

Multi-compartment benchmark and validation suites on HPC systems



Figure 1: The Piz Daint system at the Swiss National Supercomputing Center (ETHZ-CSCS)

Piz Daint is one of the High-Performance Computing (HPC) systems in the High Performance Analytics and Computing Platform (HPAC). The benchmark and validation test suite is tested daily on both the CPU and GPU partitions of Piz Daint with the CSCS continuous integration service.

The photograph was made available for download and use by CSCS under the Creative Commons Attribution-Share Alike 3.0 Unported license. (creativecommons.org/licenses/by-sa/3.0/deed.en)

Project Number:	785907	Project Title:	Human Brain Project SGA2
Document Title:	Multi-compartment benchmark and validation suites on HPC systems		
Document Filename:	D7.3.1 (D44.1 D83) SGA2 M12 SUBMITTED 190306.docx		
Deliverable Number:	SGA2 D7.3.1 (D83, D44.1)		
Deliverable Type:	Other		
Work Package(s):	WP7.3		
Dissemination Level:	PU = Public		
Planned Delivery Date:	SGA2 M12 / 31 March 2019		
Actual Delivery Date:	SGA2 M12 / 6 March 2019; ACCEPTED 22 Jul 2019		
Authors:	Benjamin CUMMING, ETHZ (P18) Stuart YATES, ETHZ (P18)		
Compiled by:	Benjamin CUMMING, ETHZ (P18)		
Contributors:	Nora ABI AKAR, ETHZ (P18), CSCS Jenkins CI support Anna LÜHRS, JUELICH (P20), SP7 Manager, SP7-internal review		
SciTechCoord Review:	Marc MORGAN, EPFL (P1)		
Editorial Review:	Guy WILLIS, EPFL (P1)		
Description in GA:	The validation and benchmark suites for multi-compartment simulators are deployed and their associated documentation can be downloaded from public repositories; the continuous integration framework is publishing up-to-date validation and benchmark results from HPC systems used in the HBP.		
Abstract:	<p>We describe NSuite, a framework for the implementation and execution of benchmarks and numerical validations for multi-compartment neuron simulators constituting the HBP SGA2 Deliverable D7.3.1. The framework specifically supports the Arbor, NEURON and CoreNeuron simulators, while allowing other simulation engines to be easily added in the future.</p> <p>The suite is specifically designed to allow easy deployment on High-Performance Computing (HPC) systems. It is also highly extensible, in that it admits the straightforward addition of new benchmarks and validation tests. As a common repository of benchmarks and tests, NSuite allows simulator users to perform performance comparisons of simulators and confirm their numerical fidelity.</p>		
Keywords:	High Performance Analytics and Computing (HPAC) Platform, High Performance Computing (HPC), Arbor, NEURON, CoreNeuron, multi-compartment, simulation, benchmarking, validation		
Target Users/Readers:	Simulation engine developers, performance-oriented simulation users, HPC centres		



Table of Contents

- 1. Introduction 4
 - 1.1 Motivation 4
 - 1.2 Example Workflows 5
 - 1.2.1 Workflow 1: Simulation Engine Developer 5
 - 1.2.2 Workflow 2: Simulation Engine Development Team 6
 - 1.2.3 Workflow 3: HPC System Validation 7
- 2. The Framework 8
 - 2.1 Supported Simulation Engines 8
 - 2.2 Building Simulation Engines 9
 - 2.3 Benchmarks 9
 - 2.4 Validation Tests 10
 - 2.5 Supported HPC Systems 10
 - 2.6 Continuous Integration 11
- 3. Status and Future Work 11

Table of Tables

- Table 1: Default versions of each simulation engine downloaded by NSuite 9
- Table 2: List of the systems that the NSuite framework has been tested on so far 11

Table of Figures

- Figure 1: The Piz Daint system at the Swiss National Supercomputing Center (ETHZ-CSCS) 1
- Figure 2: The web dashboard from the CSCS Jenkins CI instance 7
- Figure 3: Example script for running all benchmarks and validation tests 8



1. Introduction

The benchmark and validation suite, named “NSuite”, is a software framework for benchmarking and validating multi-compartment neuron simulation engines, specifically those developed and used in the HBP. This Deliverable is available in the form of an open source software repository <https://github.com/arbor-sim/nsuite> and documentation <https://nsuite.readthedocs.io>.

Three simulation engines are currently supported:

- 1) **Arbor** (<https://github.com/arbor-sim/arbor>): a performance-portable, multi-compartment simulation engine developed within the HBP. It is designed as a library for state-of-the-art simulation applications with efficient execution on contemporary High-Performance Computing (HPC) systems.
- 2) **NEURON** (<https://www.neuron.yale.edu>): the most widely used simulation engine for multi-compartment simulations; it has been available for over 30 years. It has support for running large scale HPC simulations; it has, however, limited support for modern HPC architectures.
- 3) **CoreNeuron** (<https://github.com/BlueBrain/CoreNeuron>): A refactoring of NEURON developed partially within the HBP for HPC systems. It reduces memory overheads by removing the user interface from NEURON, and implements further performance optimisations.

NSuite automates the process of building simulation engines and running performance benchmarks and validation tests on HPC systems. There are three main motives for its development:

- 1) The need for a definitive resource for comparing performance and correctness of simulation engines on different HPC systems.
- 2) The need to verify the performance and correctness of a simulation engine as its source code changes over time.
- 3) The need to test that changes to an HPC system do not affect the performance or correctness of simulation engines.

The delivered software is open source, with detailed technical documentation of its features, and how to use and extend it. This report will describe the motivation behind the framework, an overview of its structure and features, and how it will be used in the HBP research infrastructure.

1.1 Motivation

Multi-compartment neuron models are designed to model the electrical activity of a neuron in significant detail. The branching morphology of a cell is decomposed into small line segments called compartments, and ion channels and synapses are modelled explicitly on each compartment. Large network models simulate many such cells simultaneously, with cells coupled via complex event-driven, spike exchange networks and direct electrical coupling, via gap junctions. Hence, the simulation of large networks of multi-compartment models places heavy demands on HPC resources at two scales:

- 1) High per-cell overheads: complex ion channel and synapse dynamics (ODEs) solved on many compartments have high computational and memory bandwidth overheads, requiring efficient implementation on computer processors (CPUs and GPUs).
- 2) Large networks use distributed computing to fit models into memory and to complete simulations in reasonable timeframes, which requires efficient communicating and work scheduling.

The simulation of multi-compartment neuron models is a major user of HPC resources in the HBP, so understanding the performance of simulation engines on HPC systems is important for simulation developers, users and HPC service providers. To this end, NSuite provides a framework for running performance tests (benchmarks) of the supported simulation engines on diverse HPC hardware platforms.

Developers and users of simulation engines also need to validate the correctness of these engines. NSuite provides a framework for validating the accuracy of simulation engines on different platforms. Results must be validated, not only across different simulation engines, but on each platform supported by a simulation engine, to validate hardware-specific algorithms and optimisations.

NSuite implements the process of building simulation engines, and running both benchmarks and validation tests, in a simple pipeline with the following features:

- 1) Flexible selection of simulation engines to build.
- 2) Flexible selection of benchmark and validation tests to run.
- 3) Configuration of build and run options to allow customisation for any system.
- 4) Consistent representation of benchmark and validation outputs, so that they can be used either on the command line, or integrated into automated processes such as continuous integration (CI).

The narrow focus and configurability facilitate using NSuite to automate the building and testing steps in diverse automated workflows, some examples of which are given in the next section.

1.2 Example Workflows

Here we present some examples of workflow use cases incorporating the NSuite framework. The workflows are presented in the form of a *user story*, followed by a proposed solution. In each case, NSuite automates the process of building simulation engines, running benchmark and validation tests, and writing benchmark and test outputs to file. The user provides configuration inputs, and selects which simulation engine(s) to build, which tests to run and how to analyse the results.

1.2.1 Workflow 1: Simulation Engine Developer

1.2.1.1 User story

As a developer implementing a performance optimisation for the simulation engine NEURON, I want to build NEURON and run benchmarks many times during the day, as I incrementally modify the code to continuously see how my changes impact performance.

1.2.1.2 Solution

There are two phases in this task, the first is to set up the workflow, and the second is to repeatedly run this workflow. The setup phase has three steps that are performed once:

- 1) Download the NSuite framework.
- 2) Configure the framework to get the NEURON source code from your development branch in your working git repository.
- 3) Run NSuite's install step for NEURON, which will check out your branch and build it.

Once set up, implement your changes in the version of the NEURON source code that the framework checked out. Then, every time you want to test a change, perform the following steps:

- 1) Run the install step, which will rebuild NEURON with your changes.
- 2) Run the benchmarks. When the benchmark suite is run, it prints a summary of benchmark results, and saves detailed benchmark results in JSON files. The developer can use the summary, or perform further analysis of saved benchmark data.

When finished, run the validation tests to ensure that the changes are also correct, then push your changes to your development branch in the NEURON git repository.

1.2.2 Workflow 2: Simulation Engine Development Team

1.2.2.1 User story

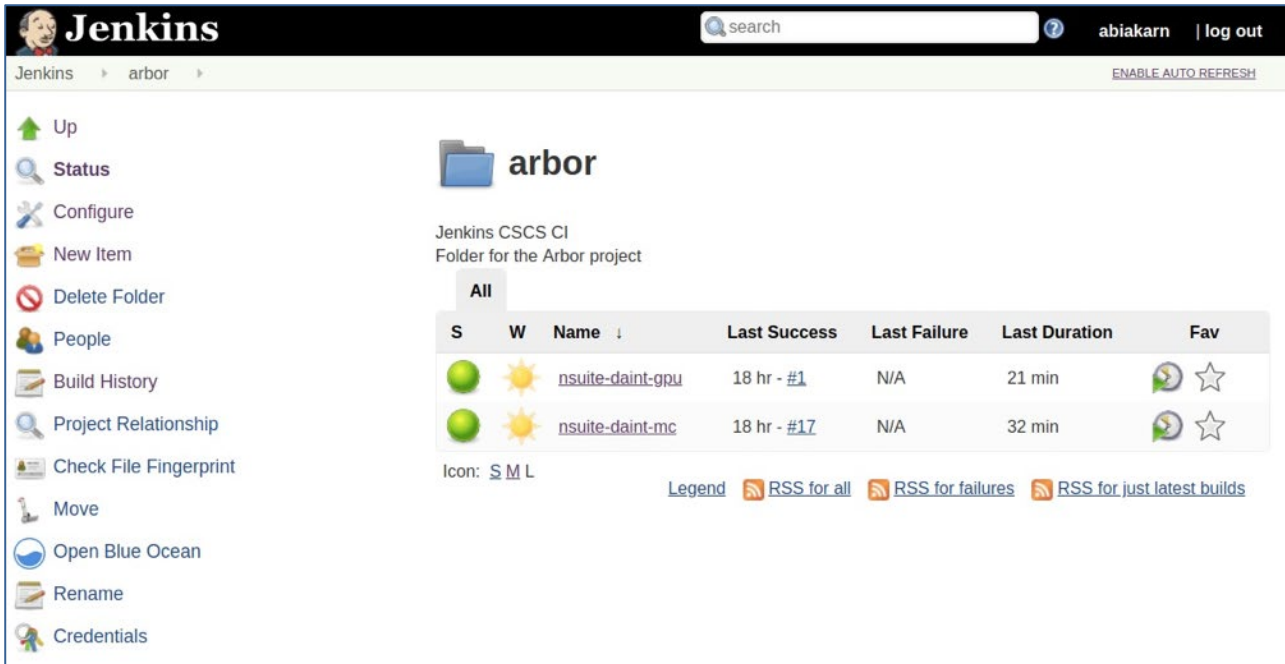
As the development team for the Arbor simulation engine, we want to automatically run benchmarks and validation tests for Arbor on all of our target architectures, on every pull request, to our GitHub repository, to ensure that proposed changes are accurate and offer adequate performance before they are merged.

1.2.2.2 Solution

The development team would start by writing scripts that will be used by the CI server to compile, run and analyse the output of Arbor in NSuite. Then, the team would configure the CI server using these scripts. In this context, “write a script” means to write code, typically in bash or Python, that will be run later by the CI framework. The following steps summarise the process of configuring the workflow:

- 1) Write an NSuite configuration script for the system. On a system with GPUs, for example, the script would configure NSuite to compile Arbor with GPU support.
- 2) Write a script that parses the output from running the benchmarks and validation tests for Arbor in NSuite. The script will report if any validation tests failed, and whether performance of the benchmarks was acceptable, and potentially collect historical performance information.
- 3) Write a configuration for the CI server that:
 - a) Checks out NSuite and runs NSuite’s build-benchmark-validate pipeline, using the custom configuration from step 1.
 - b) Then runs the script from step 2, to report if the changes should be accepted, based on whether they passed the validation tests and had acceptable performance.
- 4) Set up web hooks from GitHub, so that every pull request will trigger the CI server to run the configuration from step 3, to check out, compile and test the proposed changes.

Once configured, the CI server will now automatically test each pull request. A web-based dashboard can show the results of the tests on each target system, and present historical performance information to show how benchmark timings change over time. GitHub should also be configured to disallow merging of pull requests with failed tests. Figure 2 shows an example dashboard on the CSCS Jenkins CI server that tests both the multicore and GPU versions of Arbor on the Piz Daint supercomputer.



The screenshot shows the Jenkins web dashboard for the 'arbor' project. The dashboard includes a sidebar with navigation options like 'Up', 'Status', 'Configure', 'New Item', 'Delete Folder', 'People', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Move', 'Open Blue Ocean', 'Rename', and 'Credentials'. The main content area displays the 'arbor' folder with a table of build results for 'nsuite-daint-gpu' and 'nsuite-daint-mc'.

S	W	Name ↓	Last Success	Last Failure	Last Duration	Fav
●	☀	nsuite-daint-gpu	18 hr - #1	N/A	21 min	🔄 ☆
●	☀	nsuite-daint-mc	18 hr - #17	N/A	32 min	🔄 ☆

Below the table, there are options for 'Icon: S M L' and 'Legend' with links for 'RSS for all', 'RSS for failures', and 'RSS for just latest builds'.

Figure 2: The web dashboard from the CSCS Jenkins CI instance

Jenkins has been configured to check out NSuite daily, then compile and run the benchmarks and validation tests for the Arbor simulation engine on the multicore and GPU partitions of Piz Daint.

1.2.3 Workflow 3: HPC System Validation

1.2.3.1 User story

As system administrator on an HPC cluster, I want to run tests to verify that any changes that I make during regular maintenance or upgrade downtimes do not negatively affect application performance or correctness before handing the system back to users.

1.2.3.2 Solution

To set up the workflow before performing any maintenance, first configure NSuite for the target system and create a script that will be used to check for validation and performance regressions by following these steps:

- 1) Write a configuration script that sets up the recommended environment for building and running the simulation engines on the HPC cluster.
- 2) Run the benchmarks and validation tests and record the benchmark timings to use as a baseline to compare against after every maintenance.
- 3) Write a script that analyses NSuite output to check that validation tests passed, and that the benchmark performance is not slower than the baseline benchmark timings from step 2.

After each maintenance, run the framework with the configuration from step 1, and check that the validation tests pass and that the benchmark results are not slower than the original benchmarks using the script from step 3. Typically, this step would also be automated as part of a larger framework that runs tests for other important applications and libraries, as well as tests of system functionality, such as high-speed file systems and the cluster's network.

2. The Framework

This part of the report will give a high-level overview of the NSuite framework. It is beyond the scope of this report to go into great detail, particularly given that the framework is a *living document* that will change over time after this Deliverable is submitted. Readers interested in technical details of the framework are encouraged to check the repository <https://github.com/arbtor-sim/arbtor> and documentation at <https://nsuite.readthedocs.io>.

The framework is an open source software available on GitHub (<https://github.com/arbtor-sim/nsuite>). The workflow for running the benchmarks and validation tests follows three general steps:

- 1) **The user** downloads the framework by cloning the git repository.
- 2) **The framework** downloads and compiles at least one of the simulation engines.
- 3) **The framework** runs benchmark and validation tests on the compiled engine(s).

An example of a simple script that downloads the framework, builds all the simulation engines, and then runs benchmarks and all validation tests is presented in Figure 3.

```
# Obtain the framework from GitHub.
git clone https://github.com/arbtor-sim/nsuite.git
cd nsuite
# NSuite Downloads and installs arbor, neuron and coreneuron.
install-local.sh arbor neuron coreneuron
# NSuite runs all of the benchmarks on all the engines.
run-bench.sh arbor neuron coreneuron
# NSuite runs all of the validation tests on all the engines.
run-validate.sh arbor neuron coreneuron
```

Figure 3: Example of a script for running all benchmarks and validation tests

The framework uses bash scripts to implement each step, because bash is universally available on all the HPC, Linux and Mac OS X systems typically used by developers. Users can customise each step of the workflow, including which version of each simulation engine to use, how engines are compiled, and how benchmarks and tests are run.

2.1 Supported Simulation Engines

The default versions of each simulation engine are listed in Table 1. The framework automatically downloads or checks out the source code, depending on the type of the source. All simulation engines are built from source; there is currently no support for installing from a package or precompiled binary. The benefit of this approach is that the framework acts as a reference on how to compile and install the simulation engines, and it matches both the approach used by developers when working on source code, and that used typically by HPC centres to install software.

Table 1: Default versions of each simulation engine downloaded by NSuite

Simulation Engine	Version	Source
Arbor	v0.2	https://github.com/arbor-sim/arbor
NEURON	v7.6	https://www.neuron.yale.edu/neuron/download
CoreNeuron	v0.13	https://github.com/BlueBrain/CoreNeuron

The source (either a git repository or optionally an online zip file of the source code) and version can be customised by the user.

Adding support for a new simulation engine is relatively simple for simulation engines that:

- Support the features required to implement the benchmark and validation tests.
- Support output of performance metrics (at a minimum, the time to solution) and the model outputs required for validation (for example, voltage traces).
- Can be configured and built using tools like autotools or CMake.

A benchmark or test can be skipped if a simulation engine does not support features required to implement that test. For example, CoreNeuron currently does not support output of voltage traces, so corresponding tests that validate voltage traces are skipped.

2.2 Building Simulation Engines

The framework automatically downloads and compiles the simulation engines. The default compilation options can be customised, including the following ones:

- Enable GPU support for engines with GPU implementations.
- Enable MPI support.
- Define which compiler to use.
- Choose the Python version to use.

Some custom scripts are provided and maintained in the framework. For example, the script `nsuite/systems/daint-gpu.sh` will compile the simulation engines with GPU support on the Piz Daint supercomputer at the CSCS. The user can add their own custom scripts that set the compilation options, and configure the compilation and execution environment on the target system. The custom script is supplied as an argument to the `install-local.sh` script, which will save the configuration information to use when running benchmarks and validation tests.

2.3 Benchmarks

A performance benchmark runs a model that encapsulates one or more performance characteristics. The three main metrics for the performance of a model are enumerated below, with the engine(s) that support their measurement in parenthesis:

- 1) The time to solution (Arbor, NEURON, CoreNeuron)
- 2) The memory consumption per cell (Arbor, CoreNeuron)
- 3) The energy consumption (Arbor)

A metric is only available for a simulation engine if the engine has support for measuring and reporting on that metric. For example, Arbor is the only simulation engine that reports energy consumption, and then only on Cray systems. Benchmark outputs report time to solution for all of the simulators, and optionally reports the other metrics, if and when they are available.

The benchmarks are parameterised and, for convenience, small, medium and large parameter sets are provided for each benchmark. A generic benchmark script is provided with arguments for

specifying the simulation engines, benchmarks and parameter sets to launch. NSuite attempts to detect the optimal launch configuration on the target system, which the user can also override. The performance metrics from each benchmark are stored in a standard JSON file format that can be post-processed by the user or an automated workflow.

The first version of NSuite released with this Deliverable has two benchmarks: *ring* and *k-way*. Both benchmarks model a network of multi-compartment cells with Hodgkin-Huxley soma and passive dendrites, and randomly generated morphologies. The models are parameterised using the complexity of cell morphologies, the number of cells in the model, ring size, and the minimum delay of the network. The cell morphologies are randomised, in the sense that each cell in the model has a random morphology; however, two runs of the same benchmark will use the same set of cells, so that results between benchmarks are comparable. Morphologies are varied to avoid unrealistic benchmark timings, because models with identical morphologies can be solved much more efficiently on some architectures (e.g. Arbor on GPUs).

The ring model connects the cells in ring networks, whereby each cell has an incoming connection from the previous cell, and an outgoing connection to the next cell in the ring, such that a spike in one cell will cause a chain of spikes around the ring. By virtue of having a very simple network, this benchmark can be used to test the computational throughput of a simulation engine.

The k-way benchmark extends the ring benchmark by adding a user-specified number of synapses with exponential dynamics to each cell, and connecting them in a random network with zero weight on each connection. This benchmark generates the same solutions as the ring benchmark, by virtue of the random connections being unweighted. The spiking frequency of cells can be controlled by varying the ring size and minimum delay. This is a much more demanding benchmark, because the additional synapses on each cell require more computations, and because each spike generates many more post-synaptic events to deliver to synapses on cells.

2.4 Validation Tests

A validation test runs a particular model, representing a physical system to simulate, against one or more sets of parameters, and compares the output to a reference solution. If the output deviates from the reference by more than a given threshold, the respective test is marked as a FAIL for that simulator. Simulator output for each model and parameter set is stored in NetCDF format, where it can be analysed with generic tools.

New validation models can be added by creating a generic run script that invokes the implementation of the model for a given simulator, generates the reference data as required, and runs the comparison analysis. Simulator implementations that require compilation will be built at install time. More details can be found in the NSuite reference documentation.

In the first version of NSuite, there is one validation model implemented for Arbor and NEURON, with two parameter sets. The model comprises a single-compartment neuron with passive channels and a single synapse with exponential dynamics, which is triggered at the beginning of the simulation. The parameters are the maximum simulation time step and the initial synaptic conductance. Simulator output is compared with an explicitly integrated solution to the corresponding ordinary differential equation that can be solved exactly. CoreNeuron is not supported by this validation model, because it does not support the output of voltage traces, which is required to compare solutions. CoreNeuron runs models defined in NEURON, so support for CoreNeuron can be added easily when a version of CoreNeuron that supports voltage traces is released.

2.5 Supported HPC Systems

The framework has been tested on large HPC systems at ETHZ-CSCS and JUELICH-JSC, as detailed in Table 2. It was also run on developers' computers during development.

Table 2: List of the systems that the NSuite framework has been tested on so far

System	Architecture	Location
Daint-MC	Cray XC-40/50: Intel Haswell Xeon, NVIDIA P100 GPU	CSCS
Daint-GPU	Cray XC-40/50: Dual socket Intel Broadwell Xeon	CSCS
Tave	Cray XC-40: Intel Xeon Phi (KNL)	CSCS
JUWELS	Bull modular cluster: Dual socket Intel Skylake Xeon	JSC

NSuite includes configuration scripts for optimal performance on each system.

2.6 Continuous Integration

One of the key automated workflows for which NSuite was designed is continuous integration (CI). A continuous integration server will check out, build and test software whenever a change is made or proposed. Automating testing encourages developers to make frequent small updates, thus ensuring that bugs and performance regressions are caught early.

CI is particularly important for software that is designed to run on different HPC architectures, to ensure that changes made when targeting one architecture do not cause unforeseen problems on another platform. The CSCS runs a Jenkins CI server on its flagship Piz Daint system, which is available to developers in the HBP. At the time of writing, this CI service used the NSuite framework to test the latest version of Arbor on Piz Daint every day, the dashboard for which is in Figure 2. The scripts used to run Arbor on the CSCS Jenkins server are in the NSuite repository, and can be used as templates for running NSuite on other Jenkins servers.

3. Status and Future Work

As of the submission date of this Deliverable (SGA2 Month 12), the NSuite framework provides infrastructure for compiling the supported simulation engines and running benchmarks and validation tests. The framework also includes two benchmarks and one validation test, and custom configurations for HPC systems at JUELICH-JSC and ETHZ-CSCS.

The M12 release has focused on the framework described in this report, with few benchmarks and validation tests. Framework development will continue in the second year of SGA2, focusing on the following tasks:

- 1) Adding more benchmarks and validation tests
- 2) Integrating the NSuite framework in CI workflows for the Arbor development
- 3) Adding custom configurations for more HPC systems

This document presents a snapshot of the current NSuite framework, which will continue to evolve as more tests and systems are added.