Jan Hahne, Moritz Helias, Susanne Kunkel, Jun Igarashi, Itaru Kitayama, Brian Wylie, Matthias Bolten, Andreas Frommer, Markus Diesmann

Including gap junctions into distributed neuronal network simulations

April 1, 2016

Including gap junctions into distributed neuronal network simulations

Jan Hahne[1], Moritz Helias[2,3], Susanne Kunkel[3,4,5], Jun Igarashi[6,7], Itaru Kitayama[7,8], Brian Wylie[9], Matthias Bolten[10], Andreas Frommer[1], and Markus Diesmann[2,11,12]

[1] School of Mathematics and Natural Sciences, University of Wuppertal, Wuppertal, Germany
[2] Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6) and JARA BRAIN Institute I, Jülich Research Centre, Jülich, Germany
[3] RIKEN Advanced Institute for Computational Science, Programming Environment Research Team, Kobe, Japan
[4] Department of Computational Science and Technology, School of Computer Science and Communication, KTH Royal Institute of Technology, Stockholm, Sweden
[5] Simulation Laboratory Neuroscience, Bernstein Facility for Simulation and Database Technology, Institute for Advanced Simulation, Jülich Aachen Research Alliance, Jülich Research Centre, Germany
[6] Okinawa Institute of Science and Technology, Neural Computation Unit, Okinawa, Japan
[7] Laboratory for Neural Circuit Theory, RIKEN Brain Science Institute, Wako, Japan
[8] HPC Usability Research Team, RIKEN Advanced Institute for Computational Science, Kobe, Japan
[9] Jülich Supercomputing Centre, Jülich Research Centre, Jülich, Germany
[10] Institut für Mathematik, Universität Kassel, Kassel, Germany
[11] Department of Psychiatry, Psychotherapy and Psychosomatics, Medical Faculty, RWTH Aachen University, Aachen, Germany
[12] Department of Physics, Faculty 1, RWTH Aachen University, Aachen, Germany

Abstract. Contemporary simulation technology for neuronal networks enables the simulation of brain-scale networks using neuron models with a single or a few compartments. However, distributed simulations at full cell density are still lacking the electrical coupling between cells via so called gap junctions. This is due to the absence of efficient algorithms to simulate gap junctions on large parallel computers. The difficulty is that gap junctions require an instantaneous interaction between the coupled neurons, whereas the efficiency of simulation codes for spiking neurons relies on delayed communication. In a recent paper [15] we describe a technology to overcome this obstacle. Here, we give an overview of the challenges to include gap junctions into a distributed simulation scheme for neuronal networks and present an implementation of the new technology available in the NEural Simulation Tool (NEST 2.10.0). Subsequently we introduce the usage of gap junctions in model scripts as well as benchmarks assessing the performance and overhead of the technology on the supercomputers JUQUEEN and K computer.

Keywords: computational neuroscience, spiking neuronal network, gap junctions, waveform relaxation, supercomputer, large-scale simulation

1 Introduction

Electrical synapses, or gap junctions, are classically regarded as a primitive mechanism of neural signaling mainly of relevance in invertebrate neural circuits. Recently, advances in molecular biology revealed their widespread existence in the mammalian nervous system, such as visual cortex, auditory cortex, sensory motor cortex, thalamus, thalamic reticular nucleus, cerebellum, hippocampus, amygdala, and the striatum of the basal ganglia [8,21], which suggests their importance in brain processes as diverse as learning and memory, movement control, and emotional responses [8,21,9]. The functional roles of gap junctions in network behavior are still not fully understood, but are widely believed to be crucial for synchronization and the generation of rhythmic activity. Theoretical work suggests that the contribution of gap junctions to synchronization is versatile, as it depends on the intrinsic currents and the morphology of the neurons as well as on their interaction with inhibitory synapses [16].

Even though brain-scale neural network simulations approach the size of the brain of small primates [17] and many biophysical phenomena are already included, such as the layer-specific connectivity and spike-timing dependent synaptic plasticity, simulations with correct cell densities are still lacking gap junctions. The new technology presented in [15] will hopefully help to overcome these shortcomings. This chapter starts with a brief review of the challenges of including gap junctions into neuronal network simulators. On the basis of examples of increasing complexity we then discuss the user interface of the recently released implementation of the new technology in NEST 2.10.0 [3]. Finally we evaluate results on the performance of the implementation obtained on the JUQUEEN supercomputer and the K computer in comparison to the release (NEST 2.8.0) prior to the incorporation of the gap-junction framework. The conceptual and algorithmic work is a module in our long-term collaborative project to provide the technology for neural systems simulations [11].

2 The challenge of including gap junctions

To understand the challenge of including gap junctions into a neuronal network simulator such as NEST we need to take a look at the architecture of the simulation kernel and the underlying assumptions. Simulation codes for neuronal networks exploit the delayed and point-event like nature of the spike interaction between neurons. In a network with only chemical synapses with delays d_{ij}, the dynamics of all neurons is decoupled for the duration of the minimal delay $d_{\min} = \min_{ij}(d_{ij})$. The synaptic delays in networks of point-neuron models are the result of an abstraction of the axonal propagation time of the action potential and the time the postsynaptic potential needs to travel from the location of the synapse on the dendrite to the soma where postsynaptic potentials are assumed to interact. Hence, the dynamics of each neuron can be propagated independently for the duration d_{min} without requiring information from other neurons, such that in distributed simulations the processes need to communicate spikes only after this period [25]. Gap junctions, however, are typically represented by an instantaneous interaction between pairs of neurons of the form

$$I_{\text{gap},ij}(t) = g_{ij}\ (V_i(t) - V_j(t))\,,$$

with V_i and V_j denoting the membrane potentials of the involved neurons and g_{ij} the conductance of the gap junction, also called gap weight. The gap current I_{gap} occurs in both cells at the site of the gap junction. In point-neuron models that assume equipotentiality, the gap-junction current immediately affects the membrane potential. This is unlike the modeling of chemical synapses in point neurons, where any axonal and dendritic delays are subsumed in a retarded spike interaction. Implementing a gap junction between neuron i and j in a time-driven simulation scheme therefore requires that neuron i knows the membrane potential of neuron j and vice verse at all times. The direction of the influence mediated by a gap junction depends on the difference of the neurons' membrane potentials; one neuron is excited, the other one inhibited.

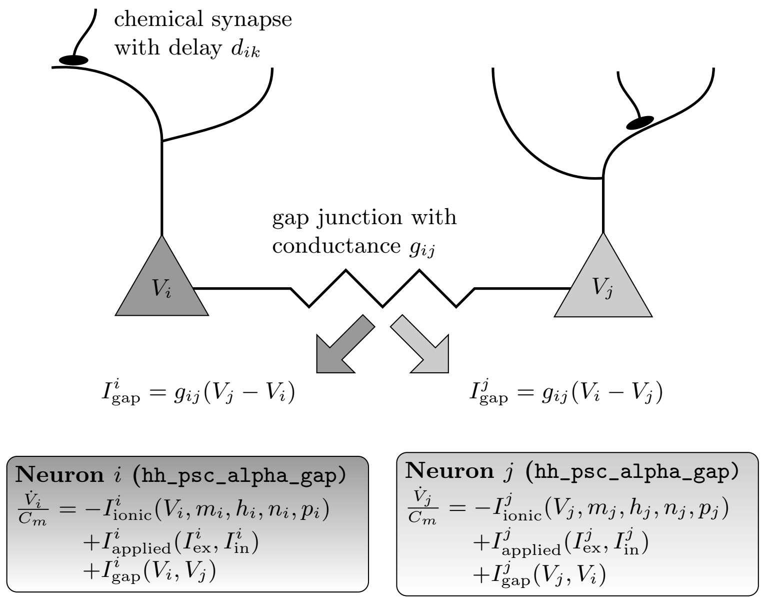
chemical synapse with delay d_{ik}
gap junction with conductance g_{ij}
V_i
V_j
$I^i_{\text{gap}} = g_{ij}(V_j - V_i)$
$I^j_{\text{gap}} = g_{ij}(V_i - V_j)$
Neuron i (hh_psc_alpha_gap)
$\frac{\dot{V}_i}{C_m} = -I^i_{\text{ionic}}(V_i, m_i, h_i, n_i, p_i) + I^i_{\text{applied}}(I^i_{\text{ex}}, I^i_{\text{in}}) + I^i_{\text{gap}}(V_i, V_j)$
Neuron j (hh_psc_alpha_gap)
$\frac{\dot{V}_j}{C_m} = -I^j_{\text{ionic}}(V_j, m_j, h_j, n_j, p_j) + I^j_{\text{applied}}(I^j_{\text{ex}}, I^j_{\text{in}}) + I^j_{\text{gap}}(V_j, V_i)$

Fig. 1. Representation of two point neurons coupled by a gap junction.

Fig. 1 illustrates the effect of a gap junction on the system of ordinary differential equations (ODEs) describing the neuronal dynamics. The originally decoupled systems of ODEs of neurons i and j are combined to a system of ODEs and can only be solved along with each other. Any additional neuron with a gap-junction connection to either i or j adds a further set of equations to the coupled system. In a biologically realistic simulation of a local cortical network each neuron has a couple of tens of gap-junction connections. In consequence the dynamics of almost all neurons are likely interrelated by one large system of ODEs. Although there are solvers like PVODE [7] of the software package SUNDIALS [19], which are specialized to the parallel solution of very large systems of differential equations of the form

$$\dot{y} = f(t, y), \qquad y(t_0) = y_0, \qquad y \in \mathbb{R}^n,$$

they cannot be employed in the context of distributed neuronal network simulations, due to the incompatible overall workflow. These solvers receive the entire system of ODEs given by the n-dimensional function $f(t, y)$ and the initial conditions y_0 as an input and integrate the dynamics by some user specified numerical method. In the more common case of an implicit numerical method, the resulting system of nonlinear algebraic equations is either solved by fixed-point iteration or by Newton iteration. The latter requires the solution of a linear system of equations. The idea of parallelization is to distribute the system of ODEs over the available computation nodes such that each node is solving a contiguous subset of the system. This is achieved by correspondingly distributing all vector operations (e.g. dot products, the calculation of norms, and linear sums) over the computation nodes. Each node computes the local part of each vector operation followed by a global MPI reduce operation for those operations where it is needed (see [7] for further details on CVODE). Thus this software conceptually uses one instance of the employed ODE solver and distributes its vector computations across the computation nodes. Instead, parallel neuronal network simulators distribute the neurons over the computation nodes. The parallelization makes use of the fact that the dynamics of the neurons without gap junctions is decoupled for the duration of the minimal delay $d_{\min}$. The solver is specified on the single-neuron level and may be different for different cell types. The membrane potentials of the gap-junction coupled neurons in the I_{gap}-term need to be approximated and communicated between the neurons at suitable times. The MPI communication between compute nodes happens collectively for all neurons on the node and only once for the duration of the minimal delay $d_{\min}$. These fundamental structural decisions are crucial for the performance of neuronal simulators and their scalability on supercomputers, where communication is expensive, because it is associated with considerable latency.

A simple approximate solution for this problem is to decrease the communication interval to the computation time step h and to communicate the membrane potentials of gap-junction coupled neurons at the beginning of each time step. This way gap currents are assumed to be constant for the duration of the time step when the ODE-system is solved. In [15] we show that the usage of this so called single-step method causes a shift in the membrane potential time course and errors in network spike rate and synchrony. An iterative framework using Jacobi waveform relaxation [24] avoids these shortcomings. The iterative method converges against the solution of the original large system and for a given integration error achieves higher performance. The framework is compatible with the propagation of neuron dynamics on the neuron level as well as communication in intervals of the d_{min}.

3 Using gap junctions in NEST 2.10.0

The NEST `Connect` routine enables neuroscientists to express a partial network structure through the connections between two sets of neurons. One dictionary specifies the connection rule and the rule-specific parameters, a second the dynamics of the interaction. The present implementation [3] accepts various connection rules from simple ones, like `all_to_all` and `one_to_one,` to random connections between the sets, such as `fixed_indegree` and `fixed_total_number`. Chemical synapses, the original research domain of NEST, mediate a directed interaction. Therefore the `Connect` routine is designed to specify directed graphs. Gap junctions, however, mediate a bidirectional interaction. Simulation code for spiking neuronal networks exploits the directedness of chemical synapses by representing synapses only on the compute node where the postsynaptic neuron resides. This enables network creation to be organized as an ideally parallelized activity without communication between nodes [25]. In this framework gap junctions need partial representations on the postsynaptic as well as on the presynaptic side to mediate the bidirectional interaction on the undirected subgraph [15]. Hence, in order to connect two neurons through a gap junction, connections in both directions need to be created. Script 1 shows corresponding code in the PyNEST [10] syntax to create two neurons connected by a single gap junction. The default algorithm of the `Connect` command is `all_to_all`. It connects the neurons specified as presynaptic (first argument) to all the neurons specified as postsynaptic. As each of the n-tuples `a` and `b` contains only a single neuron, the two `Connect` calls achieve the desired result of bidirectional connectivity. Script 2 creates the identical network in two alternative ways employing higher-level connection algorithms. The `one_to_one` algorithm connects neuron pairs specified by explicit corresponding lists of pre- and postsynaptic neurons. For our example of a single gap junction the two lists are formed from the tuple of two neurons `n` and its reverse. Line 9 again uses the algorithm `all_to_all` that has already been employed in Script 1. By using the same list for the pre- and postsynaptic neurons a fully connected network is created. Self connections are excluded by setting the `autapses` flag to false. This alternative generalizes to all-to-all connected networks of an arbitrary number of neurons independent of whether a network of chemical synapses or gap junctions is desired. All three variants ideally and automatically parallelize relying on the NEST implementation of `Connect`.

Script 1. Creation of a gap junction using the command for a directed interaction between two neurons. Two calls are required; one for each direction. Here and in the following scripts we use the syntax of the PyNEST interface [10] of the NEST simulation software as of version 2.10.0 [3]. By convention in `Connect(i,j)` the interaction is from i to j; i exerts an influence on j. This differs from the convention for connectivity matrices W_{ij} in computational neuroscience. `Create` returns an n-tuple and `Connect` accepts n-tuples, lists, and arrays of the `numpy` module as arguments for i and j. The third positional argument of `Connect` specifies the connection algorithm; it is not given here and hence falls back to its default value `all_to_all`. The fourth positional argument specifies the dynamics of the connection; as the third argument is omitted, the fourth argument needs to be assigned by its name `syn_spec`.

```
import nest

a = nest.Create('hh_psc_alpha_gap')
b = nest.Create('hh_psc_alpha_gap')
nest.Connect(a, b, syn_spec= 'gap_junction')
nest.Connect(b, a, syn_spec= 'gap_junction')
```

Script 2. Creation of a single gap junction using alternative algorithms for directed interactions between groups of neurons. Here `n[::-1]` is the Python notation for an n-tuple in reversed order. Use of a dictionary for the connection algorithm enables the specification of more details. An autapse is a connection a neuron forms with itself, which is forbidden here. Other notation as in Script 1. The first alternative is only meaningful for a single gap junction, the second generalizes to networks with all-to-all connectivity.

```
import nest

n = nest.Create('hh_psc_alpha_gap', 2)

# using algorithm 'one_to_one'
nest.Connect(n, n[::-1], 'one_to_one', 'gap_junction')

# alternative algorithm
nest.Connect(n, n,
  {'rule': 'all_to_all', 'autapses': False}, 'gap_junction')
```

We need to take more care for more complex networks. Let us consider an example where the total number of gap junctions in a given volume of cortical tissue is known. These gap junctions are randomly distributed over all possible pairs of neurons in the volume without any further constraints. In particular, a neuron does not have a gap junction with itself, but a given pair of neurons may be coupled by more than one gap junction. Script 3 shows a script imple-

Script 3. Creation of a network with a predetermined total number of gap junctions between randomly chosen pairs of neurons using a predefined connection algorithm. In a first step (line 15) the random network is created as a directed graph. The second step (lines 18-20) obtains the list of connected neuron pairs from the simulator and reshapes the data to corresponding lists of pre- and postsynaptic neurons. The final step (line 22) adds the transposed connectivity matrix to the network by supplying `Connect` with the lists of the pre- and postsynaptic neurons of the original network in reversed order. The parameters in the script result in a binomially distributed number of gap junctions per neuron with a mean of 60. The script does not work in a distributed simulation as the function `GetConnections` only returns the part of the network represented on the node executing the command; the set of incoming connections of the locally represented neurons.

```
 1  import nest
 2  import numpy as np
 3
 4  # total number of neurons
 5  N = 100
 6
 7  # total number of gap junctions
 8  K = 3000
 9
10  n = nest.Create('hh_psc_alpha_gap', N)
11
12  r = {'rule': 'fixed_total_number', 'N': K, 'autapses': False}
13  g = {'model': 'gap_junction', 'weight': 0.5}
14
15  nest.Connect(n, n, r, g)
16
17  # get source and target of all connections
18  m = np.transpose(
19      nest.GetStatus(nest.GetConnections(n),
20      ['source', 'target']))
21
22  nest.Connect(m[1], m[0], 'one_to_one', g)
```

menting this network using the algorithm `fixed_total_number` of the `Connect` command. At line 15 the network is created as a directed graph; the interaction is mediated only in one direction. `Connect` efficiently generates this network instantiating the relevant subgraphs in parallel on all of the compute nodes using parallel random number generators. Therefore on the level of the interpreter executing the script, the actual connectivity is not known. In order to create the complementary directed graph we need to retrieve the existing connections from the simulation kernel, exchange the roles of pre- and postsynaptic neurons, and create this subnetwork in addition. `GetConnections`, however, only returns the set of incoming connections of the neurons represented on the local compute node. The transpose of this subnetwork therefore, generally, has mainly non-local postsynaptic neurons which the `Connect` command ignores. Hence, Script 3 does not work in a distributed simulation. This problem occurs for any type of network where the realization is only known to the simulation engine. The alternative is to generate lists of neurons to be connected on the level of the interpreter executing the script before handing them down to the `Connect` command.

Script 4. Creation of a network with a predetermined total number of gap junctions using an explicit list of random neuron pairs. Same parameters as in Script 3. The random module of the Python Standard Library is used to independently draw K pairs of random samples from the list of all neurons (line 17). The data are in the same line reshaped into two corresponding lists of pre- and postsynaptic neurons. The first `Connect` command (line 19) interprets the first list (`m[0]`) as the presynaptic neurons. The second `Connect` command adds the transposed connectivity as in line 22 of Script 3. The script does work in a distributed simulation, but is inefficient as each compute node draws the full list of neuron pairs.

```
 1  import nest
 2  import random
 3  import numpy as np
 4
 5  # total number of neurons
 6  N = 100
 7
 8  # total number of gap junctions
 9  K = 3000
10
11  n = nest.Create('hh_psc_alpha_gap', N)
12
13  g = {'model': 'gap_junction', 'weight': 0.5}
14
15  random.seed(0)
16
17  m = np.transpose([random.sample(n, 2) for _ in range(K)])
18
19  nest.Connect(m[0], m[1], 'one_to_one', g)
20  nest.Connect(m[1], m[0], 'one_to_one', g)
```

Script 4 illustrates this approach using the `random` module of the Python Standard Library. The drawback of this script is the serialization of the connection procedure in terms of computation time and memory. Each compute node participating in the simulation needs to draw the identical full set of random numbers and temporarily represent the total connectivity in variable `m`. In the two subsequent calls of connect, each compute node only considers those neuron pairs where the postsynaptic neuron is local.

4 Performance of the NEST implementation

The gap-junction framework as described in [15] brings two major extensions to a simulation engine for biological neuronal networks such as NEST. Firstly a new event type, the so called secondary event. Secondary events are used to communicate approximations of the membrane potential time courses between neurons. They are emitted and communicated only at the end of the communication interval d_{min} and contain data for every computation time step within this interval. This data is used to approximate the membrane potential of the event-sending neuron in the event-receiving neuron. Secondly the simulation engine is extended by the ability to repeat the neuronal updates of a given time step multiple times until a stopping criterion is met. The latter is required for globally iterative solvers like the waveform relaxation scheme used here.

The design of the framework for gap junctions is guided by the requirement neither to impair code maintainability nor to impose penalties on run time or memory usage for simulations that exclusively use chemical synapses. The first requirement is addressed by the design choice to tightly integrate the novel framework with the existing connection and communication infrastructure of NEST instead of developing an independent pathway for gap-junction related data. To assess to what extent the second requirement is met we measure the performance of i) simulations exclusively using chemical synapses and ii) simulations including gap junctions. In this chapter we present benchmarks investigating the performance of NEST 2.10.0. The employed test cases are already included in [15] using a prototype branch of NEST and the JUQUEEN BlueGene/Q supercomputer at the Jülich Research Centre in Germany. For this chapter we rerun the benchmarks with NEST 2.10.0 and add results for the K computer at the Advanced Institute for Computational Science (AICS) in Kobe, Japan.

First we turn to the influence of the new capabilities of the simulation engine on simulations without gap junctions. We measure the deviation in simulation time and memory usage between the last release without the gap junction framework (NEST 2.8.0) and NEST 2.10.0. Although NEST 2.10.0 also contains other changes and new features like a framework for structural plasticity, the most time- and memory-sensitive changes are due to the gap-junction framework. The test case is a balanced random network model [5]. Fig. 2 specifies the network model and presents results for a maximum-filling scenario, where for a given machine size VP we simulate the largest possible network that just fits

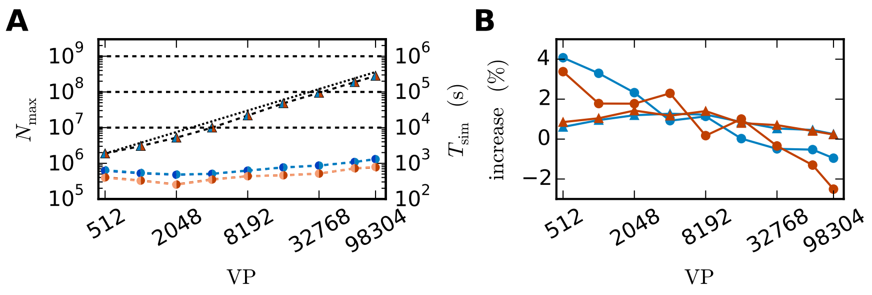

Fig. 2. Overhead of gap-junction framework for network with only chemical synapses. In this and all subsequent figures VP denotes the overall number of processes used in line with our distribution strategy (8 OpenMP threads per node). Shades of blue indicate the JUQUEEN supercomputer, while shades of red show data from the K computer. **(A)** Triangles show the maximum network size of a balanced random network model that can be simulated in the absence of gap junctions ([15], test case 3). The network consists of of 80% excitatory and 20% inhibitory leaky integrate-and-fire model neurons with alpha-shaped post-synaptic currents. Each neuron has a total number of $11,250$ (9000 excitatory, 2250 inhibitory) incoming connections. Circles show the corresponding wall-clock time (averaged over three runs) required to simulate the network for 1 second of biological time. Left semicircles indicate the results with NEST 2.8.0 without the gap-junction framework and right semicircles are obtained with the framework included (NEST 2.10.0). **(B)** Increase of time (circles) and memory consumption (triangles) of NEST 2.10.0 in percent as compared to NEST 2.8.0.

into the memory of the machine (for a discussion of different scaling-scenarios and their interpretation in the context of neural network simulations see [1]). Although the simulation scenario is maximum filling, in the presence of the gap junction framework we are able to simulate the same network size as before; the increase in memory usage is within the safety margin of the maximum-filling procedure (see [23] for details). Measured in percentage of the prior memory usage (Fig. 2B) the consumption increases by 0.2 to 1.5 percent depending on the number of virtual processes VP. The small increase of memory usage is caused by the changes to the thread-local connection infrastructure and the communication buffer. The behavior on JUQUEEN and the K computer is almost identical. The run time of the simulation increases up to 4.0 percent for simulations with a low number of VPs with an average of 1.1 respectively 0.7 on JUQUEEN and the K computer. The simulation times on the K computer show slightly higher fluctuations, although the measurements are averaged over three runs on both supercomputers. One contribution to the increase in run time is an additional check for the existence of connections using secondary events during the event delivery. A further contribution are additional initializations in the beginning of the simulation. Therefore this increase reduces at higher numbers of virtual processes due to the prolonged simulation time of these simulations.

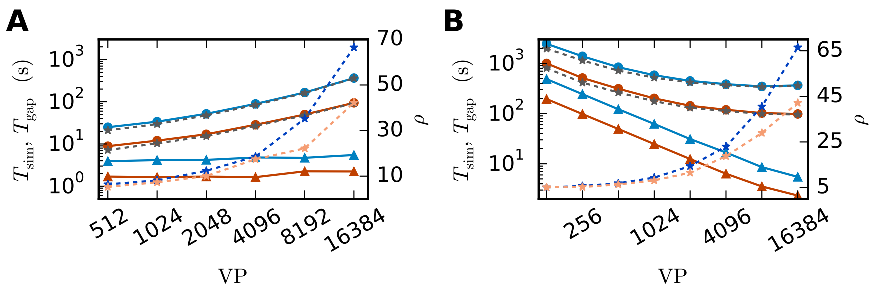

Fig. 3. Costs of gap-junction dynamics. All results are obtained with NEST 2.10.0 and communication is carried out in intervals of the minimal network delay $d_{\rm min}$ (here $d_{\rm min} = 1$ ms). The solid curves with circles show the simulation time $T_{\rm sim}$ of a network with Hodgkin-Huxley dynamics (test case 1b of [15]). The neurons have 60 gap-junctions and receive an additional current of 200.0 pA. The solid curves with triangles indicate the simulation time $T_{\rm sim}$ of the same network in the absence of gap junctions. The corresponding colored curves with asterisks show the ratio ρ of $T_{\rm sim}$ with and without gap junctions, while gray curves with asterisks show the difference $T_{\rm gap}$ of both simulation times. Simulations represent 50 ms of biological time at a step size of $h = 0.05$ms. All simulations use only a single iteration per time interval. **(A)** Weak scaling with $N = 185 \cdot \text{VP}$ neurons. **(B)** Strong scaling with $N = 185 \cdot 16384 = 3,031,040$ neurons.

Fig. 3 investigates the slowdown due to gap-junction dynamics. This is done by simulating a network with N neurons with Hodgkin-Huxley dynamics with alpha-shaped post-synaptic currents and gap-junction coupling. An additional current ensures realistic spiking behavior. For this test we only employ a single iteration per time interval, instead of using the entire iterative scheme. The obtained results are compared to the run time of a simulation without gap junctions, but otherwise identical setup. This way the difference of the two run times $T_{\rm gap}$ is the time required for the additional computational load and communication. Fig. 3A is a weak-scaling scenario. It demonstrates that the scalability of the test case is impaired by the additional communication of the secondary events. Despite the constant number of neurons per virtual process the run time increases substantially. The reason is that NEST 2.10.0 employs global communication with `MPI_Allgather` to exchange events between computation nodes. Therefore the number of received events per compute node increases with the total number of neurons. The processing of these events combined with the increased communication time leads to a substantial increase in run time. Fig. 3B studies the same setup in strong scaling with $N \approx 3 \cdot 10^7$ neurons. In this scenario the number of received events per computation node is constant, while the number of events produced by each compute node shrinks with increasing number of virtual processes. Here $T_{\rm gap}$ decreases at first and then almost stagnates for more than 2048 virtual processes. The saturation is explained by the processing and communication of the secondary events, which constitutes the major contribution to T_{gap} in this setup. As the simulation without gap junctions uses exactly the same pattern of MPI communication this is not an issue of latency, but an issue of bandwidth combined with the processing of the data. The initial decrease is due to the parallelization of the gap-junction dynamics: the computations on the single-neuron level, like the adaptive solution of the single-neurons ODE-system and the handling of incoming events are parallelized. For both scalings the behavior on JUQUEEN and the K computer is similar. The K computer benefits from the faster processors (2 GHz vs. 1.6 GHz) and the higher bandwidth per link (5 GB/s vs. 2 GB/s), but otherwise shows the same scaling behavior as JUQUEEN.

In conclusion the additional time required by simulations with gap junctions on both supercomputers is determined by the total number of neurons N. As the increase in run time is dominated by the processing and communication of the secondary events in combination with a global communication method it cannot be eliminated by using more virtual processes VP. Therefore it is advisable to use as few compute nodes as possible. In this optimal setting the communication required for gap junctions increases the simulation time of one iteration for a network of $N \approx 3 \cdot 10^7$ neurons by a factor of $\rho = 5.0$. This is, however, only the increase for a single iteration. For the accurate solution of simulations with gap junctions an iterative scheme is employed. Therefore one has to multiply the increase in simulation time T_{gap}, as displayed in Fig. 3, with the average number of iterations to receive an estimate of the overall increase in run time. For moderate gap weights the average number of iterations is about $3 - 6$ (for more details see [15]).

Conclusions

The framework for representing and simulating gap junctions in NEST 2.10.0 extends the capabilities of NEST, widens the domain of applications, and is available on supercomputers like JUQUEEN and the K computer. The iterative solver guarantees a high accuracy for network simulations with gap junctions regardless of the coupling strength. More generally, the new technology may serve as the foundation for other types of interactions requiring a continuous analog coupling as in so called rate or population models. The ability to roll backwards in time and repeat a propagation step including communication is a further generalization of the simulation engine. Nevertheless, there is still potential for optimization, both in terms of scalability and in terms of usability.

The limitation of the scalability of simulations with gap junctions arises from the need to communicate approximations of the membrane potential time courses between neurons. As the employed communication scheme uses collective MPI calls, these approximations are sent to all nodes that take part in the simulation irrespective of whether or not these nodes harbor neurons requiring this information. This situation is qualitatively similar to the spike times being collectively communicated. However, there are two quantitative differences, the number of connections per neuron (order 10,000 for chemical synapses vs. order 100 for gap junctions) and the amount of information communicated (4 Byte per spike vs. order 100 Bytes per minimum delay). Future work on the simulation code should assess the potential of targeted communication. Due to the low number of connections and their locality, directed communication may be particularly beneficial for gap-junction coupling.

In terms of usability the creation of complex bidirectional networks needs to be simplified. The present user interface requires all connected neuron pairs to be known beforehand at the level of the simulation language interpreter, for example Python, or the directed connections created by a previous `Connect` call to be obtained from the simulation engine using the `GetConnections` command. The former is inefficient as it leads to serialization, as demonstrated by Script 4. The latter, as demonstrated by Script 3, leads to code that is only correct for simulations using a single compute node; more disturbingly, distributed execution will result in incorrectly connected networks without a warning to the researcher. An exception are networks with all-to-all connectivity, as discussed in Script 2, for which a single call to connect produces the expected result for networks with chemical synapses (unidirectional interaction) as well as networks with gap junctions (bidirectional interaction). Future work should explore whether `Connect` can be informed about the intention to create unidirectional or bidirectional connections and whether the combination of specific connection dynamics with incompatible connectivity algorithms can be prevented.

Improvements to the user interface towards the expressive and safe handling of connection algorithms for networks with bidirectional connectivity have a wider scope than just networks with gap junctions. NEST already supports binary neuron models as documented in [14]. Early seminal works exploring fundamental properties of recurrently connected networks studied “symmetric”, that is bidirectional, connectivity. A prominent example is the Hopfield network [20], employing binary units. Due to the symmetric connectivity, similar to spin-glass systems, an energy function can be defined and the dynamics is relaxational, approaching the minima of the energy. Functionally these models implement associative memories [2]. Their close relation to systems of classical statistical mechanics allows an analytical treatment and the transfer of earlier results from theoretical physics [18]. Networks of binary model neurons [13] have also played an important role for the development of the theory of fluctuating activity in neural networks [12]. In recent years they experienced a revival, because they enable a systematic fluctuation expansion [6]. Moreover, a fundamental link to spiking networks has been established: both model classes can, to some approximation, be mapped to networks of units that interact by analog variables in continuous time [14]. The implementation of the latter networks of so called rate or population models [4, for a recent review] requires only a moderate extension of the technology to exchange continuous signals, as presented here. However, further work is needed on the user interface and on the implementation of a set of canonical rate models treated in the literature.

The exercise of integrating a scheme for the simulation of gap junctions into an existing code for the distributed simulation of spiking neuronal networks has not only widened the scope of biophysical phenomena now accessible to large-scale simulation, but also taught us further lessons about useful abstractions of simulation engines, expanded our knowledge on the constraints of scaling, and opened a pathway towards the design of a unified simulation engine for some of the most classical neuronal network models.

Acknowledgements

We gratefully acknowledge the NEST core team for an in-depth discussion of the user interface and Mitsuhisa Sato for hosting our activities at RIKEN AICS. Computing time on the K computer was provided through early access in the framework of the co-development program, project hp130120 of the General Use Category (2013), the Strategic Program (project hp150236, Neural Computation Unit, OIST), and MEXT SPIRE Supercomputational Life Science. The authors gratefully acknowledge the computing time on the supercomputer JUQUEEN [22] at Forschungszentrum Jülich granted by JARA-HPC Vergabegremium (provided on the JARA-HPC partition, jinb33) and Gauss Centre for Supercomputing (GCS) (provided by John von Neumann Institute for Computing (NIC) on GCS share, hwu12). Partly supported by Helmholtz Portfolio Supercomputing and Modeling for the Human Brain (SMHB), the Initiative and Networking Fund of the Helmholtz Association, the Helmholtz young investigator group VH-NG-1028, the Next-Generation Supercomputer Project of MEXT, and EU Grant 604102 (Human Brain Project, HBP). All network simulations carried out with NEST (http://www.nest-simulator.org).

References

1. van Albada, S.J., Kunkel, S., Morrison, A., Diesmann, M.: Integrating brain structure and dynamics on supercomputers. In: Grandinetti, L., Lippert, T., Petkov, N. (eds.) Brain-Inspired Computing, pp. 22–32. Springer (2014)
2. Amit, D.J.: Modeling Brain Function. Cambridge University Press, Cambridge, New York (1989)
3. Bos, H., Morrison, A., Peyser, A., Hahne, J., Helias, M., Kunkel, S., Ippen, T., Eppler, J.M., Schmidt, M., Seeholzer, A., Djurfeldt, M., Diaz, S., Morén, J., Deepu, R., Stocco, T., Deger, M., Michler, F., Plesser, H.E.: Nest 2.10.0 (Dec 2015), http://dx.doi.org/10.5281/zenodo.44222
4. Bressloff, P.C.: Spatiotemporal dynamics of continuum neural fields. Journal of Physics A: Mathematical and Theoretical 45(3), 033001 (2012), http://iopscience.iop.org/1751-8121/45/3/033001; http://www.math.utah.edu/ bresslof/publications/11-7.pdf
5. Brunel, N.: Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. J. Comput. Neurosci. 8(3), 183–208 (2000)
6. Buice, M.A., Cowan, J.D., Chow, C.C.: Systematic fluctuation expansion for neural network activity equations. Neural Comput. 22, 377–426 (2009)
7. Byrne, G.D., Hindmarsh, A.C.: PVODE, an ODE solver for parallel computers. International Journal of High Performance Computing Applications 13(4), 354–365 (1999), http://hpc.sagepub.com/content/13/4/354.short; http://acts.nersc.gov/sundials/documents/ucrl-jc-132361.pdf

8. Connors, B.W., Long, M.A.: Electrical synapses in the mammalian brain. Annu. Rev. Neurosci. 27(1), 393–418 (2004)
9. Dere, E., Zlomuzica, A.: The role of gap junctions in the brain in health and disease. Neurosci. Biobehav. Rev. 36, 206–217 (2011)
10. Eppler, J.M., Helias, M., Muller, E., Diesmann, M., Gewaltig, M.: PyNEST: a convenient interface to the NEST simulator. Front. Neuroinformatics 2, 12 (2009)
11. Gewaltig, M.O., Diesmann, M.: NEST (NEural Simulation Tool). Scholarpedia 2(4), 1430 (2007)
12. Ginzburg, I., Sompolinsky, H.: Theory of correlations in stochastic neural networks. Phys. Rev. E 50(4), 3171–3191 (1994)
13. Glauber, R.: Time-dependent statistics of the Ising model. J. Math. Phys. 4(2), 294–307 (1963)
14. Grytskyy, D., Tetzlaff, T., Diesmann, M., Helias, M.: A unified view on weakly correlated recurrent networks. Front. Comput. Neurosci. 7, 131 (2013)
15. Hahne, J., Helias, M., Kunkel, S., Igarashi, J., Bolten, M., Frommer, A., Diesmann, M.: A unified framework for spiking and gap-junction interactions in distributed neuronal network simulations. Front. Neuroinform. 9(22) (2015)
16. Hansel, D., Mato, G., Benjamin, P.: The role of intrinsic cell properties in synchrony of neurons interacting via electrical synapses. In: Schultheiss, N.W., Prinz, A.A., Butera, R.J. (eds.) Phase Response Curves in Neuroscience: Theory, Experiment, and Analysis, Springer Series in Computational Neuroscience, vol. 6, chap. 15, pp. 361–398. Springer, 1st edn. (2012)
17. Herculano-Houzel, S.: The human brain in numbers: a linearly scaled-up primate brain. Front. Hum. Neurosci. 3(31) (2009)
18. Hertz, J., Krogh, A., Palmer, R.G.: Introduction to the Theory of Neural Computation. Perseus Books (1991)
19. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: Sundials: Suite of nonlinear and differential/algebraic equation solvers. ACM Trans. Math. Softw. 31(3), 363–396 (2005), http://dblp.uni-trier.de/db/journals/toms/toms31.html#HindmarshBGLSSW05
20. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. Proc. Natl. Acad. Sci. USA 79, 2554–2558 (1982)
21. Hormuzdi, S., Filippov, M., Mitropoulou, G., Monyer, H., Bruzzone, R.: Electrical synapses: a dynamic signaling system that shapes the activity of neuronal networks. Biochim Biophys Acta 1662, 113–137 (2004)
22. Jülich Supercomputing Centre: JUQUEEN: IBM Blue Gene/Q® supercomputer system at the Jülich Supercomputing Centre. Journal of large-scale research facilities 1 (2015), http://dx.doi.org/10.17815/jlsrf-1-18
23. Kunkel, S., Schmidt, M., Eppler, J.M., Masumoto, G., Igarashi, J., Ishii, S., Fukai, T., Morrison, A., Diesmann, M., Helias, M.: Spiking network simulation code for petascale computers. Front. Neuroinformatics 8, 78 (2014), http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2014.00078/abstract
24. Lelarasmee, E.: The waveform relaxation method for time domain analysis of large scale integrated circuits: theory and applications. Memorandum p. No. UCB/ERL M82/40 (1982)
25. Morrison, A., Mehring, C., Geisel, T., Aertsen, A., Diesmann, M.: Advancing the boundaries of high connectivity network simulation with distributed computing. Neural Comput. 17(8), 1776–1801 (2005)