ORIGINAL RESEARCH ARTICLE
published: 27 April 2009
doi: 10.3389/neuro.11.010.2009

A component-based extension framework for large-scale parallel simulations in NEURON

James G. King[1], Michael Hines[2], Sean Hill[1], Philip H. Goodman[3], Henry Markram[1] and Felix Schürmann[1]*

[1] Brain Mind Institute, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[2] Department of Computer Science, Yale University, New Haven, CT, USA
[3] Department of Medicine and Program in Biomedical Engineering, University of Nevada, Reno, NV, USA

Edited by:
Erik De Schutter, University of Antwerp, Belgium; Okinawa Institute of Science and Technology, Japan

Reviewed by:
Robert C. Cannon, Textensor Limited, UK
Hugo Cornelis, University of Texas Health Science Center at San Antonio, USA

***Correspondence:**
Felix Schürmann, Brain Mind Institute, EPFL, AA/AA-VP/BBP Station 15, 1015 Lausanne, Switzerland.
e-mail: felix.schuermann@epfl.ch

As neuronal simulations approach larger scales with increasing levels of detail, the neurosimulator software represents only a part of a chain of tools ranging from setup, simulation, interaction with virtual environments to analysis and visualizations. Previously published approaches to abstracting simulator engines have not received wide-spread acceptance, which in part may be to the fact that they tried to address the challenge of solving the model specification problem. Here, we present an approach that uses a neurosimulator, in this case NEURON, to describe and instantiate the network model in the simulator's native model language but then replaces the main integration loop with its own. Existing parallel network models are easily adopted to run in the presented framework. The presented approach is thus an extension to NEURON but uses a component-based architecture to allow for replaceable spike exchange components and pluggable components for monitoring, analysis, or control that can run in this framework alongside with the simulation.

Keywords: large-scale simulation, NEURON simulator, parallel, distributed

INTRODUCTION

The growing interest in large-scale, detailed multi-compartment neuronal simulations requires increasing parallelism in neurosimulator software (Bhalla, 2008; Goddard and Hood, 1998; Hammarlund and Ekeberg, 1998; Migliore et al., 2006). Large-scale detailed modeling efforts, however, face two opposing challenges. On the one hand, the high-level of biological detail in the models requires the feature sets of a specialized simulator. On the other hand, simulation workflows at this scale require integration with a variety of visualization and analysis tools, virtual environments and the flexibility to run the simulations on either cluster-based or specialized high-bandwidth supercomputer architectures and thus the actual simulator should be abstracted to maintain sustainability of the investments.

Previous efforts to provide a standardized component framework for neuronal simulations, namely the NeoSim project (Goddard et al., 2001; Howell et al., 2003), have proposed to use component-based concepts in neuronal simulations to provide an abstraction of the specific compute engine and allow reusability of components. At the same time the NeoSim project tried to address the common model specification problem through NeuroML. The multitude of publications on that very topic though shows that defining a standardized and general network description remains a field of active research (Cornelis and DeSchutter, 2003; Crook et al., 2007; Davison et al., 2008). Trying to accommodate cutting-edge network models while providing a stringent abstraction of the software components and a common model specification poses a severe challenge and may effectively limit the applicability of such a tool at the current stage of the research (Cannon et al., 2007).

In an effort to extend simulator technology to accommodate large-scale, biologically models efficiently, the Blue Brain Project (Markram, 2006) participates in the development of the simulator NEURON (Hines and Carnevale, 1997). Those efforts ensured the efficient parallelization of the simulator software to thousands of processors as well as model reproducibility throughout the parallelization process (Migliore et al., 2006). Furthermore, features such as cache efficiency, spike compression, random number handling, or distributing individual neurons for load-balance (Hines et al., 2008a,b) as well as ports to modern platforms like IBM Blue Gene/L and IBM Blue Gene/P have been integrated into the publicly available open-source version of NEURON. NEURON, therefore, is well positioned as a primary tool for large-scale detailed neuronal simulations.

While most simulators have means to extend the operations performed *during* the integration time step of the compute engine (in the case of NEURON through the NMODL interface, Hines and Carnevale, 2000), for some extensions it might be desirable or even required to sit *outside* of the compute engine. Interfacing the simulation to an external entity, e.g. another simulation or a virtual or real environment is such a scenario where the communication handling requires access to the event information and distribution. Similarly, online analysis, i.e. analysis that is running during a simulation on the same hardware, or simulation-steering scenarios either in analysis or visualization may also require access to the flow control information of the simulation as well as access to internal variables of the compute engine.

Here we present a new extension framework that encapsulates NEURON as a compute engine while providing its own master integration loop that permits calls to an arbitrary number of user components and:

1. allows the execution of arbitrary parallel NEURON network models;

2. provides a replaceable message bus for alternative spike exchange schemes;
3. provides user-specified online monitoring, analysis, and control components.

We use a component-based architecture that encapsulates the neurosimulator and provides components for spike event communication event and components that can perform online analyses. In contrast to NeoSim, the presented framework does not address the common model specification, i.e. it leaves the complete model instantiation to the compute engine's modeling language, and therefore imposes no constraints on the user model. The latter point makes the framework immediately amenable to being used with existing models while providing stable interfaces to components requiring efficient access to the inner compute loop of a parallel simulation.

In the following, we describe the concept of such an extension framework and demonstrate a prototype implementation running at scale using the simulator NEURON, exchangeable spike event distribution components, and a component capable of online monitoring, analysis, and control (MAC component) that can be used for simulation steering. We demonstrate its usefulness in the context of large-scale network simulations of the Blue Brain Project running on 8192 processors of the EPFL Blue Gene/L supercomputer and show the performance benefits of adapting the simulator communication patterns to the underlying computer platform using the Extension Framework. We furthermore show its generality by applying it to a previously published model from ModelDB[1].

[1]http://senselab.med.yale.edu/

MATERIALS AND METHODS

The goal of the Extension Framework design is to provide a flexible simulation system in which independently developed neurosimulators may be utilized as part of a larger toolchain. To achieve this flexibility, the Extension Framework encapsulates simulator functions (Compute Engine) in a component-based architecture including an Adapter Component, a Message Bus Component, and a MAC Component (**Figure 1**). The Compute Engine is the neurosimulator developed separately from the rest of the Extension Framework. The Adapter Component provides a layer over the Compute Engine to give it a consistent interface for all other components. Although the current version of the extension framework includes an Adaptor Component specific for NEURON, the concept should be readily extensible to other simulator engines.

The Message Bus Component facilitates the communication of spikes between neurons during the course of a simulation. Communication between computing nodes is handled *via* the Message Passing Interface (MPI) developed initially by Argonne Labs (Gropp et al., 1999). The Message Bus Component can even be used to manage spike exchanges *via* TCP/IP sockets with simulators outside a running instance of the framework. The MAC Component can be used to *monitor* the simulation as it progresses and generate reports. Yet, the functionality of a MAC Component is not limited to observing; rather, it can also do advanced *analysis* and even *control* the simulation through an appropriate interface. Specialized MAC Components, so-called Real-time Agents, furthermore can provide external entities the capacity to interact with the running simulation in real-time. Another type of MAC

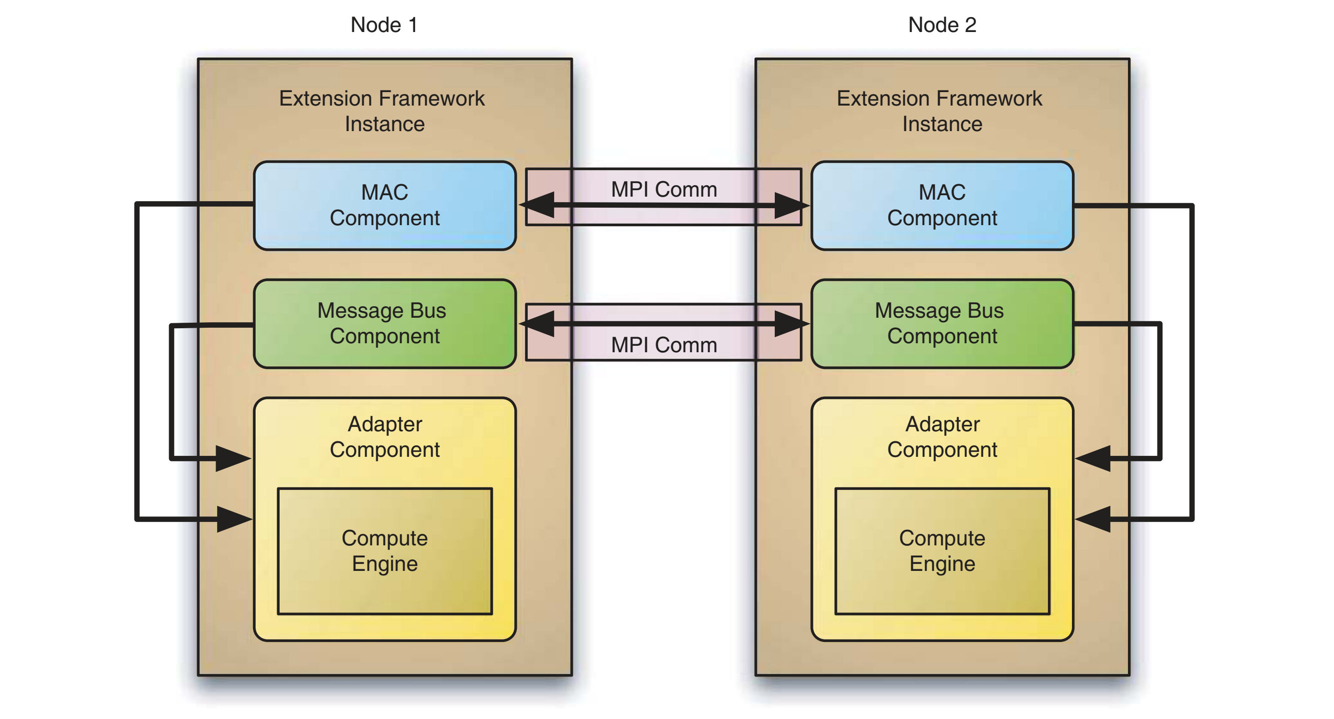


FIGURE 1 | The structural organization of component-based Extension Framework. An instance of it running on a node consists of different components. One component allows access to a Compute Engine, providing a common interface to other components. A Message Bus Component implements network connectivity, handling the exchange of APs among nodes. A MAC (monitor, analyze, control) Component may perform simulation analysis or modify simulation parameters.

Component could provide interactive visualization of a running simulation.

Unlike previous efforts to abstract neurosimulators such as NeoSim, the Extension Framework is not responsible for model configuration. Rather, it invokes the Compute Engine to instantiate a model through the simulator engine's means of model specification (e.g. HOC or Python in the case of NEURON, Hines et al., 2009) and then queries the Compute Engine through the Adapter Component for the information needed for the spike distribution. This design decision positions the presented framework as a real extension to NEURON as it is applicable to any preexisting model with essentially no modification to the model specification. Consequently, the framework does not address the distribution/load balancing issues as it instantiates the models as described in their specification.

The control flow in Extension Framework as the simulation executes is shown in **Figure 2**. During initialization of

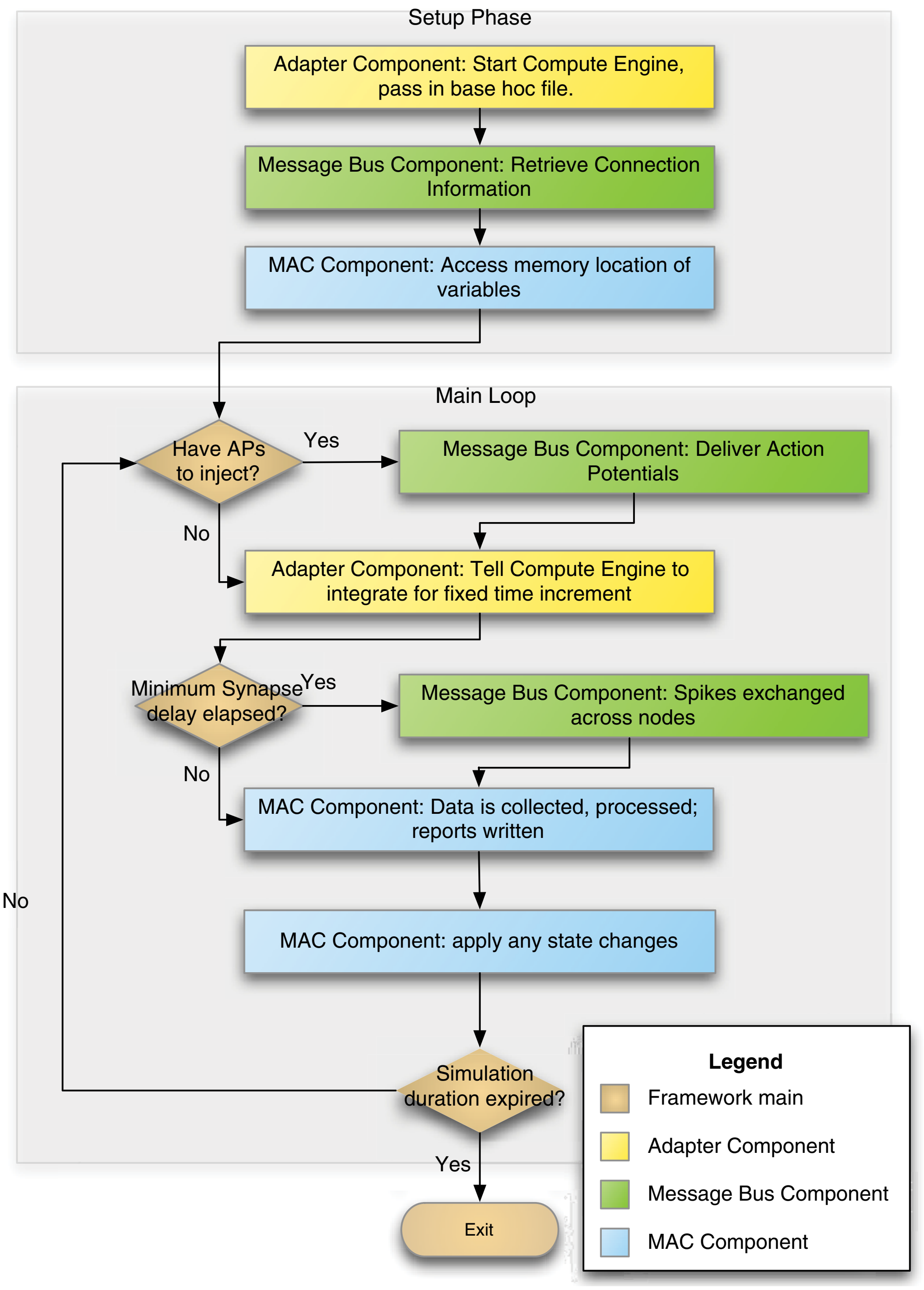

Setup Phase
Adapter Component: Start Compute Engine, pass in base hoc file.
Message Bus Component: Retrieve Connection Information
MAC Component: Access memory location of variables
Main Loop
Have APs to inject?
Yes
Message Bus Component: Deliver Action Potentials
No
Adapter Component: Tell Compute Engine to integrate for fixed time increment
Minimum Synapse delay elapsed?
Yes
Message Bus Component: Spikes exchanged across nodes
No
MAC Component: Data is collected, processed; reports written
No
MAC Component: apply any state changes
Simulation duration expired?
Yes
Exit
Legend
Framework main
Adapter Component
Message Bus Component
MAC Component

FIGURE 2 | Control flow in the Extension Framework.

the framework, it passes the native model specification to the Compute Engine and then extracts the necessary distribution/connection information through the Adapter Component. The Message Bus Component is initialized for the data exchanges that will occur whenever the minimum spike delay interval (as e.g. defined in Morrison et al., 2005) has elapsed. The MAC Component acquires the memory addresses of the variables from the simulator engine it needs to read or modify during the course of the simulation. Once all initializations have been complete, the Extension Framework moves on to the simulation loop where the main components operate.

ADAPTER COMPONENT

In order for a Compute Engine to be usable in the Extension Framework, an Adapter Component must be created as a layer, encapsulating the functionality of the Compute Engine and supplying a consistent interface so that the framework can perform necessary operations to execute a simulation. **Table 1** shows all functions the Adapter Component provides for the configuration and execution of a simulation. In principle, an Adapter Component can be built for any kind of Compute Engine as long as it provides adequate functionality; details on how it is implemented for NEURON is given in the Section 'Technical Details'.

The interface of the Adapter Component can essentially be divided into setup phase commands and simulation phase commands:

For the setup phase, the Adapter Component provides functionality to initialize the Compute Engine (`initializeComputeEngine()`) and to pass in the model description file for the Compute Engine to parse and instantiate (`setupSimulation()`). The model description file is in the Compute Engine's natural form; the Adapter Component passes the model file using a function that allows the file contents to be processed by the interpreter of the respective Compute Engine. Once the model has been interpreted and instantiated, the Adapter Component allows other components to make initial queries about the model.

Queries necessary for the Message Bus Component include acquiring information about the connections between cells using functions `sendingGids()` and `arrivingGids()`; here, a gid refers to a global unique identifier for a neuron in the simulation regardless of which node it is assigned (Migliore et al., 2006). Additionally, the function `minDelays()` is provided for deciding on the timing of spike exchange; here, the delay refers to the amount of time that has to exceed before a presynaptic cell firing can trigger the synaptic mechanism in a postsynaptic cell. The function `targetNodes()` can be used to create a Message Bus with more specific send and receive capabilities. Whereas the MAC Component gathers references to variables to be monitored during simulation run using `getVariableReferenceForReading()` and `getVariableReferenceForWriting()`.

Once all variables are accessed, the Adapter Component has the Compute Engine perform final initialization steps using the

Table 1 | Interface functions of the Adapter Component during setup and simulation phase.

Function	Parameters	Description
SETUP PHASE COMMANDS		
`initializeComputeEngine()`	Environmental Variables	Have the Adapter Component take steps necessary to initialize the Compute Engine prior to loading the model.
`setupSimulation()`	Model Description File	The Adapter Component gives the initial Model Description File to the Compute Engine so that it can instantiate the cells of the network and connect them.
`sendingGids()`	Array Pointer	Request for gids of cells on local node which send APs out. The gids are stored in the given Array Pointer. Used by Message Bus to coordinate spike exchange.
`arrivingGids()`	Array Pointer	Request for gids of cells on remotes nodes which deliver APs in. The gids are stored in the given Array Pointer. Used by Message Bus to coordinate spike exchange.
`minDelays()`	NodeID Array, Delay Array	Request for the minimum spike delay of presynaptic objectsand which node they. reside on. Used by Message Bus to coordinate spike exchange.
`targetNodes()`	Gid, Array Pointer	Request more specific information regarding which nodes a given gid sends spikes.
`getVariableReferenceForReading()`	Gid, Variable Name	Acquire access to cell values during simulation to be used for reporting or analysis.
`getVariableReferenceForWriting()`	Gid, Variable Name	Acquire access to variables from the simulator for the purpose of modifying the value during simulation.
`completeInitialization()`		Once the Message Bus Component and MAC Component have completed their setup, have the Adapter Component execute any final preparation steps on the Compute Engine so that it is ready to start simulating.
SIMULATION PHASE COMMANDS		
`integrateUntil()`	Time Stop	Adapter Component has the Compute Engine execute solver until the specified time is reached.
`receiveFireEvent()`	Gid, Time of Event	During the course of simulation,when a cell fires, the event is recorded.
`injectActionPotential()`	Gid, Time of Event, Local Flag	After spike exchange, deliver any action potentials from the indicated gid. Need to also relay if the gid is local to this node or remote.

function `completeInitialization()`, then the Extension Framework proceeds onto the simulation phase.

The simulation phase requires functionality to extract/deliver spikes and to advance the integration loop.

In order for the Extension Framework to implement a master integration loop, the Adapter Component must provide a way to control the duration of the integration in the Compute Engine; this is provided through the function `integrateUntil()`. This allows the Extension Framework to regain control after a span of time so that it can let its other components execute, such as the spike exchange *via* the Message Bus Component or examining states and modifying them *via* the MAC Component.

Before the Extension Framework can invoke the Message Bus Component, the spikes that occurred in that interval are queried from the Compute Engine and for this purpose the Adapter Component provides the function `receiveFireEvent()`. The Message Bus Component operates on those spikes and distributes them accordingly as described below; for the injection of the spikes into the Compute Engine, the Adapter Component provides the function `injectActionPotential()`.

Once the Extension Framework has finished its tasks, it invokes the Adapter Component to have the Compute Engine resume computations from where it left off, to continue evaluating the state variable equations for the next interval.

MESSAGE BUS COMPONENT

The Message Bus Component handles communications between the neurons of the network. It stores spike messages that have occurred within a current time frame that must be sent, exchanges spike buffers with other processors, and queues up synapse ids which will be activated after their spike delay has elapsed.

In the setup phase, after the model has been instantiated on the compute nodes, the Message Bus Component of the Extension Framework will be called to instantiate and configure itself using information accessed from the Compute Engine through the Adapter Component. First, the Message Bus requests information on which gids on the local node will be propagating APs through the function `sendingGids()`. Next, a list of remote gids which will be delivering APs to the local node is acquired through a call to the function `arrivingGids()`. Using these two lists, the communication patterns for sending and receiving data via MPI are established. To reduce the number of MPI invocations, the Message Bus also queries the minimum delay (`minDelays()`) on the destinations and exchanges this information across all nodes. The minimum spike delay interval determines the schedule for when the spike exchanges take place (e.g. Morrison et al., 2005).

During simulation, the Message Bus monitors the generation of spikes within any neurons on the local CPU *via* the Adapter Component through the function `receiveFireEvent()`. The Adapter Component needs a means to detect spikes as they happen on the Compute Engine. As these spike events occur, they are stored locally until they are relayed to the Message Bus Components on other CPUs such that the events arrive prior to the elapse of a minimum spike delay for any destination neuron. Any synapses that should be activated by the relayed spikes are queued into a message ring buffer until their individual delays have elapsed. The Message Bus then uses the Adapter Component's function `injectActionPotential()` to access the Compute Engine's facilities to inject the spikes into any neurons that are connected to the originating neuron once the spike delay has elapsed.

In the current version of the Extension Framework, two distinct Message Bus Components have been developed as shown in **Figures 3A,B**. Each of these Message Bus Components has distinct performance advantages depending on the computing architecture.

The first Message Bus of the Extension Framework was derived from an implementation used by the NeoCorticalSimulator (Wilson et al., 2001), a simulator designed for parallel communication on a Beowulf cluster. This implementation performs *point-to-point* communication such that a given node would communicate only with those other nodes from which it sends or receives spikes. When the simulation started, a node would use information from the

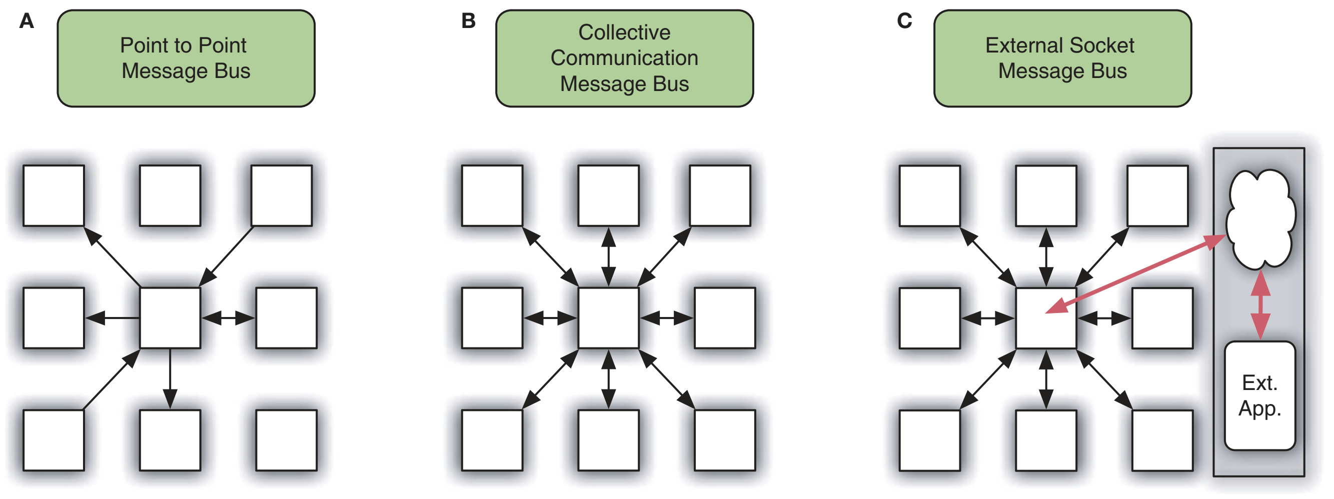

A
Point to Point Message Bus
B
Collective Communication Message Bus
C
External Socket Message Bus
Ext. App.

FIGURE 3 | Three possible implementations of the Message Bus. Each square represents a Node for computation, arrows indicate delivery of AP messages. **(A)** Point–Point communications using MPI_Send and MPI_Recv. **(B)** Collective Communication using MPI_AllGather. **(C)** Extension to B featuring external socket communication.

network connectivity description to build send and receive lists for the neurons instantiated on it to determine on which nodes to perform MPI_Send commands and on which nodes to perform MPI_Recv commands.

The second Message Bus performs *collective* communication, using the MPI_AllGather command to allow all the nodes to broadcast those neurons that have fired during the simulation time steps since the last communication. The Collective Communication Message Bus experienced improved performance since the version of MPI running on Blue Gene was specifically designed to take advantage of Blue Gene's network layout. IBM developed Blue Gene's implementation of MPI to minimize network traffic by having fewer nodes communicate redundant information (Almási et al., 2005).

A variant of the second message bus is depicted in **Figure 3C**; it extends the functionality by adding *external* communication with an external server *via* socket communication. The Extension Framework would send spike information to the server and receive back spike information from another application. It is not included in the current version of the Extension Framework.

MAC COMPONENT (MONITOR, ANALYZE, CONTROL)

The MAC Component gathers data from the simulation for either reporting or simulation management. The component acquires references to simulation parameters, periodically examines the contents of those references, and may execute changes to the values in the references.

During the setup phase, the MAC Component has to do some preliminary preparation. It uses the Adapter Component to make requests to the Compute Engine for access to certain values in the simulation which are outside the Extension Framework's memory space. Therefore, the Adapter Component provides the functions `getVariableReferenceForReading()` and `getVariableReferenceFor-Writing()` with details listed in **Table 1**.

The MAC Component can monitor state variables or simulation parameters in the Compute Engine. State variables represent the current state (including membrane potential, cellular currents, etc.) in the simulation at the current time, whereas simulation parameters are coefficients for the equations or heuristics describing the biological processes. Beyond monitoring, MAC Components can perform advanced analysis themselves or collectively using separate communication. Furthermore, MAC Components can react to the observations and make changes to direct the course of the simulation; an example of such a component can be a plasticity algorithm. Lastly, in an example scenario in which the Extension Framework is coupled with a robot through its external message bus and real-time response is required, a specific kind of MAC Component, a Real-Time Agent, could act as the interface between the simulation and the robot.

During the simulation, as the Extension Framework advances the simulation in time, the MAC Component gathers information on the state of the simulation by examining the supplied references or examining the Extension Framework memory space, too. A MAC Component may respond by altering these states or parameters values. This response can require certain conditions be met before actually triggering any changes.

Multiple independent MAC Components may be implemented and inserted into the Extension Framework simultaneously, acting separately within the simulation. MAC Components may need to work either locally or globally. A local MAC Component needs to access only the observations made on the neurons of an individual CPU. A global MAC Component must communicate through MPI with the other components across the parallel computer in order to form a more complete picture of what is happening in the circuit before determining what responses to take.

TECHNICAL DETAILS

The current version of the component-based Extension Framework is developed in C++ using MPI. It provides implementations of an Adapter Component for NEURON as well as different implementations of Message Bus Components and MAC Components.

In order to make NEURON useable as a Compute Engine in the Extension Framework, the implementation of an Adapter Component is based on three technical concepts. Firstly, the Adapter Component uses NEURON's function `hoc_valid_stmt()` as to be able to interface to arbitrary functions and model data structures through executing commands in NEURON's native interpreter language HOC or Python (Hines et al., 2009; Kernighan and Pike, 1984). Secondly, the functions of the Adapter Component used to expose variables to MAC Components are interfaced through NEURON's native mechanism extension language NMODL (Hines and Carnevale, 2000). Thirdly, to keep the connection querying independent from the instantiated model during the setup phase and to improve performance during the simulation phase, the Extension Framework makes use of special hooks within the NEURON source code. The Extension Framework is thus a combination of the three methods of runtime interoperability mentioned in Cannon et al. (2007).

While for the first and the second mechanism, no modifications to the NEURON source code are necessary, the third one requires NEURON to be configured and compiled with the flag – enable-ncs to activate certain portions of code. The functions can be categorized into two types: functions that are used during the setup phase to query the connectivity information from the NEURON compute engine; secondly, functions to extract and inject spikes during a simulation. **Table 2** lists all functions in the NEURON source code used to implement the Adapter Component. The new version of the NCS interface will be available in the NEURON 7.1 alpha distribution. The initial version of this interface has been in the NEURON source code since the publication of Migliore et al. (2006), yet for this publication, the setup phase functions were added as well as the inject mechanism modified as the previous version relied on a proprietary layout of a certain address space. Lastly, modifications were made to `nrn2ncs_outputevent()` to better clarify how it interacts when NCS mode of NEURON is used alongside the MPI features. While the function's original implementation was intended for only one parallel mode to be active at a time, either the NCS mode or the MPI mode, the updated function allows both to be used with the NCS part handling all spike delivery.

The Extension Framework requires NEURON to be compiled as a library in order to link it into one executable. NEURON's configure option – enable-ncs compiles NEURON as a library

Table 2 | Interface functions provided by NEURON once configured and compiled with the option – enable-ncs.

Function	Parameters	Description
SETUP		
`ncs_gid_sending_info()`	Array Pointer	Provide information about which gids on the local node send out APs, placing info at the indicated memory space.
`ncs_gid_receiving_info()`	Array Pointer	Provide information about which remote gids will deliver APs to this local node, placing info at the indicated memory space.
`ncs_netcon_mindelays()`	Host Array, Delay Array	Provide information about the minimum spike delays for a remote host to deliver an AP to the local node.
`ncs_target_hosts()`	Gid, Array Pointer	Provide information about which nodes a gid needs to send messages.
SIMULATION		
`ncs2nrn_integrate()`	Time Stop	Executes the solver until the indicated time has been reached
`nrn2ncs_outputevent()`	Gid, Fire Time	After a cell reaches threshold, this function is called to handle the event. With option – enable-ncs, this function is not defined by NEURON so that another entity may define it.
`ncs_netcon_count()`	Source Gid, Local Flag	Provide the number of netcons activated when the indicated gid fires an AP. A flag indicates if the source gid is local to the node or remote.
`ncs_netcon_inject()`	Source Gid, NetCon Index, Fire Time, Local Flag	Inject an AP into the indicated destination NetCon for a cell which fired. Requires the time of the event and a flag for whether the source gid is local to the node or remote.

and declares the `nrn2ncs_outputevent()` function of the NCS interface with extern status, expecting the application it is linked with to provide the implementation. Additionally, the configure option – with-paranrn is needed to activate certain code portions of NEURON that make it parallel aware. The Extension Framework has been tested on different hardware platforms from multi-processor machines, from a Beowulf cluster to an IBM Blue Gene/L (configure option – enable-bluegene). The source code will be made available on the Blue Brain website[2].

RESULTS

The Extension Framework has been run successfully using NEURON as a Compute Engine with the framework handling spike injection on different network models and hardware platforms. Small differences in timing of spike injections may occur due to the accumulation of floating point errors resulting from handling spike messages differently from a pure parallel NEURON simulation, but these minor differences have negligible impact on the inherent network spiking pattern. The time used to perform the simulation using the Extension Framework with no MAC Components is comparable to the time taken by a simulation run with pure NEURON. The additional time is taken up by overhead used to return control of the simulation to the Extension Framework and allow it to execute any MAC Components if they were enabled.

In the following, two network models are used. In order to demonstrate the usefulness of the replaceable Message Bus Components as well as the MAC Components, an unpublished Blue Brain neocortical column model with 10,000 neurons is used, which includes 200 unique morphologies consisting of approximately 600 cylindrical elements, connected via 12,500,000 conductance-based synapses, evaluated with an average of 300 electrical compartments and 10 Hodgkin-Huxley style ionic conductances per compartment at a time step of 0.025 ms. For proving the applicability of the approach to an arbitrary parallel NEURON model, the Extension Framework was used to run a previously published network model (Bush et al., 1999). Yet, instead of the originally serial version, the parallelized version used in (Migliore et al., 2006) accessible from the ModelDB model repository[3] under the accession number 64,229 was used.

COMPARISON OF MESSAGE BUS PERFORMANCE

Optimizing performance of the Extension Framework for a particular architecture is made simpler given the modular object nature of the Message Bus – the component that handles the costly communication during a simulation. We tested the *point-to-point* and *collective* Message Bus implementations in order to compare the difference in time consumed (**Figure 4**). Using the highly optimized *collective* communications developed for Blue Gene's MPI version, the Message Bus using MPI_AllGather was able to out-perform the original Message Bus using *point-to-point* communications *via* MPI_Send and MPI_Recv. The reason for this is that the latter Message Bus saw the Blue Gene flood with messages since the number of connections between a single neuron reaches so many other neurons. With each neuron sending out so many messages, then waiting to receive an equivalent amount, the simulator would spend an excessive amount of time for communication.

This superior performance may not extend to all systems as observed by performance differences when using the two Message Buses on a Beowulf cluster. This cluster is made up of a mix of architectures: 32 Dual-Processor AMD Opteron Nodes and 32 Dual-Processor Intel Pentium Xeon Nodes. The 64 Nodes are connected using Gigabit Ethernet. An Extension Framework simulation was run of shorter duration than the one Blue Gene because of the

[2]http://bluebrain.epfl.ch

[3]http://senselab.med.yale.edu/

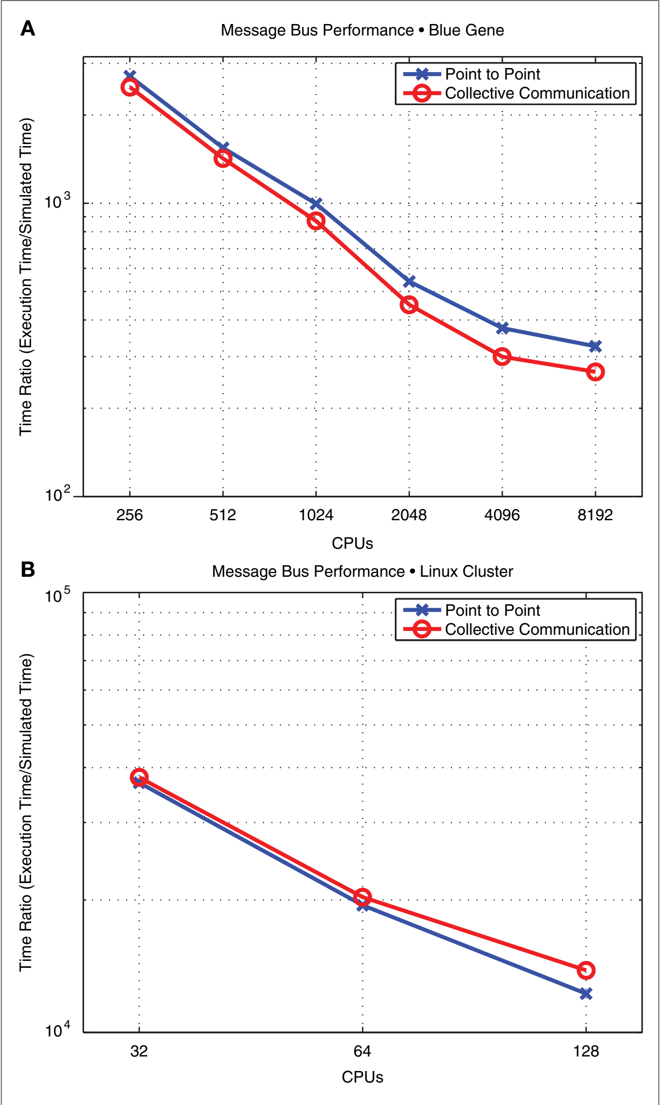


FIGURE 4 | Comparison of Message Bus performance. (A) Blue Gene collective calls (using MPI_AllGather) performance exceeds that of point-to-point (using MPI_Send and MPI_Recv) **(B)** on a Linux cluster, point-to-point version of the Message Bus achieves greater performance than collective calls.

greater resource limitation, but during this shorter simulated period, it can be observed that collective communication calls of MPI performed worse than the targeted calls using MPI_Send and MPI_Recv.

MAC COMPONENT EXAMPLE

Three MAC Components have been developed for monitoring and controlling a network simulation (King et al., 2006). The first MAC Component developed for the Extension Framework simply monitors the firing rates, f, of each neuron over a configurable time window (Gerstner and Kistler, 2002). The second MAC Component monitors the firing rate over a configurable amount of time but increases the synaptic conductance, g, for all synapses onto the neurons that fall below the target firing rate, F', by amount Δg. The third MAC Component extended the second MAC Component to monitor when the firing rate of a neuron exceeds a limit rate and lower the synaptic conductance for all synapses onto that neuron accordingly.

A series of simulations of the test network have been run using each of the three MAC Components (**Figure 5**). The time window used to determine the firing rate was 500 ms (Gerstner and Kistler, 2002). The first MAC Component simply monitors the simulation and computes the firing rate, which serves as a control condition. The second MAC Component monitors the firing rate and increases the synaptic conductances for the low-firing rate neurons. This results in a gradual increase in firing rates throughout the network, ultimately reaching and exceeding the targeted firing rate. In the case of the third MAC Component, the component increases the synaptic conductances until the targeted firing rate is reached, and as it exceeded the component acts to decrease the synaptic conductances. This results in a low frequency oscillation around the target firing rate in the network behavior as the component dynamically regulates the firing frequency of the network activity. The frequency of this oscillation depends on the monitoring window used to determine the firing rate where larger windows allow for finer grain control, reducing the degree of oscillations.

ADOPTING AN EXISTING PARALLEL MODEL FROM MODELDB

To demonstrate the simplicity of using arbitrary parallel network models specified in NEURON's HOC interpreter language, the parallel version of the model by Bush et al. (1999) was downloaded from ModelDB[4] (accession number 64,229) and run in the Extension Framework. The only necessary modification to the original model files was in the main run script init.hoc and concerned the commenting out of the parallel run command (as well as parallel run statistics) as this function is provided by the Extension Framework. All other files and the main body of the init.hoc script remain unchanged as illustrated in **Figure 6**. The spike pattern of a pure NEURON simulation and the simulation in the Extension Framework are identical.

CONCLUSIONS

The component-based Extension Framework for large-scale simulations in NEURON allows for a more flexible simulation environment where the application responsible for biophysical computations is developed separately from the details of network communication and analysis. We have presented an architecture that encapsulates the neural network simulator NEURON using an abstraction layer (Adapter Component), which permits the simulator to be extended with tailored communication components (Message Bus Components) and an on-line analysis and control framework (MAC Component). Furthermore, we demonstrated that it is possible to achieve increased communication performance during a network simulation by selecting an appropriate Message Bus for the underlying communication network. Finally, we developed an example of a MAC Component, which monitors, analyzes, and modifies an ongoing simulation providing a mechanism for dynamic control of large-scale network behavior. We demonstrated

[4]http://senselab.med.yale.edu/

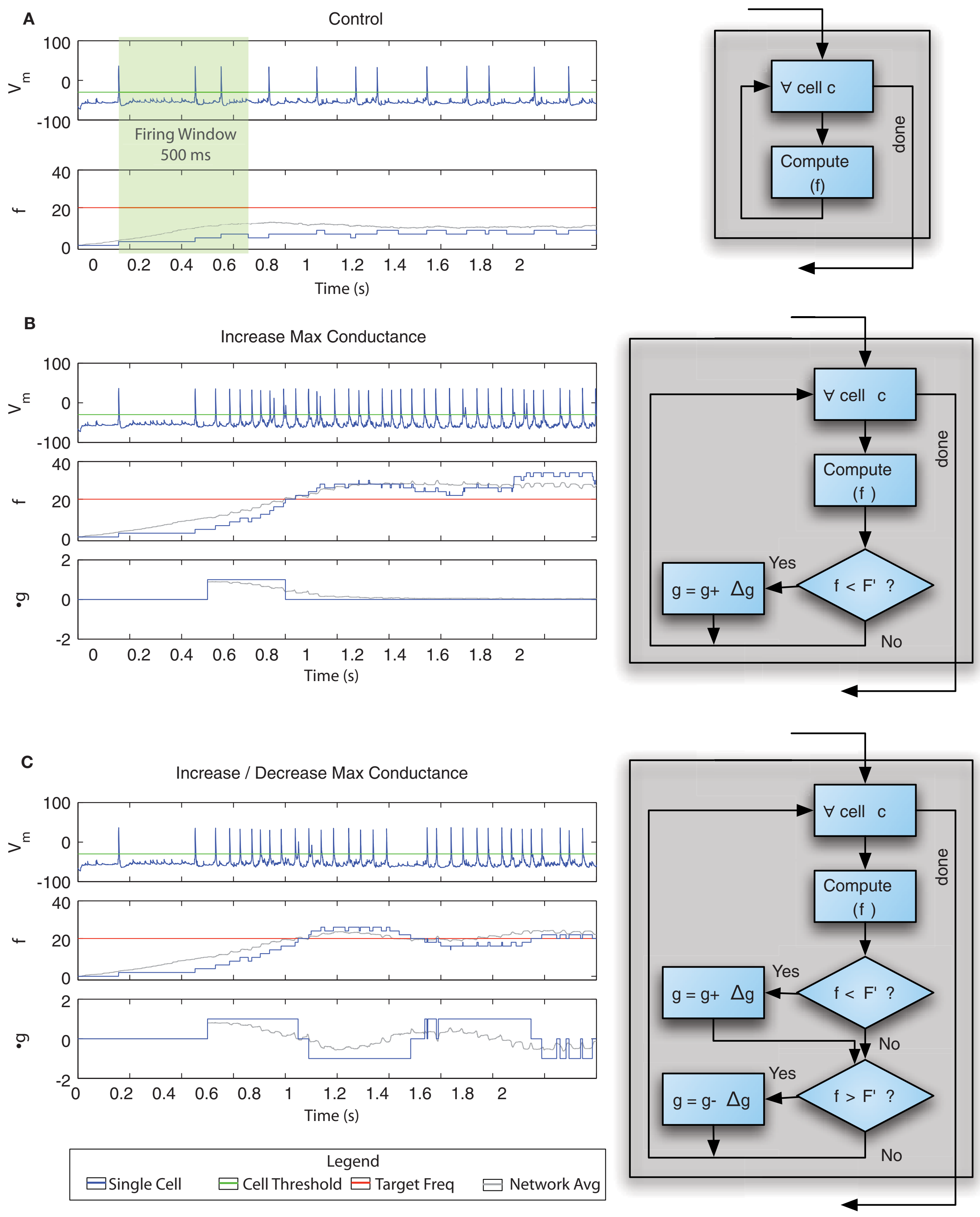
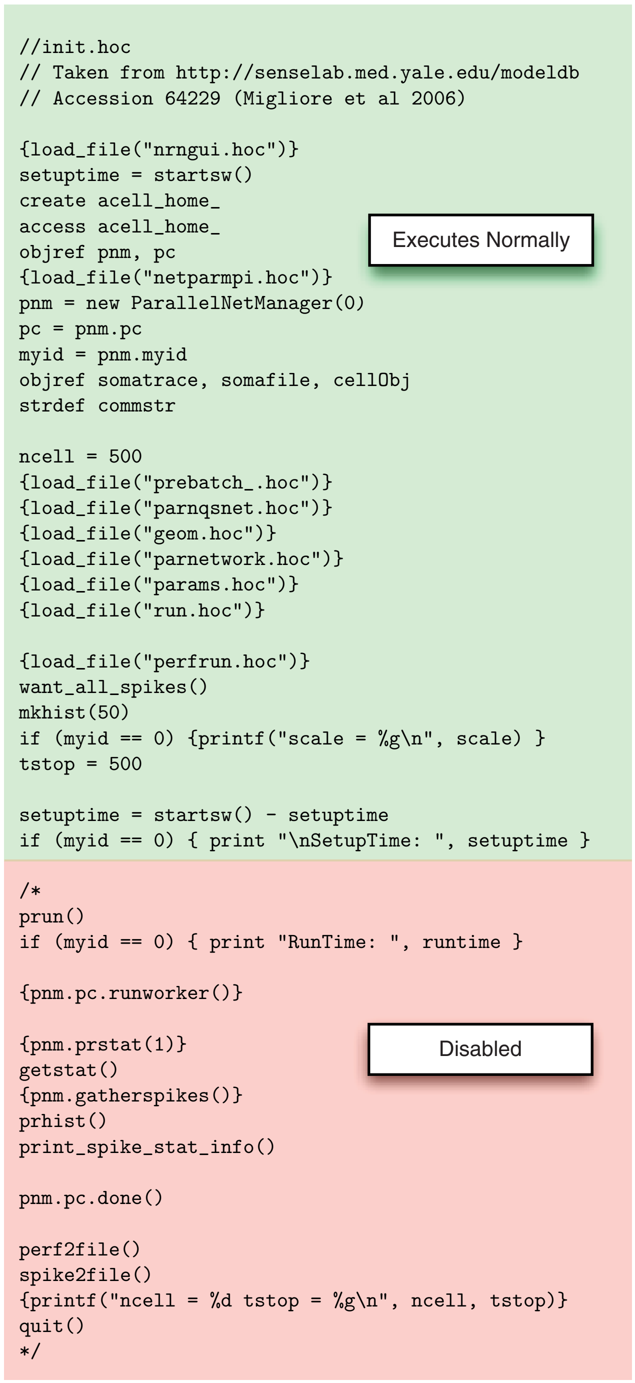


FIGURE 5 | Example MAC Component monitoring, analyzing, and controlling simulation behavior with accompanying flow diagrams illustrating component's logic. (A) Control simulation where the first graph shows the voltage trace, *Vm*, for one cell in the network, the second graph shows the firing rate, *f*, for this cell as well as the mean firing rate for the network. **(B)** MAC Component monitors firing rate and increases maximum conductance, *g*, for afferent synapses on neurons below a minimum rate, *F'*, by an amount Δ*g*; the first and second graphs correspond with those of A and the third graph shows the Δ*g* for synapses for the selected cell as well as the mean Δ*g* for all cells in the network. **(C)** Revised MAC Component still monitoring for minimum firing rate but also monitoring for maximum firing rate, scaling back maximum conductance as needed; all three graphs correspond to those of B.


Executes Normally
Disabled

```
//init.hoc
// Taken from http://senselab.med.yale.edu/modeldb
// Accession 64229 (Migliore et al 2006)

{load_file("nrngui.hoc")}
setuptime = startsw()
create acell_home_
access acell_home_
objref pnm, pc
{load_file("netparmpi.hoc")}
pnm = new ParallelNetManager(0)
pc = pnm.pc
myid = pnm.myid
objref somatrace, somafile, cellObj
strdef commstr

ncell = 500
{load_file("prebatch_.hoc")}
{load_file("parnqsnet.hoc")}
{load_file("geom.hoc")}
{load_file("parnetwork.hoc")}
{load_file("params.hoc")}
{load_file("run.hoc")}

{load_file("perfrun.hoc")}
want_all_spikes()
mkhist(50)
if (myid == 0) {printf("scale = %g\n", scale) }
tstop = 500

setuptime = startsw() - setuptime
if (myid == 0) { print "\nSetupTime: ", setuptime }

/*
prun()
if (myid == 0) { print "RunTime: ", runtime }

{pnm.pc.runworker()}

{pnm.prstat(1)}
getstat()
{pnm.gatherspikes()}
prhist()
print_spike_stat_info()

pnm.pc.done()

perf2file()
spike2file()
{printf("ncell = %d tstop = %g\n", ncell, tstop)}
quit()
*/
```

FIGURE 6 | Modifications to the hoc file from ModelDB. Setup remains normal while run is disabled to allow control to return to the Extension Framework layer.

the simplicity of adopting the Extension Framework for a pre-existing parallel NEURON model from ModelDB, where not a single line in the model description had to be changed and only the run command and run statistics had to be disabled (this could even be done automatically). Thus, other parallel network models executed in the Extension Framework can immediately take advantage of its additional functionality such as replaceable Message Bus and online MAC components.

The presented component-based Extension Framework represents a publically available version of a simulator environment *Neurodamus* developed within the Blue Brain Project (Frye et al., 2006). The design of the Extension Framework resembles some of the component-based modularity of the NeoSim project (Goddard et al., 2001; Howell et al., 2003), but the distinguishing feature is that it does not address the problem of a common model specification, which possibly impedes using state-of-the art functionality of neurosimulators. By allowing models to be specified in the simulator's specification language it allows utilizing a particular simulator's cutting edge feature set while gaining extensibility and tool chain stability. In a similar fashion, the MUSIC project (MUlti-SImulation Coordinator) seeks to create a generic interfaces between simulator cores such that the simulators can execute while under the control of a managing entity (Ekeberg and Djurfeldt, 2008). The MUSIC effort represents a project under development that also follows the idea of modularizing a simulation as described in NeoSim to allow component interaction and leaving the model specification to the respective compute engines.

While the current implementation of the Extension Framework provides an Adapter Component specific for the NEURON simulator, it should be possible to implement Adapter Components for other neurosimulators in the future. It should be noted the published version of the Extension Framework does not address the distribution of the network model on the parallel hardware architecture. It thus is targeted at models that already address the distribution in the model specification. It is conceivable that as the common model specification approaches such as NeuroML, PyNN, Neurospaces (Cornelis and DeSchutter, 2003; Crook et al., 2007; Davison et al., 2008) mature, a future extension to the Extension Framework could use a more general setup mechanism which would allow load distribution and balancing to be handled by the Extension Framework.

As large-scale detailed simulation projects go beyond the environments provided by publically available neurosimulators, the simulator engine itself needs to be integrated into a complete chain of tools. Those workflows may include powerful analysis and visualization environments (interactively and in post processing) as well as interconnects to virtual and real environments such as robotic devices and laboratory experiments. All those tools represent major developments and need to be made as independent of the simulator as possible while retaining maximum performance. The presented component-based Extension Framework for NEURON represents a working step in this direction.

ACKNOWLEDGMENTS

This work was supported by the Blue Brain Project. We thank James Frye for his initial contributions. We thank Dr. Fred Harris for his thoughtful comments on the manuscript and his work on NCS. A portion of the original NEOSIM API was reused by Michael Hines to create the prototype NCS interface to NEURON and that work was supported in part by NINDS grant NS11613.

REFERENCES

Almási, G., Archer, C., Castaños, J. G., Gunnels, J. A., Erway, C. C., Heidelberger, P., Martorell, X., Moreira, J. E., Pinnow, K., Ratterman, J., Steinmacher-Burow, B. D., Gropp, W., and Toonen, B. (2005). Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM J. Res. Dev.* 49, 393–406.

Bhalla, U. S. (2008). 'MOOSE – Multiscale Object-Oriented Simulation Environment'. Available at: http://moose.sourceforge.net/ (Retrieved April 10, 2008).

Bush, P. C., Prince, D. A., and Miller, K. D. (1999). Increased pyramidal excitability and NMDA conductance can explain posttraumatic epileptogenesis without disinhibition: a model. *J. Neurophysiol.* 82, 1748–1758.

Cannon, R. C., Gewaltig, M. O., Gleeson, P., Bhalla, U. S., Cornelis, H., Hines, M. L., Howell, F. W., Muller, E., Stiles, J. R., Wils, S., and De Schutter, E. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.

Cornelis, H., and DeSchutter, E. (2003). NeuroSpaces: separating modeling and simulation. *Neurocomputing* 52–54, 227–231.

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

Davison, A., Bru?derle, D., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). 'PyNN – a Python package for simulator-independent specification of neuronal network models'. Available at: http://neuralensemble.org/trac/PyNN (Retrieved April 10, 2008).

Ekeberg, Ö., and Djurfeldt, M. (2008). MUSIC-Multisimulation Coordinator: Request For Comments. Available from Nature Precedings <http://dx.doi.org/10.1038/npre.2008.1830.1>

Frye, J., Schürmann, F., King, J. G., Ranjan, R., and Markram, H. (2006). Neurodamus: a framework for large scale and detailed brain simulations. In 5th Forum of European Neuroscience (Vienna). FENS Abstract Vol. 3, A037.21, p. 99.

Gerstner, W., and Kistler, W. M. (2002). Spiking Neuron Models: Single Neurons, Populations, Plasticity, 1st edn. Cambridge: Cambridge University Press.

Goddard, N., Hood, G., Howell, F., Hines, M. L., and De Schutter, E. (2001). NEOSIM: portable large-scale plug and play modeling. *Neurocomputing* 38, 1657–1661.

Goddard, N. H., and Hood, G. (1998). Large-scale simulation using parallel GENESIS. In The Book of GENESIS, Bower, J. M. and Beeman, D. 2nd edn (Berlin, Springer), pp. 349–380.

Gropp, W., Lusk, E., and Skjellum, A. (1999). Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd edn. Cambridge, MA, MIT Press.

Hammarlund, P., and Ekeberg, O. (1998). Large neural network simulations on multiple hardware platforms. *J. Comput. Neurosci.* 5, 443–459.

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

Hines, M. L., and Carnevale, N. T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Comput.* 12, 995–1007.

Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Front. Neuroinformatics* 3, 1.

Hines, M. L., Eichner, H., and Schürmann, F. (2008b). Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. *J. Comput. Neurosci.* 25, 203–210.

Hines, M. L., Markram, H., and Schürmann, F. (2008a). Fully implicit parallel simulation of single neurons. *J. Comput. Neurosci.* 25, 439–448.

Howell, F., Cannon, R., Goddard, N., Bringmann, H., Rogister, P., and Cornelis, H. (2003). Linking computational neuroscience simulation tools: a pragmatic approach to component-based development. *Neurocomputing* 52–54, 289–294.

Kernighan, B. W., and Pike, R. (1984). The Unix Programming Environment, 1st edn. Eaglewood Cliffs, NJ, Prentice Hall, Inc.

King, J. G., Schürmann, F., Hill, S., and Markram, H. (2006). BlueEnvironment: Enabling Real-Time Interactions with Biologically Complex Models of the Neocortical Column. In 5th Forum of European Neuroscience (Vienna). FENS Abstract Vol. 3, A037.15, p. 98.

Markram, H. (2006). The blue brain project. *Nat. Rev. Neurosci.* 7, 153–160.

Migliore, M., Cannia, C., Lytton, W. W., Markram, H., and Hines, M. L. (2006). Parallel network simulations with NEURON. *J. Comput. Neurosci.* 21, 119–129.

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

Wilson, E. C., Goodman, P. H., and Harris, F. C. Jr (2001). Implementation of a biologically realistic parallel neocortical-neural network simulator. In Proceedings of the 10th SIAM Conference on Parallel Process for Sci. Comput. Philadelphia, USA.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 10 December 2008; paper pending published: 18 December 2008; accepted: 08 April 2009; published online: 27 April 2009.

*Citation: King JG, Hines M, Hill S, Goodman PH, Markram H and Schürmann F (2009) A component-based extension framework for large-scale parallel simulations in NEURON. Front. Neuroinform. (2009) **3**:10. doi: 10.3389/neuro.11.010.2009*

Copyright © 2009 King, Hines, Hill, Goodman, Markram and Schürmann. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.