Run-Time Interoperability Between Neuronal Network Simulators Based on the MUSIC Framework

Mikael Djurfeldt · Johannes Hjorth · Jochen M. Eppler · Niraj Dudani · Moritz Helias · Tobias C. Potjans · Upinder S. Bhalla · Markus Diesmann · Jeanette Hellgren Kotaleski · Örjan Ekeberg

Published online: 2 March 2010

Abstract MUSIC is a standard API allowing large scale neuron simulators to exchange data within a parallel computer during runtime. A pilot implementation of this API has been released as open source. We provide experiences from the implementation of MUSIC interfaces for two neuronal network simulators of different kinds, NEST and MOOSE. A multi-simulation of a cortico-striatal network model involving both simulators is performed, demonstrating how MUSIC can promote inter-operability between models written for different simulators and how these can be re-used to build a larger model system. Benchmarks show that the MUSIC pilot implementation provides efficient data transfer in a cluster computer with good scaling. We conclude that MUSIC fulfills the design goal that it should be simple to adapt existing simulators to use MUSIC. In addition, since the MUSIC API enforces independence of the applications, the multi-simulation could be built from pluggable component modules without adaptation of the components to each other in terms of simulation time-step or topology of connections between the modules.

Keywords MUSIC · Large-scale simulation · Computer simulation · Computational neuroscience · Neuronal network models · Inter-operability · MPI · Parallel processing

Introduction

Large scale neuronal network models and simulations have become important tools in the study of the brain and the mind (Albus et al. 2007; Djurfeldt et al. 2008a). Such models work as platforms for integrating knowledge from many sources of data. They help to elucidate how information processing occurs in the healthy

Electronic Supplementary Material The online version of this article (doi:10.1007/s12021-010-9064-z) contains supplementary material, which is available to authorized users.

Johannes Hjorth and Jochen M. Eppler have contributed equally to the contents of the article.

M. Djurfeldt (✉) · J. Hjorth · J. Hellgren Kotaleski · Ö. Ekeberg
School of Computer Science and Communication, Royal Institute of Technology, 100 44 Stockholm, Sweden
e-mail: mikael@djurfeldt.com

M. Djurfeldt · M. Diesmann
RIKEN Brain Science Institute, Wako-shi, 351-0198 Saitama, Japan

J. M. Eppler
Honda Research Institute Europe GmbH, Carl-Legien-Straße 30, 63073 Offenbach, Germany

J. M. Eppler · M. Helias · M. Diesmann
Bernstein Center for Computational Neuroscience, Albert-Ludwigs-Universität Freiburg, Hansastraße 9A, 79104 Freiburg, Germany

N. Dudani · U. S. Bhalla
National Centre for Biological Sciences, Bangalore, India

T. C. Potjans
Institute of Neurosciences and Medicine, Research Center Jülich, 52425 Jülich, Germany

T. C. Potjans · M. Diesmann
RIKEN Computational Science Research Program, Wako-shi, 351-0198 Saitama, Japan

brain, while perturbations to the models can provide insights into the mechanistic causes of diseases such as Parkinson's disease, drug addiction and epilepsy. A better understanding of neuronal processing may also contribute to computer science and engineering by suggesting novel algorithms and architectures for fault tolerant and energy efficient computing (see, e.g., Schemmel et al. 2008). Simulations of increasingly larger network models are rapidly developing. In principle, we have, already today, the computational capability to simulate significant fractions of the mammalian cortex (Djurfeldt et al. 2008b). A great deal of effort has been put into the development of simulation software suites (see, e.g., Brette et al. 2007). Different software packages, such as NEURON (Carnevale and Hines 2006), GENESIS (Bower and Beeman 1998) and NEST (Gewaltig and Diesmann 2007) have been developed for simulations of neuron and network models.

Depending on the scientific question asked, or on the tradition in respective computational neuroscience lab, models of various parts of the brain have been formulated using different simulators. The positive side of this diversity is that it provides a repertoire of tools where different simulators have different strengths (see e.g. Brette et al. 2007, for a review of software for spiking neuron simulations). Diversity is also good for the strong ongoing development of simulation technology. On the negative side, the reuse of models is hampered by the lack of interoperability due to the multitude of languages and simulators used. Also, reimplementation of one model in other software is in practice both time consuming and error prone (personal experience, see also Cannon et al. 2007).

Interoperability can be facilitated in several ways. One approach is to provide a model specification in some standardized format which can be understood by many simulation tools. Two developments in this direction are PyNN and NeuroML. PyNN (Davison et al. 2009) is a common programming interface enabling model scripting in the Python programming language. PyNN already supports several simulators. The computational neuroscience community has built a growing toolbox around this environment for simulation and analysis of data. NeuroML is an XML-based standard for the description of model components at various levels of the nervous system which also allows models to be described in a simulator-independent way (Crook and Howell 2007; Crook et al. 2007). Another approach, *run-time interoperability* (Cannon et al. 2007), is to allow different simulation tools to communicate data online. MUSIC (Ekeberg and Djurfeldt 2008, 2009) is a standard interface for on-line communication between simulation tools. The MUSIC project was initiated by the INCF (International Neuroinformatics Coordinating Facility, `http://www.incf.org`) as a result of the 1st INCF Workshop on Large Scale Modelling of the Nervous System (Djurfeldt and Lansner 2007). A demonstration of MUSIC's capability to couple models was presented at the INCF booth at the Society for Neuroscience Conference in Washington 2008.

Here we report on our experiences and insights from connecting two preexisting models of very different kinds; one cortical network model using integrate-and-fire units and one striatal network based on biologically detailed units. The cortical network model was implemented in NEST while the striatal network model was developed using GENESIS, but simulated here in MOOSE. By adding a MUSIC interface to each simulator and connecting them using MUSIC, we could simulate the two systems together as one multi-simulation. Connecting the two models was interesting in its own right, but this would also serve as a realistic test of how hard it is to actually achieve interoperability between two independently developed models using the new MUSIC framework.

MUSIC

MUSIC is a recently developed standard for run-time exchange of data between MPI-based parallel applications in a cluster environment (Ekeberg and Djurfeldt 2009) so that any MUSIC-compliant tool may work out-of-the-box with another. A pilot implementation was released during 2009. The standard is designed specifically for interconnecting large scale neuronal network simulators, either with each-other or with other tools. The data sent between applications can be either event based, such as neuronal spikes, or graded continuous values, for example membrane voltages.

The primary objective of MUSIC is to support *multi-simulations* where each participating application itself is a parallel simulator with the capacity to produce and/or consume massive amounts of data. Figure 1 shows a typical multi-simulation where three applications, A, B, and C, are exchanging data during runtime.

MUSIC promotes *interoperability* by allowing models written for different simulators to be simulated together in a larger system. It also enables *reusability* of models or tools by providing a standard interface. The fact that data is spread out over a number of processors makes it non-trivial to coordinate the transfer of data so that it reaches the right destination at the right time. When applications are connected in loops, timing of communication also becomes complex. The task for

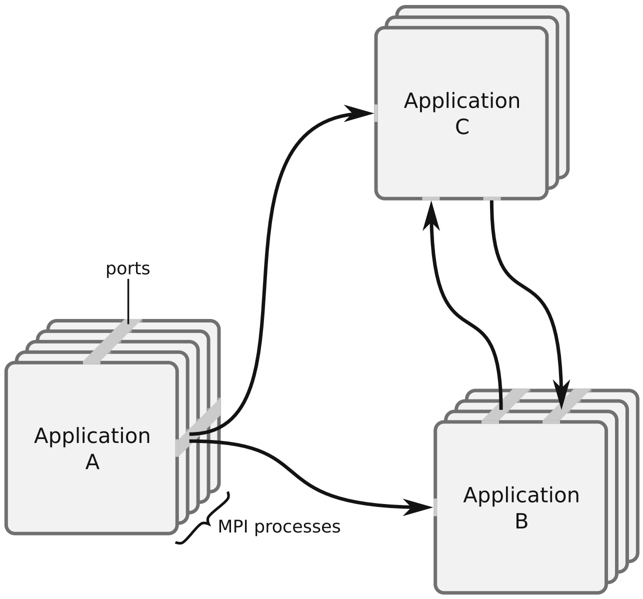


Fig. 1 Illustration of a typical multi-simulation using MUSIC. Three applications, *A*, *B*, and *C*, are exchanging data during runtime. Each application runs in a set of MPI processes. Data flows exit and enter *ports*, each spanning the set of processes of the application

MUSIC is to relieve the applications from handling this complexity.

The MUSIC pilot implementation consists of header files, a library and utilities. A MUSIC-compliant application is compiled against the MUSIC header files, linked with the MUSIC library and launched by the MUSIC launch utility (named `music`). Currently, an application needs to be compiled in a system where the full MUSIC implementation is installed. However, the resulting binary may be dynamically linked against other revisions of the MUSIC library. In a future release, standard header files can be separated so that these can be shipped together with the application.

The pilot implementation has been designed to run smoothly on state-of-the-art high-performance hardware. The software is written in C++, which is the de facto standard for current high-end hardware, and has been tested on simple multi-core machines up to massively parallel supercomputers such as the IBM Blue Gene/L. It can be automatically configured (GNU autotools) and compiled over the range of architectures tested, including 32- and 64-bit Intel-/AMD-based clusters and the Blue Gene/L.

At the beginning of this project, no simulators had been adapted to use MUSIC. In a collaborative effort, developers from the NEST and MOOSE communities worked together with the MUSIC developers in adapting these two simulators for use in a multi-simulation environment.

Phases of Execution

A multi-simulation, i.e. a set of interconnected parallel applications, is described by a MUSIC *configuration file* and executed in three distinct phases. From the simulator developers' point of view, these phases are clearly separated through the use of two main components of the MUSIC interface: the `Setup` and the `Runtime` objects.

Launch is the phase where all the applications are started on the processors. During this phase, MUSIC is responsible for interpreting the configuration file and launching the application binaries on the set of MPI processes allocated to the MUSIC job. Since MPI can be initialized only after the applications have been launched, most of this work needs to be performed outside of MPI. In particular, the tasks of accessing the command line arguments of the MUSIC launch utility and of determining the ranks of processes before MPI initialization therefore have to be handled separately for different MPI implementations.
Technically, the launch phase begins when `mpirun` launches the MUSIC launch utility and ends when the `Setup` object constructor returns. The Setup object is used for initialization and configuration and replaces the call to `MPI::Init`. (See further description below.)

Setup is the phase when all applications can publish what ports they are prepared to handle along with the time step they will use and where data will be present (where in memory and/or on what processor). During the setup phase, applications can read configuration parameters communicated via the common configuration file. At the end of the setup phase, MUSIC will establish all connections.
The setup phase begins when the `Setup` object has been created and ends when the `Runtime` object constructor returns.

Runtime is the phase when simulation data is actually transferred between applications. Via calls to `Runtime::tick()` the simulated time of the applications is kept in a consistent order.

The runtime phase begins when the `Runtime` object has been created and ends when its `finalize`() method is called.

When the application initializes MUSIC at the beginning of execution it receives the `Setup` object. This object gives access to the functionality relevant during the setup phase via its methods. When done with the setup, the application program makes the transition to the runtime phase by passing the `Setup` object as an argument to the `Runtime` object constructor which destroys the `Setup` object. The `Runtime` object provides methods relevant during the runtime phase of execution.

MUSIC requires that the application uses a communicator handed to it from the `Setup` object rather than using `MPI::COMM_WORLD` directly. This intra-communicator is used by the application to communicate within the group of processes allocated to it by MUSIC during launch, while the MUSIC library will internally use inter-communicators for communication of data between MUSIC ports. When a MUSIC-aware application is launched by `mpirun` instead of the MUSIC launch utility, the communicator returned from the `Setup` object will be identical to `MPI::COMM_WORLD`.

Communicating via MUSIC

In order to communicate via MUSIC, each participating application must be interfaced to the MUSIC API. One design goal of MUSIC has been to make it easy to adapt existing simulators. In most cases, it should be possible to add MUSIC library support without invasive restructuring of the existing code. The primary requirements on an application using MUSIC is that it declares what data should be exported and imported and that it repeatedly calls a function at regular intervals during the simulation to allow MUSIC to make the actual data transfer.

Ports and Indices

Communication between applications is handled by *ports*. Ports are named sources (output ports) or sinks (input ports) for the data flow. In the current MUSIC API, there are three kinds of ports: Continuous ports communicate multi-dimensional continuous time-series, for example membrane voltages. Event ports communicate time-stamped integer identifiers, for example neuronal spikes. Message ports communicate message strings, for example a command in a scripting language. The data to be communicated may be differently organized in process memory on the receiver side compared to the sender side. The applications may run on different numbers of processes, and, the data may be differently distributed among the sender processes and the receiver processes, as is shown in Fig. 2. How does MUSIC know which data to send where?

In MUSIC, there are two views of the data to be communicated over a connection. Data elements are enumerated differently according to these views. MUSIC uses *shared global indices* to enumerate the entire set of data to be sent over the connection while *local indices* enumerate the subset of data which is stored in the memory of a particular MPI process. Data does not need to be ordered in the same way according to the two views. For example, data stored in an array may be associated with an arbitrary subset of global indices in an arbitrary order.

The MUSIC library is informed about the relationship between global and local indices and how data is stored in memory during the setup phase. Two abstractions are used to carry this information:

The `IndexMap` maps local indices to global indices. That is, the `IndexMap` tells which parts of a distributed data array are handled by the local process and how the data elements are locally ordered.

The `DataMap` encapsulates how a port accesses its data. The `DataMap` contains an `IndexMap`. While an

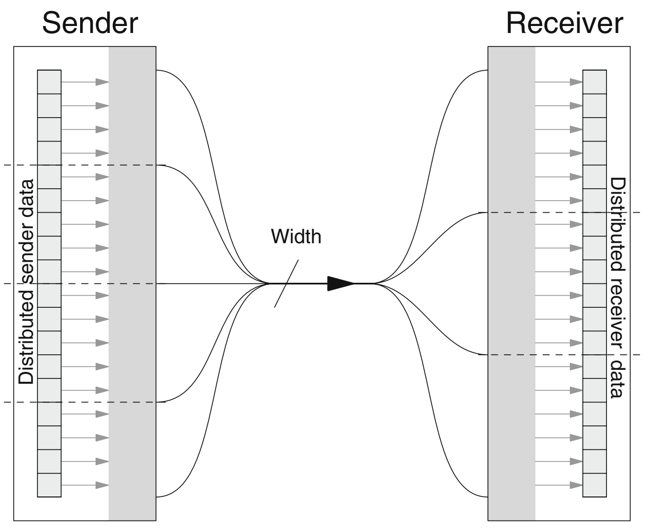


Fig. 2 Data transfer over a connection from an application running in four processes to an application running in three processes. The *light gray areas* in the sender and receiver represent the MUSIC port. *Dashed lines* divide the application into distinct processes. The *width* of the port is the total number of distinct data items being communicated from all sender processes to the receiver processes

index map is a mapping between two kinds of indices, the data map also contains information about where in memory data resides, how it is structured, and, the type of the data elements. The type is used for marshalling when running on a heterogeneous cluster.

During setup every process of the application individually provides the port with a `DataMap` (or an `IndexMap` in the case of event ports).

Events

Since event ports don't access data the same way as ports for continuous data, they do not require a full `DataMap`. Instead, an `IndexMap` is used to describe how indices in the application should be mapped to the shared global indices common to sender and receiver. The application is given the choice of using local indices or bypassing the index transformation by directly using the shared global indices when labelling events.

An event is a pair of an index identifier, either a shared global index or a local index, and a double-precision floating point time-stamp. The index usually refers to the source neuron generating a spike event. Events are given to MUSIC by the sending application through a call to the method `insertEvent`() on the port and delivered to the receiving application by MUSIC through an *event handler*. The event handler is a C++ functor given to MUSIC by the application before the simulation starts. The event handler is called by MUSIC when the application calls `tick`(). It is called once for every event delivered.

Some spiking neuronal network models include axonal delays. The MUSIC framework assumes that handling and delivery of delayed spikes occurs on the receiver side. In such a case, the receiver may allow MUSIC to deliver a spike event later than its time stamp according to local time. This *maximal acceptable latency* can be specified for a port during setup.

Application Responsibilities

One goal of MUSIC has been to limit the responsibilities imposed on each application. Here we present a step-by-step list of what an application must do in order to participate in a multi-simulation.

1. **Initiate MUSIC**
   This is done by calling the `Setup` constructor.
2. **Publish ports**
   Data available to be imported and exported is identified by creating named ports.
3. **Map ports**
   MUSIC is informed about where the actual data is located. This includes information about which processor owns each data element. For continuous data it also includes information about where in memory it is stored, while for event data it specifies how to receive events.
4. **Initiate the runtime phase**
   This is done by calling the `Runtime` constructor. At this stage, MUSIC can build the plan for communication between different processes.
5. **Advance simulation time**
   The application must call `tick`() at regular intervals to give MUSIC the opportunity to transfer data.
6. **Finalize MUSIC**
   By calling `finalize`(), all MUSIC communication is terminated.

Pre- and Post-processing

The MUSIC framework provides a uniform interface to access data from various simulators. This allows the development of pre- and post-processing tools, for example for data analysis or visualization, that are independent of data sinks or sources so that they can be re-used in different multi-simulations.

This is exemplified here by a visualizaton tool written for INCF's MUSIC demonstration at the Society for Neuroscience Conference 2008 in Washington. The tool receives events from a MUSIC event port and displays these as changes in size and color of a set of 3D spheres resembling the neurons of a neuronal network. The demonstration is described in Section "A MUSIC Multi-simulation with NEST and MOOSE" and the graphics window of the visualization tool shown in Fig. 13. The communication between the simulator and the visualization tool is set up using the MUSIC configuration file. The visualization geometry, neuron sizes and colors are specified in a separate configuration file. The camera position is automatically adjusted so that all neurons are visible.

MUSIC allows simulators to run independently of each other, in so far as one model might run ahead of another if there is only unidirectional communication between them. To cope with this, the visualization maintains its own internal clock which is a scaled version of the wall-clock time. For example, the visualization can be configured to display the simulation 100 times slower than real time. This makes the visualization independent of the relative execution time of the simulators.

Adapting NEST to MUSIC

NEST (Gewaltig and Diesmann 2007) is a simulator for heterogeneous networks of point neurons or neurons with a small number of electrical compartments. The focus of NEST is on the investigation of phenomena at the network level, rather than on the simulation of detailed single neuron dynamics.

NEST is implemented in C++ and can be used on a wide range of architectures from single- and multi-core desktop computers to super-computers with thousands of processors. It has a built-in simulation language interpreter (SLI), but can also be used from the Python programming language via an extension module called PyNEST. In this article, we use the PyNEST syntax to show the usage of the MUSIC interface in NEST. As Python does not support MPI enabled extensions out of the box, a small launcher script has to be used (see Section B in the online supplementary material). For details on PyNEST and its API, see Eppler et al. (2009).

Implementation of the NEST-MUSIC Coupling

Event sources and sinks that are located in remote MUSIC applications are represented by *proxy* nodes inside of NEST. Two separate classes of proxies are used for inbound and outgoing connections. They are derived from the base class `Node`. This means that they are created and can be connected in the network graph like all other nodes. See online supplementary material, Section A, for a more detailed description of the network representation in NEST.

To make it easier to distinguish *global ids* (NEST's identifiers for nodes) from *global indices* (MUSIC's identifiers for connections on a port, see Section "Ports and Indices"), we use the term *channel* for the concept from MUSIC in the following description and in our implementation.

Three new classes were implemented to exchange events with MUSIC. In addition, several of the existing classes were extended by data structures and algorithms for the necessary book keeping during setup and runtime phase. The following sections contain a description of the components that are involved in the MUSIC interface. See online supplementary material, Section D, for sequence diagrams that explain the interaction of the components.

Sending Events from NEST to MUSIC

The class `music_out_proxy` represents a MUSIC output port and all associated channels in NEST (see Fig. 3). It forwards the events of arbitrarily many nodes to remote MUSIC targets. One instance of this proxy is created in each NEST process for each MUSIC output port.

The name of the corresponding MUSIC output port is set as parameter `port_name` using `SetStatus()`:

```
outproxy = Create('music_out_proxy')
SetStatus(outproxy, {'port_name':
                     'spikes_out'})
```

The events of a node are forwarded to the MUSIC channel that is specified by the parameter `music_channel` during connection setup. It cannot be changed, once the connection is set up. Note that it is not allowed to connect several nodes to the same channel. The following example shows how the connections of five neurons to a MUSIC port are set up:

```
neurons = Create('iaf_neuron', 5)
Connect([neurons[0]], outproxy,
        {'music_channel': 0})
Connect([neurons[1]], outproxy,
        {'music_channel': 1})
Connect([neurons[2]], outproxy,
        {'music_channel': 2})
Connect([neurons[3]], outproxy,
        {'music_channel': 3})
Connect([neurons[4]], outproxy,
        {'music_channel': 4})
```

During connection setup in NEST, the sender checks its compatibility with the receiver by calling its `connect_sender()` function. The first argument for this function is an event of the type the sender wants to send during simulation, which is only used to select the correct variant of `connect_sender()`. The second

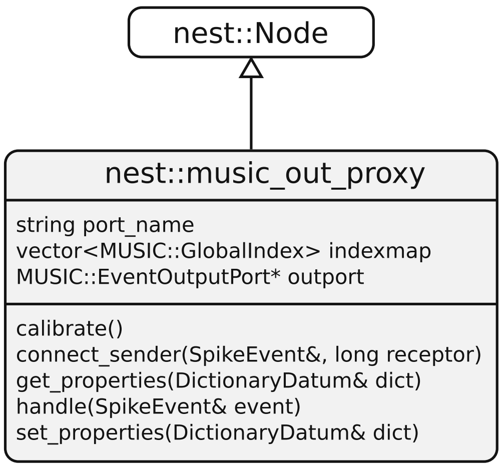
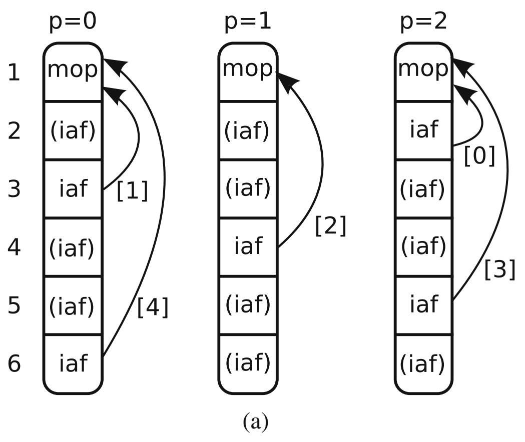

nest::Node
nest::music_out_proxy
string port_name
vector<MUSIC::GlobalIndex> indexmap
MUSIC::EventOutputPort* outport
calibrate()
connect_sender(SpikeEvent&, long receptor)
get_properties(DictionaryDatum& dict)
handle(SpikeEvent& event)
set_properties(DictionaryDatum& dict)

Fig. 3 The UML diagram shows the data members and the functions of the proxy that represents MUSIC output ports in NEST. The new class is shown in *grey*



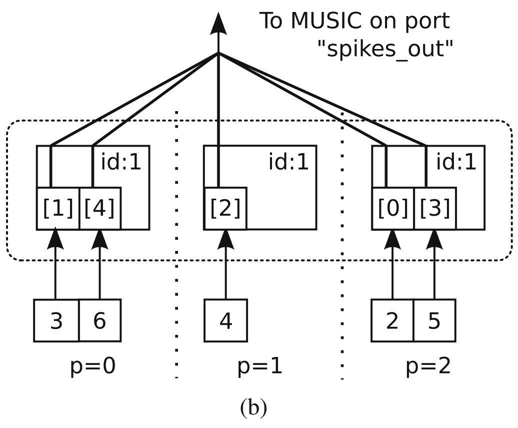


Fig. 4 **a** Nodes in NEST are distributed over the processes ($p = 0, 1, 2$). iaf denotes an integrate and fire neuron, (iaf) denotes a proxy. mop denotes a `music_out_proxy`. MUSIC channels are indicated in *square brackets* for each connection (*arrows*). **b** A sketch of the complete connectivity from the nodes (*lower squares*) over the different channels (*numbers in square brackets*) to MUSIC. The *dashed box* encloses all proxies that belong to one MUSIC output port

argument is an integer which specifies the channel the source wants to connect to, and is used to build the `indexmap`, a list that registers all channels that have to be mapped with MUSIC. The `indexmap` is built separately by each process and therefore only contains local channels.

Figure 4 shows the network in NEST after the above commands were executed using three NEST processes.

Before NEST tells MUSIC to enter the runtime phase, the port has to be mapped. This is done in `calibrate`(), which is called by NEST's scheduler on each node before the start of the simulation. This function executes the following steps:

1. Create a `MUSIC::EventOutputPort`, `outport`. This will trigger an exception if the port name is already used.
2. Create a `MUSIC::PermutationIndex` and initialize it with the data from the `indexmap`. The `PermutationIndex` informs MUSIC about the channels present on a specific process.
3. Use the `PermutationIndex` to map all local channels by calling the function `map`() on `outport`.

Events that are delivered to the proxy are passed to its `handle`() function with the event as argument. This function forwards the spikes directly to the MUSIC output port object `outport`.

Note that as the `music_out_proxy` only acts as a proxy for nodes in NEST, it does not take into account the delay of incoming connections. Synaptic interactions have to be set up in the receiving application.

Receiving Events from MUSIC in NEST

In contrast to MUSIC output ports, which are represented by single `music_out_proxys` in NEST, inbound connections require two classes: The MUSIC input port is represented by the class `MusicEventHandler` (see Fig. 5). One `MusicEventHandler` is created for each MUSIC

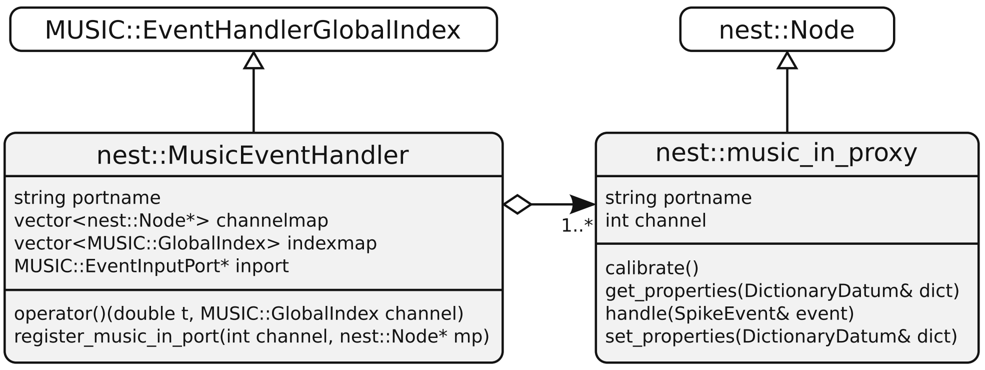

MUSIC::EventHandlerGlobalIndex
nest::Node
nest::MusicEventHandler
string portname
vector<nest::Node*> channelmap
vector<MUSIC::GlobalIndex> indexmap
MUSIC::EventInputPort* inport
operator()(double t, MUSIC::GlobalIndex channel)
register_music_in_port(int channel, nest::Node* mp)
1..*
nest::music_in_proxy
string portname
int channel
calibrate()
get_properties(DictionaryDatum& dict)
handle(SpikeEvent& event)
set_properties(DictionaryDatum& dict)

Fig. 5 The UML diagram shows the data members and the functions of the proxy that represents a channel on a MUSIC input port in NEST and its relation to the class that represents the MUSIC input port. New classes are shown in *grey*

input port in each of NEST's processes. Each channel on the port is represented by a separate `music_in_proxy` (see Fig. 5). The reason for this is that NEST's connection mechanism cannot handle different signal origins, but only different target locations on a node. This means that we cannot specify the MUSIC channel during connection setup to a single proxy (cf. Section "Sending Events from NEST to MUSIC"), but we need to set it separately as a parameter for the `music_in_proxy`, which should receive the respective input.

The `MusicEventHandler` maintains a `channelmap`, which maps the global MUSIC channel id to the address of the corresponding proxy. The `channelmap` is built incrementally during the registration of channels by `register_channel`().

Spike sources in remote MUSIC applications are represented in NEST by instances of class `music_in_proxy`. Each instance listens to exactly one channel on a MUSIC input port. This means that several proxies listen to the same port, but to different channels.

After the creation of the proxy, the port name and the channel are set using `SetStatus`(). The port name defaults to *spikes_in* for all `music_in_proxy`s.

```
in_proxies = Create('music_in_
                       proxy', 2)
SetStatus([in_proxies[0]],
          {'music_channel': 0})
SetStatus([in_proxies[1]],
          {'music_channel': 1})
```

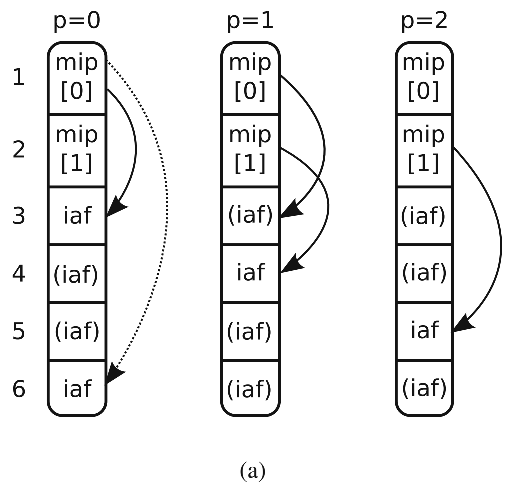
Connections from a `music_in_proxy` to other nodes can use any of NEST's built-in connection types. The following listing shows how connections are established using the high-level connection routine `DivergentConnect`() and the basic connection command `Connect`():

```
neurons = Create('iaf_neuron', 4)
DivergentConnect([in_proxy[0]],
                 [neurons[0], neurons[1]])
DivergentConnect([in_proxy[1]],
                 [neurons[1], neurons[2]])
Connect([in_proxy[0]], [neurons[3]],
        model='stdp_synapse')
```

Figure 6 shows the network representation after the above commands were executed in a setup with three NEST processes.

As the proxy itself does not know about MUSIC, we use an indirection via the `Network` class to register the proxy with the event handler for the corresponding port.

In its `calibrate`() function, the proxy registers itself with its channel index and port name with the `Network` class by calling `register_music_in_proxy`(). The network class maintains a mapping of port names to MUSIC event handlers to efficiently find the right one or create a new instance for unknown ports if an input proxy is registrered. Before the start of the simulation, all known input ports are mapped.



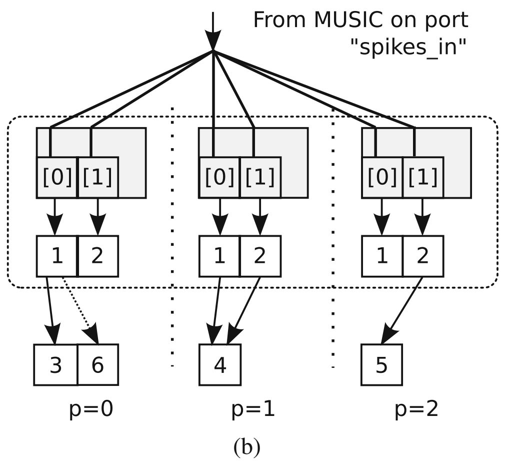


Fig. 6 **a** Nodes in NEST are distributed over the processes ($p = 0, 1, 2$). iaf denotes an integrate and fire neuron, (iaf) denotes a proxy. mip denotes a `music_in_proxy`. The *numbers on the left* indicate the global id of the nodes. MUSIC channel ids are indicated in *square brackets* for each `music_in_proxy`. The STDP connection is indicated by a *dashed arrow*. **b** A sketch of the complete connectivity from MUSIC (*channels in square brackets*) to the MUSIC event handler (*grey rectangles*) to the proxies (*squares labeled 1 and 2*) to the actual target nodes (*lower squares*). The STDP connection is indicated by a *dashed arrow*. The *dashed box* encloses all event handlers and proxies that represent a MUSIC input port

For each incoming spike, MUSIC calls `operator()` on the event handler with the time of the spike and the target channel as arguments. `operator()` creates a new `SpikeEvent` object and passes it directly to the `handle()` function of the proxy associated with the channel. This bypasses the synapse system in NEST and only informs the proxy about a new spike in a remote application. Upon arrival of new events, the `handle()` function immediately calls `Network::send()` to deliver the event to all local targets via the synapse system.

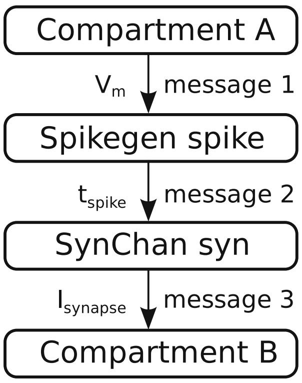


Fig. 7 Illustration of the MOOSE messaging structure. Two compartments are connected by a synapse

Adapting MOOSE to MUSIC

MOOSE (Multiscale Object Oriented Simulation Environment, available at http://moose.ncbs.res.in) is a simulator which enables the development and simulation of biologically detailed models of neuronal and biochemical networks. It is a multiscale simulator, as it lets a modeller build a model by coupling components from different levels of detail—from single molecules to whole neurons. It achieves this by coordinating calculations between specialized numerical engines which are suited for each level of detail.

To discuss how MUSIC compatibility was added to MOOSE, it will be helpful to outline how MOOSE functions. MOOSE inherits an object-oriented framework from the GENESIS simulator (Bower and Beeman 1998) for describing models and simulations. In this framework, the user sets up a simulation by putting together the right building blocks ("MOOSE objects"), which are instances of the respective MOOSE classes.

Objects in MOOSE communicate with each other by means of "messages". A message is a persistent connection between two objects, which allows them to exchange information during a simulation. An example of messaging is shown in Fig. 7 which depicts how one can model synaptic transmission in MOOSE. A `SpikeGen` object called `spike` monitors the membrane potential `Vm` of the presynaptic compartment `A`, via the message labelled `message 1`. When this membrane potential crosses a certain threshold, `spike` interprets it as an action potential and sends the spike time to the `SynChan` (short for "Synaptic Channel") object called `syn`. This triggers the opening of the synaptic channel, and `syn` sends the synaptic current to the postsynaptic compartment `B`, via the message `message 3`.

MOOSE provides a user- and developer-friendly framework to run parallel simulations on a cluster. It hides MPI-based communication behind an interface so that sending and receiving information to and from foreign objects looks the same to the developer as with local objects. For the user, the design is such that a serial simulation script can be run in parallel right away, without any changes. In particular, for inspecting and manipulating objects and their fields and messages, the same script commands work in serial and in parallel operation. During object creation, a load-balancer decides which process the object should be created on.

Implementation of the MOOSE-MUSIC Coupling

New Classes

Five new MOOSE classes were created to allow MOOSE to exchange spike times with MUSIC:

- `Music`—This is a singleton class with exactly one instance automatically created at the start of a MOOSE session.
  This object is responsible for making most of the basic MUSIC API calls in the correct order. This includes appropriate initialization and finalization by managing the MUSIC `Setup` and `Runtime` objects. Also, during a simulation, this object calls MUSIC's `tick()` function periodically, separated by a user-specified time interval.
  While this `Music` object carries out the above without user intervention, it also provides an interface which the user can use to create new MUSIC ports for sending and receiving spike-event information.
- `InputEventPort`—An instance of this class is created when the user calls a function of the above `Music` class to declare readiness to receive spike-event information. Upon creation, this object finds out the width of the corresponding MUSIC port by making a MUSIC API call. A corresponding

number of instances of the `InputEventChannel` class (described next) are then created.

- `InputEventChannel`—Instances of this class act as proxies within MOOSE of the spike-generating entities in the sending application. They receive spike-time information relayed by MUSIC and recreate the original spike-train by emitting the spike-times locally. Hence, within MOOSE, they appear as bona fide spike-generating objects which can connect to, e.g. a `SynChan` object, and send spike messages just like a `SpikeGen` object can.
- `OutputEventPort`—This class is analogous to the `InputEventPort`, and is instantiated by the user when MOOSE should act as a spike-generating application. Like before, an instance of this class creates the same number of instances of the following `OutputEventChannel` as is its own width.
- `OutputEventChannel`—Objects of this class can receive spike-time messages from other MOOSE objects, like a `SynChan` object can. Upon receiving a spike, an `OutputEventChannel` object passes it on to MUSIC, which forwards it to interested applications.

Note that if the user creates a port of width m in a parallel simulation with n processes, then m channel objects (i.e., instances of `InputEventChannel` or `OutputEventChannel`) will have to be distributed among the n processes. An algorithm is built into the port classes (i.e., `InputEventPort` and `OutputEventPort`) to carry out this distribution, without the need for the user's knowledge. At present, this algorithm is simple: the list of m is divided into n blocks of size approximately equal to m/n, and channels within each block are created on a separate node. In the future, this algorithm can be improved by placing the channel objects in the same process as the objects they connect to.

Interfacing with MUSIC in MOOSE

With the above classes at hand, it is simple for a user to incorporate MUSIC sources and sinks in a simulation. The user carries out the following steps to set up MOOSE-MUSIC communication:

- Specifying `tick`() rate—The user provides a time interval which is used to call MUSIC's `tick`() function periodically. This is done by setting up a "clock" with the desired time interval as its clock-rate, and attaching it to the instance of the singleton `Music` class. See online supplementary material, Section E, for an example MOOSE script, which has commands to carry out all steps mentioned here.
- Adding ports—The user declares the ports through which MOOSE can receive and send data via MUSIC. One script command has to be issued for every port added.
- Connecting MUSIC with the model—With the above two steps done, the user has MUSIC sources and sinks available as native MOOSE objects. From here on, it is intuitive for a user to route MUSIC-originating and MUSIC-destined data to- and from desired entities in a model. This is done by simply adding messages, in the usual MOOSE fashion, between objects representing MUSIC, and objects constituting the model. Note that in adding a message, the user need not do anything special if the source and destination objects are situated in different processes, since MOOSE will carry

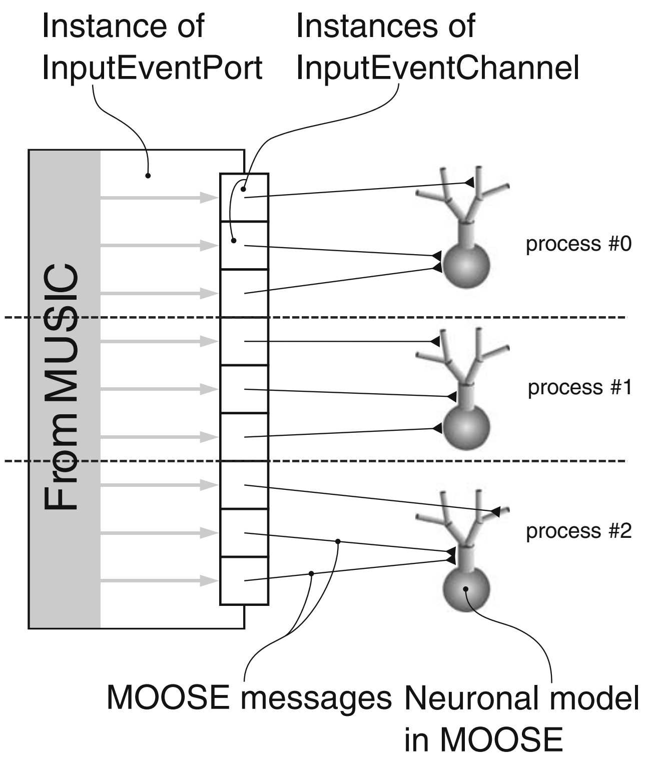


Fig. 8 A model in MOOSE receiving spike-time information from MUSIC. An object of type `InputEventPort` handles spike-times relayed by MUSIC. Objects of type `InputEventChannel` act as proxies for the spike-generating entities in the foreign application. The proxies forward the spikes to targets in the model via messages. Note that it is possible for a message to connect a proxy and its target even if both are in separate processes. It is most efficient, however, if they are on the same process

out the correct setup internally. This situation is depicted in Fig. 8.

Performance and Application

The adaptation of NEST and MOOSE to MUSIC allows us to test the performance of MUSIC and to apply the framework to a multi-simulation connecting two very different models. We test the performance of MUSIC in two typical multi-simulation examples: (1) an asymmetric multi-simulation benchmark with one large-scale model that is connected bidirectionally to a second program that runs on a single process (see Fig. 9a) and (2) a symmetric multi-simulation benchmark with MUSIC connecting two large-scale models each running on multiple processes (see Fig. 9b). For simplicity, we use NEST for the benchmarks presented here. As a complete application example, we present a multi-simulation that connects two very different models: a cortical network model based on integrate-and-fire units in NEST and a striatal network model based on multi-compartmental units with Hodgkin-Huxley formalism in MOOSE.

In Section "Benchmarking MUSIC with a Cortical Network Model in NEST" we describe consecutively the model definition and the performance of the cortex model, the asymmetric multi-simulation benchmark and the symmetric multi-simulation benchmark. Section "A MUSIC Multi-simulation with NEST and MOOSE" then describes the multi-simulation connecting the cortex and the striatum network and shows simulation results.

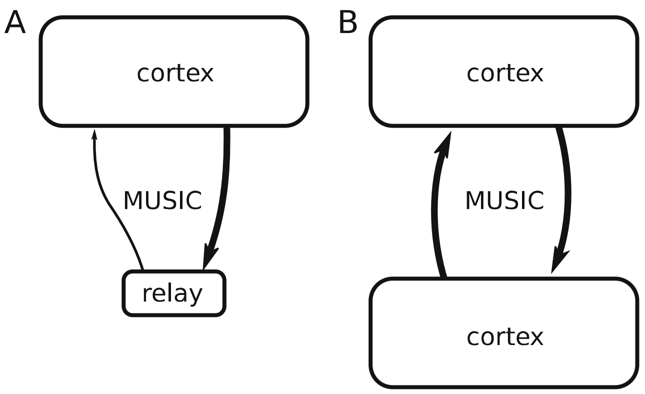


Fig. 9 Benchmark models. **a** Asymmetric benchmark model consisting of one large-scale cortex model and a single process relay model. Inter-model communication via MUSIC is bidirectional but asymmetric, mainly from the cortex model to the relay model. **b** Symmetric benchmark model consisting of two interconnected large-scale cortex models. Communication via MUSIC is symmetric between the two models

Benchmarking MUSIC with a Cortical Network Model in NEST

We use a layered cortical network model (Potjans and Diesmann 2008) in NEST in order to assess the performance of MUSIC for large-scale simulations. It consists of 80,000 integrate-and-fire units divided among four layers (2/3, 4, 5 and 6) and around 0.3 billion synapses. Each layer contains one excitatory (e) and one inhibitory (i) population. Populations are connected randomly with layer- and type-specific connection probability. In all benchmark simulations, we use static synapses. The integration step size is 0.1 ms and the minimal delay in the network is `min_delay` = 0.8 ms. The network exhibits asynchronous irregular activity with layer- and type-specific firing rates for stationary, homogeneous background input. The mean firing rates range from below 1 Hz to maximally 8 Hz (Potjans et al. 2009).

Performance of the Cortex Model

From the computational perspective, simulating this model is a rather lightweight job on modern compute clusters. We simulate the model with NEST on a ×86 cluster consisting of 23 nodes: each node is equipped with two AMD Opteron 2834 Quad Core processors with 2.7 GHz clock speed and running Ubuntu Linux. The nodes are connected via InfiniBand; the MPI implementation is OpenMPI 1.3.1. Our simulation setup first distributes the processes to nodes, resulting in a single process per machine and InfiniBand port up to 23 processes. Figure 10a shows the computing time per second of biological time as a function of the number of cores on this system: black squares show the data for the default installation of NEST, gray diamonds when linking NEST during compilation to MUSIC. The overlap of the data points shows that the performance of NEST is not impeded when using the MUSIC communicator; the performance is the same when NEST is compiled by default with the configure switch `−−with−music`. In both cases, the simulation time of the layered cortex model scales supralinearly up to 20 cores and linearly up to 24 cores, yielding a simulation time of only 8 s per second of biological time. A further increase of the number of processors still improves the simulation time; using 48 cores results in a simulation time of around 5 s. The suboptimal scaling when increasing the number of processes from 24 to 32 is due to limited memory bandwidth that comes into play when multiple processes run on a single compute node. Beyond 32 processes the scaling is again close to optimal linear scaling. This simulation represents the

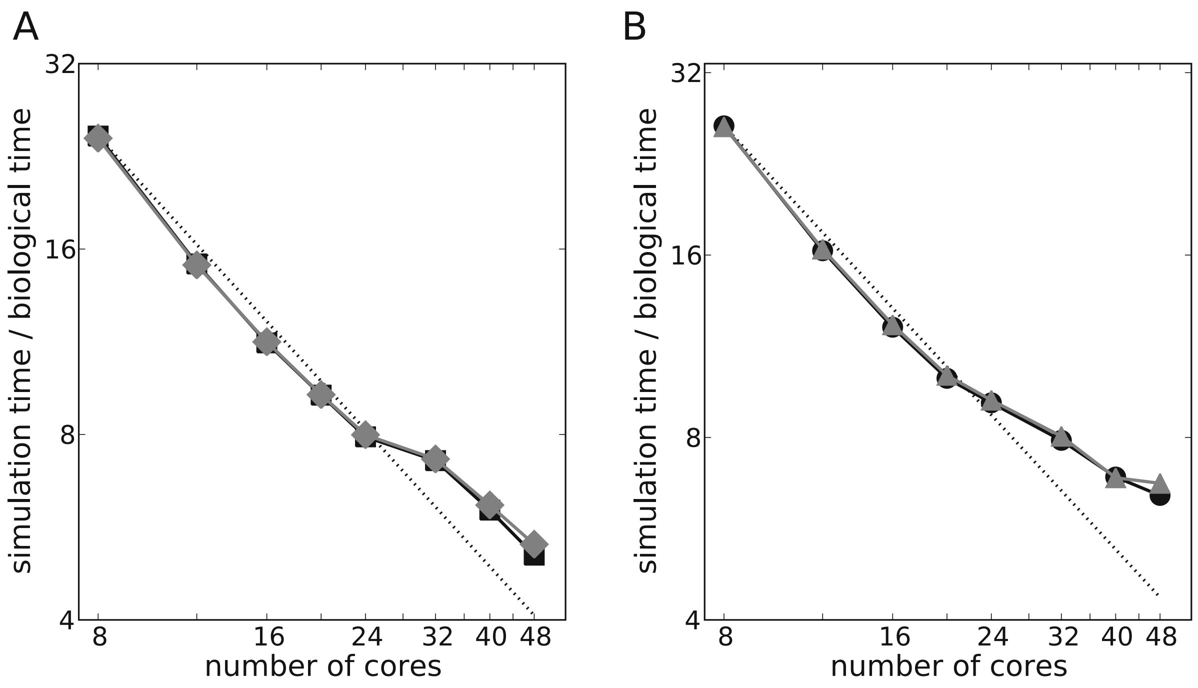


Fig. 10 Performance of the layered cortical network model and the asymmetric multi-simulation benchmark. **a** Computing time per second of biological time as a function of the number of compute cores. *Gray diamonds* show the performance of the cortex model simulation when NEST is compiled without MUSIC, *black squares* the performance when compiled with MUSIC. The *dotted line* indicates the expectation for linear speed-up. **b** Computing time per second of biological time of the asymmetric multi-simulation benchmark. The number of cores corresponds to the number of cores used for the cortex model without the additional core for the relay network. The shown data corresponds to $N_{\mathrm{MUSIC}} = 8$ (*black circles*) and $N_{\mathrm{MUSIC}} = 71,000$ (*gray triangles*); the *dotted line* gives the expectation for linear speed-up

control simulation for the asymmetric multi-simulation benchmark.

Asymmetric Multi-simulation Benchmark

The asymmetric multi-simulation benchmark consists of two models implemented in NEST: the cortex model and a basic relay model (see Fig. 9a). The models are coupled bidirectionally via MUSIC, i.e. both models send/receive spike events to/from the other model. Basically, we record spikes from any population in the layered network and transmit them to the relay model. The relay model takes few of the transmitted spikes and sends them back to the sender population. The communication is layer- and type-specific: we transmit the spikes to/from any population in the cortex model separately.

The relay model is kept minimal. It consists of one `parrot_neuron` for every population in the cortex model; this neuron immediately emits a spike for every spike it receives. The `parrot_neuron`s receive a subset of the spike trains transmitted from its population in the cortex model from the corresponding `music_in_proxy`s in the relay model and sends its spikes to the corresponding `music_out_proxy`. Therefore we always have a fixed number of eight MUSIC channels transmitting spikes from the relay model to the cortex model.

The implementation of the benchmark requires changes to the cortex model script and the relay script. But as the communication with MUSIC is carried out by nodes—`music_out_proxy`s and `music_in_proxy`s—the multi-simulation NEST scripts do not differ fundamentally from scripts describing stand-alone simulations. We create and connect `music_out_proxy`s and `music_in_proxy`s for any population in the model as described in Sections "Sending Events from NEST to MUSIC" and "Receiving Events from MUSIC in NEST", respectively. In addition, we have to set the acceptable latency with the `SetAcceptableLatency`() command. Care has to be taken in order to arrive at consistent parameters in the NEST scripts and the corresponding MUSIC script of the multi-simulation. On the level of a multi-simulation, these parameters are the number of MUSIC channels per population going from the cortex model to the relay model and vice versa—we call the total number of efferent MUSIC channels of the cortex model N_{MUSIC}.

On the level of the NEST scripts, we account for the asymmetry of a single `music_out_proxy` with many `music_channel`s per population on the one hand and many `music_in_proxy`s with the corresponding `music_channel`s on the other hand.

Altogether, the asymmetric multi-simulation benchmark extends the stand-alone cortex model by the following parameters:

- $N^{i,x}_{\mathrm{MUSIC}}$: number of efferents of the cortex model per population ($i \in \{2/3, 4, 5, 6\}$, $x \in \{\mathrm{e}, \mathrm{i}\}$). $N_{\mathrm{MUSIC}} = \sum_{i,x} N^{i,x}_{\mathrm{MUSIC}}$ corresponds to the number of MUSIC channels from the cortex model to the relay model and therefore also to the number of `music_in_proxy`s in the relay model
- $k^{i,x}_{\mathrm{MUSIC}}$: number of connections between the $N^{i,x}_{\mathrm{MUSIC}}$ `music_in_proxy`s and the corresponding `parrot_neuron` in the relay model.

Figure 10b shows the performance of the asymmetric multi-simulation benchmark for $N_{\mathrm{MUSIC}} = 8$, $k^{i,x}_{\mathrm{MUSIC}} = 1$ (black circles) and for $N_{\mathrm{MUSIC}} = 71,000$, $k^{i,x}_{\mathrm{MUSIC}} = 4$ (gray triangles). For better comparison, we give here the number of cores used for the cortex model, the relay model is simulated on one additional core.

We find that the additional costs due to the MUSIC interfaces and due to the communication of the two models via MUSIC are very small. The multi-simulation scales supralinearly up to 20 cores and the relative increase in simulation time is well below 10% of the simulation time of the control simulation without MUSIC. Further increasing the number of cores still improves the simulation time below 7 s; only when using 48 cores, the additional costs lead to an earlier onset of the saturation of the simulation time. The excellent performance holds for the minimal case where we only transmit the spikes of a single neuron from every population, but also when transmitting almost all spikes created by the cortex model: We do not observe a dependence of the number of MUSIC channels/the number of transmitted spikes for this benchmark.

Symmetric Multi-simulation Benchmark

The symmetric multi-simulation benchmark increases the demands on software and hardware considerably. It consists of two reciprocally connected cortex models (see Fig. 9b). Each model connects, as in the asymmetric multi-simulation benchmark, via in total N_{MUSIC} MUSIC channels to the other model. The incoming spike trains project on $k^{i,x}_{\mathrm{MUSIC}}$ neurons of the corresponding population. We choose very low synaptic weights for the connections between the two models in order to not interfere with the dynamics of the layered network. The random connectivity of the networks requires MUSIC to route events not only between single machines but rather in an all-to-all fashion.

The implementation of this benchmark does not require any changes to the cortex model script with respect to the asymmetric multi-simulation benchmark. The only changes affect the MUSIC script, configuring two interconnected and equally sized NEST simulations of the same model.

Figure 11a shows the performance of the symmetric multi-simulation benchmark. The given number of cores corresponds to the multi-simulation with both models. The control simulation for this benchmark (black squares) is defined by the multi-simulation of both cortex models without any connections between the models ($N_{\mathrm{MUSIC}} = 0$). Only the first data point (16 cores) corresponds to the situation with a single process per InfiniBand port. Still, we observe linear scaling up to 24 cores. Beyond this, the simulation time scales up to 96 cores, yielding a simulation time of 6 s per second of biological time.

The minimal benchmark ($N_{\mathrm{MUSIC}} = 8$, $k^{i,x}_{\mathrm{MUSIC}} = 1$, dark gray diamonds) also exhibits excellent scaling, but the simulation time increases by 1.3 ± 0.4 s. This difference in simulation time, however, does not show a clear dependence on the number of cores. Communicating 1,000 spike trains for every population ($N_{\mathrm{MUSIC}} = 8,000$, $k^{i,x}_{\mathrm{MUSIC}} = 1,000$, light gray circles) results in an additional increase of 1.3 ± 0.4 s, again with excellent scaling and no clear dependence of the increase in simulation time of the number of cores.

In order to understand this increase in simulation time, we simulate the symmetric multi-simulation benchmark for various values of N_{MUSIC} (keeping $k^{i,x}_{\mathrm{MUSIC}} = 1,000$ constant) and convert the number of MUSIC channels with the measured firing rates of the different populations in the cortex model into the MUSIC spike rate, the total number of spikes that is transmitted in one biological second from one cortex model to the other. Figure 11b shows the simulation time per second of biological time as a function of this MUSIC spike rate for a fixed number of cores (data obtained with 32 cores in the multi-simulation is shown in black, with 64 cores in dark gray). The dashed lines indicate the control multi-simulations with two unconnected cortex models ($N_{\mathrm{MUSIC}} = 0$). While the asymmetric multi-simulation benchmark is independent of the MUSIC spike rate (see above), the simulation time does depend on the MUSIC spike rate for the symmetric multi-simulation. Note, however, that this number is representing the spike rate transmitted via MUSIC

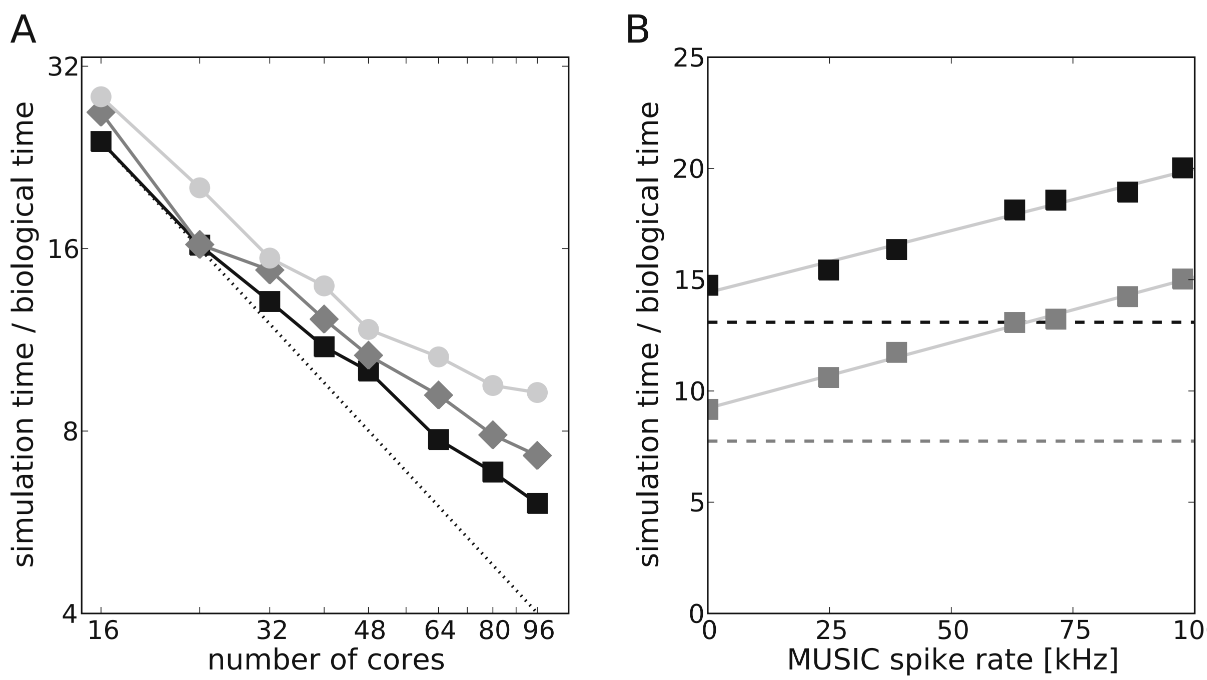


Fig. 11 Performance of the symmetric multi-simulation benchmark. **a** Simulation time per second of biological time as a function of the total number of compute cores for both network models. *Black squares* show the performance of the control ($N_{\text{MUSIC}} = 0$), *dark gray diamonds* the benchmark's performance for $N_{\text{MUSIC}} = 8$ and *light gray circles* for $N_{\text{MUSIC}} = 8,000$. The *dotted line* indicates the expectation for linear speed-up of the control. **b** Simulation time per second of biological time as a function of the MUSIC spike rate. Simulations with 32 cores are indicated in *black*, with 64 cores in *dark gray*. *Dashed lines* indicate the control ($N_{\text{MUSIC}} = 0$) and *squares* show the data for the symmetric multi-simulation benchmark with *light gray lines* showing the corresponding linear fits

in every direction and that the information has to be routed to all processes running the corresponding cortex model. We find that the simulation time increases linearly with the MUSIC spike rate (light gray lines indicate the linear fit). For 32 cores, the simulation time increases by 0.56 s when increasing the MUSIC spike rate by 10 kHz. For 64 cores, this number is only slightly increased to 0.6 s/10 kHz in the MUSIC spike rate.

A MUSIC Multi-simulation with NEST and MOOSE

A MUSIC multi-simulation was performed by connecting the layered cortical network model in NEST to a striatal network model in MOOSE. Activity of both simulations were visualized using the tool described in Section "Pre- and Post-processing" (see Fig. 12).

For the live demonstration, we reduced the size of the layered cortical network model to 8,000 neurons. The output consisted of spike events generated in the excitatory population of layer 5 that were exported through a MUSIC port.

The striatal network model was built using multi-compartmental units with Hodgkin-Huxley formalism and consisted of ten striatal medium spiny projection neurons with 189 compartments each (Wolf et al. 2005; Hjorth et al. 2008) and ten fast spiking interneurons with 127 compartments each (Hellgren Kotaleski et al. 2006). The cell models were ported from NEURON

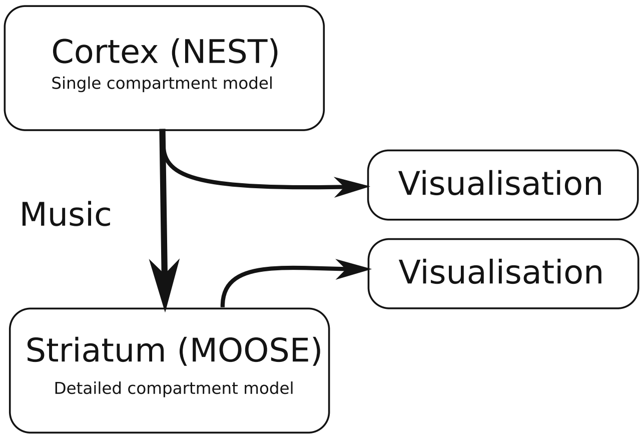


Fig. 12 Schematic of run-time interoperability for a cortico-striatal model. The cortical model simulated in NEST uses MUSIC to send spikes to the striatal model in MOOSE. In addition, two visualization processes receive the spike information from both NEST and MOOSE

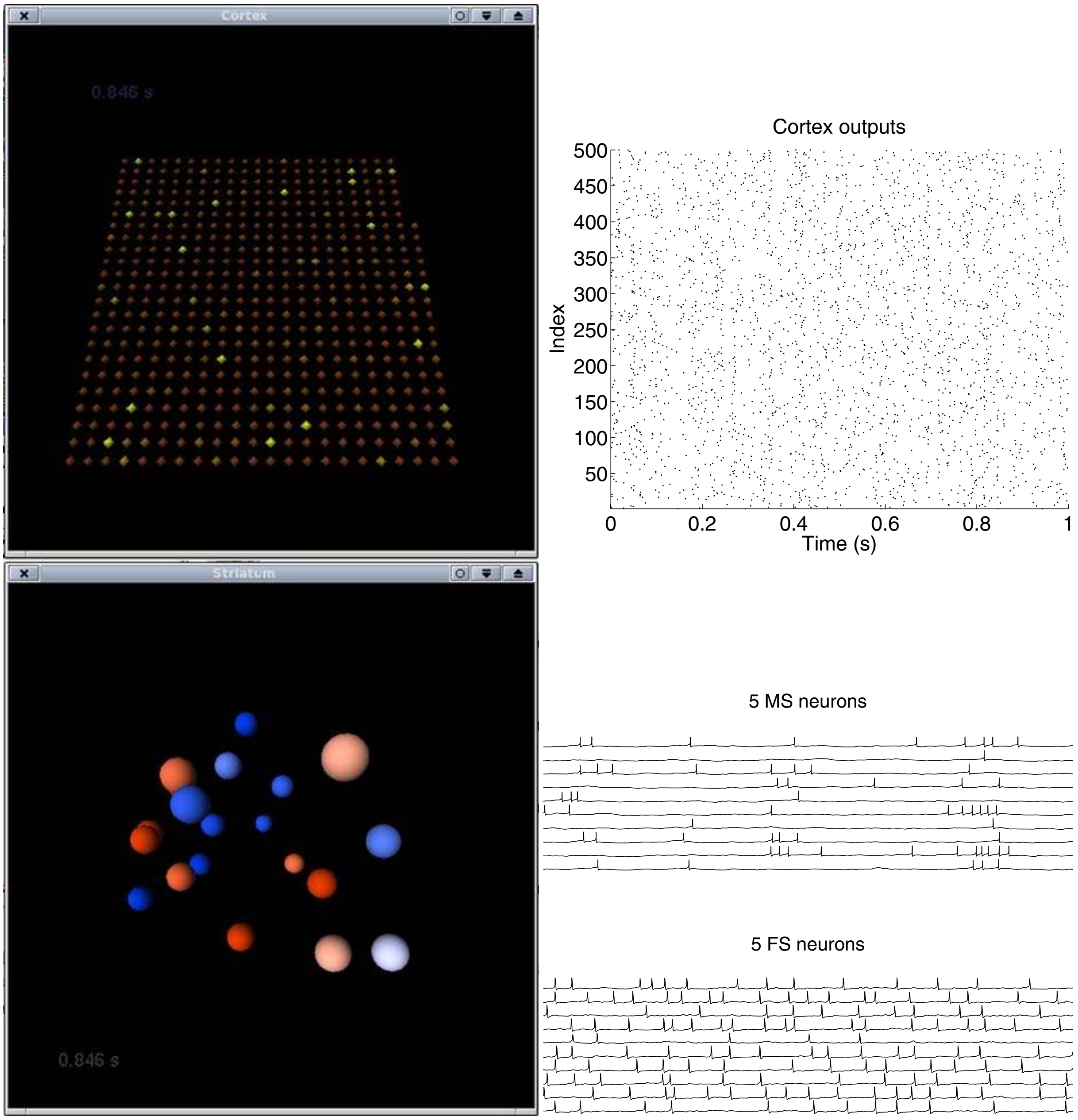


Fig. 13 Results from the multi-simulation described schematically in Fig. 12. To the *left*, two window captures from 3D visualizations of the cortex and striatum model are shown. In the *upper half* of the figure, 500 outputs from the cortex model in NEST are visualized on a planar grid, the radii and intensity of the color of the neurons increase when they spike. In the *lower part*, 10 MS (*red*) and 10 FS (*blue*) neurons in the striatal network are visualized in the same manner. To the *right* are a raster plot of the cortical activity and voltage traces for the MS and FS neurons

and GENESIS, respectively, to MOOSE. In this reduced version of the striatum for the MUSIC demonstration, no GABAergic connections were included between the two neuron populations. A MUSIC input port delivered spike events to both populations.

A short MUSIC configuration file described the multi-simulation and specified connections between the cortex output port and the striatum input port, and also connections from both models to one instance each of the visualization tool. Figure 13 shows captures of windows from the simulation tool instances together with simulation results from each model.

Since the MUSIC API enforces independence between the applications, the multi-simulation could be built from the cortex model and the striatum model without changes to their simulation scripts in other respects than the creation of MUSIC ports and the addition of the cortico-striatal projection on the receiver side. Spike events from NEST could easily be routed to MOOSE as well as a visualization process without further changes to the simulation scripts.

Discussion

The multi-simulation described in the previous section is a demonstration of how MUSIC can promote interoperability between models written for different simulators and how these can be re-used to build larger model systems. Alternative approaches to run-time interoperability are object-oriented frameworks (as illustrated by MOOSE itself; see Cannon et al. 2007) and using a common standard model description language.

Object-oriented frameworks provide APIs for services such as solvers, scheduling of events and communication, while specialized modules correspond to entities in the neuronal model. In comparison, the MUSIC API is slim, essentially only providing what is necessary to support communication through MUSIC ports. In a sense, the approach of MUSIC is orthogonal to that of an object-oriented framework, implying that these approaches can, in fact, be combined, as illustrated by the MOOSE simulation in this article. Writing a module for an object-oriented framework usually means a commitment to that framework. On one hand, the object-oriented framework lifts some of the burden of implementation by providing services. On the other hand, it will only be possible for the module to communicate with other modules in the same object-oriented framework. In contrast, any simulator or tool supporting the MUSIC interface can be connected to the rest of the set of tools supporting MUSIC. In fact, any module written for an object-oriented framework which supports MUSIC will also be possible to connect to such tools supporting MUSIC.

One example of a framework targeting a similar problem domain as MUSIC is the component-based extension framework by King et al. (2009). This framework provides three APIs, one for a *compute engine*, exemplified by a specially compiled version of NEURON (Carnevale and Hines 2006), a *message-bus component*, allowing the encapsulation of a spike communication algorithm, and, a *monitoring, analysis and control component*. This framework could, as MUSIC, be used to set up multi-simulations and promote interoperability and re-use of existing components. While both solutions are non-exclusive in the sense that they could potentially co-exist with each other and/or other communication frameworks, MUSIC does not require the re-organization of an existing simulator into a library providing the compute engine API and is in this way less invasive. Also, MUSIC abstracts connectivity at two levels, as ports and as shared global indices within ports, thereby making it possible to easily connect pluggable components into different configurations specified by a configuration file. The component engine API leaves the handling of the lowest level of connectivity entirely to the user (in the form of NEURON "gids"). This creates dependencies between the configurations of components of a multi-simulation so that the re-use of a tool requires a different mapping of gids.

The approach of a common standard model description language, such as PyNN (Davison et al. 2009) or NeuroML (Crook and Howell 2007; Crook et al. 2007), enables the same model description to be used with different simulators. This circumvents the difficulty of reimplementing models when moving them from one software to another. This approach also has the strength that it makes the model future-proof. But even in the presence of such a standard, we cannot combine two models of different kinds (for example a model based on integrate-and-fire units and a model based on Hodgkin-Huxley formalism) if our favorite software does not support both forms of modeling. Ultimately, we must recognize the value in specialized tools optimized for a particular purpose. A scripting language environment such as Python can bind tools together by loading them as libraries (Ray and Bhalla 2008). MUSIC is another alternative. Thus, we again see that MUSIC should be seen as providing orthogonal functionality.

Apart from interoperability, MUSIC also provides efficient communication between parallel applications enabling multi-simulation of large-scale neuronal systems.

One of the strengths of the MUSIC API design is that it allows for establishing a deterministic communication schedule which removes the need for handshaking. This has also been exploited in the implementation. The downside of this design choice is that new MUSIC ports cannot be added once the simulation is run, as MUSIC cannot change back to the setup phase once the runtime was entered. It is conceivable, though, that a future version of the standard could allow for changes to the communication graph during simulation without requiring handshaking during communication.

The adaptation of NEST to MUSIC was straightforward. The changes were not extensive and fell naturally into the existing structure of the code. MUSIC concepts such as ports were mapped to NEST proxies, MUSIC events could be routed by the proxies into the standard spike event delivery mechanisms. In the NEST simulator kernel, only five of the existing compilation units were affected: the scheduler and the units for MPI communication, network administration, scripting language binding and error handling. New compilation units were added for the MUSIC event handler and the NEST representations of MUSIC ports (`music_out_proxy`, `music_in_proxy`). The handling of the MUSIC `Setup` and `Runtime` objects was encapsulated in NEST's `Communicator` class.

NEST implements an error handling strategy based on C++ exceptions. Several new exception classes have been added to be used upon errors related to MUSIC. Unfortunately it is not possible to recover from errors during a MUSIC multi-simulation, as interactive simulations are not supported. Therefore NEST uses the function `MPI_Abort`() to quit the simulator upon errors. This also quits all remote applications.

Very little had to be changed in MOOSE to adapt it to MUSIC, and the changes here fit naturally into the MOOSE code structure. The five classes mentioned in Section "New Classes" were defined in fewer than 1000 lines of C++ code, and no changes were made in the basic MOOSE infrastructure. This was possible due to the compact MUSIC API, and was facilitated by the modular design of MOOSE.

Where possible, MOOSE allows the user to handle MUSIC related errors. For example, the user can inspect the `isConnected` field on MOOSE objects representing MUSIC ports, and choose to quit the simulation, or continue without MUSIC communication, in case a port was found to have not been connected successfully. At present, MPI exceptions are left unhandled by MOOSE, causing MOOSE to abort in case of errors at the MPI level.

While MUSIC supports communication of events, continuous values and messages, currently only spike event communication has been implemented in NEST and MOOSE.

The asymmetric multi-simulation benchmark (Section "Asymmetric Multi-simulation Benchmark", Fig. 10) shows that linking with MUSIC and using the MUSIC communicator does not affect performance. It also shows that normal communication loads through MUSIC ports do not add significantly to simulation time. In order to test performance under heavy communication load, the symmetric multi-simulation benchmark (Section "Symmetric Multi-simulation Benchmark", Fig. 11) provided a situation where every MUSIC channel in one application communicates spikes to neurons on every MPI process in the other application. MUSIC adapts both the spatial and temporal communication scheme to the topology of the multi-simulation, but the pilot implementation of the MUSIC library only uses pair-wise MPI `Send`() and `Receive`(). For a uni-directional one-to-one projection this would mean communication in one step at longer intervals. For the symmetric benchmark this instead implies a complete pair-wise exchange at every min-delay (minimum axonal delay between the applications). This partly accounts for the difference between no connectivity (black squares) and $N_{\text{MUSIC}} = 8$ (dark grey diamonds) in Fig. 11a. However, the linear dependence on the number of spikes transmitted via MUSIC in Fig. 11b can also be attributed to the additional load due to the collection and delivery of spikes by `music_out_proxys` and `music_in_proxys` in NEST. While the pair-wise communication gives most efficiency for multi-simulations that do not require all-to-all communication, a future version of the library could switch to the use of, for example, `Allgather`() when the number of inter-process communication pairs are of O(#processes). Another interesting development would be to use non-blocking communication over the MUSIC library inter-communicators between `tick`() calls, during the time when the application is computing or communicating.

We conclude that MUSIC fulfills the design goal that it should be simple to adapt existing simulators to use MUSIC. In addition, since the MUSIC API enforces independence of the applications, the multi-simulation could be built from pluggable component modules without adaptation of the components to each other in terms of simulation time-step or topology of connections between the modules. Preliminary results from benchmarks of two reciprocally connected large-scale versions of the layered cortical network model (one magnitude larger than the model simulated in this article) also indicate good performance and scaling behavior. We would like to encourage the community to continue building on a sharable base of MUSIC-enabled simulators and tools for the easy construction of multi-simulations.

Acknowledgements Partially funded by DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, EU Grants FP6-2004-IST-FETPI-015879 (FACETS) and FP7-HEALTH-2007-A-201716 (SELACT), the Next-Generation Supercomputer Project of MEXT (Japan), and the Helmholtz Alliance on Systems Biology (Germany). The MUSIC standard and software is provided and supported by the International Neuroinformatics Coordinating Facility (INCF).

Information Sharing Statement The MUSIC software is distributed under the GPLv3 license and can be downloaded from http://software.incf.org/software/.

References

Albus, J. S., Bekey, G. A., Holland, J. H., Kanwisher, N. G., Krichmar, J. L., Mishkin, M., et al. (2007). A proposal for a decade of the mind. *Science, 317*(5843), 1321.

Bower, J. M., & Beeman, D. (1998). *The book of GENESIS: Exploring realistic neural models with the GEneral NEural SImulation System* (2nd Ed.). New York: Springer.

Brette, R., Rudolph, M., Carnevale, N. T., Hines, M. L., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience, 23*, 349–398.

Cannon, R. C., Gewaltig, M.-O., Gleeson, P., Bhalla, U. S., Cornelis, H., Hines, M. L., et al. (2007). Interoperability of neuroscience modeling software: Current status and future directions. *Neuroinformatics, 5*(2), 127–138.

Carnevale, N. T., & Hines, M. L. (2006). *The NEURON Book*. U.K.: Cambridge University Press.

Crook, S., Gleeson, P., Howell, F., Svitak, J., & Silver, R. A. (2007). MorphML: Level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics, 5*, 96–104.

Crook, S. M., & Howell, F. W. (2007). XML for data representation and model specification in neuroscience. *Methods in Molecular Biology, 401*, 53–66.

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: A common interface for neuronal network simulators. *Frontiers in Neuroinformatics, 2*, 1–10.

Djurfeldt, M., Ekeberg, Ö., & Lansner, A. (2008a). Large-scale modeling—a tool for conquering the complexity of the brain. *Frontiers in Neuroinformatics, 2*, 1–4. doi:10.3389/neuro.11/001.2008.

Djurfeldt, M., & Lansner, A (2007). Workshop report: 1st INCF workshop on large-scale modeling of the nervous system. Nature Precedings. Available from http://dx.doi.org/10.1038/npre.2007.262.1.

Djurfeldt, M., Lundqvist, M., Johansson, C., Rehn, M., Ekeberg, Ö., & Lansner, A. (2008b). Brain-scale simulation of the neocortex on the BlueGene/L supercomputer. *IBM Journal of Research and Development, 52*, 31–42.

Ekeberg, Ö., & Djurfeldt, M. (2008). Music—multisimulation coordinator: Request for comments. Available from Nature Precedings http://dx.doi.org/10.1038/npre.2008.1830.1.

Ekeberg, Ö., & Djurfeldt, M. (2009). *MUSIC—Multi-Simulation Coordinator, users manual* (1st Ed.). Stockholm, Sweden: INCF, Karolinska Institutet, Nobels väg 15 A, SE-171 77, February 2009. http://software.incf.org/software/music.

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., & Gewaltig, M. (2009). PyNEST: A convenient interface to the NEST simulator. *Frontiers in Neuroinformatics, 2*, 12. doi:10.3389/neuro.11.012.2008.

Gewaltig, M.-O., & Diesmann, M. (2007). NEST (Neural Simulation Tool). *Scholarpedia, 2*(4), 1430.

Hellgren Kotaleski, J., Plenz, D., & Blackwell, K. T. (2006). Using potassium currents to solve signal to noise problems in inhibitory feedforward networks of the striatum. *Journal of Neurophysiology, 95*(1), 331–341.

Hjorth, J., Zilberter, M., Oliveira, R. F., Blackwell, K. T., & Hellgren Kotaleski, J. (2008). Gabaergic control of backpropagating action potentials in striatal medium spiny neurons. *BMC Neuroscience, 9*(Suppl 1), P105. doi:10.1186/1471-2202-9-S1-P105.

King, J. G., Hines, M., Hill, S., Godman, P. H., Markram, H., & Schürmann, F. (2009). A component-based extension framework for large-scale parallel simulations in NEURON. *Frontiers in Neuroinformatics, 3*, 1–11.

Potjans, T. C., & Diesmann, M. (2008). Consistency of *in vitro* and *in vivo* connectivity estimates: Statistical assessment and application to cortical network modeling. In *Soc. Neurosci. Abstr.* (Vol. 38, pp. 16.1). Washington, DC, U.S.A.

Potjans, T. C., Fukai, T., & Diesmann, M. (2009). Implications of the specific cortical circuitry for the network dynamics of a layered cortical network model. *BMC Neuroscience, 10*(Suppl 1), P159.

Ray, S., & Bhalla, U. S. (2008). PyMOOSE: Interoperable scripting in python for MOOSE. *Frontiers in Neuroinformatics, 2*, 6. ISSN 1662-5196, URL: http://www.ncbi.nlm.nih.gov/pubmed/19129924, PMID: 19129924. doi:10.3389/neuro.11.006.2008.

Schemmel, J., Fieres, J., & Meier, K. (2008). Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008* (pp. 431–438).

Wolf, J. A., Moyer, J. T., Lazarewicz, M. T., Contreras, D., Benoit-Marand, M., O'Donnel, P., et al. (2005). NMDA/AMPA ratio impacts state transitions and entrainment to oscillations in computational model of the nucleus accumbens medium spiny projection neuron. *Journal of Neuroscience, 25*(40), 9080–9095.