Efficient parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers

Hans Plesser, Jochen Eppler, Abigail Morrison, Markus Diesmann, Marc-Oliver Gewaltig

2007

Preprint:

This is an accepted article published in Euro-Par 2007 Parallel Processing. The final authenticated version is available online at: https://doi.org/[DOI not available]

Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers

Hans E. Plesser[1], Jochen M. Eppler[2,3], Abigail Morrison[4], Markus Diesmann[3,4], and Marc-Oliver Gewaltig[2,3]

[1] Dept. of Mathematical Sciences and Technology, Norwegian University of Life Sciences, PO Box 5003, 1432 Ås, Norway, hans.ekkehard.plesser@umb.no
[2] Honda Research Institute, Offenbach/Main, Germany
[3] Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, Freiburg, Germany
[4] RIKEN Brain Science Institute, Wako-shi, Saitama, Japan

Abstract. To understand the principles of information processing in the brain, we depend on models with more than 10^5 neurons and 10^9 connections. These networks can be described as graphs of threshold elements that exchange point events over their connections.
From the computer science perspective, the key challenges are to represent the connections succinctly; to transmit events and update neuron states efficiently; and to provide a comfortable user interface. We present here the neural simulation tool NEST, a neuronal network simulator which addresses all these requirements. To simulate very large networks with acceptable time and memory requirements, NEST uses a hybrid strategy, combining distributed simulation across cluster nodes (MPI) with thread-based simulation on each computer. Benchmark simulations of a computationally hard biological neuronal network model demonstrate that hybrid parallelization yields significant performance benefits on clusters of multi-core computers, compared to purely MPI-based distributed simulation.

1 Introduction

The neuronal networks in our brains can be described as weighted, directed graphs, with neurons as nodes and synaptic connections as edges. Neurons communicate by sending and receiving point events (spikes) through their connections (synapses). In the mammalian cortex, each neuron sends connections to about 10^4 other neurons and receives connections from as many. Just 1 mm^3 cortex contains some 10^5 neurons with 10^9 connections [1]. This represents a threshold size for simulations, as a realistic number of synapses per neuron can be combined with realistic sparseness (connection probability ~ 0.1). Brain function emerges from the spatio-temporal patterns of neuronal spike activity, but

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 672–681, 2007.

the principles are poorly understood. Progress in understanding brain function therefore depends on simulation studies of large cortical networks.

In large neuronal networks, we can neglect the geometric and biophysical complexity of individual nerve cells and describe neurons as point-like objects with a dynamic state governed by a set of ODEs. The most common state variable is the membrane potential V, which is affected by spikes that arrive at the neuron's synapses. Whenever V crosses a threshold value V_{th}, the neuron produces a spike, which is transmitted to all adjacent neurons with a delay of a few milliseconds. Each connection can have a different delay and weight. Weights may evolve as a result of neuronal activity, a phenomenon known as synaptic plasticity, the biological substrate of learning. The spikes of an individual neuron are rare and occur at rates of 1–50 Hz, whereas the rate of incoming spikes is of the order of 100 kHz due to some 10^4 incoming connections.

Simulating large-scale neuronal networks poses several challenges: (i) 10^9–10^{12} connections must be stored; this requires a distributed representation. (ii) A large number of spikes must be buffered until they are transmitted across the network. (iii) Simulation results must be reproducible down to the level of membrane potentials and spike times. (iv) The object-oriented implementation must be appropriate for the problem domain and allow network and machine level optimizations such as efficient caching.

In this contribution, we describe how the Neural Simulation Tool NEST [2] addresses these issues to efficiently simulate neuronal networks of more than 10^5 neurons and 10^9 synapses. In section 2 we discuss how NEST represents nodes and connections, before describing the update and communication algorithms in section 3. Section 4 demonstrates the performance of our hybrid approach. NEST is available from `www.nest-initiative.org`.

2 Network Representation

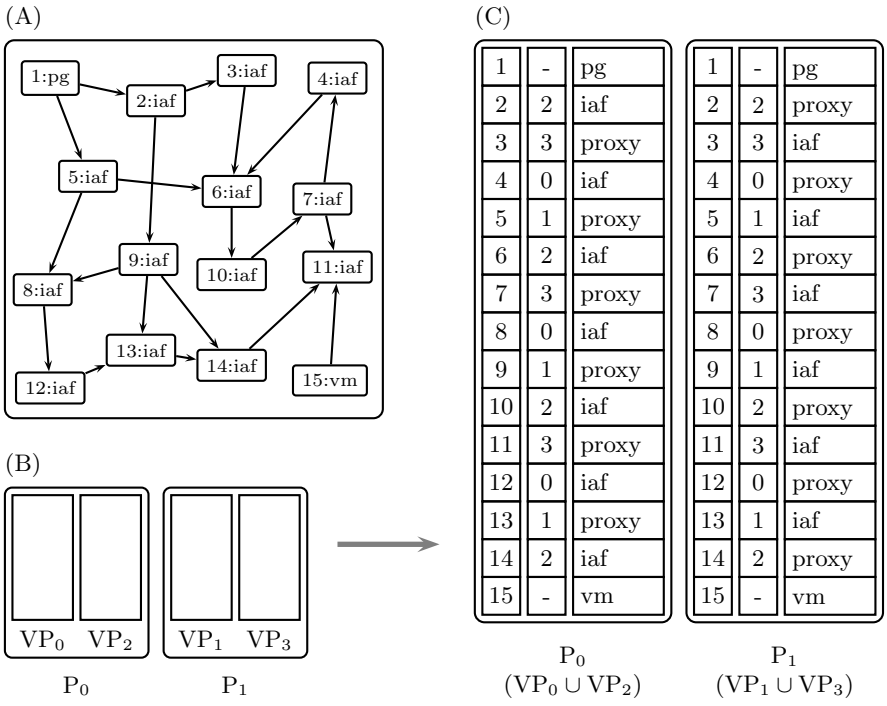
A network model consists of nodes, connections, and events, each represented by an abstract base class. Models for neurons and devices inherit from `class Node` and implement the state vector, the internal dynamics, and the responses to different types of events.

NEST distributes a network model over N_{VP} *virtual processes*. A virtual process is a POSIX thread that lives in one of N_{MPI} MPI processes [3, 4]. The total number of virtual processes N_{VP} is the number of MPI processes times the number of threads per process: $N_{\mathrm{VP}} = N_{\mathrm{MPI}} \times N_{\mathrm{Thrd}}$. NEST ensures that for a given number of virtual processes N_{VP}, all simulations of a model yield identical results, independent of the combination of N_{MPI} and N_{Thrd}.

Neuron models are often stochastic, consuming many random numbers. To distribute the load of random number generation while obtaining identical simulation results for different combinations of N_{MPI} and N_{Thrd} on $N_{\mathrm{VP}} = \mathrm{const}$ virtual processes, we give each virtual process its own random number generator. By default we use a lagged Fibonacci generator, because it can be seeded to produce non-overlapping sequences of random numbers [5].

2.1 Nodes and Proxies

In a network with n nodes, each node is given a unique integer `gid` in order of creation, and is assigned to a virtual process such that `vid` = `gid` mod N_{VP}. Each virtual process manages the memory for its nodes. In addition, each virtual process has a vector of size n, called *node list*, with pointers to its own nodes and pointers to proxies for nodes that belong to other virtual process. Within an MPI process, we can collapse the node lists of all its virtual processes into a single list to conserve memory, as illustrated in figure 1.



1	-	pg
2	2	iaf
3	3	proxy
4	0	iaf
5	1	proxy
6	2	iaf
7	3	proxy
8	0	iaf
9	1	proxy
10	2	iaf
11	3	proxy
12	0	iaf
13	1	proxy
14	2	iaf
15	-	vm

1	-	pg
2	2	proxy
3	3	iaf
4	0	proxy
5	1	iaf
6	2	proxy
7	3	iaf
8	0	proxy
9	1	iaf
10	2	proxy
11	3	iaf
12	0	proxy
13	1	iaf
14	2	proxy
15	-	vm

Fig. 1. Distributed and multi-threaded network representation. (A) The network as a directed graph. (B) Sketch of the distribution of four virtual processes onto two MPI processes, P_0 and P_1. (C) Collapsed node lists of the two MPI processes containing the nodes of two VPs each. The first column shows the node's `gid`, the second contains the VP a node is assigned to; '`-`' indicates that a node is created for each VP. The third column contains the node type: `pg`, Poisson spike generator device; `vm`, voltmeter device; `iaf`, integrate-and-fire model neurons.

This representation has the following advantages: (i) we can access each node directly with its `gid` as index to the node list on any virtual process; (ii) each virtual process knows the type of each node in the network; (iii) there is no memory overhead for pure multi-threaded simulations.

For a network of n nodes, each requiring S_N bytes of memory, and proxy nodes requiring S_P bytes, the required memory is given by:

$$M_{\text{nodes}} = \frac{n}{N_{\text{MPI}}} S_N + \left(n - \frac{n}{N_{\text{MPI}}} \right) S_P \ . \tag{1}$$

Typical values are $S_N = 480$ bytes and $S_P = 56$ bytes. Thus, for 10 and more MPI processes, more than half of all memory occupied by objects is occupied by proxy nodes, and more than 90% for more than 80 MPI processes. In absolute numbers, though, M_{nodes} is only around 70 MB for a network of 10^6 nodes distributed across 100 processes, which is negligible compared to the memory required for the connections in the network, cf. sec. 2.2. From a performance aspect, a node list filled with mostly proxies could become suboptimal if the node list became so large that it could no longer be cached efficiently. In this case, a fast hashing lookup may become more efficient.

Devices So far we assumed that nodes correspond to neurons. We will now discuss nodes representing devices, such as spike injectors and recorders. There are two types of devices: recording devices and stimulation devices. Recording devices measure the state of one or more neurons and write the data to disk. If nodes on different virtual processes are assigned to the same device, each of the virtual processes gets its own instance of the recording device. This has two advantages: (i) the measured data need not be transmitted between virtual processes; (ii) each device instance can write to its own local disk. Stimulation devices supply signals to one or more neurons, thereby manipulating their state. Again, each virtual process has its own instance of a given stimulation device to reduce the amount of data that must be exchanged between virtual processes. For some stimulation devices which produce random signals we must ensure that all instances in different virtual processes produce identical signals. Thus, these devices cannot use the random number generator of the virtual process, but have their own random number generators, initialized with identical seeds.

2.2 Connections

A connection between two nodes is defined by at least four numbers: the `gid` of the sending node, the `gid` of the receiving node, a weight, and a delay. More complicated connections have weights that change, depending on the activity of the connected nodes.

Different types of connections can be implemented by classes that derive from the abstract base class `Connector`. These can implement arbitrarily complex dynamics, provided they only depend on the previous state, the time since the last event, and information available from the target node. The most important applications are synaptic depression [6] and spike-timing dependent plasticity (STDP) [7]. An algorithm for STDP suitable for distributed computing can be found in [8].

Because the connections dominate the memory requirements of large networks, NEST splits them up such that each virtual process only stores the incoming connections to its own nodes. A vector inside each `Connector` contains pointers to all local target nodes, along with the delay (integer), and weight (double). The memory required for connections per MPI process is:

$$M_{\text{conn}} = \frac{n \times c \times S_C}{N_{\text{MPI}}} \,, \tag{2}$$

where c is the number of outgoing connections per node and S_C the memory per connection. For connections with constant weight and delay, $S_C = 32$ bytes; plastic synapses require more memory. A network of 10^6 nodes with 10^4 connections each thus requires 32 GB connection memory per process if distributed across 10 MPI processes, but only 3.2 GB per process if distributed across 100 processes.

A `ConnectionManager` stores the connections to all virtual processes of an MPI process in a three-dimensional data-structure. The first dimension is the thread number (virtual process) of the target node. The second dimension is the `gid` of the source node, and the third dimension is the index of the connection type. This memory layout has two advantages: (i) we can construct networks with heterogeneous synaptic dynamics; (ii) it is optimal for multi-threaded event delivery (cf. sec. 3.2) and the efficient implementation of synaptic dynamics [8].

3 Network Update and Event Exchange

Conceptually, NEST evaluates the network model on an evenly spaced time-grid $t_i := i \cdot \Delta$ and at each point, the network is in a well-defined state S_i. Starting at an initial state S_0, a global state transfer function $\mathrm{U}(S)$ propagates the system from one state to the next, such that $S_{t+\Delta} \leftarrow \mathrm{U}(S_t, \Delta)$. As a side effect of $\mathrm{U}(S_t)$, nodes create events that must be delivered to the target nodes after a delay that depends on the connection.

NEST evaluates a network model using the following algorithm:

1: $T \leftarrow 0$
2: **while** $T < T_{\text{stop}}$ **do**
3: **parallel on all** $vp \in N_{\text{VP}}$ **do**
4: deliver all events due
5: call $\mathrm{U}(S_T)$ for all nodes
6: **end parallel**
7: exchange events between VPs
8: increment network time: $T \leftarrow T + \Delta$
9: **end while**

The N_{VP} virtual processes evaluate steps 4 and 5 in parallel, and in step 6 they synchronize to exchange their events in step 7.

Although NEST uses a discrete event framework, it does not use a global event-driven update [9, 10]. Event-driven simulation assumes that the communication between nodes is rare and the update of a node is expensive. This does not

hold for biological networks, however: If a typical cortical neuron receives spikes from $\sim 10^4$ other neurons at a rate of ~ 10 Hz, the average interval between spike arrivals is ~ 0.01 ms. However, for most neuron models integration steps of $h \sim 0.1$ ms are sufficient. Thus event-driven update would need one order of magnitude *more* updates than time-driven update; see [11] for details.

In the following we describe in more detail (i) how to maximize the time increment Δ and (ii) how to collect, exchange, and deliver events between virtual processes.

3.1 Exploiting Delays for Cache-Efficient Update

Nodes affect each other by exchanging events that arrive at their destination with a delay $d_{ij} > 0$. The time period Δ is the largest permissible temporal desynchronization between any two nodes in the network. Δ may be increased as long as this does not change the order of events. This is equivalent to a system of distributed clocks that synchronize each other with events. Lamport showed that the smallest transmission delay $d_{\min}$ defines the interval at which clocks must be synchronized to maintain the order of events [12]. Accordingly, NEST sets Δ to $d_{\min}$. During this period, all nodes are effectively decoupled.

Most neural simulators use the integration step h of the neuron dynamics as the time increment. Maximizing Δ, typically to ~ 1 ms, i.e. about 10 times larger than h, has two advantages: (i) the virtual processes can run independently for a longer time, thereby reducing the number of synchronizations and thus the communication overhead; (ii) the state-update of each node can run a few tens of integration steps *en bloc*, keeping all required data in the CPU's L1 cache.

3.2 Global Event Exchange

NEST does not transmit individual events between virtual processes, as there are far too many. Instead, for each node that produced an event, the following information is transmitted: the `gid` of the sending node and the time at which the event occurred (address event representation [13]). All other connection parameters, such as the list of weights, delays and targets, are available at each virtual process. With this information, the virtual processes reconstruct the actual events and deliver them to their destinations.

We describe below the buffering and transmission of spike events constrained to a discrete time grid $t_n = nh$. This scheme is easily extended to spikes at arbitrary times [11].

Sender-Side Buffering Each MPI process has a three-dimensional buffer (*spike register*) to record the nodes that produced a spike-event during the update interval Δ. The first dimension represents the VP, so that they can write without collisions. The second dimension represents the time of the event with one entry per integration step h. The third dimension is a list of `gid`s, one for each spike on a given thread at a given time. The total number of spikes per

virtual process per update interval is small: even assuming 10^6 neurons firing at 10 Hz and distributed across 20 VPs, only some 500 spikes occur per VP and update interval. With 4 threads per MPI process, the spike register occupies less than 20 kB.

Spike Exchange and Delivery Before spikes are exchanged between MPI processes, they are copied from the spike register to a communication buffer as follows: their `gid`s are written to the buffer, ordered by the integration time step at which the spikes were generated. Sentinels separate spikes generated during different steps. Since the number of integration steps per update cycle is fixed, the receiver can reconstruct the spike time from the sentinels. Each process also maintains buffers to receive the `gid`s from other processes. Once all buffers are set up, the spike buffers are exchanged between MPI processes by simultaneous pairwise exchange using CPEX [14–16].

Each virtual process delivers the spikes to its nodes in the parallel step 4 of the update algorithm (sec. 3). For each entry of the communication buffer, which now contains both local and remote spikes, it executes the following algorithm.

1: $n_{\text{sentinels}} \leftarrow 0$
2: **if** entry is sentinel **then**
3: $\quad n_{\text{sentinels}} \leftarrow n_{\text{sentinels}} + 1$
4: **else**
5: $\quad$ calculate t_{spike} from network time and $n_{\text{sentinels}}$
6: $\quad$ **for all** `tgt` $\in$ local targets **do**
7: $\quad\quad$ send spike time, weight, delay to `tgt`
8: $\quad\quad$ `tgt` stores spike in its ring buffer according to delay [14].
9: $\quad$ **end for**
10: **end if**

4 Performance

The scaling of large-scale simulations of neural networks depends significantly on the computational load of the individual neuron. The more complex the neuron, the better the scaling, as the ratio of local computation to communication costs increases. We therefore consider the following benchmark to be a hard problem in the field of distributed neural network simulations: the computation load is low, because the neuron and synapse models are simple, but the communication load is high, as the network has a biologically realistic connection density.

Benchmark Simulation The network consists of 12500 leaky integrate-and-fire neurons (80% excitatory, 20% inhibitory), each receiving input from 10% of all neurons, mediated by alpha-shaped current-injecting synapses with a synaptic delay of 1 ms (total number of synapses: 15.6×10^6). The neurons are initialized with random membrane potentials and receive a constant DC input adjusted to sustain asynchronous irregular firing at 12.7 Hz [17]. For a complete network specification and numerics, see [11].

Simulation times were measured on a cluster of Sun X4100 compute nodes equipped with two dual-core 2.4 GHz AMD Opteron 280 processors, 8GB RAM, and Mellanox MTS2400 Infiniband interconnect under SuSE Linux Enterprise Server 9 using the Scali MPI Connect 4.4 library. Threads were bound to CPU cores using the `taskset` command.

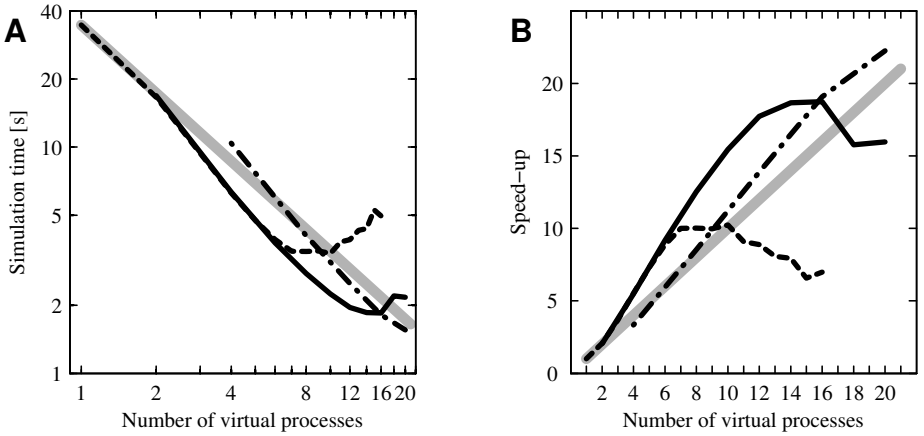


Fig. 2. Performance of different parallelization strategies as a function of the number of virtual processes. Single-thread MPI processes, dashed line; MPI processes with 2 threads, solid line; MPI processes with 4 threads, dash-dotted line. (A) Simulation time for one biological second in double-logarithmic representation. (B) Speed-up. The gray diagonal indicates the slope for a linear speed-up in both cases. Data obtained for simulations of 10 s biological time with a time step of 0.1 ms, averaged over 5 trials.

Results Figure 2 clearly demonstrates that the parallelization strategy significantly affects the scaling and absolute run-time of the simulation. A purely MPI-parallelized simulation shows supra-linear speed-up up to 8 virtual processes, rapidly saturates, and then undergoes a significant decrease in performance. The supra-linear speed-up is due to increasingly efficient caching [14], and the saturation in performance is due to the communication overhead.

By using a hybrid strategy with two threads per MPI process, such that both threads are bound to the same CPU, the number of MPI processes is halved. This reduces the number of send/receive operations per communication step by a factor of four and results in a performance which is better than the single-threaded case for numbers of virtual processes greater than eight. The performance of this hybrid strategy remains supra-linear up to 16 virtual processes, thus substantially reducing the absolute simulation time.

Reducing the number of MPI processes further by increasing the number of threads per MPI process to four leads to worse performance for small numbers of

virtual processors. This is due to the fact that memory allocation is performed by a single thread on each MPI process; as a result of the NUMA architecture, memory access is sub-optimal for the two threads on the non-allocating processor. The role of memory access is corroborated by simulating with two threads per MPI process as above, but binding the threads to different CPUs. This results in a performance which lies between that of the two-thread same-CPU variant discussed above and that of the four-thread variant (data not shown). This analysis is further supported by benchmarks performed on a Sun V40z server with four dual-core 2.2GHz AMD Opteron 875 processors, in which the threads used during simulation were placed at arbitrary cores relative to the thread constructing the network. Simulation times increased with increasing memory-access distance between the core used for construction and those used for simulation [18]. The costs of the sub-optimal memory access outweigh the benefits of decreasing the number of packets until 16 virtual processes, after which the four-thread variant becomes the most efficient simulation strategy.

5 Conclusions

Supra-linear scaling for a distributed biological neural network simulation was demonstrated for the first time in [14]. This result has since been confirmed by several other laboratories. In the present work we show that a hybrid approach to neural network simulation, combining multi-threading and distributed computing techniques, achieves an even better performance than a purely distributed solution. This suggests that the infrastructure of NEST is appropriate for future generations of multiprocessor, multi-core clusters.

The problem studied here was chosen to be particularly hard with respect to communication. In studies with larger neural networks or with more complex dynamics, NEST performance saturates at much larger numbers of processors: Simulation time for a network of 10^5 neurons with 10^9 synapses, driven by Poisson background input, shows supra-linear scaling up to 80 virtual processes on the same hardware. Other laboratories have shown good scaling of large-scale simulations on systems with thousands of processors, albeit on less hard problems [19, 20]. The scaling of NEST on such systems remains to be investigated.

The benchmarking results demonstrate the importance of sophisticated memory allocation on modern NUMA machines. Future work on NEST will be concerned with improving memory access times in a hybrid message-passing and multi-threading environment and further optimizing communication with respect to number of packets and latency hiding.

Acknowledgements HEP acknowledges Anita Woll for the execution of benchmark tests. Benchmarks were performed on supercomputing equipment at the Norwegian University of Life Sciences and the University of Freiburg. Partially funded by DAAD/NFR 313-PPP-N4-lk, DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, and EU Grant 15879 (FACETS).

References

1. Braitenberg, V., Schüz, A.: Cortex: Statistics and Geometry of Neuronal Connectivity. Second edn. Springer, Berlin (1998)
2. Gewaltig, M.O., Diesmann, M.: NEST. Scholarpedia (2007)
3. Message Passing Interface Forum: MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee (1994)
4. Lewis, B., Berg, D.J.: Multithreaded programming with pthreads. Sun Microsystems, Mountain View (1998)
5. Knuth, D.E.: The Art of Computer Programming. Third edn. Volume 2. Addison-Wesley, Reading, MA (1998)
6. Thomson, A.M., Deuchars, J.: Temporal and spatial properties of local circuits in neocortex. Trends Neurosci **17** (1994) 119–126
7. Bi, G.Q., Poo, M.M.: Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. J Neurosci **18** (1998) 10464–10472
8. Morrison, A., Aertsen, A., Diesmann, M.: Spike-time dependent plasticity in balance recurrent networks. Neural Comput **19** (2007) 1437–1467
9. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation. Second edn. Academic Press, Amsterdam (2000)
10. Brette, R., et al.: Simulation of networks of spiking neurons: A review of tools and strategies. J Comput Neurosci (in press) (2007)
11. Morrison, A., Straube, S., Plesser, H.E., Diesmann, M.: Exact subthreshold integration with continuous spike times in discrete time neural network simulations. Neural Comput **19** (2007) 47–79
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (1978) 558–565
13. Lazzaro, J.P., Wawrzynek, J., Mahowald., M., Sivilotti, M., Gillespie, D.: Silicon auditory processors as computer peripherals. IEEE Transactions on Neural Networks **4** (1993) 523–528
14. Morrison, A., Mehring, C., Geisel, T., Aertsen, A., Diesmann, M.: Advancing the boundaries of high connectivity network simulation with distributed computing. Neural Comput **17** (2005) 1776–1801
15. Tam, A., Wang, C.: Efficient scheduling of complete exchange on clusters. In: 13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000), Las Vegas (2000)
16. Gross, J., Yellen, J.: Graph Theory and its Applications. CRC Press (1999)
17. Brunel, N.: Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. J Comp Neurosci **8** (2000) 183–208
18. Woll, A.: Performance analysis of an MPI- and thread-parallel neural network simulator. Master's thesis, Norwegian University of Life Sciences (2007)
19. Djurfeldt, M., Johansson, C., Ekeberg, Ö., Rehn, M., Lundqvist, M., Lansner, A.: Massively parallel simulation of brain-scale neuronal network models. Technical Report Technical Report TRITA-NA-P0513, KTH, School of Computer Science and Communication Stockholm, Stockholm (2005)
20. Migliore, M., Cannia, C., Lytton, W.W., Markram, H., Hines, M.: Parallel network simulations with NEURON. J Comp Neurosci **21** (2006) 119–223