LETTER **Communicated by Nigel Goddard**

Advancing the Boundaries of High-Connectivity Network Simulation with Distributed Computing

Abigail Morrison
abigail@biologie.uni-freiburg.de
Computational Neurophysics, Institute of Biology III and Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, 79104 Freiburg, Germany

Carsten Mehring
Carsten.Mehring@biologie.uni-freiburg.de
Department of Zoology, Institute of Biology I and Berstein Center for Computational Neuroscience, Albert-Ludwigs-University, 79104 Freiburg, Germany

Theo Geisel
geisel@chaos.gwdg.de
Department of Nonlinear Dynamics, Max-Planck-Institute for Dynamics and Self Organization, 37018 Göttingen, Germany

Ad Aertsen
aertsen@biologie.uni-freiburg.de
Neurobiology and Biophysics, Institute of Biology III and Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, 79104 Freiburg, Germany

Markus Diesmann
diesmann@biologie.uni-freiburg.de
Computational Neurophysics, Institute of Biology III and Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, 79104 Freiburg, Germany

The availability of efficient and reliable simulation tools is one of the mission-critical technologies in the fast-moving field of computational neuroscience. Research indicates that higher brain functions emerge from large and complex cortical networks and their interactions. The large number of elements (neurons) combined with the high connectivity (synapses) of the biological network and the specific type of interactions impose severe constraints on the explorable system size that previously have been hard to overcome. Here we present a collection of new techniques combined to a coherent simulation tool removing the fundamental obstacle in the computational study of biological neural networks: the enormous number of synaptic contacts per neuron. Distributing an individual simulation over multiple computers enables the investigation of networks orders of magnitude larger than previously possible. The

software scales excellently on a wide range of tested hardware, so it can be used in an interactive and iterative fashion for the development of ideas, and results can be produced quickly even for very large networks. In contrast to earlier approaches, a wide class of neuron models and synaptic dynamics can be represented.

1 Introduction

It has long been pointed out (Hebb, 1949; Braitenberg, 1978) that cortical processing is most likely carried out by large ensembles (assemblies) of nerve cells, whereby the membership of neurons in various assemblies is exhibited through the complex correlation structure of their spike trains (von der Malsburg, 1981, 1986; Abeles, 1982, 1991; Palm, 1990; Aertsen, Gerstein, Habib, & Palm, 1989; Gerstein, Bedenbaugh, & Aertsen, 1989; Singer, 1993, 1999). Theoretical studies (Shadlen & Newsome, 1998; Diesmann, Gewaltig, & Aertsen, 1999; Salinas & Sejnowski, 2000; Kuhn, Aertsen, & Rotter, 2003) have demonstrated that the cortical neuron is indeed sensitive to the higher-order correlation structure of their input. With the experimental technology for multiple-single unit recordings becoming routinely available for animals involved in behavioral tasks, appropriate network models need to be constructed to interpret the results.

Due to the nonlinear and stochastic nature of neural systems, simulations have become a research tool of major importance in the developing field of computational neuroscience (Dayan & Abbott, 2001; Koch, 1999; Koch & Segev, 1989). A number of simulation tools have been developed (Genesis: Bower & Beeman, 1997; Neuron: Hines & Carnevale, 1997; XPP: Ermentrout, 2002) and are in widespread use. They are general purpose in the sense that they maintain a layer of abstraction between the neuron model to be simulated and the machinery implementing the network interaction; that is, they are not neuron or network model specific. The primary focus of these tools is small networks of detailed neuron models. One exception is SpikeNET (Delorme & Thorpe, 2003), which is specialized for a specific class of large networks of spiking neurons.

A major barrier in the simulation of mammalian cortical networks has been the large number of inputs (afferents) a single neuron receives. Assuming a biologically realistic level of connectivity, each neuron should have of the order of 10^4 afferents. In order to ensure that the network is sufficiently sparse, a connection probability of 0.1 should be assumed, resulting in a minimal network size of 10^5 neurons, corresponding to roughly a cubic millimeter of cortex (Braitenberg & Schüz, 1998). Such a network contains of the order of 10^9 synapses and has proved to be beyond the memory capacity of the computers available to many researchers. Furthermore, even given a computer with sufficient memory resources, the amount of time required to construct and simulate such networks and

perform reasonable parameter scans is not conducive to rapid scientific progress.

Faced with these problems, many researchers have opted to use scaled networks, in which the connection probability remains constant and the synaptic weights are scaled in some manner with the inverse of the total number of connections a neuron receives. This approach is not without its risks; although the mean or, alternatively, the standard deviation of the subthreshold activity can be held constant, this is not true for correlations and combinatorial measures. Furthermore, in such networks, the memory requirement increases quadratically with the number of neurons. Clearly, these disadvantages hold only until the minimal network size is attained, at which point synaptic scaling need no longer be applied and memory requirements increase only linearly with increasing network size.

In this letter, we describe without reference to a particular implementation language how distributed computing (i.e., simultaneous execution on multiple processors, where the addressable memory of one processor is not visible to the others) can be used to acquire the memory resources to surpass the threshold of 10^5 neurons and reach a simulation speed suitable for practical work.

To our knowledge, this is the first description of a general-purpose simulation scheme that allows the routine investigation of networks of spiking neurons with biologically realistic levels of connectivity. Our design consists of a highly efficient distributed algorithm, based on the serial (i.e., running on one processor) simulation kernel first described in Diesmann, Gewaltig, and Aertsen (1995). Despite the use of a distributed algorithm, a serial interface is presented to the researcher. In addition to the increases in network size and simulation speed enabled by our approach, a key feature of our design is its flexibility due to object orientation (the implementation language being C++; Stroustrup, 1997). Recent advances in simulation technique have tended to focus on current-based integrate-and-fire point neurons (Mattia & Del Giudice, 2000; Lee & Farhat, 2001; Reutimann, Giugliano, & Fusi, 2003); using our technology, the researcher is not bound by any of these restrictions and can just as easily use conductance-based neurons (e.g., Chance, Abbott, & Reyes, 2002; Destexhe, Rudolph, & Pare, 2003; Kuhn, Aertsen, & Rotter, 2004), non-integrate-and-fire models (Hawkes, 1971), simple compartment-based models (Larkum, Zhu, & Sakmann, 2001; Kumar, Kremkow, Rotter, & Aertsen, 2004), or implement a neuron model of his or her own. Whereas complex compartmental models could theoretically be implemented, there is no support for generating them as in Neuron (Hines & Carnevale, 1997) or Genesis (Bower & Beeman, 1997), and so such models remain out of the reach of the current version. The main constraint is the restriction to the class of neuron models where the interaction between neurons is noninstantaneous (finite delays) and mediated by point events (spikes). Similarly, a large range of synaptic dynamics can be employed, and a network may be entirely heterogeneous in terms of the neuron and synapse models used.

In section 2, we describe the representation and construction of a network, including compression techniques. On this basis, we explain the distributed algorithm for solving the dynamics in section 3. After a brief discussion of the treatment of pseudorandom numbers in section 4, we provide benchmarks for our simulation technology in section 5 using relevant simulation examples and hardware, demonstrating its excellent scalability with respect to number of processors, network activity, and network size on computer clusters and parallel computers (for the purpose of this article, we reserve the term *parallel computer* for shared memory architectures). Section 6 summarizes the key concepts of our simulation scheme and discusses our approach in the light of future directions of neuroscience research and upcoming computer architectures.

Source code detail is out of the scope of this letter, as is simulation infrastructure such as writing data to files. In the following, the term *machine* refers to one processor, addressing memory that is assumed to be invisible by other machines. Thus, a computer with two processors and shared memory would be regarded as two machines. The term *list* is taken in its intuitive meaning of a sequential ordering, which need not necessarily be implemented as the data structure known as *list* (Aho, Hopcroft, & Ullman, 1983).

The research on a distributed simulation kernel described in this article is a module in our long-term collaborative project to provide the technology for neural systems simulations (Diesmann & Gewaltig, 2002). The application of the technology described here has already enabled interesting insights into the nature of cortical dynamics (Mehring, Hehl, Kubo, Diesmann, & Aertsen, 2003; Aviel, Mehring, Abeles, & Horn, 2003; Tetzlaff, Morrison, Geisel, & Diesmann, 2004).

Preliminary results have been presented in abstract form (Morrison et al., 2003).

2 Representation of Network Structure

2.1 A Generic Network. Consider a generic network of spiking point model neurons. In order to represent this, it is helpful to consider the synapses as being separate entities from the neurons. If a synapse is triggered by a spike from its presynaptic neuron, it transmits this information in the form of an event with weight w and delay d to its postsynaptic neuron. Each neuron is assigned a unique index, and we say that the synapse that transmits a spike from neuron i to neuron j is an axonal synapse of i but a dendritic synapse of j. Thus, for a serial algorithm, a list of neurons, each possessing a list of its axonal synapses, whereby each synapse contains the index of its postsynaptic neuron, would suffice to represent the network structure completely. This is analogous to the adjacency lists commonly used to represent graphs (Gross & Yellen, 1999) and is illustrated in Figure 1. Inorder to allow maximum flexibility in the structure and heterogeneity of

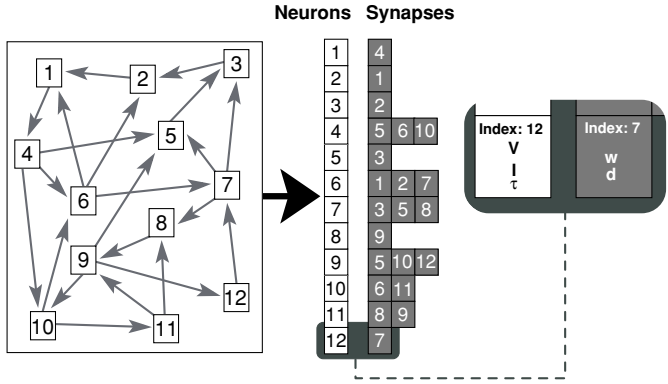

Figure 1: Data structures for a serial simulation scheme. The network comprises N uniquely indexed neurons (left column), and each neuron is assigned a list of axonal synapses (right column). Each neuron contains its own state variables, as does each synapse (illustrated by close-up). In addition, each synapse contains the index of its postsynaptic neuron.

the networks that can be simulated, an object-oriented approach is highly advantageous, in which each individual neuron and synapse maintains its own parameters, as depicted in the close-up in Figure 1, and performs its own dynamics rather than being subject to a global algorithm.

For a distributed algorithm, matters are somewhat more complicated. The most obvious requirement for distributing a simulation is that the neurons are distributed. The simplest possible load balancing is to assign an equal number of neurons to each machine. As there may be different neuron models with different associated computational costs in the network, this assignment is performed using a modulo operation, so that contiguous blocks of neurons (the most intuitive way of defining neuron populations) are dealt out among all the machines, resulting in a good estimate of the fairest load distribution. For simplicity, here and in the remainder of the article, we ignore this assignment and assume that neurons 1 to N/m, where m is the number of machines, are located on the first machine, neurons $N/m + 1$ to $2N/m$ on the second machine, and so on. Clearly, the synapses must also be distributed. We distribute the axonal synapses of each neuron as follows: on each machine, there are N lists of synapses, one for each neuron in the network. A synapse from neuron i to neuron j is stored in the ith list on the machine owning neuron j. Alternatively, from a biological point of view, we say the axon of a neuron is distributed, but its dendrite is local, as anticipated in Mattia and Del Giudice (2000). This seems less intuitive than distributing the dendrite and keeping the axon local, but confers a considerable advantage in communication costs, as discussed in section 3.

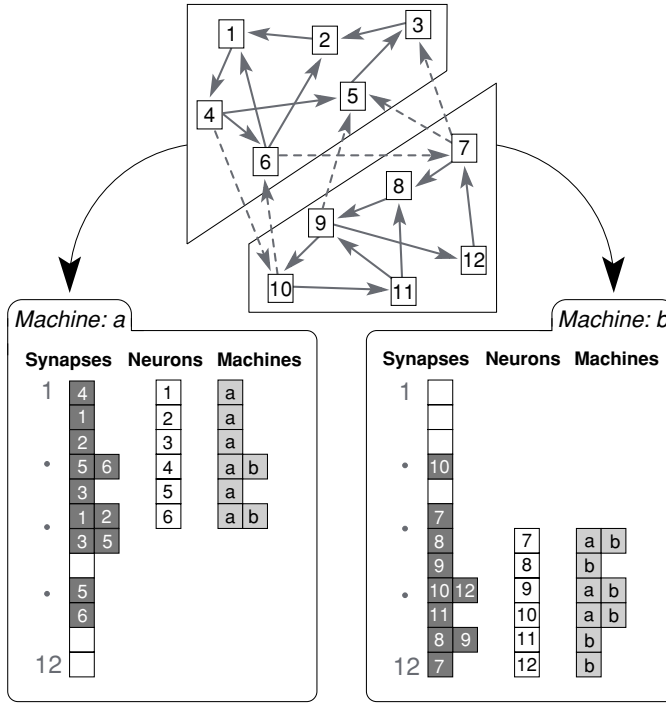

Figure 2: Data structures for a distributed simulation scheme. As an example, the network shown in Figure 1 is distributed (top) over two machines. Each machine (bottom panels) contains $N/2$ neurons (center column) and N lists of synapses (left column), one list for each neuron in the network. All synapses in the ith list are synapses from neuron i to neurons on the local machine. The dashed arrows indicate connections that cross machine boundaries. In addition to the data structures required for the serial algorithm (see Figure 1), each neuron is assigned a list of machines (right column). The machine list for neuron i specifies all the machines on which neuron i has targets (i.e., all machines where the ith synapse list is not empty).

Instead of a list of axonal synapses, each neuron is assigned a list of the machines on which it has targets; that is, the machine list for i contains exactly those machines on which the ith synapse list is not empty. This scheme is depicted in Figure 2.

Further network elements not shown in Figure 2 include neuron input devices such as current generators and observation devices such as membrane potential recorders and spike detectors, which interact with any or all of the local neurons.

2.2 Network Construction. As the synapses are represented on the same machine as their postsynaptic neuron, the network must also be connected from the postsynaptic side: for each neuron, its afferent neurons must be specified and a synapse added to the local synapse list of each of those neurons. Once the inputs for each neuron have been established, one *complete exchange* between the machines (see section 3.4) suffices to propagate the connectivity information necessary to construct the machine list for each neuron. Constructing the network is therefore a fully parallelizable activity that results in a near-linear speed-up (see Figure 7). This is an important feature of our technology, as without parallelization, the wiring of a network can account for a large fraction of the total run time, especially as biological levels of connectivity are approached. Despite the highly parallel nature of the construction, the user needs no knowledge about the location of the neurons and can specify connections as if it were a serial application. On the most basic level, a command `Connect`(i, j) is ignored almost instantaneously on all machines except the one owning neuron j. For higher-level connection commands, such as one for connecting a random network, we make use of the fact that a machine can determine at the beginning of such a routine which neurons belong to it. The desired method can then be applied to all neurons within the appropriate bounds without having to check ownership of each neuron, thus achieving an even higher degree of code parallelization.

2.3 Network Compression. Simulations of large biological neural networks are by their very nature highly consumptive of memory. Although the emphasis of this article is on the use of distributed computing to provide the necessary memory resources, it is clear that any compression of the network structures will be beneficial, as it increases the size of network that can be investigated on any given hardware and decreases the simulation time for any given network. We therefore explain briefly the general principles of how redundancy can be reduced without special knowledge of the network structure in order to reduce the memory demands of a simulation.

We will consider the simplest synapse possible, a static synapse consisting of a constant weight w, a constant delay d, and the index of the postsynaptic neuron i. Ignoring for simplicity the memory overhead involved in creating the synapse object, a naive representation of a synapse would therefore require a number of bytes $M_s = M_w + M_d + M_i$, and a network of N neurons with K synapses each would require NKM_s bytes purely for the synapses. To give some idea of the scale of the problem, typical values for M_w, M_d, and M_i on a 32-bit system are 8, 4, and 4 bytes, respectively, and a network containing $N = 10^5$ neurons with a connectivity of $K = 10^4$ synapses each would require 16 GB. However, instead of storing the postsynaptic index for each synapse in a list, we keep the list sorted in order of increasing index and store just the differences between them. If this difference is less than 255, it can be stored in 1 byte. If the difference happens

to be too large, an appropriate entry is made in an overflow list. In fact, in most applications, this is a rare occurrence. In the network mentioned above, if connected using a uniform distribution, the average difference between neurons is 10. If networks of orders of magnitude larger than this were to be investigated, spatial structure would have to be taken into consideration to avoid obtaining a biologically unrealistic sparseness. It should be noted that the requirement of keeping the target lists sorted is fulfilled without extra costs if, given indices j_1, j_2 of neurons located on a particular machine with $j_1 \leq j_2$, the commands establishing the connections are issued in the sequence ..., `Connect`(i_1, j_1), ..., `Connect`(i_2, j_2), ... the order and location of the presynaptic neurons being irrelevant. This technique reduces the amount of memory for each synapse to $M_s \simeq M_w + M_d + 1$.

Further compression can be achieved by considering the distributions of the synaptic delays and weights. In the best-case scenario, as far as memory is concerned, each neuron makes only one kind of axonal synapse: $w_{ij} = w_i$ and $d_{ij} = d_i$. In this case, the parameters have to be stored only once per list rather than once per synapse, resulting in $M_s \simeq 1$ and reducing the memory requirements for the above example network to approximately 1 GB. In the next best case, each neuron makes axonal synapses with weights and delays that take on only a few possible values (see, e.g., Brunel, 2000). Here, several synaptic lists will be initialized, one for each combination encountered. This produces compression almost as good as in the ideal case. However, there is a non-negligible memory overhead associated with the construction of each list, so for a broad distribution of delays, it is more efficient to represent them in terms of their difference from a reference value. For many applications, this difference can be expressed in 1 byte, resulting in $M_s \simeq M_w + 1 + 1$. Similar to the representation of the postsynaptic neurons, a value too far away from the reference value can be stored in an overflow list. This kind of compression is applicable only to discrete-valued variables such as delay. There is currently no way to compress a continuous distribution.

These methods can be easily extended to more complicated synapse objects, but the onus is on the user to be aware of the redundancies in the network to be investigated and choose the appropriate kind of compression. As illustrated above, depending on the heterogeneity of the network, eliminating redundancy can reduce the memory requirements significantly, so that even large networks can be simulated on hardware available to modest budgets. Conversely, with access to large clusters or parallel machines, it is possible to investigate networks orders of magnitude larger than previously possible.

3 Simulation Dynamics

3.1 Time Driven or Event Driven? A Hybrid Approach to Simulation. The two classic approaches to simulation are time driven and event driven, also known as synchronous and asynchronous algorithms. We pursue a

hybrid strategy whereby the neurons are updated at every time step, but a synapse is updated only if its presynaptic neuron produces a spike. A purely event-driven algorithm such as that proposed by Mattia and Del Giudice (2000) is unsuitable for our purposes because it places too many restrictions on the classes of neurons that can be simulated. For example, any neuron class in which a spike induces a continuous postsynaptic current would be very difficult to implement if a neuron is updated only on the arrival of a spike event. Furthermore, the motivation for these algorithms is the fact that the state of a current-based integrate-and-fire (IAF) neuron can be directly interpolated between events, where events are assumed to be rare. This perceived computational advantage dwindles rapidly as the frequency of events increases: a neuron with 10^4 afferent connections firing at just 1 Hz receives events at a rate of 10,000 Hz and is therefore at least as expensive to simulate event driven as on a time grid with a step size of 0.1 ms. In Reutimann et al. (2003), a solution to the problems of rise times and event saturation is presented for IAF neurons at the cost of large look-up tables and restrictions on the type of background population activity. By updating the neurons in fixed time steps and applying exact integration techniques where possible (Rotter & Diesmann, 1999), we maintain a highly flexible simulation environment at no grave computational cost. As a consequence of this scheme, spike times are constrained to the time grid and are expressed as multiples of h, the time step or computational resolution. It should, however, be noted that a neuron can perform arbitary dynamics within this time step, including using an internal resolution much finer than the one the spikes are constrained to.

Conversely, a purely time-driven algorithm is equally unsuitable. Synapses are by far the most numerous elements in a network. This is the case even when synaptic scaling is employed (i.e., using fewer but stronger synapses), but particularly so if biologically realistic connectivity is assumed. Clearly, updating 10^9 synapses every time step would have catastrophic consequences for simulation times. Fortunately, although the above consideration that events are rare is not valid for neurons, it is valid for individual synapses—in the situation described above, a synapse processes events at just 1 Hz. Assuming that the synaptic state can be calculated from its previous state, the time since its last update, and information available from the postsynaptic neuron, a synapse need be updated only when it transfers a spike event. In fact, a wide range of synaptic dynamics falls into this category, including synaptic depression (Thomson & Deuchars, 1994), synaptic redistribution (Markram & Tsodyks, 1996), and spike-time-dependent plasticity (Bi & Poo, 1998, 2001), for a review see Abbott and Nelson (2000).

By combining the flexibility of a time-driven algorithm with the speed of an event-driven algorithm, a highly functional and fast simulation environment is achieved.

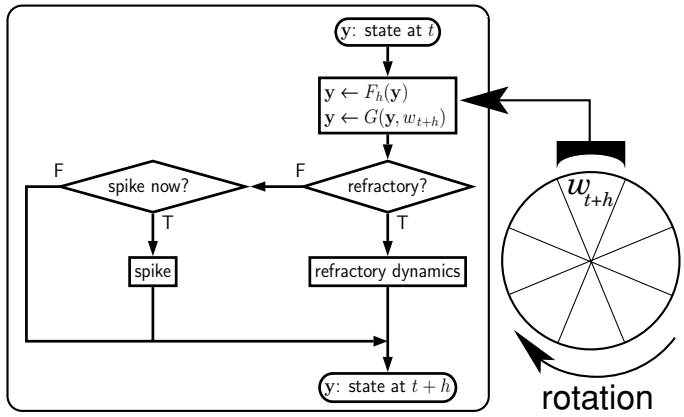

Figure 3: Update of neuron state. The flowchart (left) defines the sequence of operations required to propagate the state $\mathbf{y}$ of an individual neuron by one time step h. Operator F_h performs the subthreshold dynamics, and operator G modifies the state according to the incoming events. Event buffers (only one is shown, right) contain incoming events for the neuron. They are rotated at the end of each time step so that for any simulation time t, the current read position (indicated by the black read head symbol) always provides the input scheduled to arrive at simulation time $t + h$.

3.2 Neuron Update. At each time step, each neuron is updated from its state at time t, $\mathbf{y}_t = (y_1, y_2, \ldots, y_n)_t$, to its state at time $t + h$, where h is the temporal resolution of the simulation. This update is performed on the basis of $\mathbf{y}_t$ and w_{t+h}, the summed weight of all events arriving at time $t + h$.

A flowchart of the update is shown in Figure 3, a concrete example is given in Diesmann, Gewaltig, Rotter, and Aertsen (2001), and the theory for grid-based simulation is developed in Rotter and Diesmann (1999). First, the subthreshold dynamics of the system is applied to the state vector; that is, the state of the neuron is calculated without taking new events into consideration. Next, w_{t+h} is read out of the neuron's event buffer, also shown in Figure 3. The event buffer can be thought of as a primitive looped tape device; at each time step, the current value can be read off and erased. At the end of a time step, all event buffers are rotated one segment so that the next value is available to be read in the next time step. Only one such buffer is depicted; in fact, the number of buffers is dependent on the dynamics of the neuron model. The provisional new state of the neuron is then updated on the basis of w_{t+h}. Now the provisional state of the neuron reflects the values valid at time $t + h$ with respect to the neuron's subthreshold dynamics. If at this point the state of the neuron fulfills its spiking criteria (e.g., passing a threshold potential), the state is updated once again according to the appropriate spike generation dynamics (such as resetting the membrane

potential). The emitted spike is assigned to the time $t + h$, and the information that it has occurred must be transmitted to the neuron's distributed axonal synapses. This calculation of the new state of the neuron is defined by the individual neuron model rather than by a global algorithm. In many cases, it is possible to avoid using computationally expensive differential equation solvers by applying a propagator matrix to the neuron state to integrate exactly (Rotter & Diesmann, 1999).

Clearly, the use of event buffers, already present in the original serial version (Diesmann et al., 1995; Gewaltig, 2000), obviates the requirement for a centralized event queuing system; an event of weight w due to arrive at the neuron d time steps in the future can be added to the event buffer d steps upstream from the current reading position. Obviously, causality requires a nonzero transmission delay, that is, $d \geq 1$. It is important to note the implicit assumption made here that the weights of incoming events can be summed, as each ring buffer segment contains just one value, which is incremented by the weights of successive incoming events. It does not, however, imply that an incoming event may only cause a discontinuous jump in a state variable of the neuron. The interpretation of the weights is left up to the individual neuron model. For example, one model may interpret the weights as the magnitude of a jump in the membrane potential and another as the maximum value of an alpha function (Jack, Noble, & Tsien, 1983; Bernard, Ge, Stockley, Willis, & Wheal, 1994) describing the change of conductance. Nor does it imply that all the events received by a neuron necessarily induce identical dynamics. Models with different time constants for excitatory and inhibitory input, for example, or with several compartments (Kumar et al., 2004) are implemented by giving the neuron access to several such event buffers.

3.3 Index Buffering. At first glance it seems as if communication between machines should take place after every time step in order to convey the information of which neurons spiked during that time step. Fortunately, this is not so. If the minimum synaptic delay is $d_{\min} \cdot h$, then a neuron spiking at time t cannot have an effect on any postsynaptic neuron at a time earlier than $t + d_{\min} \cdot h$. Therefore, if the spikes can be stored maintaining their temporal order, it is sufficient to communicate in intervals of $d_{\min}$ time steps. This also represents the maximum possible communication interval; any greater communication interval would result in events arriving with delays longer than those specified by the user. This communication scheme is completely independent of the temporal resolution; simulating on a finer time grid does not increase the frequency of communication. Maintaining the temporal ordering is easily done. In each time step, the indices of all spiking neurons can be stored in a buffer as illustrated in Figure 4, and at the end of the time step, a marker is inserted into the buffer to separate the spikes of successive time steps. Note that if the axonal synapses were local and the dendritic synapses distributed, the synaptic weight, delay,

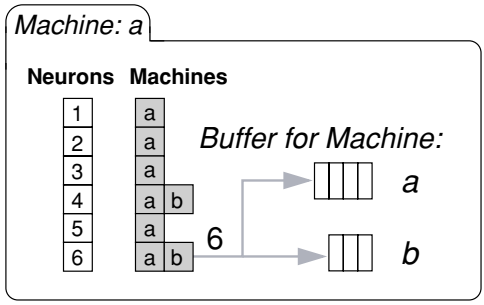

Figure 4: Target machine-specific buffering of local events. In this example, neuron 6 located on machine a (see Figure 2) produces a spike. Its list of target machines contains the identifiers for machines a and b. The index 6 is appended to the index buffers for these machines.

and index of every target neuron would have to be communicated, at a cost of $M_c = K \cdot (M_w + M_d + M_i)$ per spike. Using a representation of the network structure where the postsynaptic neuron maintains the information about the weights and delays of incoming connections, as is the case in several simulators (e.g., Bower & Beeman, 1997), would reduce this cost to $M_c = K \cdot M_i$. Assuming the same sizes of the synaptic parameters as in section 2.3, this amounts to a reduction factor of 4. However, in our representation, the entire synaptic structure, including the index of the postsynaptic neuron, is stored physically on the postsynaptic side but logically on the presynaptic side (see section 2.1). This means that it suffices to send merely the index of the source neuron to every machine on which it has a target, so the communication cost is no longer proportional to the connectivity of the neuron but to the number of machines: $M_c = m \cdot M_i$. This is a reduction factor of K/m, which for biologically realistic levels of connectivity can be of the order of 10^4 and so adequately justifies the otherwise counterintuitive decision to distribute the axonal rather than dendritic synapses.

A further reduction in communication bulk results from sending spike information only to where it is needed. In Figure 4, each neuron is shown to have a list of the machines on which it has target neurons. This information can be used to filter the indices into machine-specific buffers. At the end of the $d_{\min}$ interval, these buffers can be exchanged with the corresponding machines. By distributing axonal synapses, communicating in intervals of $d_{\min}$, and sending spike information only to where it is needed, a communication scheme is achieved with both minimal bulk and frequency.

3.4 Buffer Exchange. The communication itself is performed using routines from the Message Passing Interface (MPI) library (Pacheco, 1997). The two basic communication types are blocking and nonblocking. In blocking

communication, a `Send` (b) on machine a requires a corresponding call of `Receive` (a) on machine b, and both machines wait until these calls have been successfully completed. In nonblocking communication, the data to be sent from machine a to machine b are written into a buffer that can be picked up by b when it is ready, and a continues with its next instruction without waiting. The latter paradigm is potentially more efficient; we use the former as it is more robust: the MPI library definition does not fully specify how data that have not yet been retrieved are to be buffered. Furthermore, as each of the machines needs to receive event buffers from every other machine before it can continue and the amount of time spent in communication is small compared to the amount of time required to update the neurons (see section 5), using nonblocking communication could result in only a minimal increase in performance.

Given m machines, there are consequently $m(m-1)/2$ individual bidirectional exchanges that need to be ordered carefully in order to prevent deadlock (for example, if a tries to send to b, while b tries to send to c and c tries to send to a). This is equivalent to an edge coloring problem, where each machine is a vertex of a fully connected graph, and each edge represents the exchange of buffers between the two machines it connects. Each vertex may have only one edge of each color, and edges that are the same color correspond to exchanges that can be carried out in parallel without causing deadlock. The Complete Pairwise EXchange (CPEX) algorithm (Tam & Wang, 2000; Gross & Yellen, 1999) is a simple constructive process that produces sets of edges to enable the graph to be colored with the minimum of colors or, equivalently, the order of exchanges for maximally efficient communication with no deadlock.

In Figure 5A, the ordering of exchanges is illustrated for a network with five machines. It should be noted that the algorithm is more efficient if an even number of machines is involved (see Figure 5B), resulting in $m-1$ communication steps (colors) rather than m steps for odd m, where in every step, one machine is idle.

3.5 Event Delivery. The received buffers are then sequentially processed by reading off the indices one by one and activating the corresponding synapses, as illustrated in Figure 6. If the index of neuron i is read off before the first marker has been read, this means that neuron i spiked $d_{\min}$ time steps ago. The synapses of i are activated, and for each postsynaptic neuron j, an event of weight w_{ij} with synaptic delay $d_{ij} \cdot h$ is produced. This weight is then written to the appropriate event buffer of neuron j, $d_{ij} - d_{\min}$ time steps on from the current reading position, thus maintaining the correct temporal ordering of events. Indices between the first and second markers correspond to neurons that spiked in the second time step following the last buffer exchange; accordingly, the resulting events have their synaptic delays decremented by $d_{\min} - 1$. This process continues until all indices from the buffer have been read off, at which point the next buffer can be processed in

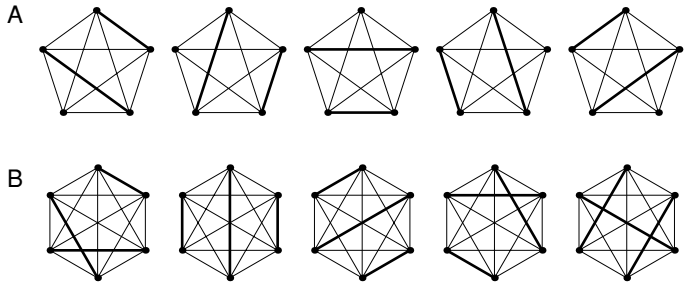

Figure 5: Illustration of the complete pairwise exchange (CPEX) algorithm as an edge coloring problem. Machines correspond to the nodes of the graph (filled circles) and communication routes to the edges (lines). (A) The five communication steps required for a computer cluster with $m = 5$ machines are represented by the sequence of graphs. In each step (from left to right), two pairs of machines (connecting edges highlighted by thick lines) exchange their messages. In the edge coloring terminology, this corresponds to the application of a different color. Using this odd number of machines, the progress of the algorithm can be visualized by the clockwise rotation of the highlighted parallel edges. (B) The $m - 1$ communication steps required for $m = 6$ machines. Same display as in *A*.

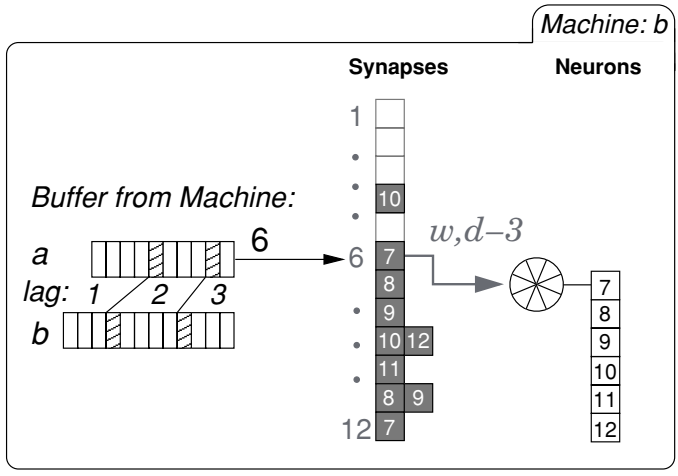

Figure 6: Delivery of received events. In this example, $d_{\min} = 3$, and so as the index 6 is read out of the section of the buffer received from machine a (left) before the first marker (hatched block), neuron 6 spiked three time steps ago. This information is passed to the synapse list of neuron 6 on this machine (center). This list contains a synapse with the postsynaptic neuron 7. It produces an event of weight w and delay d, which is placed in the event buffer (cf. Figure 3) of neuron 7 (right) not d, but $d - 3$ segments on from the current reading position, to take account of the communication lag of three time steps.

the same way. Once all buffers have been processed, the next $d_{\min} \cdot h$ period in the life of the network is simulated, and the cycle begins again.

4 Random Numbers

Many networks require a massive amount of random numbers for their simulation and construction. In addition to the usual pitfalls of pseudorandom number generation, this presents extra problems in a distributed environment. The ideal solution should be able to produce sequences of random numbers for each machine such that each sequence is itself uncorrelated, and each sequence is uncorrelated to any other sequence. Furthermore, it should be possible to perform a simulation on a different hardware or with a different number of machines, and obtain identical results. A central random number server is not an appropriate solution for this application, as this would result in a bottleneck due to the sheer volume of numbers required. A partial solution is provided by the use of random number generators (RNGs), which produce independent trajectories for different seeds (Knuth, 1997). Such RNGs are available from the GNU Scientific Library (Galassi, Gough, & Jungman, 2001), which, moreover, ensures platform independence.

The solution is completed by the introduction of pseudoprocesses. The number of pseudoprocesses N_{pp} is specified at run time, when the number of machines used m is also known. They are assigned to the machines such that pseudoprocess p is on machine $p \bmod m$, and each machine has an equal number of pseudoprocesses, thereby constraining m to be a factor of N_{pp}. Each pseudoprocess is assigned one RNG with a unique seed. Each neuron is assigned to a pseudoprocess such that neuron n is assigned to pseudoprocess $n \bmod N_{\text{pp}}$, and all the random numbers required for that neuron are drawn from the corresponding RNG. As the algorithm to assign the neurons to the pseudoprocesses depends solely on N_{pp}, identical simulation results will be obtained for any m fulfilling the constraint. In this way, we ensure a fast, safe production of random numbers, independent of both the platform and the number of machines used.

5 Performance

We tested the software on four architectures, chosen to reflect the kind of hardware currently available:

- Elderly PC cluster (8×2 processors, 100 MBit Ethernet, Intel Pentium 0.8 GHz, 256 kB cache)
- Recent PC cluster (20×2 processors, Dolphin/Scali network, Intel Xeon, 2.8 GHz, 512 kB cache)
- Compaq GS160 (16 processors, Alpha 0.7 GHz, 8 MB cache)

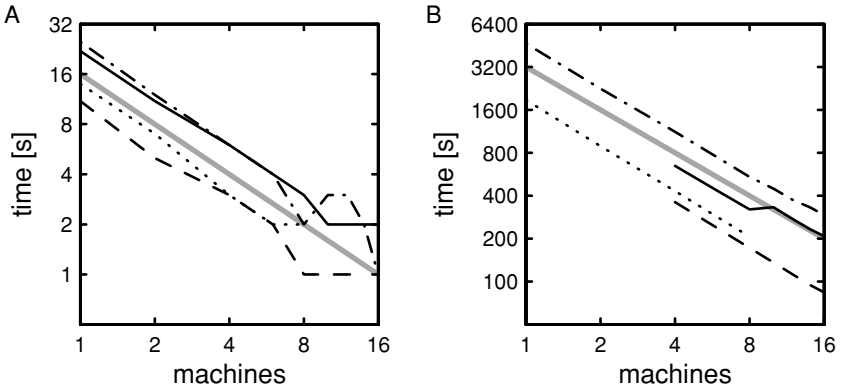

Figure 7: Scalability of wiring with respect to number of processors: elderly PC cluster, solid line; recent PC cluster, dashed line; GS160, dash-dotted line; GS1280, dotted line. See the text for architecture and simulation details. (A) Wiring time for the 10^4 network against number of processors, log-log representation. The gray line indicates slope for a linear speed-up. (B) As in *A* but for the 10^5 network.

- Compaq GS1280 (8 processors, Alpha 1.15 GHz, 1.75 MB associative cache)

The following simulations were performed on the three different architectures for several different numbers of processors:

- 10^4 low rate network: 10,000 neurons with 1000 random afferent connections each and an average spike rate of around 2.5 Hz (dynamics as described in Brunel, 2000) was simulated for 10 biological seconds.
- 10^4 high rate network: as above, but with an unrealistically high average spike rate of around 250 Hz.
- 10^5 low rate network: 100,000 neurons with 10,000 random afferent connections each and an average spike rate of around 2.5 Hz (dynamics as described in Brunel, 2000) was simulated for 1 biological second.

These simulations were chosen to demonstrate the scalability of the software with respect to the number of processors for networks of significantly different sizes and activities.

In Figure 7 the wiring times for the two different network sizes are plotted against the number of machines used. The double logarithmic representation reveals the exponent of the dependence. In both cases, the wiring time scales linearly with the number of machines. For the smaller network (see Figure 7A), a saturation for large numbers of machines seems to be visible; however, the times measured are close to the resolution of measurement

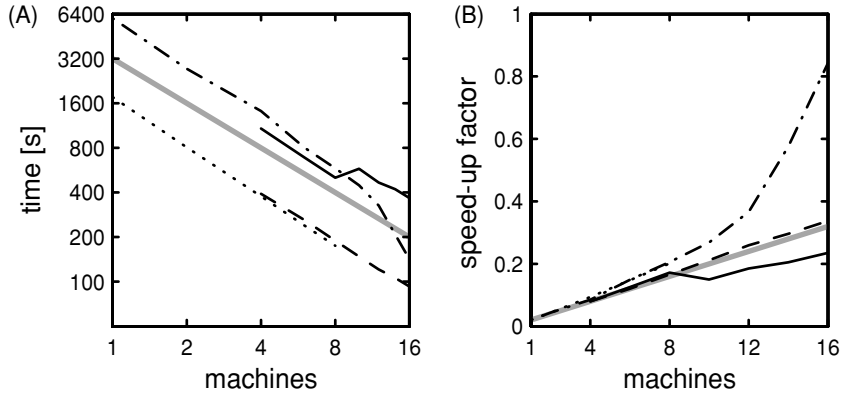

Figure 8: Scalability of simulation time with respect to number of processors: elderly PC cluster, solid line; recent PC cluster, dashed line; GS160, dash-dotted line; GS1280, dotted line. See the text for architecture and simulation details. (A) Simulation time for the 10^5 network against number of processors, log-log representation. The gray line indicates the slope for a linear speed-up. (B) Corresponding speed-up factor against number of processors. The diagonal (broad gray line) corresponds to linear speed-up.

(1 second). For the larger network (see Figure 7B), no saturation is observed. In fact, no saturation is visible even when using 40 machines of the modern PC cluster (not shown). In contrast to the other architectures, the elderly PC cluster slows when increasing from 8 to 10 machines. This is due to the fact that the two processors on each board share a memory bus. For 8 or fewer machines, the application can be distributed such that only one processor on each board is running it. Above this point, both processors are in use on at least one board, which leads to a significant reduction in efficiency. If the application is distributed such that both processors are in use on all contributing boards, then a supralinear behavior is seen for all numbers of machines, but at the cost of larger absolute run times.

In Figure 8A the simulation time of the 10^5 network is plotted against the number of machines used. All tested architectures show a steeper slope than that expected for a linear speed-up (gray line), commonly considered to be the maximum speed-up possible for a distributed application. This surprising result is due to the fact that the amount of fast cache memory available increases linearly with the number of processors. For our nondeterministic algorithm, the exploitation of this memory more than compensates for the memory and communication overheads that accrue as a result of distributing the simulation. In particular, in the range of machines tested, the communication overheads are negligible. Even when simulating the 10^4 high rate network on the elderly PC cluster (i.e., the maximimum amount

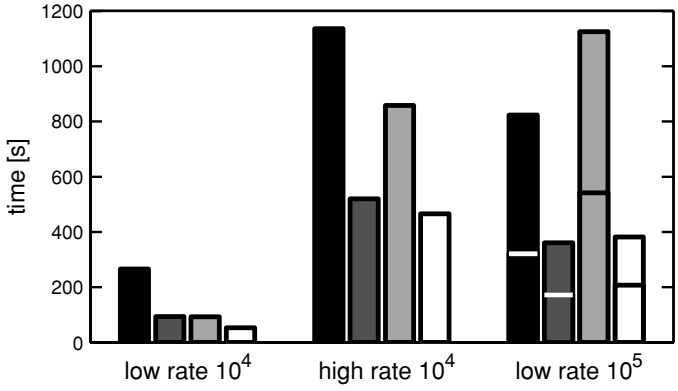

Figure 9: Comparison of architectures. Total time to run the three different types of simulation on four different architectures using eight processors. See the text for architecture and simulation details. In each block, the black bar refers to the elderly PC cluster, the dark gray bar to the recent PC cluster, the light gray bar to the GS160, and the white bar to the GS1280. The left block shows the run times of the 10^4 low-rate network, the middle block the run times of the 10^4 high-rate network, and the right block the run times of the 10^5 network. For the 10^5 network, the lower part of each bar shows the construction time of the network and the upper part the simulation time. For the 10^4 networks, the construction time is negligible.

of communication with respect to the number of neurons, with the slowest communication hardware), the time spent communicating amounted to less than 0.5% of the total runtime. The corresponding speed-up curves (see Wilkinson & Allen, 2004), that is, how many times faster the application runs with m processors than with 1 processor as a function of m, are plotted in Figure 8B. In the case of the PC clusters, the application is too large to be addressed by one 32-bit processor; therefore, the curves have been normalized appropriately. In this representation, the supralinear scaling is indicated by the fact that all the curves lie above the gray diagonal, indicating a linear speed-up, except for the elderly PC cluster for large n, as explained above. Similar results (not shown) were obtained for the 10^4 networks with low and high spike rates, whereby the supralinear behavior is more pronounced at high rates. At high rates, the efficiency of writing to the event buffers becomes an increasingly crucial factor, so the exploitation of the cache resource plays a much more important role than for low rate simulations. In the case of the recent PC cluster, we have been able to test up to 40 processors, and even at this high number, no saturation of the speed-up was observed, resulting in total run times for the 10^5 network of less than 2 minutes. In both panels, the reduction in efficiency of the elderly PC cluster caused by two processes competing for memory access is clearly

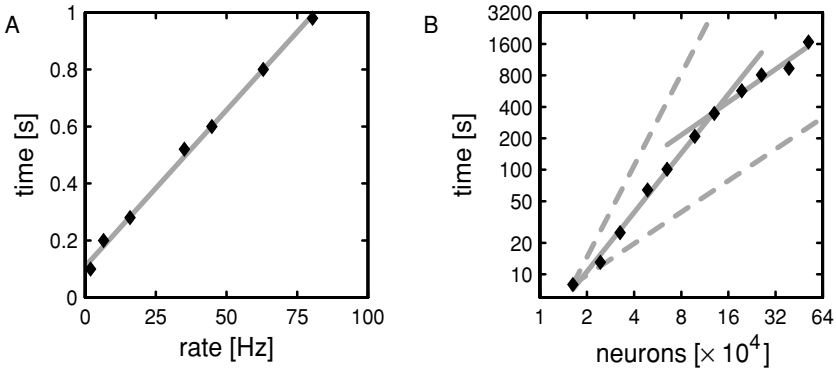

Figure 10: Scalability with respect to activity and network size. (A) Simulation time (diamonds) on eight processors of the GS1280 for 1 biological second of a 10^4 network plotted against average spike rate λ. The gray line is a linear fit to the data. (B) Simulation time (diamonds) on eight processors of the GS1280 for 1 biological second of low-rate networks (3.9 Hz) against number of neurons, log-log representation. The number of synapses per neuron increases linearly with the number of neurons (i.e., constant connection probability) until a biologically realistic connectivity is reached (at 13×10^4), after which the number of synapses per neuron remains constant. The gray lines are linear fits to the two regimes, with slopes of 1.88 and 1.05, respectively. The lower dashed gray line indicates the expected run time increase assuming a linear dependence of the run time on the number of neurons; the upper dashed gray line indicates the expected run-time increase assuming a quadratic dependence on the number of neurons.

visible. Again, supralinear behavior is observed for the entire series if the application is redistributed as described above, but at the cost of higher absolute run times.

A comparison of the run times for the different types of simulation is given in Figure 9. The total run time for all combinations of simulations and architectures is depicted for the case that eight processors are used. In all cases, the simulation finishes in less than 20 minutes. The recent PC cluster with its high-speed, low-latency network, and rapid clock speed has a clear advantage over both the elderly PC cluster and parallel computer and is at this number of processors comparable to the modern parallel computer.

To demonstrate the scalability of the software with respect to network activity and size, we varied one parameter while holding the others constant. Figure 10A shows that the software scales linearly with the average spike rate λ. An excellent scaling behavior is also seen with respect to the number of neurons in the network with constant rate and connection probability (see Figure 10B). It lies between the linear scaling due to the increase in the number of neurons and the quadratic scaling due to the increase in the

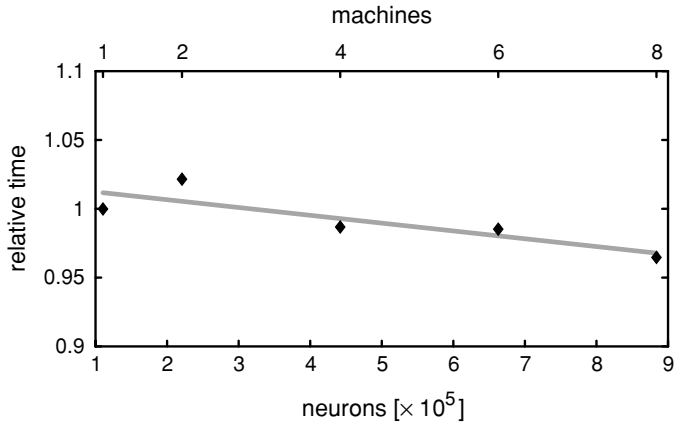

Figure 11: Scalability with respect to problem size. The ratio of serial simulation time and parallel simulation time (vertical) is shown as a function of network size (horizontal, bottom) and number of machines (horizontal, top) for the GS1280. The network size is increased from 110,500 to 884,000, keeping the number of neurons per machine and the number of synapses per neuron constant. The gray line represents a linear fit exhibiting a slope of -0.0057.

number of synapses. For increases in network size above 10^5, a near-linear increase is observed; having reached biological levels of complexity, the total number of synapses in the network increases only linearly.

We have discussed how simulation time scales with number of processors for a fixed network size (see Figure 8) and how the simulation time scales with the network size for a fixed number of machines (see Figure 10B). Another useful performance measure is the scaled speed-up (Wilkinson & Allen, 2004), where the network size is increased linearly with the number of machines. The motivation is that with a larger number of machines available, it should be possible to address a proportionally larger problem in the same time as the original problem on one machine. The scaled speed-up characterizes to what extent this assumption holds. Figure 11 shows that for the number of processors available on the parallel computer tested, the scaled speed-up remains close to 1, indicating excellent scalability of the software and the problem. The small, systematic linear decline of scaled speed-up presumably originates from the increased absolute amount of memory access and communication load.

6 Discussion

We described a scheme for the efficient distributed simulation of large heterogeneous spiking neural networks. In contrast to earlier approaches,

the simulation technology enables the investigation of recurrent networks with levels of connectivity and sparseness characteristic for the mammalian brain. A network of 100,000 neurons permits a biologically realistic number of synapses per neuron (of the order of 10^4) while simultaneously adhering to a biologically realistic constraint on the connection probability between two neurons (of the order of 0.1 in the local volume). In this sense, such a network constitutes a threshold size for realistic simulations of cortical networks. The technology described in this article easily overcomes this threshold on available hardware, requiring wall clock times suitable for routine use. For larger neural systems, the required computer memory and wall clock time scale merely linearly (as opposed to quadratically) with the number of neurons.

The neural network to be simulated can be distributed over many computers. Execution time and the computer memory required on an individual machine scale excellently with the number of machines used. As a consequence of the considerations above, larger networks can be simulated, or a reduction in execution time achieved, simply by adding a proportionate number of computers to the system. In addition to the distributed simulation of the dynamics, an important feature of our technology is the parallel generation of the network structure. A serial construction of the network would severely limit the speed-up, as the time required to construct the network can constitute a considerable fraction of the total execution time (see Figure 9). A similar argument holds for the generation of random numbers, which can also easily become the component limiting the speed-up. Consequently, the generation of random number is parallelized, and care is taken that simulation results are independent of the number of machines used to carry out the simulation.

The efficiency of the simulation scheme results from exploiting the fact that the interaction of network elements is noninstantaneous and mediated by point events (spikes). The frequency and bulk of the communication between machines are independent of the computation step size (precision of the simulation). The simulation scheme profits from the fact that the fast cache memory increases proportionally with the number of machines, reducing the ratio between the locally required working memory and the locally available cache (Wilkinson & Allen, 2004). Surprisingly, the increase in simulation speed gained more than compensates the overhead due to the communication in a distributed environment. Overall, a supralinear speed-up is observed, justifying the use of large clusters of computers. A detailed quantitative investigation of cache effects is outside the scope of this study. However, it is evident from the results presented that when deciding on hardware for a simulation project using our scheme, not only the clock speed of the processors but also the amount of cache memory should be considered.

We pointed out in section 3.1 that neither of the textbook simulation schemes, discrete time and event-driven algorithms, is optimal for the

simulation of biological neural networks. Instead, only a carefully adjusted hybrid of both leads to a satisfactory run-time behavior. A similar observation is made with respect to the class design of the software components (cf. the comment on this observation in Gamma, Helm, Johnson, & Vlissides, 1994). Only a well-balanced mixture of objects from the problem domain (neurobiology) and from the machine-oriented domain of parallel algorithms leads to a design that appropriately compromises between the heterogeneity of biological structure, usability of the software, and the efficiency of the simulation on today's computer hardware. Both observations argue for a pragmatic and undogmatic usage of software design principles and the selection of an implementation language supporting multiple paradigms (Stroustrup, 1994). Ideally, the interface with which the researcher, attempting to implement a new neuron model, is confronted would be expressed in terms of neuroscience concepts and objects, while the objects of the software layer below are optimized for cache exploitation and efficient communication. We have made some first steps in this direction, but further research is required to work out how this approach can consistently be applied to the different components of the simulation scheme.

Future work on this topic falls broadly into two categories: optimization and functionality. With respect to the former, the clearly observable cache effects suggest that much could be gained by optimizing the data structures accordingly. We are currently testing various alternative representations of the network structure and update schemes in order to enhance the cache usage, particularly for low numbers of machines. Furthermore, the load balancing currently carried out by the simulation kernel is minimal and relies on the static uniform distribution of network elements and their types over the available machines. Thus, efficient use of the hardware resources requires homogeneous computer clusters. While in the context of high-performance computing this does not represent a major constraint, the limitation becomes relevant when more structured networks are investigated with large differences in the communication load between and within subnetworks. The next step in addressing the problem of load balancing would be to provide user-level control over the mapping of network elements to machines. Finally, the results presented in this article demonstrate that it is possible and efficient to execute our code on computers with multiple processors. However, the use of a communication protocol developed under the constraints of distributed computing (Pacheco, 1997) does not fully exploit the existence of a working memory addressable by all processors. In the framework of the NEST initiative (www.nest-initiative.org) we are developing the simulation technology for computers with multiple processors. Current trends in computer hardware toward clusters of multiprocessor machines, whereby each machine has a small number of processors and support for multithreading (Butenhof, 1997) in the individual processors, makes a hybrid simulation kernel using multithreading locally and message passing between computers increasingly interesting.

One factor limiting the usability of the software described here is that the protocol for a particular simulation experiment must be specified by a C++ program. In a parallel line of work, we are developing a simulation language interpreter enabling the interactive specification and manipulation of neural systems simulations (Diesmann & Gewaltig, 2002). It remains to be investigated how the distributed simulation kernel can be combined with this interpreter. Other current work (Morrison, Hake, Straube, Plesser, & Diesmann, 2005) focuses on extending the functional range of the technology through the incorporation of precise (off-grid) spike times (see Hansel, Mato, Meunier, & Neltner, 1998; Rotter & Diesmann, 1999; Shelley & Tao, 2001, for discussion) and spike-time-dependent plasticity (Morrison, Aertsen, & Diesmann, 2004) into the simulation scheme. In future projects, structural plasticity and the interaction of spiking neural networks with modulatory chemical subsystems will also be addressed.

Acknowledgments

We acknowledge constructive discussions with Denny Fliegner, Stefan Rotter, Masayoshi Kubo, and the members of the NEST collaboration (in particular Marc-Oliver Gewaltig and Hans Ekkehard Plesser). This work was partially funded by the Volkswagen Foundation, GIF, BIF, DIP F1.2, DAAD 313-PPP-N4-lk, and BMBF Grant 01GQ0420 to BCCN Freiburg. All simulations have been carried out using the parallel computing facilities of the Max-Planck-Institute for Fluid Dynamics (now MPI for Dynamics and Self Organization) in Göttingen and the Agricultural University of Norway in Ås. Part of the work was carried out when A. M. and M. D. were based at the Max-Planck-Institute for Fluid Dynamics. The initial distributed version of the simulation software was developed by Mehring in the context of Mehring, et al. (2003).

References

Abbott, L. F., & Nelson, S. B. (2000). Synaptic plasticity: Taming the beast. *Nat. Neurosci.*, *3*(Suppl.), 1178–1183.

Abeles, M. (1982). *Local cortical circuits: An electrophysiological study*. Berlin: Springer-Verlag.

Abeles, M. (1991). *Corticonics: Neural circuits of the cerebral cortex*. Cambridge: Cambridge University Press.

Aertsen, A., Gerstein, G., Habib, M., & Palm, G. (1989). Dynamics of neuronal firing correlation: Modulation of "effective connectivity." *J. Neurophysiol.*, *61*(5), 900–917.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Reading, MA: Addison-Wesley.

Aviel, Y., Mehring, C., Abeles, M., & Horn, D. (2003). On embedding synfire chains in a balanced network. *Neural Comput.*, *15*(6), 1321–1340.

Bernard, C., Ge, Y. C., Stockley, E., Willis, J. B., & Wheal, H. V. (1994). Synaptic integration of NMDA and non-NMDA receptors in large neuronal network models solved by means of differential equations. *Biol. Cybern.*, *70*, 267–273.

Bi, G.-q., & Poo, M.-m. (1998). Activity-induced synaptic modifications in hippocampal culture: Dependence on spike timing, synaptic strength and cell type. *J. Neurosci.*, *18*, 10464–10472.

Bi, G., & Poo, M. (2001). Synaptic modification by correlated activity: Hebb's postulate revisited. *Annu. Rev. of Neurosci.*, *24*, 139–166.

Bower, J. M., & Beeman, D. (1997). *The book of GENESIS: Exploring realistic neural models with the GEneral NEural SImulation System* (2nd ed.). New York: TELOS, Springer-Verlag.

Braitenberg, V. (1978). Cell assemblies in the cerebral cortex. In R. Heim & G. Palm (Eds.), *Theoretical approaches to complex systems*. Berlin: Springer.

Braitenberg, V., & Schüz, A. (1998). *Cortex: Statistics and geometry of neuronal connectivity* (2nd ed.). Berlin: Springer-Verlag.

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.*, *8*(3), 183–208.

Butenhof, D. R. (1997). *Programming with POSIX threads*. Reading, MA: Addison-Wesley.

Chance, F. S., Abbott, L. F., & Reyes, A. D. (2002). Gain modulation from background synaptic input. *Neuron*, *35*, 773–782.

Dayan, P., & Abbott, L. F. (2001). *Theoretical neuroscience*. Cambridge, MA: MIT Press.

Delorme, A., & Thorpe, S. (2003). Spikenet: An event-driven simulation package for modeling large networks of spiking neurons. *Network: Comput. Neural Systems*, *14*, 613–627.

Destexhe, A., Rudolph, M., & Pare, D. (2003). The high-conductance state of neocortical neurons in vivo. *Nat. Rev. Neurosci.*, *4*, 739–751.

Diesmann, M., & Gewaltig, M.-O. (2002). NEST: An environment for neural systems simulations. In T. Plesser & V. Macho (Eds.), *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001* (pp. 43–70). Göttingen: Ges. für Wiss. Datenverarbeitung.

Diesmann, M., Gewaltig, M.-O., & Aertsen, A. (1995). *SYNOD: An environment for neural systems simulations: Language interface and tutorial* (Tech. Rep. GC-AA-/95-3). Tel Aviv: Weizmann Institute of Science.

Diesmann, M., Gewaltig, M.-O., & Aertsen, A. (1999). Stable propagation of synchronous spiking in cortical neural networks. *Nature*, *402*, 529–533.

Diesmann, M., Gewaltig, M.-O., Rotter, S., & Aertsen, A. (2001). State space analysis of synchronous spiking in cortical neural networks. *Neurocomputing*, *38–40*, 565–571.

Ermentrout, B. (2002). *Simulating, analyzing, and animating dynamical systems: A guide to Xppaut for researchers and students (software, environments, tools)*. Philadelphia: Society for Industrial and Applied Math.

Galassi, M., Gough, B., & Jungman, G. (2001). *Gnu scientific library: Reference manual*. Bristol, UK: Network Theory Ltd.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented Software*. Reading, MA: Addison-Wesely.

Gerstein, G. L., Bedenbaugh, P., & Aertsen, A. (1989). Neuronal assemblies. *IEEE Trans. Biomed. Eng.*, *36*, 4–14.

Gewaltig, M.-O. (2000). *Evolution of synchronous spike volleys in cortical Networks: Network simulations and continuous probabilistic models*. Aachen, Germany: Shaker.

Gross, J., & Yellen, J. (1999). *Graph theory and its applications*. Boca Ratan, FL: CRC Press.

Hansel, D., Mato, G., Meunier, C., & Neltner, L. (1998). On numerical simulations of integrate-and-fire neural networks. *Neural Comput.*, *10*(2), 467–483.

Hawkes, A. G. (1971). Point spectra of some mutually exciting point processes. *Journal of the Royal Statistical Society (London) B*, *33*, 438–443.

Hebb, D. O. (1949). *Organization of behavior: A neurophysiological theory*. New York: Wiley.

Hines, M., & Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput.*, *9*, 1179–1209.

Jack, J. J. B., Noble, D., & Tsien, R. W. (1983). *Electric current flow in excitable cells*. New York: Oxford University Press.

Knuth, D. E. (1997). *The art of computer programming: Seminumerical algorithms* (3rd ed., Vol. 2). Reading, MA: Addison-Wesley.

Koch, C. (1999). *Biophysics of computation: Information processing in single neurons*. New York: Oxford University Press.

Koch, C., & Segev, I. (1989). *Methods in neuronal modeling*. Cambridge, MA: MIT Press.

Kuhn, A., Aertsen, A., & Rotter, S. (2003). Higher-order statistics of input ensembles and the response of simple model neurons. *Neural Comput.*, *15*(1), 67–101.

Kuhn, A., Aertsen, A., & Rotter, S. (2004). Neuronal integration of synaptic input in the fluctuation-driven regime. *J. Neurosci.*, *24*, 2345–2356.

Kumar, A., Kremkow, J., Rotter, S., & Aertsen, A. (2004). Synaptic integration in a 3-compartment model of layer 5 pyramidal neurons. *FENS Abstr.*, 2, A014.27.

Larkum, M., Zhu, J., & Sakmann, B. (2001). Dendritic mechanisms underlying the coupling of the dendritic with the axonal action potential initiation zone of adult rat layer 5 pyramidal neurons. *J. Neurophysiol.*, *533 (pt. 2)*, 447–466.

Lee, G., & Farhat, N. H. (2001). The double queue method: A numerical method for integrate-and-fire neuron networks. *Neural Networks*, *14*, 921–932.

Markram, H., & Tsodyks, M. (1996). Redistribution of synaptic efficacy between neocortical pyramidal neurons. *Nature*, *382*(6594), 807–810.

Mattia, M., & Del Giudice, P. (2000). Efficient event-driven simulation of large networks of spiking neurons and dynamical synapses. *Neural Comput.*, *12*(10), 2305–2329.

Mehring, C., Hehl, U., Kubo, M., Diesmann, M., & Aertsen, A. (2003). Activity dynamics and propagation of synchronous spiking in locally connected random networks. *Biol. Cybern.* *88*(5), 395–408.

Morrison, A., Aertsen, A., & Diesmann, M. (2004). *Stability of plastic recurrent networks*. Paper presented at the Monte Verita Workshop on Spike-Timing Dependent Plasticity, Monte Verita, Ascona, Switzerland.

Morrison, A., Hake, J., Straube, S., Plesser, H. E., & Diesmann, M. (2005). Precise spike timing with exact subthreshold integration in discrete time network simulations. In *Proceedings of the 30th Göttingen Neurobiology Conference*, Neuroforum Supplement 1, pp. 205B.

Morrison, A., Mehring, C., Diesmann, M., Aertsen, A., & Geisel, T. (2003). Distributed simulation of large biological neural networks. In *Proceedings of the 29th Göttingen Neurobiology Conference*, (p. 590).

Pacheco, P. S. (1997). *Parallel programming with MPI*. San Francisco: Morgan Kaufmann.

Palm, G. (1990). Cell assemblies as a guideline for brain research. *Conc. Neurosci.*, *1*, 133–148.

Reutimann, J., Giugliano, M., & Fusi, S. (2003). Event-driven simulation of spiking neurons with stochastic dynamics. *Neural Comput.*, *15*, 811–830.

Rotter, S., & Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biol. Cybern.*, *81*(5/6), 381–402.

Salinas, E., & Sejnowski, T. J. (2000). Impact of correlated synaptic input on output firing rate and variability in simple neuronal models. *J. Neurosci.*, *20*(16), 6193–6209.

Shadlen, M. N., & Newsome, W. T. (1998). The variable discharge of cortical neurons: Implications for connectivity, computation, and information coding. *J. Neurosci.*, *18*(10), 3870–3896.

Shelley, M. J., & Tao, L. (2001). Efficient and accurate time-stepping schemes for integrate-and-fire neuronal networks. *J. Comput. Neurosci.*, *11*(2), 111–119.

Singer, W. (1993). Synchronization of cortical activity and its putative role in information processing and learning. *Annu. Rev. Physiol.*, *55*, 349–374.

Singer, W. (1999). Time as coding space. *Curr. Opin. Neurobiol.*, *9*(2), 189–194.

Stroustrup, B. (1994). *The design and evolution of C++*. Reading, MA: Addison-Wesley.

Stroustrup, B. (1997). *The C++ programming language* (3 ed.). Reading, MA: Addison-Wesley.

Tam, A., & Wang, C. (2000). Efficient scheduling of complete exchange on clusters. In G. Chaudhry & E. Sha (Eds.), *13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000)*. Cary, NC: International Society for Computers and Their Applications.

Tetzlaff, T., Morrison, A., Geisel, T., & Diesmann, M. (2004). Consequences of realistic network size on the stability of embedded synfire chains. *Neurocomputing*, *58–60*, 117–121.

Thomson, A. M., & Deuchars, J. (1994). Temporal and spatial properties of local circuits in neocortex. *TINS*, *17*, 119–126.

von der Malsburg, C. (1981). *The correlation theory of brain function* (Internal Rep. 81-2). Göttingen: Max-Planck-Institute for Biophysical Chemistry.

von der Malsburg, C. (1986). Am I thinking assemblies? In G. Palm & A. Aertsen (Eds.), *Brain theory* (pp. 161–176) Berlin: Springer-Verlag.

Wilkinson, B., & Allen, M. (2004). *Parallel programming: Techniques and applications using networked workstations and parallel computers* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

Received July 20, 2004; accepted January 31, 2005.