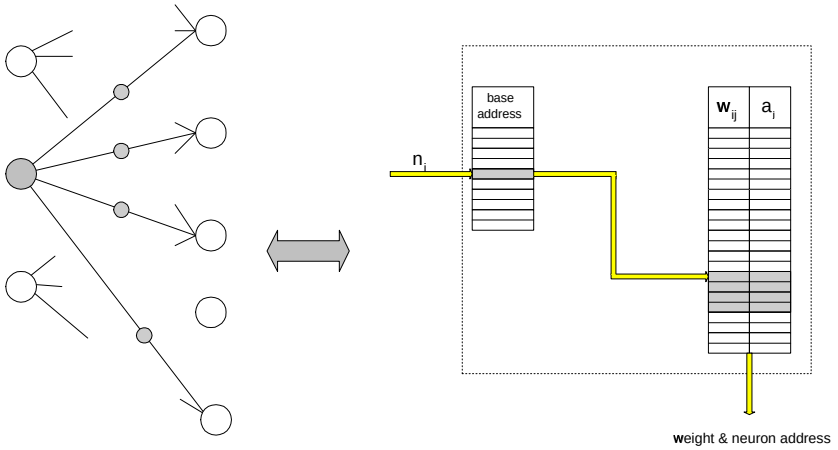

Figure 9.3: Connectivity List

n_i : address of sending neuron i
a_i, w_{ij} : address/weight of receiving neuron j

address (see Fig. 18). The items in the lists are datasets consisting of weights and addresses. The addresses a_j in the list may denote the neuron n_j, to which n_i sends a spike or from which n_i receives a spike. Obviously, the sender-oriented connectivity will be preferred for an algorithm like A_2 or A_3 where we have to know the neurons to which a sending neuron will send a spike. However, for a learning mechanism of the Hebbian type also the reverse direction is required. Hence, a second receiver-oriented list - which will be sender-oriented for the learning phase - is needed. This is equivalent to the need for the transposed weight matrix using weight matrices to store the connectivity.

Connectivity as in Fig. 9.4 follows some simple deterministic rules and is quite regular for neurons of one layer. It will be called *regular connectivity* (RC). So all neurons in a layer are connected according to the same simple rules. Hence, we can compute the regular connections using these rules instead of storing them. On-line computation of the connections reduces the required amount of storage for connection lists drastically [Roth et al., 1997]. Furthermore, in some cases the weight vectors for all neurons in a layer are similiar (weight sharing). We only need to store one *regular weight vector* per layer for the whole network. Unfortunately the on-line computation itself requires several operations per connection depending on the hardware platform. Therefore, this approach is only useful for computers with limited I/O-bandwidth and on-chip memory, e.g. the CNAPS described in Section 9.6.2.

9.5 Mapping Neural Networks on Parallel Computers

In the next section we will examine the performance of different hardware platforms like workstations, DSPs, neurocomputer and supercomputer for the simulation of spiking neural networks. Neurocomputers, supercomputers, and sometimes DSPs are parallel computers. They consist of a few up to thousand and more processing elements (PEs) that can communicate and cooporate to solve large problems quickly.

But how can we map a specific network of spiking neurons on a parallel computer in order

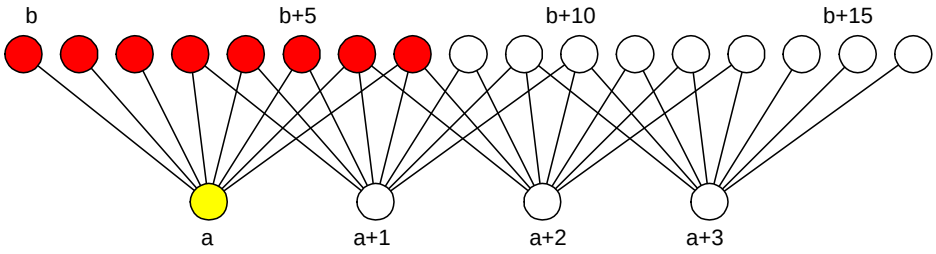

Figure 9.4: Regular Connections

to achieve the maximum performance? The key concepts of an efficient mapping in order to maximize the performance will be load balancing, minimizing inter-PE communciation and minimizing synchronization between the PEs. Furthermore, the mapping should be scalable both for different network sizes, and for different numbers of processing elements.

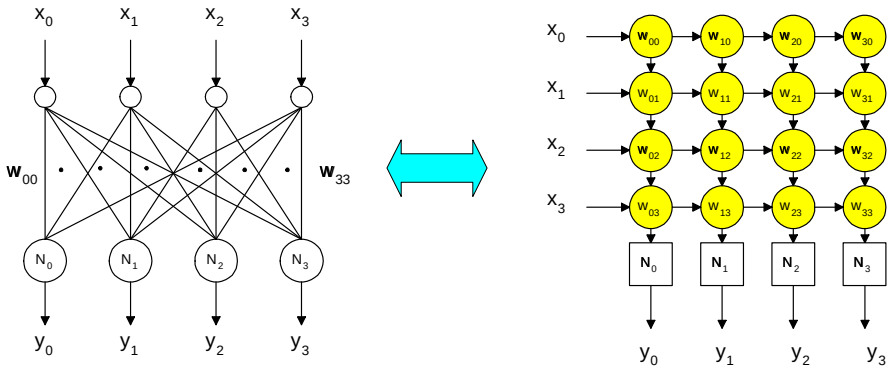

Figure 9.5: Presentation of the Neural Network

Some basic concepts for an efficient mapping will be discussed in the next subsections. A more detailed discussion concerning the mapping of conventional neural networks can be found e.g. in [Ienne, 1997]. The weight matrix presentation of a simple neural network (four neurons with four synapses each) used in the following is shown in the right of Fig. 9.5, while the left side shows the conventional presentation of the same network. In the case of the Spike Response model with only one unique response function, the rectangle N denotes the computation of the response function. The circle w_{ij} stands for the computation of the synapse: $y_i = w_{ij} \cdot u_j + y_{i-1}$ where y_{i-1} is the result from the preceeding synapse.

9.5.1 Neuron-parallelism

Let us assume we would like to simulate a network with N spiking neurons on a parallel computer with N_{PE} PEs. We will illustrate the mapping with $N = 4$ and $N_{PE} = 4$ in the following sections. A common approach for this configuration is shown in Fig. 9.6. The synapses of one neuron and the output function are mapped to the same PE and N_{PE} neurons are computed in parallel. We will call it *neuron-parallel* or simply *n-parallel*.

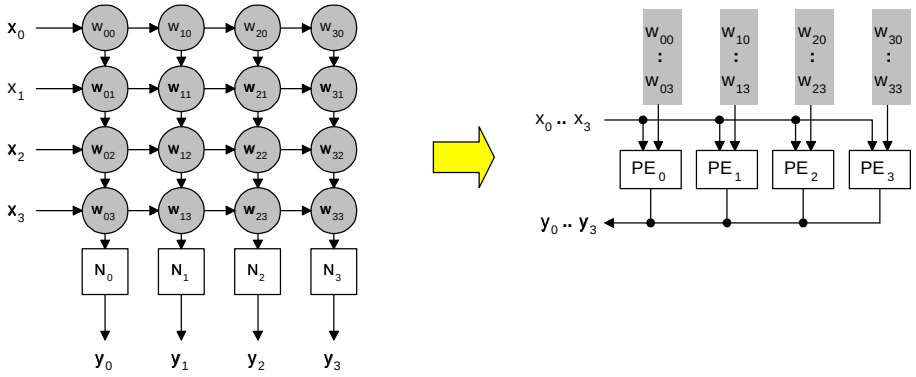

Figure 9.6: N-Parallelism

9.5.2 Synapse-parallelism

Instead of mapping the synapses of one neuron to the same PE they can be mapped to different PEs. So, N_{PE} synapses -not necessarily from the same neuron- will be computed in parallel. The output functions of a neuron can be processed either on a seperate PE serially (mid part of Fig. 9.7) or can be distributed over the PEs (right part of Fig. 9.7). Hence, we will call the first kind of mapping *s*(ynapse)-*parallel* and the latter *sn-parallel*. Small circles in Fig. 9.7 denote

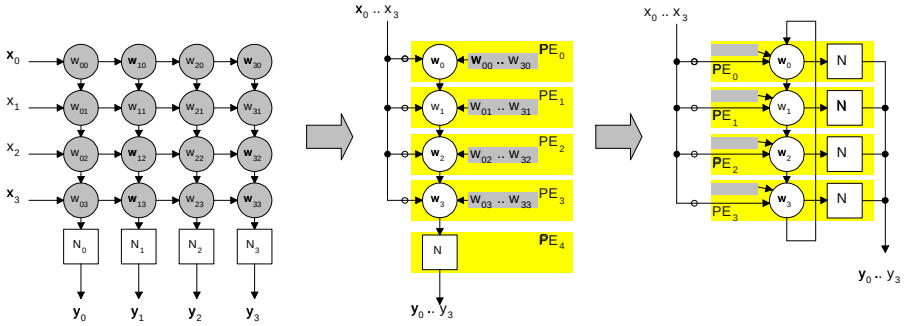

Figure 9.7: S- and SN-Parallelism

registers which are only enabled every fourth cycle. So, PE_0 receives only input signal x_0, PE_0 only x_1 .... It should be noted that for the Output Phase the actual weight values have to be collected from several other PEs. This could be done either by distributed accumulation like in mid/right part of Fig. 9.7 or by local accumulation on the PE computing the neuron. The latter would require a high coommunication bandwidth.

9.5.3 Pattern-parallelism

In order to reduce the required communication bandwidth *p(attern)-parallelism* has been used for conventional networks such as MLPs with backpropagation learning. Each PE of the parallel processor simulates the response of the network for a different input pattern. Unfortunately, different patterns will be presented to the spiking neural network at different times nT and the

state of the network for time $(n+1)T$ is not independent of the state for time nT. Therefore, we have to simulate pattern after pattern and can not use *p-parallelism*!

9.5.4 Partitioning of the network

An important aspect of mapping is which synapse (in the case of *s-/sn-parallel*) or which neuron (in the case of *n-/sn-parallel*) exactly is mapped to which processing element for a specific hardware architecture in order to achieve a balanced workload of all processing elements.

Discussing this scenario, the degree of parallelism -the ratio N_{PE}/N- is important. A commonly used distinction exists between fine-grain and coarse-grain parallelism. In the case of spiking neural networks we will define it as follows:

fine-grain parallelism: $N_{PE} \approx N$
coarse-grain parallelism: $N_{PE} \ll N$

Furthermore, the synchronization of the parallel computer has to be taken into account: SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data). At any given time all processors of a SIMD computer execute the same operation (single instruction) on different data (multiple data). Hence, all PEs of a SIMD computer are synchronized. On a MIMD computer, all PEs could execute different operations (multiple instruction) on different data (multiple data). There is a need for synchronisation between PEs only in the case of data exchange.

Concerning a SIMD computer we have to remember that for any given network state neither all synapses/neurons (A_2/A_3) nor all IPs (A_3) have to be computed. Assuming spatial and temporal correlated activity, the active neurons of one time step nT are forming one or more local activity clusters. Furthermore, a high percentage of neurons receiving spikes in $(n+1)T$ also belong to these activity clusters. In the worst case, the only activity cluster is mapped to one PE which has to compute all the active synapses/IPs/neurons while the other PEs are idle. In order to avoid this worst case scenario we could use *sender-oriented s- or sn-parallel mapping*. All synapses w_{ij} receiving spikes from neuron n_j will be equally distributed to PEs. Hence, all PEs have to update their synaptic values during the Input Phase. So the workload for the Input Phase will be balanced.

In the case of n-parallelism, we could use the following *block-based* mapping scheme for n-parallelism (see Fig. 9.8). The network layers l_i will be divided in blocks of n_b adjacent neurons

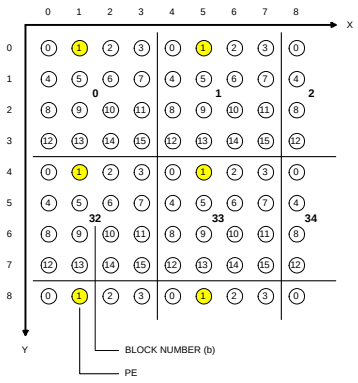

Figure 9.8: Network Mapping Sheme

with $n_b \leq N_{PE}$. Each neuron of a block bn is mapped to a adifferent PE, while neurons of different blocks but the same x/y position relative to their block are mapped to the same PE. The neurons on a PE are distinguished by their block and layer number (see Fig. 9.8). Thus, the coordinates $C_0 = (x, y, l)$ can be transformed into to the coordinates $C_1 = (np, bn, b)$, where np denotes also the PE to which the neuron will be mapped.

np	is the one-dimensional neuron position relative to the block
bn	is the block number
l	is the layer number

Now, neurons of an active cluster will be mapped to different PEs and the workload is more likely to be balanced for the Filter and Output Phase. Even the balancing of the workload for the Input Phase will be enhanced in the case of local connectivity.

Concerning the implementation on a coarse-grain MIMD parallel computer a huge network can be decomposed into smaller parts which will be computed on different PEs. Due to the MIMD architecture there is no need for synchronization. However, the communication bandwidth will now become the main bottleneck. Therefore, only n-parallel mapping is suitable. Furthermore, in order to minimize the communication the interconnectivity of the neurons can be used as the main criterion for the decomposition of the neural network. Hence, neurons with high interconnectivity with each other, regardless to which layer they belong, will be mapped to the same PE. An example for this decomposition can be found in [PGENESIS], which describes an implementation for PGENESIS.

9.6 Performance Study

After the explanations of some basic concepts for the simulation of spiking neural networks on digital computers, we want to take a look at the performance of real computers. As a model network for our performance evaluation we chose the neural network presented by Reitboeck, Stoecker et al. [Reitböck et al., 1993]. The network performs a basic segmentation task. It consists of a two-dimensional layer of neurons which receive an input spike from a corresponding pixel of the input image. Each neuron is a Spike-Response Neuron with 3 filters each and is connected to its 90 nearest neighbors and recurrently to a inhibitory neuron. For our comparative studies we used different network sizes varying from 8kN up to 512kN. Further distinctions will be made between RC and NRC networks. This is of particular interest regarding hardware with limited on-chip memory capacity where the connectivity has to be stored locally.

9.6.1 Single PE workstations

We used algorithm A_3 for the implementation of the model network on single PE workstations. For an integer implementation, different fixed point number representations are necessary. Hence, a large number of shift operations is required for the computation of the network. This may be the reason why we noticed a similar performance for both the integer and the floating point implementation. Therefore, we used a floating implementation for further investigations. Our results in Fig. 9.9 and the execution profiles indicate that the Filter Phase accounts for the major workload on single PE workstations [Jahnke et al., 1995]. Furthermore, the results for SUN Ultra-1 (166 MHz) show that available single PE computer do not have enough performance for real-time simulation of spike-processing neural networks where simulation times of less than 1 millisecond are required (see chapter 9.2.1).

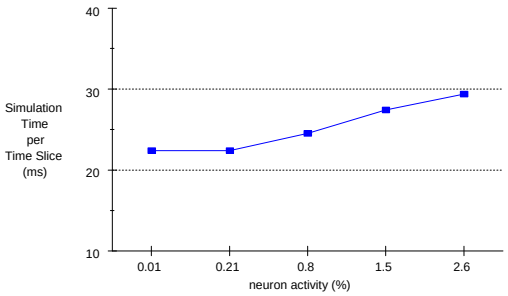

Figure 9.9: Simulation Times on SUN Ultra of a 128k network

9.6.2 Neurocomputer

The **CNAPS/256** (Adaptive Solutions) is a SIMD parallel computer consisting of 256 16b PEs (50 MHz) with 4KB local memory each, communicating via two 8b buses (see Fig. 9.10). The implementable network size is limited to $nmax_{rc} \approx 32$kN for RC networks and to $nmax_{nrc} \approx 500N$ for NRC networks. For the latter, the whole connectivity has to be stored in local memory. Overcoming these limits with extensive use of external memory leads to an unacceptable decrease of performance and a speedup $\ll 1$ over a single PE, since 256 PEs would need to communicate with the external memory via two 16b buses only. Furthermore, the bottle-neck of inter-PE-communication requires that the connectivity of all neurons mapped to one PE must be either stored or computed directly on PE. Therefore, a sender-oriented connectivity is not useful, since the whole network topology would need to be stored on each PE leading to an early exploitation of the limited on-chip memory resources.

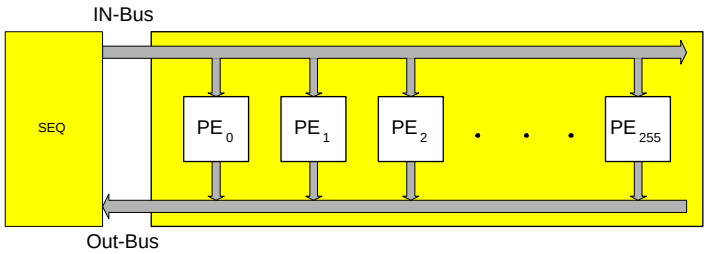

Figure 9.10: Neurocomputer CNAPS/256

However, using the block-based mapping scheme described above for only regular connectivity, we could avoid purely receiver-oriented connections. In a block-based mapping scheme each neuron is unambiguously defined by its coordinates (np, bn, l). Broadcasting this identification code of a spike-emitting neuron to the PEs, each PE could detect in a sender-oriented fashion, which local neuron receives a spike. This could be done with low effort by using the np values -taking into account the block borders- for the computation of local neurons connected to the sending neuron.

```
1 foreach timestep in TS do
2   foreach proc_element in K do
3     foreach active_neuron in AN(proc_element) do
4       SentToBus(active_neuron)
5       begin ParallelCode
6         ComputeConnectedNeuron(ative_neuron)
7       end ParallelCode
8   foreach neuron in N do
9     ParallelCode ...
```

In order to evaluate the speedup through parallelization of this algorithm, we measured computation times while varying N and/or N_{PE}. Our results indicate, that the speedup is approximately linear to N_{PE} as long as inter-PE-communication is not a dominating factor. This is true for network activity $< 1\%$. Staying within these limits, a speedup of up to 15 over single PE computers has been measured (see Fig. 1). The performance could be further increased by using microcoding. Thereby, real-time requirements could be fulfilled for RC networks with up to 16kN. However, the use of CNAPS/256 for real-time applications is limited by the low network complexity as indicated by the values of $nmax_{rc}$ and $nmax_{nrc}$.

The **SYNAPSE** (Siemens AG) is a systolic array processor optimized for matrix-matrix operations. SYNAPSE achieves a very high performance for computing highly connected, conventional neural networks using s- and p-parallel mapping [Ramacher et al., 1991]. The simulation of conventional, static neural networks in general constitute a compute-bounded problem. However, our algorithms are I/O-bounded. So, with barely any on-chip memory and no sufficient bandwidth to communicate with external memory, systolic array processors like SYNAPSE are inadequate for the simulation of spiking neural networks.

9.6.3 Parallel computers

The **TMS320C80** (Texas Instruments) is a digital signal processor including 4 MIMD 32b PEs with 12kB local memory each and one 32b PE for vector processing. All PEs are communicating via a crossbar structure (see Fig. 9.11). All network parameter of RC networks up to 20kN

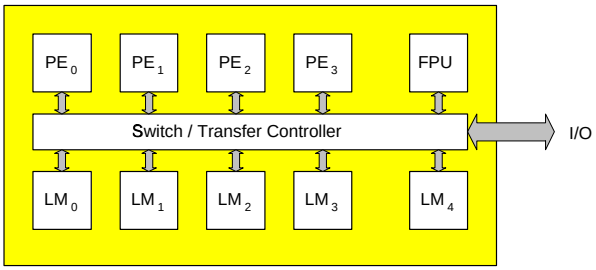

Figure 9.11: Digital Signal Processor TMS320C80

could be stored in local memory. Using algorithm A_3 and n-parallel mapping, a performance similar to the CNAPS could be achieved (see Fig. 1). On one hand, CNAPS can profit from a higher number of PEs. On the other hand, the MIMD architecture of the TMS320C80 allows a high workload for input and decay and output phases without using a specific mapping scheme. Also compared to CNAPS, the crossbar structure allowd better inter-PE-communication which is also diminished due to the larger on-chip memory per PE of the TMS320C80. Simulating

larger RC networks or NRC networks > 1 kN requires extensive use of external memory. Even the high memory bandwidth (400 MB/s) of the TMS320C80 is not sufficient and the simulation times approach those of single PE computers.

The **4xP90** is a MIMD parallel computer with four Pentium P90 and shared memory architecture. Neither the on-chip cache is sufficient to store all network parameters nor are the cache algorithms appropriate to achieve a good hit-rate. Hence, concerning the simulation of spiking neural networks the problem arises that more than one PE needs access to the shared memory at a given time period. Therefore we noticed a poor speed-up of less than 1.3 over a single P90 implementation and a performance similiar to the performance of a single-PE SUN Ultra.

The **SP2** is a MIMD parallel computer with up to 256 R6000 PEs, local memory architecture and high speed 16-to-16 switches for inter-PE communication. N-parallel mapping and sender-oriented connectivity has been used for the implementation [Mohraz, 1997]. On the one hand, a single PE exhibited significantly less performance than e.g. a P90 which must be a result of very unsatisfactory compiler settings and requires further investigations. On the other hand, we noticed a speedup of only 1.12/1.23/1.25 using 2/4/8 PEs over a single PE implementation. Hence, the performance is mainly limited by inter PE communication (computation and communication could not be done in parallel on SP2) which results in the observed poor PE-scaling behavior.

The Connection Machine **CM-2** is a SIMD parallel computer with 16k 1b PEs and hypercube architecture. It has been shown by E.Niebur et.al. that networks with up to 4M simple Spike Response neurons with one filter function can be simulated efficiently on the CM-2 [Niebur and Brettle, 1993]. While n-parallelism and sender-oriented connectivity is most efficient for Filter/Output Phase, only a few neurons are active at a given time step of Input Phase and most of the PEs are idle. Niebur et al. showed that sn-parallelism is more efficient for the CM-2. We already mentioned the inefficieny of sn-parallelism for Filter/Output Phase but this does not matter for the CM-2 with its powerful communication network. On the basis of the results of [Niebur and Brettle, 1993] we estimated the simulation times in Fig. 9.12 for our model network. Even for large networks the real-time requirements are met.

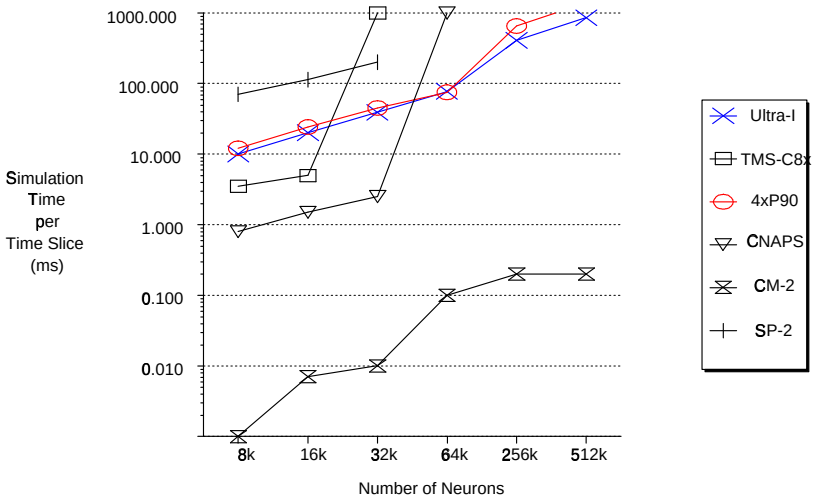

Figure 9.12: Evaluation Results

9.6.4 Results of the performance study

As Fig. 9.12 shows, only supercomputers like the CM-2 exhibit enough performance for real-time simulation of large-scale spiking neural networks. The simulation times indicate that the main reason for the poor performance of other parallel computers is the limited I/O-bandwidth which is a result of the I/O-bounded character of computing spiking networks. This was a motivation to develop dedicated hardware for the simulation of spike-processing neural networks which has been presented recently [Jahnke et al., 1996].

9.7 Conclusion

We have discussed implementation issues concerning the digital simulation of spiking neural networks.

A method has been presented to calculate requisite fixed point precision. Fixed point representation becomes crucial for simulation on neurocomputers, neurochips and digital signal processors.

Three general algorithms for the simulation of spiking neural networks have been shown: a straight-forward approach, an algorithm using spike-event list and a more sophisticated approach by computing only the non-negligible parts of each neuron. Furthermore, on-line computation of the network connectivity could be useful in order to implement algorithms.

We investigated the performance of our algorithms on available computers such as CNAPS, SYNAPSE, TMS320C80 and CM-2. Our results indicate that only a supercomputer exhibits enough performance to compute spike-processing neural networks in real-time.

Therefore, we see two directions for future work. On the one hand, an efficient simulation enviroment for spiking neural networks should be available for a broad class of parallel computers and heterogeneous workstation cluster. This could be a distributed and event-driven simulator that will provide fast but also detailed analysis of larger spiking neural networks [Grassmann, 1997]. On the other hand, there is still the need for dedicated hardware which is currently under developement [Jahnke et al., 1996]. Such hardware could be used either as a low-cost simulator or for real-world applications where both real-time behavior, low power consumption and small size would be needed.

Bibliography

[neuro simulators] A list of available neural network simulators. http://www.neuronet.ph.kcl.ac.uk/neuronet/software/software.html.

[PGENESIS] Examples for PGENESIS. http://www.psc.edu/Packages/PGENESIS/examples.html.

[MATLAB, 1995] MATLAB - User's Guide (1995), Prentice Hall, Englewood Cliffs.

[Blum, 1992] Blum A. (1992) *Neural Networks in C++*. Wiley, New York.

[Grassmann, 1997] Grassmann, C., (1997). Private Communication, Institut für Informatik II, Universität Bonn.

[Hartmann et al., 1995] Hartmann, G., Frank, G., Schaefer, M., and Wolff, C. (1995). SPIKE 128K - An accelerator for dymamic simulation of large puls-coded networks. *Proc. MicroNeuro'97* edited by H. Klar, A. König, U. Ramacher, University of Technology Dresden, 130–139.

[Hammerstrom, 1990] Hammerstrom, D. (1990). A VLSI architecture for high-performance, low-cost, on-chip learning, *Proc. Int. Joint Conf. on Neural Networks 1990 (IJCNN'90)*, Lawrence Erlbaum Associates Publishers, 537–543.

[Ienne, 1997] Ienne, P. (1997). Digital connectionist hardware: Current problems and future challenges. *Proc. Int. Work-Conf. on Artificial and Natural Neural Networks 1997 (IWANN'97)*, edited by J. Mira, R. Moreno-Diaz, J. Cabestany, Springer, 688–713.

[Jahnke et al., 1995] Jahnke, A., Roth, U., and Klar H. (1995). Towards Efficient Hardware for Spike-Processing Neural Networks. *Proc. World Congress on Neural Networks 1995*, 460–463.

[Jahnke et al., 1996] Jahnke, A., Roth, U., and Klar H. (1996). A SIMD/dataflow architecture for a neurocomputer for spike-processing neural networks (NESPINN). *Proc. MicroNeuro'96*, IEEE Computer Society Press, 232–237.

[Lazarro et al., 1993] Lazarro, J. and Wawrzynek, J. (1993). Silicon auditory processors as computer peripherals. *Advances in Neural Information Processing Systems*, Morgan Kaufmann Publishers, 5:820–827.

[McClelland and Rummelhardt, 1988] McClelland, J. L. and Rummelhardt, D. E. (1988). *Explorations in Prallel Distributed Processing.* MIT Press.

[Freeman, 1994] Freeman, J. A. (1994). *Simulating Neural Networks with Mathematica.* Addison-Wesley.

[Niebur and Brettle, 1993] Niebur, E. and Brettle, D. (1993). Efficient simulation of biological neural networks on massively parallel supercomputers with hypercube architecture. *Advances in Neural Information Processing Systems*, Morgan Kaufmann Publishers, 6:904–910.

[Mohraz, 1997] Mohraz, K., Schott, U., and Pauly, M. (1997). Parallel simulation of pulse-coded neural networks. *Proc. Int. Association for Mathematics and Computers in Simulation (IMACS) World Congress 1997 on Scientific Computation, Modeling and Applied Mathematics*, Wiss.-und-Technik, 6:523-528.

[Ramacher et al., 1991] Ramacher, U., Beichter, L., and Brüls, N. (1991). Architecture of a general purpose neural signal processor. *Proc. Int. Joint Conf. on Neural Networks 1991 (IJCNN'91)*, IEEE Service Center, I:443–446.

[Reitböck et al., 1993] Reitböck, H. J., Stöcker, M., and Hahn, C. (1993). Object separation in dynamic neural networks. *Proc. Int. Conf. on Neural Networks 1993 (ICNN'93)*, IEEE Service Center, II: 638–641.

[Roth et al., 1995] Roth, U., Jahnke, A., and Klar, H. (1995). Hardware requirements for spike-processing neural networks. *Proc. Int. Work-Conf. on Artificial and Natural Neural Networks 1995 (IWANN'95)*, edited by J. Mira, F. Sandoval, Springer, 720–727.

[Roth et al., 1997] Roth, U., Eckardt, F., Jahnke, A., and Klar, H. (1997). Efficient on-line computation of connectivity: Architecture of the connection unit of NESPINN. *Proc. MicroNeuro'97*, edited by H. Klar, A. König, U. Ramacher, University of Technology Dresden, 31–39.

[Zell et al., 1993] Zell, A., Huebner, R. et al. (1993). SNNS: An efficient simulator for neural nets. *Proc. Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems 1993 (MASCOTS'93)*, edited by H. Schwetmann, Society for Computer Simulation, 17–20.