KTH Computer Science
and Communication

School of Computer Science and Communication
TRITA-NA-P0513 • CBN • ISSN 0348-2952

Massively parallel simulation of brain-scale neuronal network models

Mikael Djurfeldt, Christopher Johansson, Örjan Ekeberg,
Martin Rehn, Mikael Lundqvist, and, Anders Lansner

Computational Biology and Neurocomputing

CSC is a co-operating departement between the
Royal Institute of Technology and Stockholm University

Mikael Djurfeldt, Christopher Johansson, Örjan Ekeberg,
Martin Rehn, Mikael Lundqvist, and, Anders Lansner
Massively parallel simulation of
brain-scale neuronal network models

Report number: TRITA-NA-P0513, CBN
Publication date: December 2005
E-mail of author: djurfeldt@nada.kth.se

Reports can be ordered from:

Computational Biology and Neurocomputing
School of Computer Science and Communication (CSC)
Royal Institute of Technology (KTH)
SE-100 44 Stockholm
SWEDEN

telefax: +46 8 790 09 30
http://www.csc.kth.se/

Massively parallel simulation of brain-scale neuronal network models

Mikael Djurfeldt, Christopher Johansson, Örjan Ekeberg,
Martin Rehn, Mikael Lundqvist, and, Anders Lansner
KTH – School of Computer Science and Communication
and
Stockholm University
SE-100 44 Stockholm
SWEDEN

December 23, 2005

Abstract

Biologically detailed computational models of large-scale neuronal networks have now become feasible due to the development of increasingly powerful massively parallel supercomputers. We report here about the methodology involved in simulation of very large neuronal networks. Using conductance-based multicompartmental model neurons based on Hodgkin-Huxley formalism, we simulate a neuronal network model of layers II/III of the neocortex. These simulations, the largest of this type ever performed, were made on the Blue Gene/L supercomputer and comprised up to 8 million neurons and 4 billion synapses. Such model sizes correspond to the cortex of a small mammal. After a series of optimization steps, performance measurements show linear scaling behavior both on the Blue Gene/L supercomputer and on a more conventional cluster computer. Results from the simulation of a model based on more abstract formalism, and of considerably larger size, also shows linear scaling behavior on both computer architectures.

Contents

1 Introduction

Biologically detailed computational models of large-scale neuronal networks have now become feasible due to the development of increasingly powerful massively parallel supercomputers. In this report, we describe the methodology involved in simulations of a neuronal network model of layers II/III of the neocortex. The model uses Hodgkin-Huxley-style formalism and comprises millions of conductance-based multi-compartmental neurons and billions of synapses. Such model sizes correspond to the cortex of a small mammal. We also describe techniques involved in simulating even larger cortex-like models based on a more abstract formalism.

After some background and discussion of concepts relevant to the methodology of large and full scale neuronal network simulation, section 2 describes the kind of supercomputer architectures we have used for our simulations, while section 3 shows how we have adapted our software tool to these computers and to large problem sizes. Section 4 briefly describes the scalable cortical attractor memory network model used in our performance tests and shows examples of scaling results obtained. Section 5 describes and shows results for the more abstract cortex-like model. The report is concluded with a discussion in section 6 and conlusions in section 7.

1.1 Computational modeling in neuroscience

Computational modeling has come to neuroscience to stay. As modeling techniques become increasingly integrated with experimental neuroscience research, we will see more knowledge extracted from existing experimental data. Quantitative models help to explain experimental observations and seemingly unrelated phenomena. They assist in generating experimentally testable hypotheses and in selecting informative experiments.

Super computers are becoming evermore powerful. In this report we show that it is possible to simulate a substantial part of a small mammal's neocortex with fairly biologically detailed compartmental neuron models. This type of models are essential when linking cognitive functions to their underlying neurophysiological correlates. They are becoming an important aid in achieving a better understanding of normal function as well as psychiatric and neurological disorders.

1.2 Could computer science learn from neuroscience?

The nature of the neocortex is inherently parallel, since it is based upon a large number of small computing elements. Furthermore, neural systems, and neocortex in particular, are good examples of very advanced and robust, massively parallel, systems that perform much better on a great number of tasks than today's algorithms. This suggests that models of neural systems, especially their more abstract versions, are interesting to study also from a computer science perspective.

1.3 Level of abstraction

When modeling any physical system, there is a choice of which state variables and which parameters to include. In this sense, a model can focus on particular aspects of reality and model those aspects with a certain level of detail. For example, should we model the currents through individual species of ion channels in the cell membrane, or is a more abstract model, where we only consider the average state of activity in the cell, sufficient? The choice is mainly governed by the type of questions we want to be able to address.

In computational neuroscience, the term "biologically detailed" typically refers to models which employ conductance based multi-compartmental model neurons with a number of different ion channel types represented. Abstract network models, on the other hand, are often based on connectionist type graded output units. In this case, a single unit in the model may even represent multiple neurons, like a cluster of cells in a cortical minicolumn.

When we consider the methodological problems involved in simulating very large networks, especially problems of parallelization, there are many similarities between the detailed and abstract modeling approaches. In this report, we therefore study both types of models and characterize some of the options for designing simulators with a good scaling performance on cluster computers with a large number of nodes.

1.4 Why do we need large-scale models?

Real vertebrate neuronal networks typically comprise millions of neurons and billions of synaptic connections. They have a complex and intricate structure including a modular and layered layout at several levels, e.g., cortical minicolumns, macrocolumns, and areas. It can be useful to model a single module or microcircuit in isolation, for example in relation to a

correspondingly reduced experimental *in vitro* preparation such as a cortical slice. But such a module is, by definition, a component in a much larger network. In a real nervous system it is embedded in a mosaic of similar modules and receives afferent input from multiple other sources. We may therefore need to consider an entire network of modules.

However, computational studies have, partly due to limited computing resources, often focused on the cellular and small network level, e.g., the emergence and dynamics of local receptive fields in the primary visual cortex. Modeling at the network level poses specific challenges, but it is clear that it is inevitable to take global and dynamic network interactions into account in order to understand the functioning of a neuronal system.

Natural dynamics

It is indeed hopeless to understand a global brain network from modeling a local network of some hundred cells, like a cortical minicolumn, or from dramatically sub-sampling the global network by letting one model neuron represent an entire cortical column or area. One reason is that using small or sub-sampled network models lead to unnatural dynamics.

The network model is comprised of model neurons tuned after their real counterparts. These model neurons need sufficient synaptic input current to become activated. In a small network, model neurons are bound to have very few presynaptic neurons. Thus, either it is necessary to exaggerate connection probabilities, or, synaptic conductances—most of the time both.

This results in a network with few and strong signals circulating, in stark contrast to the real cortical network, where many weak signals interact. Such differences tend to significantly distort the network dynamics. For example, artificial synchronization can easily arise, which is a problem especially since synchronization is one of the more important phenomena one might want to study. By modeling the full network with a one-to-one correspondence between real and model neurons such problems are avoided.

Computo-pharmacology

In biologically detailed large-scale neuronal network models, the network effects of pharmacological manipulations can be studied in great detail. Does the addition of the drug alter the firing patterns of a particular neuron? What happens to the activity of a large ensemble of neurons? It might even be possible to study functional properties, such as memory function, and the enhancement of these. Other types of manipulations are possible, such as the addition or removal of different cell types, or, the rearrangement of connectivity patterns.

From ion channels to phenomenology

Large-scale network models will help to bridge the gap between the neuronal and synaptic phenomena and the phenomenology of global brain states and dynamics as observed in extracellular recordings, LFP, EEG, MEG, voltage-sensitive dyes, BOLD etc. But large-scale models demand tremendous amounts of computing power.

In summary, computational neuroscience will benefit greatly from the current development of new affordable massively parallel computers, likely to enter the market in the next few years. This development constitutes an enabling technology for the modeling of complex and large-scale neuronal networks.

1.5 Parallel neuronal network simulators

The development of parallel simulation in computational neuroscience has been relatively slow. Today there are a few publicly available simulators, but they are far from as general, flexible, and documented as commonly used simulators like Neuron (Hines and Carnevale, 1997) and Genesis (Bower and Beeman, 1998). For Genesis there is PGenesis and the development of parallel version of Neuron has started. In addition there exists simulators like NCS (Frye, 2005), NEST (Morrison et al., 2005), and our own parallelizing simulator SPLIT (Hammarlund and Ekeberg, 1998). However, they are in many ways still on the experimental and developmental stage.

In the following we will describe a parallel simulation study using the SPLIT simulator and one using a more recently developed special purpose simulator. But we start with a description of the computer architectures on which these simulators have been developed.

2 Cluster computers

The simulations in this report were mainly performed on two supercomputer installations: 1. The Dell Xeon cluster *Lenngren* at PDC, KTH, Stockholm—a rather conventional cluster computer containing 442 dual processor nodes. 2. The Blue Gene/L supercomputer (hereafter denoted *BG/L*) at the Deep Computing Capacity on Demand Center, IBM, Rochester, USA.

BG/L (Gara et al., 2005) represents a new breed of cluster computers where the number of processors, instead of the computational performance of individual processors, is the key to higher performance. The power consumption of a processor, and, thus, the heat generated, increases quadratically with clock frequency. By using a lower frequency, the amount of heat generated decreases dramatically. Therefore, CPU chips can be mounted more densely and need less cooling equipment. A system built this way can be made to consume less material and less power for an equal computational performance.

While a BG/L system can contain up to 65536 processing nodes, with 2 processors per node, the largest simulations described in this report were performed on one BG/L rack containing 1024 nodes. Our simulations on Lenngren ran on a maximum of 256 nodes. The theoretical peak performance of a processor core in BG/L, running at 700 MHz, is 2.8 Gflops/s. This presumes using special compiler support for the double floating point unit in the PowerPC 440 core. Since we chose not to explore that in this project, we should instead count on a theoretical peak performance of 1.4 Gflops/s. This is to be compared to 6.8 Gflops/s for one processor in the Lenngren cluster. A node in the BG/L cluster has 512 MiB of memory compared to 8 GiB for a node in the Lenngren cluster. A set of BG/L nodes can be configured to run in *co-processor mode*, where one of the two processors in a node run application code in most of the available memory ($\approx$ 501 MiB) while the other processor manages communication, and *virtual node mode*, where both processors run application code with half of the memory ($\approx$ 249 MiB) allocated to each. Thus, for our purposes, a Lenngren node has 4.9 times more computing power than a BG/L node in virtual node mode and 16 times more memory.

When running our code, we did not see any significant difference in performance between a set of nodes running in co-processor mode and half of that set running in virtual node mode. We therefore only used co-processor mode in cases where we needed more memory per node.

BG/L has three different inter node communication networks: a 3D torus network for point-to-point communication, a tree structured network for collective communication, and a barrier network. The nodes in the Lenngren cluster are connected with Infiniband which is a two-layered switch fabric.

3 The SPLIT simulator

3.1 Original design

The SPLIT simulator (Hammarlund and Ekeberg, 1998) was developed in the mid 90's with the aim to explore how to efficiently use the resources of various parallel computer architectures for neural simulations. Since different computer architectures benefit from different kinds of optimizations, the code uses programming techniques which encapsulate such optimizations and hardware specific features from the rest of the program, and the neural network model specification in particular. SPLIT has also served as a platform for experiments with communication algorithms.

SPLIT comes in the form of a C++ library which is linked into the user program. The SPLIT API is provided by an object of the class `split` which is the only means of communicating with the library. The user program specifies the model using method calls on the `split` object. The user program is serial, and can be linked with a serial or parallel version of the library. Parallelism is thus completely hidden from the user. In the parallel case, the serial user program runs in a master process which communicates, through mechanisms internal to the SPLIT library, with a set of slave processes. On clusters, SPLIT uses PVM or MPI.

The library exploits data locality for better cache-based performance. In order to gain performance on vector architectures, state variables are stored as sequences. It uses techniques such as adjacency lists for compact representation of projections and AER (Address Event Representation; Bailey and Hammerstrom (1988)) for spike events.

Perhaps the most interesting concept in SPLIT is its asynchronous design: On a parallel architecture, each slave process has its own simulation clock which runs asynchronously with other slaves. Any pair of slaves only need to communicate at intervals determined by the smallest axonal delay in connections crossing from one slave to the other.

The cells in a neural model can be distributed arbitrarily over the set of slaves. This gives great freedom in optimizing communication so that densely connected neurons reside on the same CPU and so that axonal delays between pairs of slaves are maximized. The asynchronous design, where a slave process does not need to communicate with every other slave at each time step, then gives two benefits: 1. By communicating more seldom, the communication overhead is reduced. 2. By allowing slave processes to run out of phase, the amount of time waiting for communication is decreased.

3.2 Optimizing SPLIT for cluster computers

The recent work on large-scale models, described in this report, has exposed a set of bottlenecks in the original SPLIT library. This is not unexpected, since current model sizes were quite unthinkable at the time when the SPLIT library was conceived. In addition a set of bugs has been discovered and fixed. We report below about the major changes to the code.

3.2.1 Stricter adherence to the MPI standard

The SPLIT library makes extensive use of the point-to-point communication operations `MPI_Send` and `MPI_Recv`. On the architectures on which the original SPLIT library was developed, smaller messages were buffered and the `MPI_Send` call did not block. The original author used this implementation dependent assumption in a few instances in the code. A typical case is one where a many-to-many communication is performed by first posting all sends and then all receives. On several modern architectures, this assumption does not hold for message sizes used in SPLIT. Therefore, and in order to remove uncertainties regarding code portability, we have changed the code in all instances so that it strictly adheres to the MPI standard. Thus, `MPI_Send` is not assumed to return before a corresponding `MPI_Recv` has been posted. In the following case, where the use of `MPI_Alltoall` was, for reasons having to do with code and class structure, precluded, this meant implementing the communication scheme shown in Figure 1.

If the communication network is not overloaded the new scheme still has time complexity $O(p)$, where p is the number of processes. While there certainly are schemes with better time complexity (Tam and Wang, 2000; Gross and Yellen, 1998), the simplicity of the chosen one is better from the perspective of code maintainability, but may become problematic for setup time when the number of processes approaches hundreds of thousands.

```
// Original code:

for (i = 0; i < n_processes; ++i)
  // send to process i

for (i = 0; i < n_processes; ++i)
  // receive from process i


// New code:

for (i = 0; i < local_rank; ++i)
  // receive from process i

for (i = local_rank; i < n_processes; ++i)
  // send to process i

for (i = 0; i < local_rank; ++i)
  // send to process i

for (i = local_rank; i < n_processes; ++i)
  // receive from process i
```

Figure 1: Old and new version of simple communication scheme for all-to-all communication during setup.

3.2.2 Identifying scaling problems—memory and time complexity $O(n_s)$

In the original design, synapses are created and their parameters set from the master process. Associated with this is a set of data structures with size proportional to the number of synapses, n_s, giving memory complexity $O(n_s)$. In models where n_s approaches hundreds of millions, the required amount of memory comes close to what is available on a single processing node on many architectures—we have a memory scaling problem. When models include billions of synapses, it becomes necessary to remove all such data structures. This includes data structures for storage of synaptic parameters before distribution to the slaves and various kinds of transformation tables used to quickly map distributed vector entities, for example synaptic conductance, from index on the master node, *global index*, to the corresponding process rank and index within that process, *local index* (see discussion below).

Remember also that, since the original SPLIT library confines all model specification to the master process, model specification is serial. We therefore also have a scaling problem in the time domain since the time complexity for synapse setup becomes $O(n_s)$. This was clearly noticeable in our simulations where synapse setup time soon began to dominate setup time for larger models.

3.2.3 Distributing creation of synapses and setting of their parameters to slave processes

An obvious solution to the memory and time complexity problems described above is to distribute the creation of synapses and setting of their parameters

to the slave processes, since setup code would gain speedup from parallelization and data structures proportional to the number of synapses only would need to cover synapses relevant to the neurons represented on a particular slave. This means that we need a conceptually simple way to run user code in the slave processes. Below, we will describe a first attempt at supporting this. Then we will describe a way to shield the user code from issues of parallelization which arise due to this change. In our tests, these changes reduced model setup time from hours to minutes for the larger models. Most importantly, it changed the time complexity of setup time from $O(n_s)$ to $O(n_s/p)$ where n_s is the number of synapses and p the number of slave processes.

3.2.4 The call-back mechanism—a conceptually simple way to parallelize user code

In the original design, the user program, running in the master process, specifies all parameters and initiates all stages of simulation through predefined functions in the SPLIT API. In the new design, a *call-back mechanism* introduces the possibility of executing arbitrary code at the slaves, by allowing the user to define functions which can be called in all slave processes at the command of the user program. Using such functions, *callbacks*, the user can freely mix serial and parallel sections in the main program.

In order not to deal directly with slave processes, a neural network model is conceived as consisting of a number of parts, each governing the update of the state variables of a disjoint subset of neurons in the model. Such a part is represented as a *simulation unit*, which, in the parallel case, is handled by one slave process.

A simulation unit object, `split_unit`, provides a subset of the SPLIT API normally provided by the `split` object. A callback receives a pointer to the relevant `split_unit` object as argument. The user program can also supply an arbitrary amount of data in the callback call, which is then distributed by the call-back mechanism to all slaves.

3.2.5 Hiding parallelism—the connection set algebra

According to SPLIT's design, parallelism should be hidden from the user, both conceptually and practically. A user program is able to arbitrarily link with either a serial or parallel version of the SPLIT library. How can we, when synapse setup is distributed, maintain a clean separation between issues of the model itself and issues of parallelism so that the user code only needs to be concerned with the model? In particular, we don't want the user code to deal with how cells in the model are distributed over the set of slave processes.

The solution to this problem involves the abstraction *connection set* and, the simulation unit abstraction and call-back mechanism described above. The synaptic connectivity is defined by the user in a callback, in a way which hides parallelism by using expressions from a *connection set algebra*. Since this is a novel construction, it will now be described in some detail.

A connection set object in essence represents an infinite connection matrix. In practice, it is used to represent a type of connection structure. The connection set object obtains the sets of pre- and postsynaptic neurons it governs through the simulation unit object. Conceptually, this acts as cutting out the relevant piece of the infinitely large connection matrix. Subsequently, the connection set object can work as an iterator over existing connections and can be used repeatedly in the user program to create synapses and set their parameters.

A basic set of connection sets are provided with the SPLIT library. This includes full connectivity, diagonal structure, uniformly random connectivity (existence of a connection determined by a fixed probability), and, Gaussian structure. The user can implement his own connection structures by inheritance, but a much simpler and more expressive method is provided in the connection set algebra which can be conceived as a set of operators on infinite connection matrices. As a simple example, consider the case where we want a 70% probability that a neuron connects to any other neuron in a population, but no connection to itself. This can now be expressed as:

```
random_uniform (0.7) - diag ()
```

where `random_uniform` and `diag` are connection set objects.

With the connection set abstraction, the user can focus on the connectivity structure itself while details such as which subset of neurons the code is working with is completely hidden. Note also how the simulation unit abstraction and call-back mechanism aid in hiding parallelism—the user does not need to deal with how many simulation units there are, or how neurons are distributed between them. These novel mechanisms have low overhead and are cache efficient since they do not depend on large tables in main memory.

3.2.6 The cell_map abstraction

To specify synapses, communicate spike events, etc., neurons need to be identified. An axon should connect a certain presynaptic neuron to a certain postsynaptic neuron. A spike is destined for a certain neuron. As mentioned above, SPLIT uses global and local indexes to refer to neurons. This design was influenced by two factors: 1. The best way to identify a neuron is task dependent. When specifying synapses, we most likely want to refer to neurons as members of a population in the model being simulated. However, when communicating spikes, it is more practical to refer to neurons as members of population fragments located in particular slave processes. Thus, we can reduce complexity if we are allowed to use multiple ways to address neurons. 2. SPLIT is designed to allow for arbitrary mappings of neurons to slave processes. Thus, we need a granularity at the neuron level when determining to which slave a neuron belongs. The global index of a neuron is used to locate it within the context of the entire population while its local index locates it within a particular slave. Using global indexes, we can avoid mixing the complexity of the synaptic connection structure with the complexity of the mapping of the model onto the slaves.

As indicated above, in the original code, the master process is responsible for transforming between global and local indexes. When synapse setup is distributed, the slaves need to be made aware of how the cells in the model are distributed over the slave processes. To this end, we have devised an abstraction, `cell_map`, which handles index transformations both in the direction from global to local index and the reverse. During setup, when the user program supplies the mapping from neurons to slaves, the `cell_map` object takes this information and distributes itself to the slaves. In each slave, the local `cell_map` object then builds the tables it needs for efficient index transformations. The ultimate goal is that the `cell_map` object will handle all functions associated with the mapping from neurons to slaves. This has not yet been fully implemented.

3.3 Optimizing SPLIT for Blue Gene/L

In this section, we will describe our experiences with the BG/L architecture and the changes we made to SPLIT in order to achieve good scaling behavior.

Before running on the BG/L architecture our experience was based on running the updated version of SPLIT, described above, on cluster computers at PDC, KTH. Although scaling results on the Lenngren cluster were slightly unpredictable, likely due to the communication network being shared with other applications in a queue system, they appeared good in that setup time seemed approximately constant for runs on different number of nodes, while speedup for simulation time was roughly linear.

BG/L represented a challenge in the sense that we now had to start imagining simulations using tens of thousands of processors while our previous practical experience was limited to less than 200. The other main consideration was the smaller amount of memory available to each processor.

3.3.1 Porting SPLIT to Blue Gene/L

The port of SPLIT to BG/L was trivial. It basically consisted of finding the correct arguments for the GNU configure script (compiler flags, library locations etc). This is noteworthy, considering that the BG/L is a novel computer architecture, and has two major reasons: 1. The software stack on the BG/L nodes is based on GNU software, for example GLIBC, and provides a Linux-like run-time environment. 2. The BG/L specific inter node communication hardware is abstracted using the MPI API.

3.3.2 A scaling problem in model setup

The first scaling results for a model with 330000 neurons and 161 million synapses showed that setup time scaled linearly with the number of processors used on BG/L (see upper trace in Figure 2). This was unexpected, since our results on Lenngren so far indicated a constant setup time. We then performed runs on Lenngren with up to 384 processors and could indeed confirm a linear dependence on the number of MPI processes (Figure 2, third trace). Because SPLIT slave processes run asynchronously, it was difficult to pin down exactly which code region was responsible, but initial debugging indicated that the setup of cell parameters was the culprit. Since this part of model setup is computationally intensive and involves much communication between master and slaves, and was already identified as a scaling problem of the future due to its time complexity $O(n_c)$, it was natural to distribute cell setup as we had previously done with synapse setup. This resulted in the second trace in Figure 2. The linear scaling still remained, however.

3.3.3 Adding a barrier call to the SPLIT API

In order to isolate the problem, we added a barrier call to the SPLIT API which synchronize processes using `MPI_Barrier`:

```
split->barrier ()
```

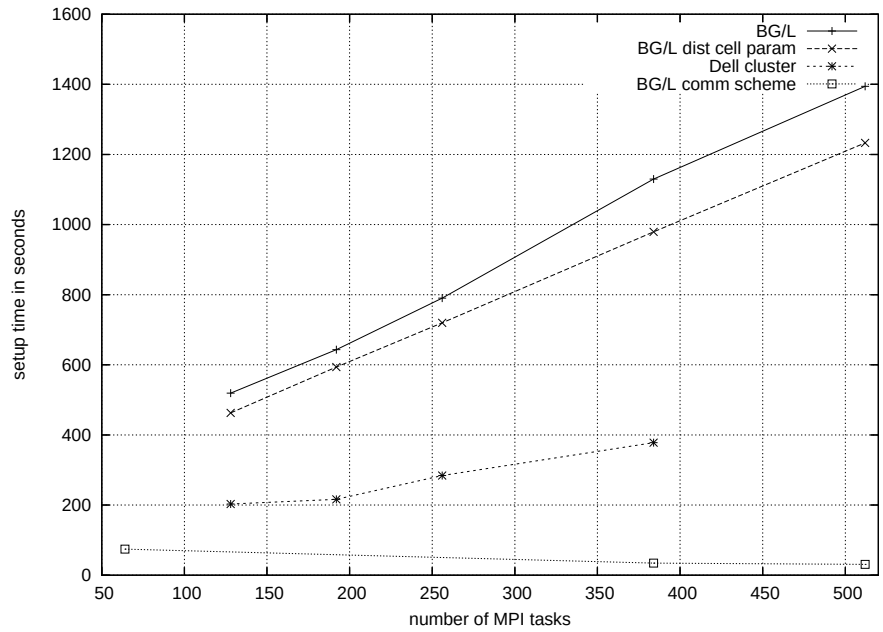

Figure 2: Scaling of setup time for a model with 330000 neurons. Upper trace: Initial result on BG/L. 2nd trace: Scaling after distributing setup of cell parameters to the slave processes. 3rd trace: Initial result on Dell cluster. 4th trace: Final result, after correcting problem in communication scheme implementation. Setup time now decreases with increasing number of processors.

The implementation involved extending the internal communication protocols in SPLIT, but the effort was well motivated by its utility for debugging. By using the barrier call to synchronize processes between each call to the SPLIT API, the scaling problem could be located to one particular member function (`map`). This made us discover that the implementation of the communication scheme for connecting spike communication objects, a simplified version of which is shown in Figure 1, did not work as intended, but introduced unintended dependencies between sender-receiver pairs. After correcting this, we obtained the scaling scaling curve shown in the fourth trace in Figure 2.

3.3.4 A scaling problem during simulation

The upper trace in Figure 3 shows timings for one second of simulated time in a model with 330000 neurons and 161 million synapses. Note that the addition of more processors does not lead to decreased simulation time for simulations on more than 767 processors. In order to pin down this problem, we linked SPLIT with an MPI trace library provided by IBM. In addition to giving detailed logs of communication events, this library provides various kinds of statistics of communication during a simulation. In a histogram over sizes of communication packets, and the blocking time associated with those messages, we could observe that packets of size 80 bytes were over-represented in number and responsible for a large fraction of simulation time. With this clue, we could identify the cause of the scaling problem: The normal state of asynchronous operation of the slaves was repeatedly interrupted by synchronization caused by the code which collects membrane potential data for logging to file at the master process. By decreasing the frequency of communication of such logging data, we could verify that this was indeed the problem (lower trace in Figure 3).

3.3.5 Removing data structures with memory complexity $O(n_c)$

With the above scaling problems resolved, we could now run models consisting of up to 4 million neurons. At this point, we encountered a *memory* scaling problem. The SPLIT library still contained many data structures in the master process, proportional to the number of neurons, n_c. This presented a barrier for the absolute size of model we could handle. Some of these structures were related to index transforma-

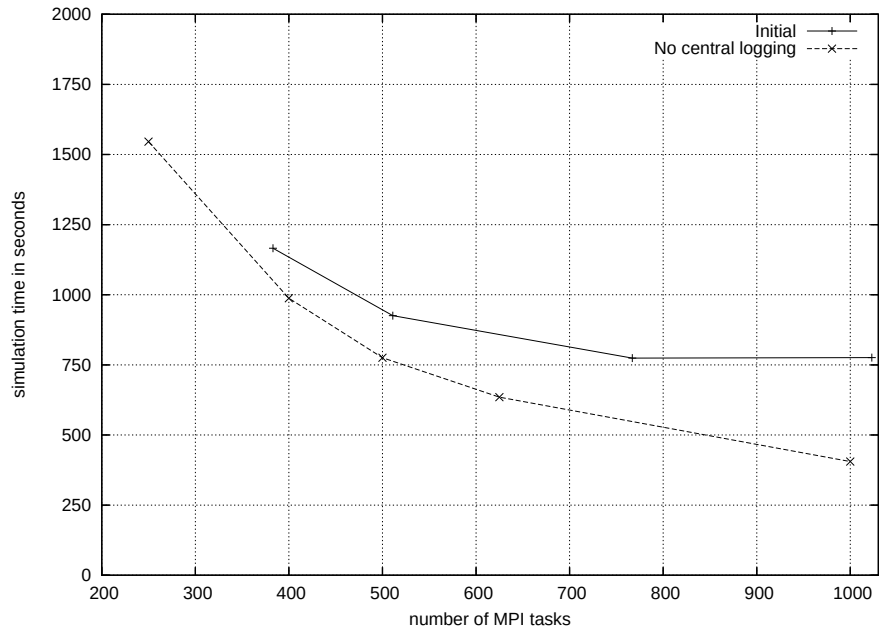

Figure 3: Scaling of simulation time for a model with 330000 neurons. Upper trace: Initial result. No speedup above 767 MPI tasks. Lower trace: Final result, after decreasing frequency of communication associated with central logging of membrane potentials.

tions, as described above, and could be eliminated, but this was only part of the problem, as we will see in section 3.3.6.

3.3.6 Implementation of a new protocol for specification of data logging

A thorough review of data structures in the master process with regard to memory complexity showed that the largest amount of memory was consumed by the mechanisms for specifying which model state variables should be logged to file. The reason was that this mechanism operated at the level of individual state variables. We therefore implemented a novel mechanism where logged variables could be specified at the level of *sets* of variables. This change, which included extensions to the internal communication protocol in SPLIT, together with the changes described in the previous section, substantially improved memory scaling. Without further changes, the library is now capable of handling models consisting of tenths of millions of cells and tenths of billions of synapses.

3.3.7 Implementation of a distributed data logging mechanism for binary data

Since we are interested in synthesizing various measures of activity within the cortical sheet, like EEG, we need an efficient way to log large amounts of data to file. For example, one of our experiments, where we simulate 20 seconds of activity of a model with 82500 neurons, generates 10 billion data points.

There are three problems associated with this: 1. Since all previous mechanisms for data logging was based on the master process collecting data from the slaves, the concentration of data to the master process, and the serialization that this implies, would lead to overloading of the master, and time would be lost due to slave processes waiting for communication. 2. The high rate of data to be collected from the slave would lead to disturbance of the normal asynchronous state of the slaves, as described above. 3. The previous mechanisms for data logging used a text file format, which would be too voluminous.

A new data logging mechanism was therefore implemented, which writes binary data to the BG/L parallel file system, *at each slave*. Data can be logged for an arbitrary number of state variables for subsets of neurons and to files specified by the user program. File names are suffixed with the slave rank. We have written tools to locate and extract specific traces of data points from these files, and to convert them for further processing in tools like MatLab.

4 A scalable network model of the neocortex

The model described in this section is the result of an integration of functional constraints given by a theoretical view of the neocortex as an associative attractor memory network (a *top-down* strategy; see Churchland and Sejnowski (1992), ch. 1), and, empirical constraints given by cortical anatomy and physiology (*bottom-up*).

The view of the cortex as an attractor network has its origin more than fifty years back in Hebb´s theories of cell assemblies (see, e.g, Fuster (1995) for a review). It has been mathematically instantiated in the form of the Willshaw-Palm (Willshaw and Longuet-Higgins, 1970; Palm, 1982) and Little-Hopfield models (Hopfield, 1982) and has subsequently been elaborated on and analyzed in great detail (Amit, 1989; Hertz et al., 1991).

The olfactory cortex (Haberly and Bower, 1989) as well as the hippocampus CA3 field (Treves and Rolls, 1994) have previously been percieved and modeled as prototypical neuronal auto-associative attractor memory networks. More recently, sustained activity in an attractor memory of a similar kind has been proposed to underlie prefrontal working memory, although in this case the attractor state itself and not the connectivity matrix is assumed to hold the memory (Compte et al., 2000).

Our view of cortical associative memory has been expressed in the form of an abstract neural network model (Lansner et al., 2003; Sandberg et al., 2002, 2003). An implementation of this model is described in section 5 of this report. The computational units in the model correspond to the anatomical minicolumns, suggested by Mountcastle (1978) and described by Peters and Sethares (1991), and to the orientation minicolumns in the primary visual cortex.

In a previous series of papers we used the ideas from the abstract model in a biologically detailed model of a memory network architecture in layers II/III of the neocortex (Fransén and Lansner, 1995, 1998; Lansner and Fransén, 1992). We found that the model could perform critical cell assembly operations such as pattern retrieval, completion and rivalry.

However, the abstract framework also suggests modularity in terms of a hypercolumnar organization of a similar kind to that described by Hubel and Wiesel (Hubel and Wiesel, 1977) for the primary visual cortex, that is, bundles of about 100-200 minicolumns forming one hypercolumn. In the present model, we have continued the development of the biologically detailed model by adding a hypercolumnar structure. We have also added a second type of interneuron and model the hypercolumn in *full scale*, meaning that there is a one-to-one correspondence between model neurons and real neurons of the selected types in layers II/III of the minicolumn.

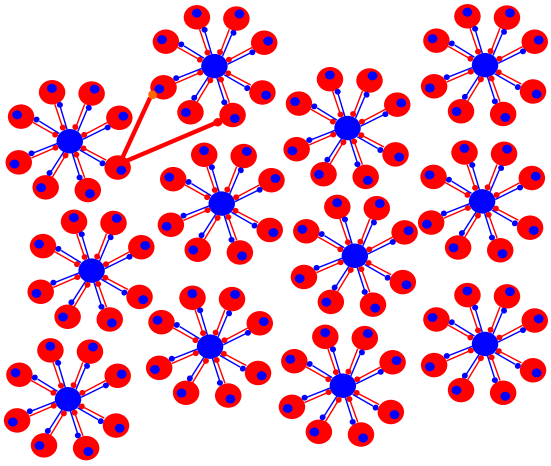

Figure 4: Cartoon of a network with 12 hypercolumns and 8 minicolumns each. Each hypercolumn has 8 circularly arranged minicolumns, each comprising 30 densely connected model pyramidal cells. The small disc in each minicolumn represents 2 RSNP cells that receive input from distant pyramidal cells and inhibit local ones. The large disc at the center of each hypercolumn represents a population of basket cells. The negative feedback circuitry between pyramidal and basket cells is indicated. An excitatory long-range minicolumn-to-minicolumn connection originates in pyramidal cells in the presynaptic minicolumn and targets pyramidal cells in the postsynaptic minicolumn. An inhibitory minicolumn-to-minicolumn connection originates in the same way but targets the RSNP cells in the postsynaptic minicolumn, which, in turn, provide a di-synaptic inhibition onto the pyramidal cells in the receiving minicolumn.

As in the abstract model, each hypercolumn operates like a soft winner-take-all module in which activity is normalized among the minicolumns, such that the sum of activity is kept approximately constant. We propose that the lateral inhibition mediated by basket cells may achieve this normalization in cortex. Normalization models of a similar kind have recently been proposed for the primary visual cortex (Blakeslee and McCourt, 2004).

Since our model is built using the connection set algebra tool described in section 3.2.5, it is easily scalable. That is, we can easily adjust structure parameters such as the number of minicolumns per hyper-

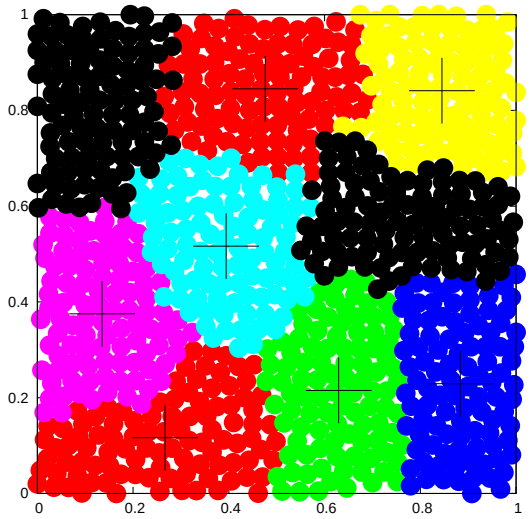

Figure 5: Spatial layout of minicolumns in a network with 9 hypercolumns (centers marked with a cross), each comprising 100 minicolumns. Each small disc represents one minicolumn. Minicolumns with the same color belong to the same hypercolumn.

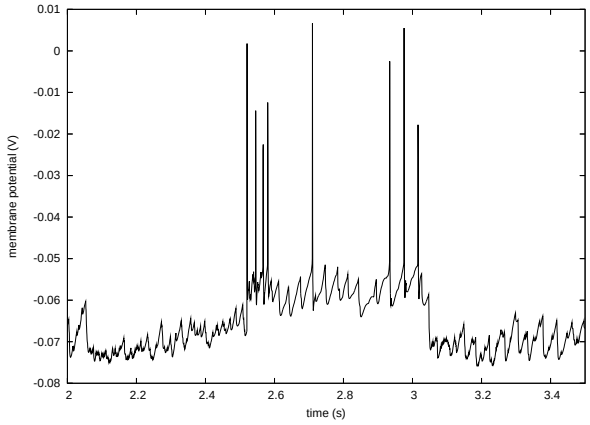

Figure 6: Model pyramidal neuron switching from DOWN to UP state and back. The mean membrane potential becomes elevated at $t = 2.5$ s due to network activity when the neuron starts participating in an active attractor state. The attractor state was activated through simulated electrical stimulation of other member neurons. (The varying height of action potentials is an artefact due to a large plotting time step.)

column, or the total number of hypercolumns in the network. Our model contains scaling equations for connection probabilities which ensure that different cell types see similar currents in an attractor state for different model size and structure. For natural structure parameters, connection probabilities correspond to empirical estimates of cortical connectivity (see Lundqvist et al. (2006) for details). Figure 4 shows, in an abstract fashion, the structure of a version of our network model consisting of 12 hypercolumns with 8 minicolumns each (all simulations in this report use 100 minicolumns per hypercolumn except where noted).

The cell models used in the simulations described here are conductance based and multi-compartmental of intermediate complexity . They are to a large extent similar to those developed previously (Fransén and Lansner, 1995, 1998). The cells included are layers II/III pyramidal cells and two different types of inhibitory interneurons, assumed to correspond to horizontally projecting basket cells and vertically projecting double bouquet and/or bipolar cells (Douglas and Martin, 2004; Kawaguchi and Kubota, 1993; Markram et al., 2004). Following Kawaguchi (1995) we will here refer to the latter type as regular spiking non-pyramidal or RSNP cells.

The pyramidal cells are modeled with six compartments and the inhibitory interneurons with three. Voltage-dependent ion channels for Na^+, K^+, Ca^{2+}, and Ca^{2+}-dependent K^+-channels are modeled using Hodgkin-Huxley formalism.

The excitatory synapses between pyramidal cells and between pyramidal and RSNP cells are of a mixed kainate/AMPA and NMDA type while the pyramidal to basket cell synapse only has a kainate/AMPA component and the inhibitory synapses are of $GABA_A$ type.

The network connectivity is set up to store a number of memory patterns (attractor states) such as would have resulted from long-term plasticity using a Hebbian learning rule (Sandberg et al., 2002). The synaptic plasticity actually included in the model is restricted to fast synaptic depression of pyramidal to pyramidal synapses.

Axonal delays are determined by using a geometric model of the cortical sheet grown using cellular automata. Figure 5 shows the spatial layout of minicolumns in a network with 9 hypercolumns. For further details see Lundqvist et al. (2006).

The raster plot in Figure 7 shows a typical example of network dynamics in our simulations. The network spends a few hundreds of milliseconds in each attractor state. For simplicity, the memory patterns stored in this network are orthogonal—every minicolumn only belongs to a single pattern. Since patterns compete within each hypercolumn, this gives rise to

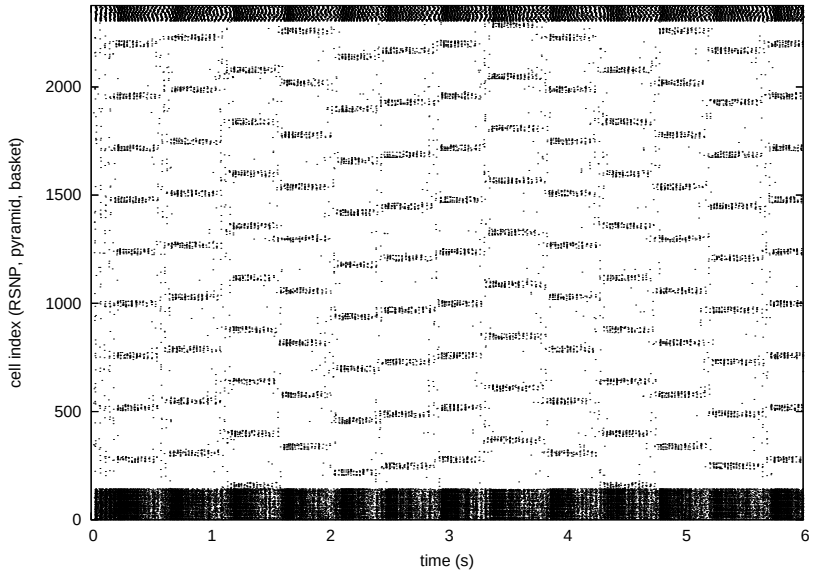

Figure 7: Raster plot for network with 9 hypercolumns and 8 minicolumns per hypercolumn. The lower portion of the raster shows activity in all RSNP cells, mid portion shows pyramidal cells, and, upper portion basket cells. The long-range pyramidal-pyramidal and pyramidal-RSNP synapses store orthogonal memories. The network can be seen jumping spontaneously between these memory states.

the structure of regular bands in the figure—only one minicolumn wins in each hypercolumn. Depending on the level of background drive, attractor states can either be attained spontaneously (as in the figure) or by afferent input to neuronal members of the memory pattern. The average spike frequency for pyramidal cells participating in an active memory state is around 6 Hz, compared to a base line firing rate of 0.2 Hz. Competition between attractors is resolved within 30-40 ms.

A striking feature of our model is that this dynamics, which corresponds well to data on *UP states* (see Figure 6) seen *in vivo* (Anderson et al., 2001) and in slice preparations (Cossart et al., 2003), is preserved over all model sizes and structures we have yet simulated. That is, even for the largest simulation described in this report, a global memory pattern can be correctly recalled within tens of milliseconds.

4.1 Results

When considering how to use super-computing power to boost performance of a neuronal network simulator, there are two main directions in which to trim the software: On one hand, we can use the increased capacity for computing to run *larger* models. On the other, we can use it to compute *faster*. In this work we have primarily focused on the first possibility. After the adaptations of our software tool SPLIT described in section 3, we obtained generally good scaling performance, particularly for large models where computation time at the nodes dominates overhead due to communication.

Figure 8 shows a scaling diagram for a version of the model described in section 4 containing 4 million cells and 2 billion synapses. Real pyramidal cells have in the order of 10000 synapses. The number of synapses in our model is lower than this due to the fact that we only consider connections within one layer of a local area, and, that the inter-minicolumn connectivity contains a small number of memory patterns due to their orthogonality. Note, though, that the number of *active* synapses is the same as in a network with a more natural connection matrix.

Setup and simulation wall-clock times for 1 second of simulated activity are shown for various number of slave processes. The model comprises 1225 hypercolumns with 100 minicolumns per hypercolumn, Note how, even for a model of this size, setup time decreases with increasing number of processes, in accordance with the earlier results described in section 3.3.3.

Note also that simulation time is roughly inversely proportional to the number of processes. This is shown more clearly in the speedup diagram in Figure 9, which, for practical reasons, is normalized rel-

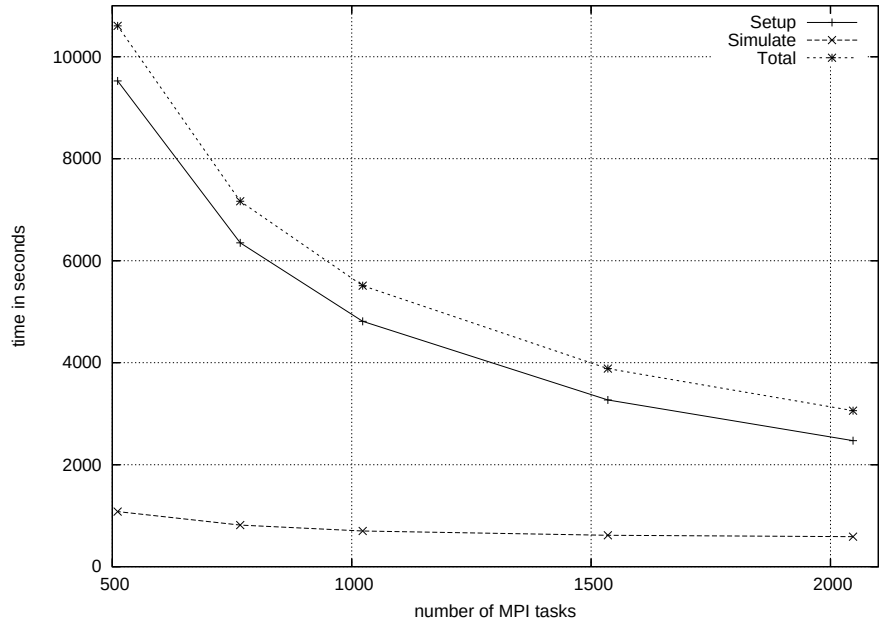

Figure 8: Scaling diagram for model with 4 million cells and 2 billion synapses. Wall-clock setup and simulation time for 1 sec of simulated activity.

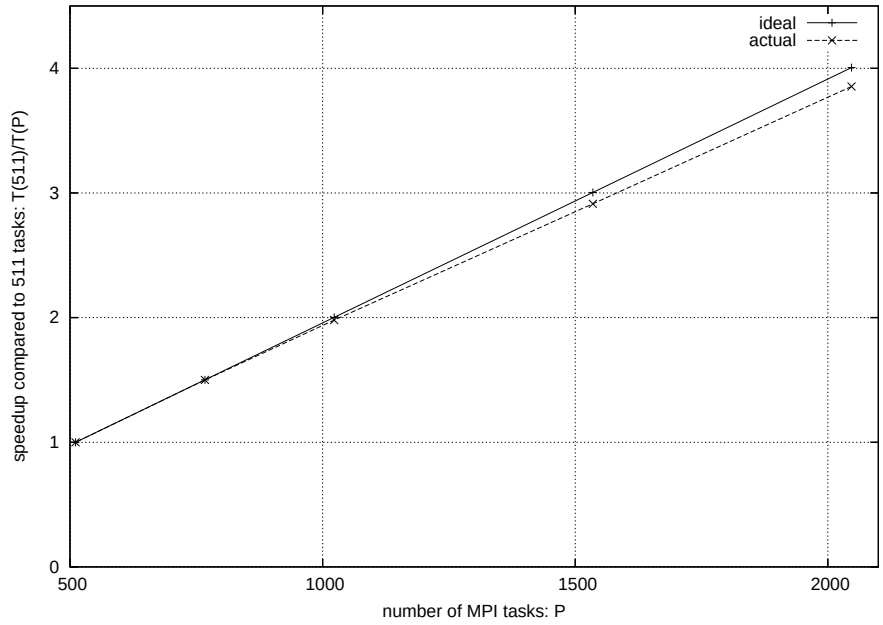

Figure 9: Speedup for model with 4 million cells and 2 billion synapses.

# procs.	Setup	Simulation
1024	2272	9489
2048	1582	4749

Table 1: Setup and simulation times in seconds for a model with 8 million cells and 4 billion synapses on 1024 and 2048 BG/L processors.

ative to the data point for 511 processes rather than 1. The lower trace shows that scaling is nearly linear.

The largest model we have simulated so far is a network consisting 2401 hypercolumns (see table 1 for setup and simulation times on 1024 and 2048 processors of one full BG/L rack). This model comprises 8 million neurons and 4 billion synapses.

Even for the largest models, the brain-like dynamics we have observed (see section 4 and Lundqvist et al. (2006)) in the network is preserved, although some phenomena show quantitative changes which we report on separately (Djurfeldt et al., 2006).

5 An abstract model of neocortex

Here we present a connectionist abstract model of cortex, and its implementation, which provides functional constraints for the biologically detailed model described in section 4. Both the computational and memory requirements for this model are smaller than for its more biologically detailed counterpart. Instances of this model that are a few orders of magnitude larger than the biologically detailed version can run in close to real time. Areas of interest for this type of connectionist model are to explore, test, and, verify theories on how the neocortex processes information in cases which are difficult to investigate with a computationally more demanding model.

As described in section 4, the abstract model (Johansson and Lansner, 2004a,b) on which the biologically detailed model is based, makes the assumption that a cortical minicolumn is the smallest functional unit in cortex. Each such minicolumn is modeled by a single unit in the abstract network. One hundred such units are grouped into a hypercolumn. The functional role of the hypercolumn is to normalize the activity in its constitutive units to one. The average neuron in the mammalian brain has approximately 8000 synapses, and a cortical minicolumn has on average 100 neurons (Johansson and Lansner, 2004b). With the assumption that five synapses are required to link two minicolumns together, the average minicolumn potentially has 120000 incoming connections (Johansson and Lansner, 2004a).

In this report we study networks of randomly connected units. This type of connectivity is very general and is often used in attractor neural networks. In turn, the dynamics of attractor neural networks are hypothesized to be a first and very rough approximation of the cortical dynamics (Rolls and Treves, 1998; Palm, 1982).

In the abstract model, the hypercolumn has a computational grain size that maps very well onto the computational resources in a cluster computer. The hypercolumn has a constant number of units and connections, and in turn constant computational requirements.

5.1 Bayesian Confidence Propagating Neural Network

The units and synapses of the abstract model are implemented with a Bayesian Confidence Propagating Neural Network (BCPNN) (Lansner and Ekeberg, 1989; Lansner and Holst, 1996; Sandberg et al., 2002). This type of neural network can be used to implement both feed-forward classifiers and attractor memory. In the latter case, it is similar to a Hopfield network, the main difference being the local structures imposed by the hypercolumns. It has a Hebbian type of learning-rule, which is local in both space and time (only the activity of the pre- and postsynaptic units at one particular moment are needed to update the weights) and therefore it can be efficiently parallelized. Further, the network can be used with both unary-coded activity (spiking activity) and real-valued activity, $o \in (0, 1)$. In the current implementation we use spiking activity that allows for efficient address event communication (AER) (Bailey and Hammerstrom, 1988; Mortara and Vittoz, 1994; Deiss et al., 1999; Mattia and Giudice, 2000). The network has N units grouped into H hypercolumns with U_h units in each. Here, h is the index of a particular hypercolumn and Q_h is the set of all units belonging to hypercolumn h.

The computations of the network can be divided into two parts; training (Figures 10–12) and retrieval (equations 1–3). In the training phase the weights, w_{ij}, and biases, β_j, are computed. In the retrieval phase the units' potential and activities, o_i, are updated. The training phase is presented for three different levels of computational complexity. Real-time performance for the computations is set to one update of both phases in 10 ms. Next, we first discuss the retrieval and then, in section 5.1.1, the training phase.

In the retrieval phase a process called relaxation

is run in which the potential and activity is updated. When using the network as an autoassociative memory the activity is initialized to a noisy or a partial version of one of the stored patterns. The relaxation process has two steps; first the potential, m, is updated (eq. 2) with the current support, s (eq. 1). Secondly, the new activity is computed from the potential by randomly setting a unit active according to the softmax distribution function in eq. 3. The parameters $\tau_m = 10$ and $G = 3$ are fixed throughout.

$$s_j = \log(\beta_j) + \sum_{h=1}^{H} \log\left(\sum_{k \in Q_h} w_{kj} o_k\right) \quad (1)$$

$$\tau_m \frac{dm_j}{dt} = s_j - m_j \quad (2)$$

$$o_j \leftarrow \frac{e^{Gm_j}}{\sum_{k \in Q_h} e^{Gm_k}} : j \in Q_h \quad (3)$$
$$\text{for each} \quad h = \{1, \ldots, H\}$$

5.1.1 Three implementations of the training phase

Here we present three different implementations of the learning phase, each with different computational complexity. Up to three different state variables are used to represent a unit; which are called Z-, E-, and P-trace variables. A connection can have up to two state variables; P- and E-trace. In the simplest implementation, with the lowest complexity, the units' P-trace variables are computed by leaky integrators directly from the input as in Figure 10. Here, S is the input and P is the P-trace state variable. This implementation uses delayed update of both unit and connection variables. Further, these trace variables are efficiently computed in the logarithmic domain (Johansson and Lansner, 2006a,b). The sparse connectivity is implemented with adjacency lists. We call this a P-type implementation. A connection need 10 bytes of memory; 4 bytes for storing P_{ij}, 4 bytes for the index of the presynaptic unit, and 2 bytes for the time stamp associated with the delayed update.

In a slightly more complex implementation, called Z-type, the units' state variables, Z and P, are computed with two leaky integrators as in Figure 11. Here, we only implement the state variables of the connections with delayed update. A connection in the Z-type network requires as much memory as in the P-type.

In most complex implementation, called E-type, two leaky integrators compute the state variables of the connections and three leaky integrators compute the state variables of the units as shown in Figure 12. Here, it is not possible to use delayed update for any

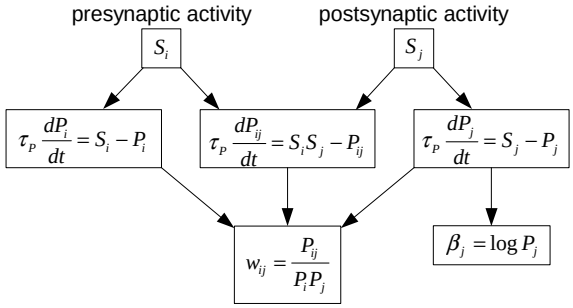

Figure 10: The minimal set of equations needed to implement an associative BCPNN, called P-type.

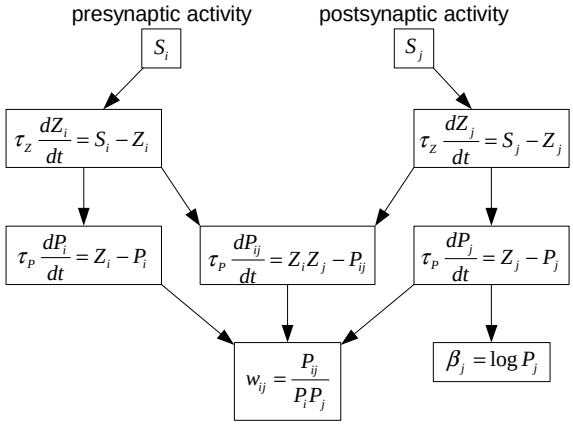

Figure 11: The equations used to compute the weights, w_{ij}, and biases, β_j, for the Z-type network.

of the state variables and hence the training time increases considerably. In each iteration during training all connections are updated. A connection need 12 bytes of memory; 4 bytes for storing P_{ij}, 4 bytes for storing E_{ij}, 4 bytes for the index of the presynaptic unit, and 2 bytes for the timestamp.

5.1.2 Implementation details

The abstract model was implemented with a focus on efficiency. Four techniques were used to achieve good performance; delayed updating of the units and connections, computation of the leaky integrators in the logarithmic domain, adjacency lists for effective indexing of the units in the sparse connectivity matrix, and AER for effective communication. We used the programming language C with calls to two standard routines in the MPI application programming interface; `MPI_Bcast()` and `MPI_Allgather()`.

Further, a fixed-point arithmetic implementation

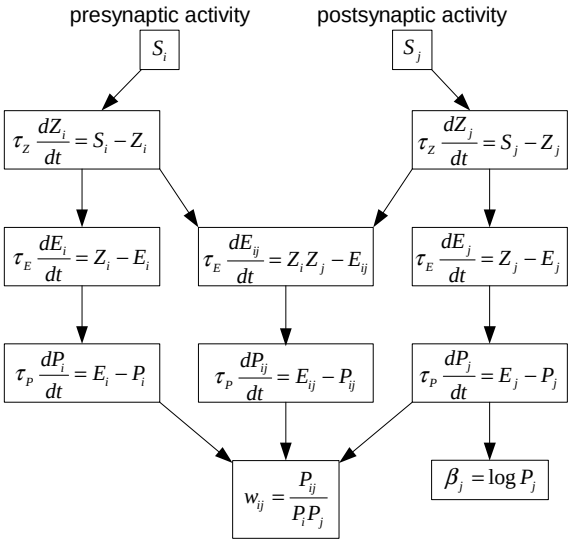

Figure 12: The equations of the network with the highest complexity, called E-type.

of the BCPNN has been developed. The target use for this implementation is in hardware implementations but it can also be advantageous to use in other applications because it reduces the memory requirements with 30%.

5.2 Results

Since a node on BG/L only has 1/16 of the memory of a node on Lenngren, the scaling experiments on BG/L were set up to use 16 times more nodes than corresponding experiments on Lenngren. This means that the programs filled the memory on every node on both clusters. Scaling was studied under the condition that the problem size was increased together with the number of processors, which is referred to as *scaled speedup*. If the program parallelizes perfectly the scaled speedup measure is constant with increasing number of processors.

We did not use the *speedup* measure where the running time of a problem of constant size is plotted as a function of the number of processors is increased, the reason being that the application studied has a relatively large and fixed part that runs on each node. When the problem size is scaled more than an order of magnitude, this part influences the running times for the smaller sized networks.

Neither did we plot the scaling of the parallel program relative to an optimal implementation on a single processor. This is because the problem is very memory intensive and it is not possible to run on a single processor. We need to run it on a cluster computer not only for the processing power but also for the available memory resources.

	Train	Retrieval	Total
P-type	0.0130	0.0196	0.0326
Z-type	0.0270	0.0825	0.109
E-type	2.29	0.0538	2.344

Table 2: The iteration times for three different network complexities on 128 processors on Lenngren. Time is measured in seconds.

	Train	Retrieval	Total
P-type	0.007	0.0184	0.0254
Z-type	0.018	0.0736	0.0916
E-type	0.718	0.0415	0.756

Table 3: The iteration times for three different network complexities on 2048 processors ($16 \cdot 128$) on BG/L. Time is measured in seconds.

The problem size was varied in two ways; Firstly, the number of units was increased, each having a constant number of connections. Secondly, the number of connections per unit was increased, holding the number of units constant. Here, we present scaling results for both types of scaling.

5.2.1 Running times for a mouse sized network

Here we run a network with 2048 hypercolumns and $1.2 \cdot 10^5$ connections per unit. This network is 30% larger than a mouse cortex equivalent network (Johansson and Lansner, 2004b,a).

Table 2 shows the iteration times on 128 processors on Lenngren. On each processor 16 hypercolumns were allocated. Running the Z-type network was 3.3 times slower and E-type network was 72 times slower than a P-type network.

Table 3 shows the iteration times on 2048 processors on BG/L. On each processor one hypercolumn was allocated. Running the Z-type network was 3.6 times slower and E-type network was 30 times slower than a P-type network.

The more complex units and connections were, the better the scaling became because more computations were done in each iteration. In the case of the most simplistic network, the computations needed to administrate the sparse connectivity and to compute the resident part of the network on each processor were noticeable.

The retrieval times for the Z-type network were

the longest because two logarithms are computed for each weight in the synaptic summation for each unit. In case of the P-type network no logarithm needs to be computed in the synaptic summation, and for the E-type one logarithm is computed for each weight in the synaptic summation.

5.2.2 Scaling of Hypercolumns

In Figure 13 the scaling, on Lenngren, of the network when the number of hypercolumns was increased is plotted. The number of connections per unit was fixed to $4 \cdot 10^4$. The retrieval times scaled linearly and with a factor smaller than the rate by which the number of processors was increased. The training times scaled linearly, but slightly faster, than the rate by which the number of processors was increased for the P- and Z-type networks. The Z-type network does not have delayed update of the units' state variables and, consequently, it has a poorer scaling than the P-type network. In case of the E-type network, the training times scaled no faster than the increase in processing power. This was because the training phase of the E-type network is much more computationally demanding than for the other two network types.

In Figure 14, the scaling on BG/L is showed for the same three networks but run on 16 times more processors. The general results were the same with one exception; the training times for the computationally intensive E-network increase slightly faster than the increase in processing power. The probable cause is that the execution time of the resident part of the problem was becoming proportionally larger since the problem is dived into a smaller computational grain size.

5.2.3 Scaling of Connections

In Figure 15 the scaling, on Lenngren, of a P-type network when the number of connections per unit was increased is plotted. The number of hypercolumns was fixed to 2048. Figure 16 shows the iteration times on BG/L, where 16 times more processors were used that on Lenngren. From these results we can conclude that neither the training nor the retrieval times scales faster than the increase in processing power when more connections are added.

5.2.4 A network model of record size

The largest simulation was a P-type network run on 256 nodes on Lenngren. This network had 1.6 million units and 200 billion connections. The iteration time of the training phase was 51 ms and for the retrieval

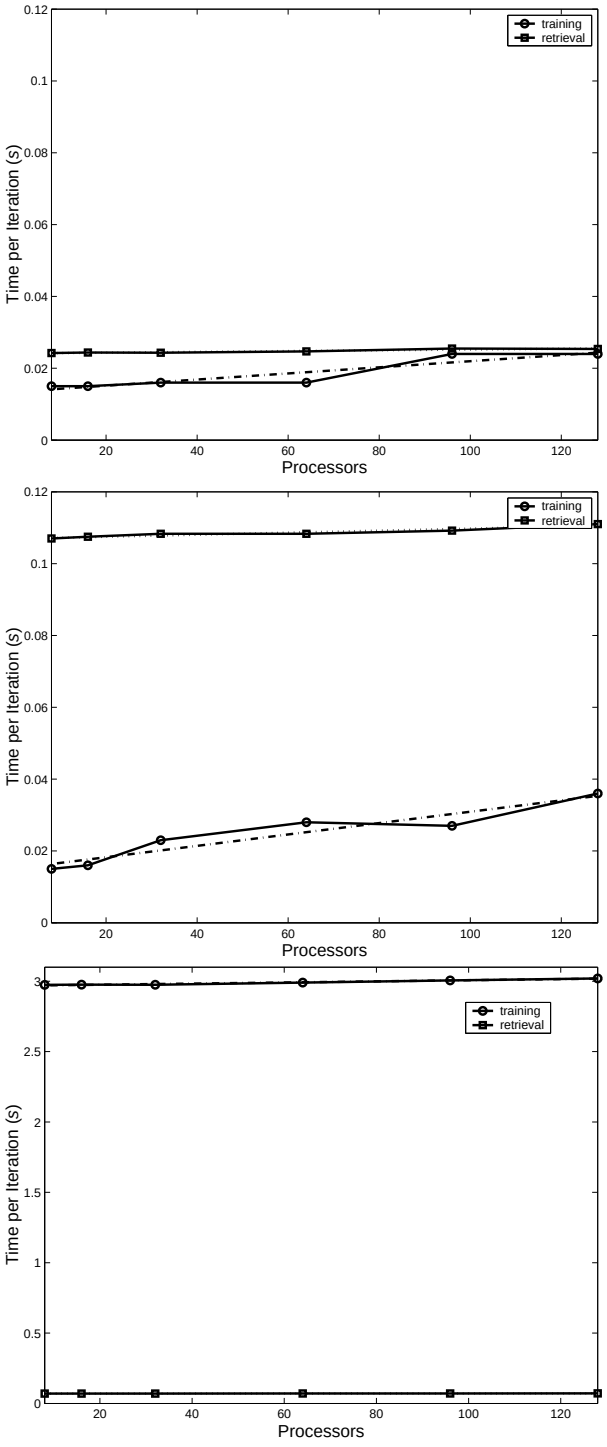

Figure 13: The iteration times on Lenngren for networks with 64 hypercolumns per processor. The top plot is for P-type, the middle for Z-type, and the lower plot is for E-type networks. The number of connections per unit was fixed to 40000.

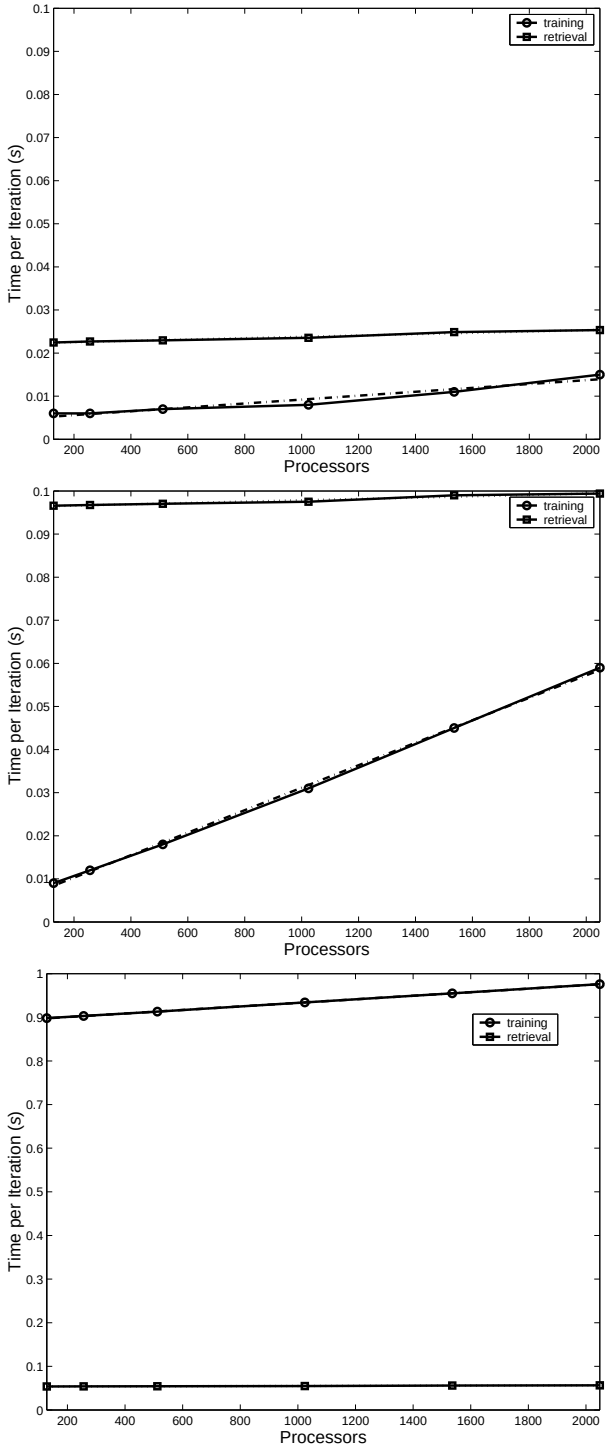

Figure 14: The iteration times on BG/L for networks with four hypercolumns per processor. The top plot is for P-type, the middle for Z-type, and the lower plot is for E-type networks. The number of connections per unit was fixed to 40000.

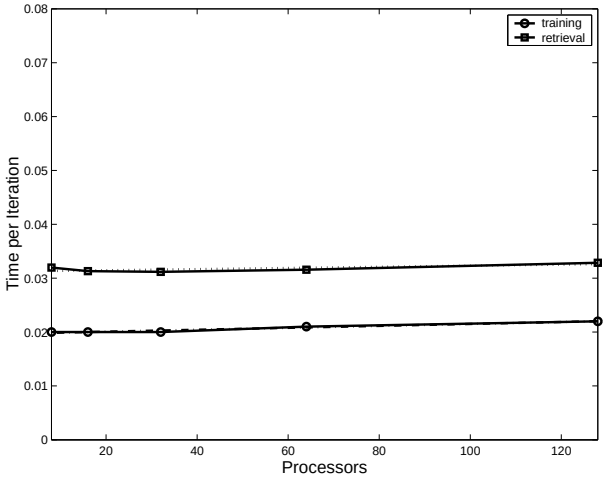

Figure 15: The iteration times on Lenngren for a P-type network with a constant number of 2048 hypercolumns.

phase it was 61 ms, which is close to real-time performance. The network was used for doing pattern completion and noise reduction on color images. A testimony of the code's efficiency comes from setting the unofficial record in power use on the Lenngren cluster. Lenngren actually used more power to run this network than it used when running the benchmark programs for the top-500 supercomputers list. The power usage peaked at around 120 kW, which meant that the average connection drew $6 \cdot 10^{-7}$ W. Translated into the biological equivalent this is roughly 10^{-8} W per synapse. The corresponding figure for the human brain is $3 \cdot 10^{-14}$ W per synapse given that the human body consumes energy at a rate of 150 W (Henry, 2005) and that 20% of this energy is used by the 10^{11} neurons and 10^{15} synapses of the brain (Kandel et al., 2000).

6 Discussion

A reason sometimes given for staying away from large scale neuronal network models is that there would be no point to build a biophysically detailed model of a brain scale neuronal network since the model would be as complex as the system it represents and equally hard to understand. This is a grave misconception, since, on the contrary, if we actually had an exact quantitative model of a human brain in the computer, which, of course, is impossible, it would be extremely beneficial. Obviously, since we would have full access to every nitty-gritty detail in this "brain" that would

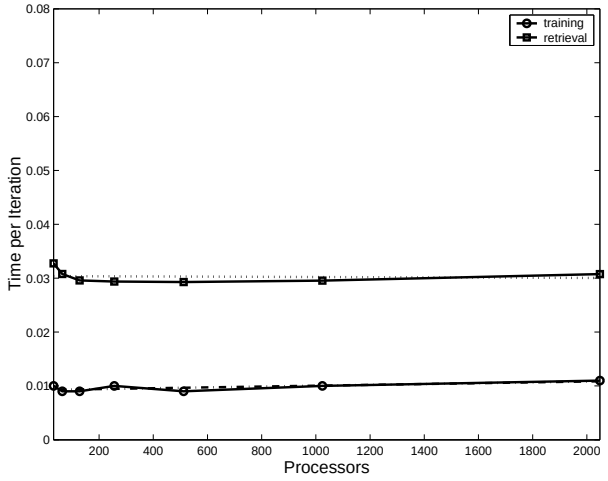

Figure 16: The iteration times on BG/L for a P-type network with a constant number of 2048 hypercolumns.

dramatically speed up progress in understanding of this extremely complex structure. Unfortunately, or perhaps fortunately, this is a complete utopia, and we have to bear with much simpler models in the foreseeable future.

6.1 Trade-off between numbers and complexity

The starting point for network modeling is the component cells and their synaptic interactions, including transmission, conduction delays, and, synaptic plasticity. The level of complexity of model neurons now becomes more of an issue. Given our limited computational resources, there is a trade-off between complexity of the cell models and the size of the network. It is a painful fact that a high-end PC of today gets unbearably slow as model network size grows beyond some hundred cells.

There is some light in the tunnel, however, since one reason for avoiding large-scale or full-scale simulations is now gradually disappearing. Although the reduction of physical dimensions of integrated circuits and increasing clock speed may be approaching their limits, the number of processors operating in parallel in computers is now starting to increase dramatically. Parallel computing is beginning to reach the consumer market resulting in a drop in prices. It is quite likely that we will have desktop boxes with hundreds of parallel processors within the next five years. Since neuronal networks represent computationally homogeneous computational structures, parallel simulation is relatively straightforward. Thus, in the near future, computer power will no longer prevent us from putting together and study large and even full-scale models of global brain networks.

6.2 Number of free parameters

A real and unavoidable, but disturbing, fact is that the number of free parameters increases as you go from the single cell model to a network with a number of cell types and different types of interactions between them. Even rather simplistic, conductance based multi-compartmental model neurons comprise tens of equations and hundreds of parameters. The number of synapses in a big network model are many more and they too require some equations and parameters. Thus, any brain-scale network model would contain on the order of billions of parameters. Even with advanced, automated, parameter search it would be hopeless to find a reasonably adequate combination of these massive amounts of parameters. This appears to put unreachable demands on the availability of experimental data to constrain the model. Fortunately, the situation is not that bad. Neuronal networks are typically described as comprising a quite limited number of cell types, with basically the same properties but some variation within the population. The parameter used for one neuron of a certain type is likely to do for the others as well, possibly with some compact description of a distribution around a mean. This holds for the synaptic interactions as well, though now we need to consider pairs of cell types. Moreover, synaptic conductances are not determined arbitrarily, but are presumably the result of the action of some kind of, so far unknown, learning rule coupling them to historical network activity.

Thus, the good news is that the number of truly free parameters is to some extent independent of the actual number of neurons and synapses used in a model. The same averages, distributions and learning rules hold for the huge network model as well as the tiny one. Knowledge about the distribution of cell and synaptic properties of course becomes important. On the other hand, the actual details of, e.g., dendritic arborization of one individual cell is less important than in single cell modeling, since this is a typical thing that varies within the population. What really increases with increasing numbers is the match between model and reality.

6.3 Cost of complex cell models in large scale simulations

Large-scale network models shift the balance so that synaptic complexity takes over as the limiting factor. Somewhat unintuitive, for really large networks it comes with little extra cost to have complex cell models! Contrary to the case for single cells and small networks, the solution methods used for, e.g., dendritic integration does not add significantly to the cost of computation, since synaptic computation dominates.

Furthermore, perhaps surprisingly, parallel neural simulation is often bound by local computation and not by inter-neuronal communication. A factor of major influence on communication speed is whether neuronal interactions can be represented by spiking events or if they are graded. It is crucial for scalability that the action potential can be represented as a binary event and that we can use AER (Address Event Representation) based communication (Bailey and Hammerstrom, 1988). As soon as we need to incorporate graded interaction of some sort in the cell-to-cell interaction, like gap junctions or sub-threshold transmitter release, parallel simulations needs to be organized very differently and the neuronal interaction becomes potentially performance limiting. Even so, parallelization of the model will be beneficial in this case as well, and good simulator design will be able to provide reasonable simulation times in such cases as well. Given this development we are likely to see an increasing use of complex network modeling in neuroscience in the coming years.

6.4 Mapping the model onto the Blue Gene/L torus

In BG/L, every node is connected to six neighbours through the 3D torus network. Communication is particularly efficient with nodes a small number of hops away. Since SPLIT makes extensive use of pairwise communication, it should be possible to gain efficiency from this but we have not yet explored how to map our model onto the torus. As described above, the SPLIT library already has support for specification of the mapping from neurons to slave processes. The solution should be fairly straightforward since our cortex model has an inherent 2D geometry where most data is exchanged between 2D-neighbors. The problem thus reduces to folding a 2D sheet onto the 3D torus.

7 Conclusions

Our study demonstrates that supercomputing power is now sufficient to simulate neuronal networks with numbers of model neurons and synapses approaching those in the brains of small mammals. Most importantly, this can be done without totally sacrificing the level of biological detail in neuron and synapse models. The model neurons used in our simulations were multi-compartmental and of a conductance based, Hodgkin-Huxley type, representing an intermediate level of biological detail.

The largest network model simulated so far comprises eight million neurons and four billion synaptic connections. Considering that we only simulate specific cell types in one cortical layer, this corresponds to 240000 minicolumns or almost half of the rat neocortex (Johansson and Lansner, 2004a). To our knowledge this is, by far, the largest simulation of this type ever performed. It takes about one and a half hour to simulate one second of dynamic network activity. Simulation time is dominated by local computation and there is a significant potential gain of further parallelization of local computation.

We have also shown that an abstract, connectionist type, model of neocortex has linear scaling characteristics, both when the number of units and connections is increased. Generally the model scales as ZH, where Z is the number of connections per unit and H the number of hypercolumns. The largest network simulated had about a million units and 200 billion connections. It takes about ten seconds to simulate one second of network activity, including synaptic plasticity. Finally, we conclude that the simulated network requires five to six orders of magnitude more energy per synapse than its biological counterpart; the human brain.

Four techniques were used to achieve good performance; delayed update, computation of the leaky integrators in the logarithmic domain, adjacency lists for effective indexing of the units in the sparse connectivity matrix, and AER for effective communication. For future explorations of the brain's cognitive functions, this type of abstract model is an important tool.

Finally, we conclude that the simulation tools described here can efficiently utilize the massive parallelism in today's fastest computer, Blue Gene/L.

8 Acknowledgments

We would like to thank Carl G. Tengwall and Erling Weibust, Blue Gene Solutions, IBM Deep Computing EMEA, IBM Svenska AB, and, Cindy Mestad, James

C. Pischke, Steven M. Westerbeck and Michael Hennecke for excellent support when running on the Blue Gene/L installation at the Deep Computing Capacity on Demand Center, IBM, Rochester. Many thanks also to PDC, KTH, and, SNIC, for access to the Dell cluster Lenngren.

References

Amit, D. (1989). *Modeling Brain Function: The World of attractor neural networks*. Cambridge University Press, New York.

Anderson, J. S., Lampl, I., Gillespie, D. C., and Ferster, D. (2001). Membrane potential and conductance changes underlying length tuning of cells in cat primary visual cortex. *J Neurosci*, 21(6):2104–12.

Bailey, J. and Hammerstrom, D. (1988). Why vlsi implementations of associative vlcns require connection multiplexing. In *International Conference on Neural Networks*, San Diego, U.S.A.

Blakeslee, B. and McCourt, M. E. (2004). A unified theory of brightness contrast and assimilation incorporating oriented multiscale spatial filtering and contrast normalization. *Vision Res*, 44(21):2483–503.

Bower, J. M. and Beeman, D. (1998). *The book of GENESIS: Exploring realistic neural models with the GEneral NEural SImulation System*. Springer-Verlag, New York, 2 edition. ISBN 0-387-94938-0.

Churchland, P. S. and Sejnowski, T. J. (1992). *The Computational Brain*. The MIT Press, Cambridge, Massachusetts.

Compte, A., Brunel, N., Goldman-Rakic, P. S., and Wang, X. J. (2000). Synaptic mechanisms and network dynamics underlying spatial working memory in a cortical network model. *Cereb Cortex*, 10(9):910–23.

Cossart, R., Aronov, D., and Yuste, R. (2003). Attractor dynamics of network up states in the neocortex. *Nature*, 423(6937):283–8.

Deiss, S. R., Douglas, R. J., and Whatley, A. M. (1999). A pulse-coded communication infrastructure for neuromorphic systems. In Maass, W. and Bishop, C. M., editors, *Pulsed Neural Networks*, pages 157–178. The MIT Press, Cambridge, Massachusetts.

Djurfeldt, M., Rehn, M., Lundqvist, M., and Lansner, A. (2006). Attractor dynamics in a modular network model of neocortex. In preparation.

Douglas, R. J. and Martin, K. A. (2004). Neuronal circuits of the neocortex. *Annu Rev Neurosci*, 27:419–51.

Fransén, E. and Lansner, A. (1995). Low spiking rates in a population of mutually exciting pyramidal cells. *Network: Computation in Neural Systems*, 6(2):271–288.

Fransén, E. and Lansner, A. (1998). A model of cortical associative memory based on a horizontal network of conneted columns. *Network: Computation in Neural Systems*, 9:235–264.

Frye, J. (2005). http://brain.cse.unr.edu/ncsdocs/.

Fuster, J. M. (1995). *Memory in the Cerebral Cortex*. The MIT Press, Cambridge, Massachusetts.

Gara, A., Blumrich, M. A., Chen, D., Chiu, G. L.-T., Coteus, P., Giampapa, M. E., Haring, R. A., Heidelberger, P., Hoenicke, D., Kopcsay, G. V., Liebsch, T. A., Ohmacht, M., Steinmacher-Burow, B. D., Takken, T., and Vranas, P. (2005). Overview of the Blue Gene/L system architecture. *IBM Journal of Reasearch and Development*, 49(2/3):195–212.

Gross, J. L. and Yellen, J. (1998). *Graph theory and its applications*. CRC Press, Boca Raton, Florida, 2 edition.

Haberly, L. B. and Bower, J. M. (1989). Olfactory cortex: model circuit for study of associative memory? *Trends Neurosci*, 12(7):258–64.

Hammarlund, P. and Ekeberg, O. (1998). Large neural network simulations on multiple hardware platforms. *J Comput Neurosci*, 5(4):443–59.

Henry, C. (2005). Basal metabolic rate studies in humans: measurement and development of new equations. *Public Health Nutr*, 8(7A):1133–52.

Hertz, J., Krogh, A., and Palmer, R. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood, CA.

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

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, USA*, 79:2554–2558.

Hubel, D. H. and Wiesel, T. N. (1977). Ferrier lecture. functional architecture of macaque monkey visual cortex. *Proc R Soc Lond B Biol Sci*, 198(1130):1–59.

Johansson, C. and Lansner, A. (2004a). Towards cortex sized artificial nervous systems. In Negoita, M. G., Howlett, R. J., and Jain, L. C., editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 3213 of *Lecture Notes in Artificial Intelligence*, pages 959–966, Berlin. Springer.

Johansson, C. and Lansner, A. (2004b). Towards cortex sized attractor ann. In Ijspeert, A. J., Masayuki, M., and Naoki, W., editors, *First International Workshop on Biologically Inspired Approaches to Advanced Information Technology*, volume 3141 of *Lecture Notes in Computer Science*, pages 63–79, Berlin. Springer.

Johansson, C. and Lansner, A. (2006a). A fixed-point arithmetic implementation of an exponentially weighted moving average. Submitted to IEEE Trans. on Neural Networks.

Johansson, C. and Lansner, A. (2006b). Towards cortex sized artificial neural systems. Submitted to Neural Networks.

Kandel, E. R., Schwartz, J. H., and Jessell, T. M., editors (2000). *Principles of Neural Science*. McGraw-Hill, New York, 4th edition.

Kawaguchi, Y. (1995). Physiological subgroups of nonpyramidal cells with specific morphological characteristics in layer II/III of rat frontal cortex. *J. Neurosci.*, 15:2638–2655.

Kawaguchi, Y. and Kubota, Y. (1993). Correlation of physiological subgroupings of nonpyramidal cells with parvalbumin- and calbindind28k-immunoreactive neurons in layer v of rat frontal cortex. *J Neurophysiol*, 70(1):387–96.

Lansner, A. and Ekeberg, Ö. (1989). A one-layer feedback artificial neural network with a bayesian learning rule. *International Journal of Neural Systems*, 1(1):77–87.

Lansner, A. and Fransén, E. (1992). Modeling hebbian cell assemblies comprised of cortical neurons. *Network: Computation in Neural Systems*, 3:105–119.

Lansner, A., Fransén, E., and Sandberg, A. (2003). Cell assembly dynamics in detailed and abstract attractor models of cortical associative memory. *Theory Biosci*, 122:19–36.

Lansner, A. and Holst, A. (1996). A higher order Bayesian neural network with spiking units. *International Journal of Neural Systems*, 7(2):115–128.

Lundqvist, M., Rehn, M., Djurfeldt, M., and Lansner, A. (2006). Attractor dynamics in a modular network model of neocortex. *Network: Computation in Neural Systems*. Submitted.

Markram, H., Toledo-Rodriguez, M., Wang, Y., Gupta, A., Silberberg, G., and Wu, C. (2004). Interneurons of the neocortical inhibitory system. *Nat Rev Neurosci*, 5(10):793–807.

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

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Computation*, 17:1776–1801.

Mortara, A. and Vittoz, E. (1994). A communication architecture tailored for analog vlsi artificial neural networks: intrinsic performance and limitations. *IEEE Transactions on Neural Networks*, 5(3):459–466.

Mountcastle, V. B. (1978). An organizing principle for cerebral function: The unit module and the distributed system. In Edelman, G. M. and Mountcastle, V. B., editors, *The mindful brain*. The MIT Press, Cambridge, Massachusetts.

Palm, G. (1982). *Neural assemblies. An alternative approach to artificial intelligence*. Springer.

Peters, A. and Sethares, C. (1991). Organization of pyramidal neurons in area 17 of monkey visual cortex. *J Comp Neurol*, 306(1):1–23.

Rolls, E. T. and Treves, A. (1998). *Neural Networks and Brain Function*. Oxford University Press, New York.

Sandberg, A., Lansner, A., Petersson, K. M., and Ekeberg, Ö. (2002). Bayesian attractor networks with incremental learning. *Network: Computation in neural systems*, 13:179–194.

Sandberg, A., Tegner, J., and Lansner, A. (2003). A working memory model based on fast hebbian learning. *Network*, 14(4):789–802.

Tam, A. and Wang, C. (2000). Efficient scheduling of complete exchange on clusters. In *13th International Conference on Parallel and Distributed Computinh Systems*, Las Vegas, USA.

Treves, A. and Rolls, E. T. (1994). Computational analysis of the role of the hippocampus in memory. *Hippocampus*, 4(3):374–91.

Willshaw, D. and Longuet-Higgins, H. (1970). Associative memory models. In Meltzer, B. and Michie, O., editors, *Machine Learning*, volume 5. Edinburgh University Press, Edinburgh, Scotland.