frontiers | Frontiers in Neuroinformatics

ORIGINAL RESEARCH
published: 11 May 2022
doi: 10.3389/fninf.2022.837549

A Modular Workflow for Performance Benchmarking of Neuronal Network Simulations

Jasper Albers[1,2]*, *Jari Pronold*[1,2], *Anno Christopher Kurth*[1,2], *Stine Brekke Vennemo*[3], *Kaveh Haghighi Mood*[4], *Alexander Patronis*[4], *Dennis Terhorst*[1], *Jakob Jordan*[5], *Susanne Kunkel*[3], *Tom Tetzlaff*[1], *Markus Diesmann*[1,6,7] and *Johanna Senk*[1]

[1] Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6) and JARA-Institute Brain Structure-Function Relationships (INM-10), Jülich Research Centre, Jülich, Germany, [2] RWTH Aachen University, Aachen, Germany, [3] Faculty of Science and Technology, Norwegian University of Life Sciences, Ås, Norway, [4] Jülich Supercomputing Centre (JSC), Jülich Research Centre, Jülich, Germany, [5] Department of Physiology, University of Bern, Bern, Switzerland, [6] Department of Physics, Faculty 1, RWTH Aachen University, Aachen, Germany, [7] Department of Psychiatry, Psychotherapy and Psychosomatics, School of Medicine, RWTH Aachen University, Aachen, Germany

Modern computational neuroscience strives to develop complex network models to explain dynamics and function of brains in health and disease. This process goes hand in hand with advancements in the theory of neuronal networks and increasing availability of detailed anatomical data on brain connectivity. Large-scale models that study interactions between multiple brain areas with intricate connectivity and investigate phenomena on long time scales such as system-level learning require progress in simulation speed. The corresponding development of state-of-the-art simulation engines relies on information provided by benchmark simulations which assess the time-to-solution for scientifically relevant, complementary network models using various combinations of hardware and software revisions. However, maintaining comparability of benchmark results is difficult due to a lack of standardized specifications for measuring the scaling performance of simulators on high-performance computing (HPC) systems. Motivated by the challenging complexity of benchmarking, we define a generic workflow that decomposes the endeavor into unique segments consisting of separate modules. As a reference implementation for the conceptual workflow, we develop `beNNch`: an open-source software framework for the configuration, execution, and analysis of benchmarks for neuronal network simulations. The framework records benchmarking data and metadata in a unified way to foster reproducibility. For illustration, we measure the performance of various versions of the `NEST` simulator across network models with different levels of complexity on a contemporary HPC system, demonstrating how performance bottlenecks can be identified, ultimately guiding the development toward more efficient simulation technology.

Keywords: spiking neuronal networks, benchmarking, large-scale simulation, high-performance computing, workflow, metadata

OPEN ACCESS

Edited by:
Ludovico Minati,
Tokyo Institute of Technology, Japan

Reviewed by:
Justin Wozniak,
Argonne National Laboratory (DOE), United States
David Phillip Nickerson,
The University of Auckland, New Zealand

***Correspondence:**
Jasper Albers
j.albers@fz-juelich.de

Received: *16 December 2021*
Accepted: *11 March 2022*
Published: *11 May 2022*

Citation:
Albers J, Pronold J, Kurth AC, Vennemo SB, Haghighi Mood K, Patronis A, Terhorst D, Jordan J, Kunkel S, Tetzlaff T, Diesmann M and Senk J (2022) A Modular Workflow for Performance Benchmarking of Neuronal Network Simulations. Front. Neuroinform. 16:837549. doi: 10.3389/fninf.2022.837549

1. INTRODUCTION

Past decades of computational neuroscience have achieved a separation between mathematical models and generic simulation technology (Einevoll et al., 2019). This enables researchers to simulate different models with the same simulation engine, while the efficiency of the simulator can be incrementally advanced and maintained as a research infrastructure. Increasing computational efficiency does not only decrease the required resources of simulations, but also allows for constructing larger network models with an extended explanatory scope and facilitates studying long-term effects such as learning. Novel simulation technologies are typically published together with verification—evidence that the implementation returns correct results—and validation—evidence that these results are computed efficiently. Verification implies correctness of results with sufficient accuracy for suitable applications as well as a flawless implementation of components confirmed by unit tests. For spiking neuronal network simulators, such applications are simulations of network models which have proven to be of relevance for the field. In a parallel effort, validation aims at demonstrating the added value of the new technology for the community. To this end, the new technology is compared to previous studies on the basis of relevant performance measures.

Efficiency is measured by the resources used to achieve the result. Time-to-solution, energy-to-solution and memory consumption are of particular interest. For the development of neuromorphic computing systems, efficiency in terms of low power consumption and fast execution is an explicit design goal: simulations need to be able to cope with limited resources, for example, due to hardware constraints. Real-time performance, meaning that simulated model time equals wall-clock time, is a prerequisite for simulations interacting with the outer world, such as in robotics. Even faster, sub-real-time simulations enable studies of slow neurobiological processes such as brain development and learning, which take hours, days, or more in nature. High-performance computing (HPC) benchmarking studies usually assess the scaling performance of the simulation architecture by incrementally increasing the amount of employed hardware resources (e.g., compute nodes). In weak-scaling experiments, the size of the simulated network model is increased proportionally to the computational resources, which keeps the workload per compute node fixed if the simulation scales perfectly. Scaling neuronal networks, however, inevitably leads to changes in the network dynamics (van Albada et al., 2015b). Comparisons between benchmarking results obtained at different scales are therefore problematic. For network models of natural size describing the correlation structure of neuronal activity, strong-scaling experiments (in which the model size remains unchanged) are more relevant for the purpose of finding the limiting time-to-solution. For a formal definition of strong and weak scaling refer to page 123 of Hager and Wellein (2010) and for pitfalls in interpreting the scaling of network simulation code see van Albada et al. (2014). When measuring time-to-solution, studies distinguish between different phases of the simulation, in the simplest case between a setup phase of network construction and the actual simulation phase of state propagation. Such benchmark metrics not only depend on the simulation engine and its options for time measurements (see, e.g., Jordan et al., 2018; Golosio et al., 2021), but also on the network model. The simulated activity of a model may not always be stationary over time, and transients with varying firing rates are reflected in the computational load. For an example of transients due to arbitrary initial conditions see Rhodes et al. (2019), and for an example of non-stationary network activity, refer to the meta-stable state of the multi-area model described by Schmidt et al. (2018a). Studies assessing energy-to-solution need to specify whether only the power consumption of the compute nodes is considered or interconnects and required support hardware are also accounted for (van Albada et al., 2018).

The omnipresence of benchmarks in studies on simulation technology demonstrates the relevance of efficiency. The intricacy of the benchmarking endeavor, however, not only complicates the comparison between these studies, but also reproducing them. Neuroscientific simulation studies are already difficult to reproduce (Crook et al., 2013; McDougal et al., 2016; Rougier et al., 2017; Gutzen et al., 2018; Pauli et al., 2018; Gleeson et al., 2019), and benchmarking adds another layer of complexity. Reported benchmarks may differ not only in the structure and dynamics of the employed neuronal network models, but also in the type of scaling experiment, soft- and hardware versions and configurations, as well as in the analysis and presentation of the results. **Figure 1** illustrates the complexity of benchmarking experiments in simulation science and identifies five main dimensions: "Hardware configuration", "Software configuration", "Simulators", "Models and parameters", and "Researcher communication". The following presents examples specific to neuronal network simulations, demonstrating the range of each of the five dimensions.

Different simulators, some with decades of development, allow for large-scale neuroscientific simulations (Brette et al., 2007). We distinguish between simulators that run on conventional HPC systems and those that use dedicated neuromorphic hardware. Prominent examples of simulators for networks of spiking point-neurons are `NEST` (Morrison et al., 2005b; Gewaltig and Diesmann, 2007; Plesser et al., 2007; Helias et al., 2012; Kunkel et al., 2012, 2014; Ippen et al., 2017; Kunkel and Schenck, 2017; Jordan et al., 2018; Pronold et al., 2021, 2022) and `Brian` (Goodman and Brette, 2008; Stimberg et al., 2019) using CPUs; `GeNN` (Yavuz et al., 2016; Knight and Nowotny, 2018, 2021; Stimberg et al., 2020; Knight et al., 2021) and `NeuronGPU` (Golosio et al., 2021) using GPUs; `CARLsim` (Nageswaran et al., 2009; Richert et al., 2011; Beyeler et al., 2015; Chou et al., 2018) running on heterogeneous clusters; and the neuromorphic hardware `SpiNNaker` (Furber et al., 2014; Rhodes et al., 2019). `NEURON` (Carnevale and Hines, 2006; Migliore et al., 2006; Lytton et al., 2016) and `Arbor` (Akar et al., 2019) aim for simulating morphologically detailed neuronal networks.

The hardware and software configurations used in published benchmark studies are diverse because both underlie updates and frequent releases. In addition, different laboratories may not have access to the same machines. Therefore, HPC benchmarks are performed on different contemporary compute clusters or supercomputers. For example, `NEST` benchmarks have been

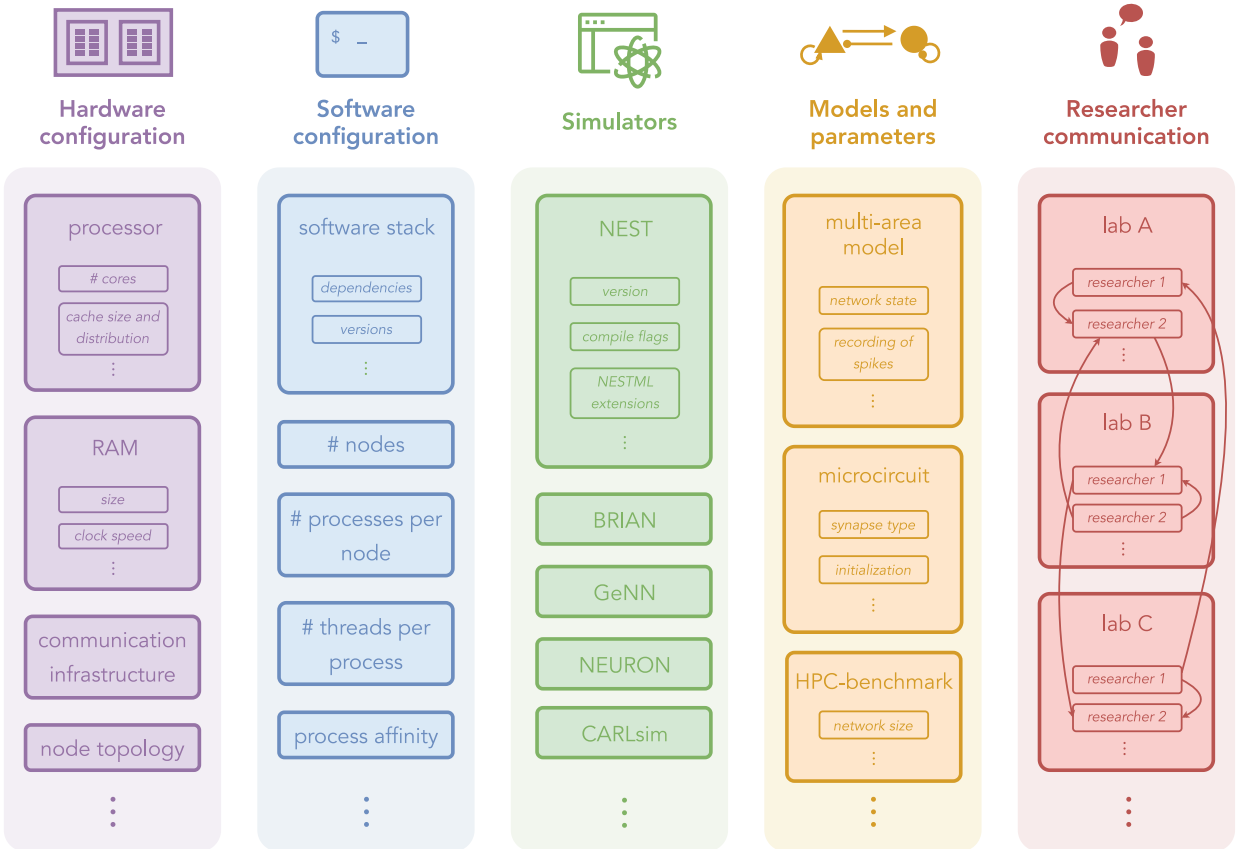

FIGURE 1 | Dimensions of HPC benchmarking experiments with examples from neuronal network simulations. Hardware configuration: computing architectures and machine specifications. Software configuration: general software environments and instructions for using the hardware. Simulators: specific simulation technologies. Models and parameters: different models and their configurations. Researcher communication: knowledge exchange on running benchmarks.

conducted on the systems located at Research Center Jülich in Germany but also on those at the RIKEN Advanced Institute for Computational Science in Japan (e.g., Helias et al., 2012; Jordan et al., 2018). To assess the performance of GPU-based simulators, the same simulation is typically run on different GPU devices; from low-end gaming GPUs to those installed in high-end HPC clusters (Knight and Nowotny, 2018; Golosio et al., 2021). This variety can be beneficial; performing benchmark simulations on only a single system can lead to unwanted optimization toward that type of machine. However, comparing results across different hard- and software is complicated and requires expert knowledge of the compared technologies in order to draw reasonable conclusions.

The modeling community distinguishes between functional models, where the validation is concerned with the questions if and how well a specific task is solved, and non-functional models, where an analysis of the network structure, dynamics, and activity is used for validation. Simulating the same model using different simulation engines often results in activity data which can only be compared on a statistical level. Spiking activity, for example, is typically evaluated based on distributions of quantities such as the average firing rate, rather than on precise spike times (Senk et al., 2017; van Albada et al., 2018). Reasons for that are inevitable differences between simulators such as different algorithms, number resolutions, or random number generators, combined with the fact that neuronal network dynamics is often chaotic, rapidly amplifying minimal deviations (Sompolinsky et al., 1988; van Vreeswijk and Sompolinsky, 1998; Monteforte and Wolf, 2010). The most frequently used models to demonstrate simulator performance are balanced random networks similar to the one proposed by Brunel (2000): generic two-population networks with 80% excitatory and 20% inhibitory neurons, and synaptic weights chosen such that excitation and inhibition are approximately balanced, similar to what is observed in local cortical networks. Variants differ not only in the parameterization but also in the neuron, synapse, and plasticity models, or other details. Progress in NEST development is traditionally shown by upscaling a model of this type, called "HPC-benchmark model", which employs leaky integrate-and-fire (LIF) neurons, alpha-shaped post-synaptic currents, and spike-timing-dependent plasticity (STDP) between excitatory neurons. The detailed model description and parameters can be

found in Tables 1–3 of the Supplementary Material of Jordan et al. (2018). Other versions include a network of Izhikevich model neurons and STDP (Izhikevich, 2003) used by Yavuz et al. (2016) and Golosio et al. (2021), the COBAHH model with Hodgkin-Huxley type neurons and conductance-based synapses (Brette et al., 2007) used by Stimberg et al. (2020), and a version with excitatory LIF and inhibitory Izhikevich model neurons where excitatory synapses are updated with STDP and inhibitory-to-inhibitory connections do not exist is used by Chou et al. (2018). Even though balanced random networks are often used for weak-scaling experiments, they describe the anatomical and dynamical features of cortical circuits only at a small spatial scale and the upscaling affects the network dynamics (see van Albada et al., 2015b as indicated above). At larger scales, the natural connectivity becomes more complex than what is captured by this model type. Therefore, models of different complexity need to be benchmarked to guarantee that a simulation engine performs well across use cases in the community. In addition to the HPC-benchmark model, this study employs two more elaborate network models: the "microcircuit model" proposed by Potjans and Diesmann (2014) and the "multi-area model" by Schmidt et al. (2018a). The microcircuit model is an extension of the balanced random network model with an excitatory and an inhibitory neuron population in each of four cortical layers with detailed connectivity derived from experimental studies. The model spans 1 mm^2 of cortical surface, represents the cortical layers at their natural neuron and synapse densities, and has recently been used to compare the performance of different simulation engines; for instance, `NEST` and `SpiNNaker` (Senk et al., 2017; van Albada et al., 2018; Rhodes et al., 2019); `NEST`, `SpiNNaker`, and `GeNN` (Knight and Nowotny, 2018); and `NEST` and `NeuronGPU` (Golosio et al., 2021). The multi-area model comprises 32 cortical areas of the visual system where each is represented by an adapted version of the microcircuit model; results are available for `NEST` (van Albada et al., 2021) and `GeNN` (Knight and Nowotny, 2021). Comparing the performance of the same model across different simulators profits from a common model description. The simulator-independent language `PyNN` (Davison et al., 2009), for example, enables the use of the same executable model description for different simulator back ends. Testing new technologies only with a single network model is, however, not sufficient for general-purpose simulators and comes with the danger of optimizing the code base for one application, while impairing the performance for others.

Problems to reproduce the simulation outcome or compare results across different studies may not only be technical but also result from a miscommunication between researchers or a lack of documentation. Individual, manual solutions for tracking the hardware and software configuration, the simulator specifics, and the models and parameters used in benchmarking studies have, in our laboratories, proven inefficient when scaling up the number of collaborators. This effect is amplified if multiple laboratories are involved. Similar inter-dependencies are also present between the other four dimensions of **Figure 1**, making it hard to produce long-term comparable results; the exhibited intricacy of benchmarking is susceptible to errors as, for instance, small details in parameterization or configuration may have a large impact on performance.

Standardizing benchmarks can help to control the complexity but represents a challenge for the fast-moving and interdisciplinary field of computational neuroscience. While the field had some early success in the area of compartmental modeling (Bhalla et al., 1992) and Brette et al. (2007) made initial steps for spiking neuronal networks, neither a widely accepted set of benchmark models nor guidelines for performing benchmark simulations exist. In contrast, benchmarks are routinely employed in computer science, and established metrics help to assess the performance of novel hardware and software. The `LINPACK` benchmarks (Dongarra et al., 2003), for example, were initially released in 1979, and the latest version is used to rank the world's top supercomputers by testing their floating-point computing power (TOP500 list). Although this strategy has been successful for many years, it has also been criticized as misguiding hardware vendors toward solutions with high performance in formalized benchmarks but disappointing performance in real-world applications[1]. For the closely related field of deep learning, Dai and Berleant (2019) summarize seven key properties that benchmarking metrics should fulfill: relevance, representativeness, equity, repeatability, cost-effectiveness, scalability, and transparency. There exist standard benchmarks for machine learning and deep learning applications such as computer vision and natural language processing with standard data sets and a global performance ranking. The most prominent example is `MLPerf`[2] (Mattson et al., 2020). Another example is the High Performance `LINPACK` for Accelerator Introspection (`HPL-AI`) benchmark[3] which is the mixed-precision counterpart to the `LINPACK` benchmarks. Ostrau et al. (2020) propose a benchmarking framework for deep spiking neural networks and they compare results obtained with the simulators `Spikey` (Pfeil et al., 2013), `BrainScales` (Schemmel et al., 2010), `SpiNNaker`, `NEST`, and `GeNN`.

For measuring and comparing the scaling performance of large-scale neuronal network model simulations, there exists, to our knowledge, no unifying approach, yet. Recently, more laboratories make use of established simulators rather than developing their own, and computing resources have become available and interchangeable. The resulting increase in the size of user-communities comes with the demand for even more flexible and efficient simulators with demonstrated performance. To keep up with this progress, we see the need for a common benchmarking framework. We envision a consistently managed array of standard benchmark models together with standard ways for running them. The five dimensions outlined above lend themselves to a modular framework integrating distinct components which can be updated, extended, or replaced independently. The framework needs to cover all steps of the benchmarking process from configuration, to execution, to handling of results. For enabling

[1] https://www.technologyreview.com/2010/11/08/199100/why-chinas-new-supercomputer-is-only-technically-the-worlds-fastest
[2] https://mlcommons.org
[3] https://www.icl.utk.edu/hpl-ai

comparability and reproducibility, all relevant metadata and data need to be tracked. In this work, we develop a conceptual benchmarking workflow that meets these requirements. For a reference implementation named beNNch, we employ the JUBE Benchmarking Environment[4] and the simulator NEST in different versions (Gewaltig and Diesmann, 2007), and we assess the time-to-solution for the HPC-benchmark model, the microcircuit model (Potjans and Diesmann, 2014), and the multi-area model (Schmidt et al., 2018a) on the contemporary supercomputer JURECA-DC (Thörnig and von St. Vieth, 2021). The goal of this study is to set the cornerstone for reliable performance benchmarks facilitating the comparability of results obtained in different settings, and hence, supporting the development of simulators.

The Results section of this manuscript formalizes the general concepts of the benchmarking workflow (Section 2.1), implements these concepts into a reference benchmarking framework for the NEST simulator (Section 2.2), and applies the framework to generate and compare benchmarking data, thereby making a case for the relevance of benchmarking for simulator development (Section 2.3.1). After a discussion of our results in Section 3, Section 4 provides details of specific performance optimizations addressed in this work.

2. RESULTS

2.1. Workflow Concepts

We devise a generic workflow for performance benchmarking applicable to simulations running on conventional HPC architectures. The conceptual workflow depicted in **Figure 2** consists of four segments which depend on each other in a sequential fashion. The segments are subdivided into different modules which are related to the specific realizations used in our reference implementation of the workflow (Section 2.2). We use the term "workflow" to describe abstract concepts that are of general applicability with regard to benchmarking efforts, and "framework" to refer to the concrete software implementation we have developed. Further, we make the distinction between "internal" and "external" modules. Internal modules are considered essential building blocks of the workflow while external modules can be exchanged more readily. The following introduces each of the workflow's conceptual segments and explains how the proposed solution addresses the identified problems (cf. **Figure 1**).

2.1.1. Configuration and Preparation

The first of the four workflow segments consists of five distinct modules that together set up all necessary prerequisites for the simulation. First, the installation of the simulation software and its dependencies is handled by "software deployment", while "machine configuration" specifies parameters that control the simulation experiment conditions, as for example, how many compute nodes to reserve. Together, these two modules target the problem dimensions "hardware configuration", "software configuration", and "simulators". Addressing "models and parameters", the module "model" provides the network model implementation, while "model configuration" allows for passing parameters to the model such as the biological model time to be simulated, thereby separating the model from its parameters. Finally, the "user configuration" module confines user-specific data, such as file paths or compute budgets, to a single location.

2.1.2. Benchmarking

The second segment encompasses all modules related to actually running the benchmark simulation. Compute clusters typically manage the workload of the machine via queuing systems; therefore, compute-intensive calculations are submitted as jobs via scripts which define resource usage and hold instructions for carrying out the simulation. In the workflow, this is handled by the module aptly named "job script generation". Here, the first link between modules comes into play: the workflow channels model, user and machine configuration to create a job script and subsequently submit the script to the job queue via the module "job submission". With the simulator software prepared by the software-deployment module, "job execution" performs the model simulation given the job-submission parameters. While a simulation for neuroscientific research purposes would at this point focus on the output of the simulation, for example, neuronal spike times or voltage traces, benchmarking is concerned with the performance results. These are recorded in the final benchmarking module called "data generation".

2.1.3. Data- and Metadata Handling

A core challenge in conducting performance benchmarks is the handling of all produced data and metadata. While the former type of data here refers to the results of the performance measurements, the latter is an umbrella term describing the circumstances under which the data was recorded according to the dimensions of benchmarking (**Figure 1**). Since executing multiple simulations using different configurations, software, hardware, and models is an integral part of benchmarking, data naturally accumulates. Recording the variations across these dimensions leads to a multitude of metadata that needs to be associated to the measured data. Standardized formats for both types of data make the results comparable for researchers working with the same underlying simulation technology. The workflow segment "Data- and metadata handling" proposes the following solution. First, the raw performance data, typically stemming from different units of the HPC system, are gathered and unified into a standardized format, while the corresponding metadata is automatically recorded. Next, the metadata is associated to the unified data files, alleviating the need for manually keeping track of parameters, experiment choices and software environment conditions. While there are different possible solution for this, attaching the relevant metadata directly to the performance-data files simplifies filtering and sorting of results. Finally, "initial validation" allows for a quick glance at the results such that erroneous benchmarks can be swiftly identified.

[4]https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/_node.html

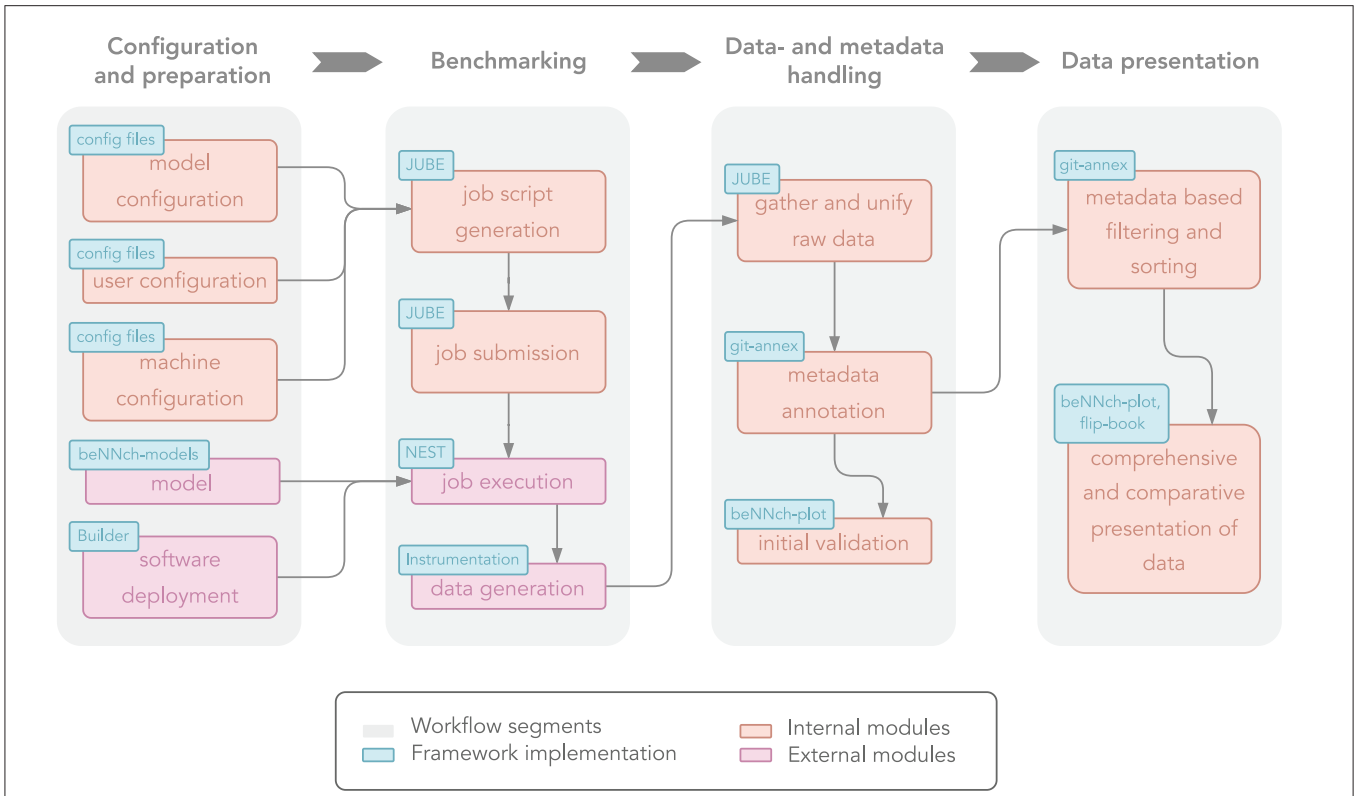

FIGURE 2 | Conceptual overview of the proposed benchmarking workflow. Light gray boxes divide the workflow into four distinct segments, each consisting of multiple modules. Internal modules are shown in orange and external ones in pink. Blue boxes indicate their respective realization in our reference implementation.

2.1.4. Data Presentation

This final workflow segment addresses the challenge of making the benchmarking results accessible and comparable such that meaningful conclusions can be drawn, thereby aiming to cope with the complexity that "Researcher communication" introduces. In a first step, "metadata based filtering and sorting" allows the user to dynamically choose the results to be included in the comparison. Here, dynamic means that arbitrary cuts through the hypercube of metadata dimensions can be selected such that the filtered results only differ in metadata fields of interest. Second, the data is presented in a format for which switching between benchmarks is intuitive, key metadata is given alongside the results, and data representation is standardized. The presentation of data should be comprehensive, consistent, and comparative such that the benchmarking results are usable in the long term. Thereby, the risk of wasting resources through re-generation of results is eliminated, making the corresponding software development more sustainable.

2.2. beNNch: A Reference Implementation

Building on the fundamental workflow concepts developed in Section 2.1, we introduce a reference implementation for modern computational neuroscience: beNNch[5]—a benchmarking framework for neuronal network simulations. The framework serves not only as a proof-of-concept, but also provides a software tool that can be readily used by neuroscientists and simulator developers. While beNNch is built such that plug-ins for any neuronal network simulator can be developed, we specifically implement compatibility with the NEST simulator (Gewaltig and Diesmann, 2007) designed for simulating large-scale spiking neuronal network models. In the following subsections, we detail software tools, templates, technologies, and user specifications needed to apply beNNch for benchmarking NEST simulations. Each of the conceptual modules of **Figure 2** is here associated with a concrete reference.

2.2.1. Builder

Reproducible software deployment is necessary for repeatability and comparability of the benchmarks. In favor of the usability of the benchmarking framework, however, we need to abstract non-relevant information on the hardware architecture and the software tool chain. The tool set is required to install software in a platform independent way and should not depend on a particular flavor of the operating system, the machine architecture or overly specific software dependencies. Additionally, it needs to be able to make use of system-provided tools and libraries, for example, to leverage machine specific MPI implementations. beNNch uses

[5]https://github.com/INM-6/beNNch

the tool Builder[6] for this purpose. Given a fixed software stack and hardware architecture, Builder provides identical executables by deriving the install instructions from "plan files". Integration with other package management systems such as easy_build (Geimer et al., 2014) or Spack (Gamblin et al., 2015) is achieved by using the same environment module systems[7]. Thereby, the required user interaction is minimized and, from a user perspective, installation reduces to the configuration of installation parameters. Given a specified variation of the software to be benchmarked, beNNch calls Builder to deploy the requested software. In doing so, Builder checks whether the software is already available and otherwise installs it according to the specifications in the plan file. The depth to which required dependencies need to be installed and which mechanisms are used depend on the conventions and prerequisites available at each execution site. For any installation, the used software stack—including library versions, compiler versions, compile flags, etc.—are recorded as metadata.

2.2.2. NEST

beNNch implements compatibility with the NEST simulator (Gewaltig and Diesmann, 2007), enabling the performance benchmarking of neuronal network simulations at the resolution of single neurons. The NEST software is complex, and the consequences of code modifications for performance are often hard to predict. NEST has an efficient C++ kernel, but network models and simulation experiments are defined via the user-friendly Python interface PyNEST (Eppler et al., 2009; Zaytsev and Morrison, 2014). To parallelize simulations, NEST provides two methods: for distributed computing, NEST employs the Message Passing Interface (MPI, Message Passing Interface Forum, 2009), and for thread-parallel simulation, NEST uses OpenMP (OpenMP Architecture Review Board, 2008).

2.2.3. Instrumentation

We focus our performance measurements on the time-to-solution. Acquiring accurate data on time consumption is critical for profiling and benchmarking. To this end, we make use of two types of timers to collect this data: the timers are either built-in to NEST on the C++ level, or they are included on the Python level as part of the PyNEST network-model description. The latter type of timers are realized with explicit calls to the function time.time() of the Python Standard Library's time. To achieve consistency throughout the framework, we use standardized variable names for the different phases of the simulation. **Figure 3** shows the simulation flow of a typical NEST simulation. During "network construction", neurons and auxiliary devices for stimulation and recording are created and subsequently connected according to the network-model description. Afterwards, in the course of "state propagation", the network state is propagated in a globally time-driven manner. This comprises four main phases which are repeated until the entire model time has been simulated: update of neuronal states, collocation of spikes in MPI-communication buffers, communication of spikes, and delivery of the received spikes to their respective thread-local targets. NEST's built-in timers provide a detailed look into the contribution of all four phases of state propagation, while timers on the Python level measure network construction and state propagation.

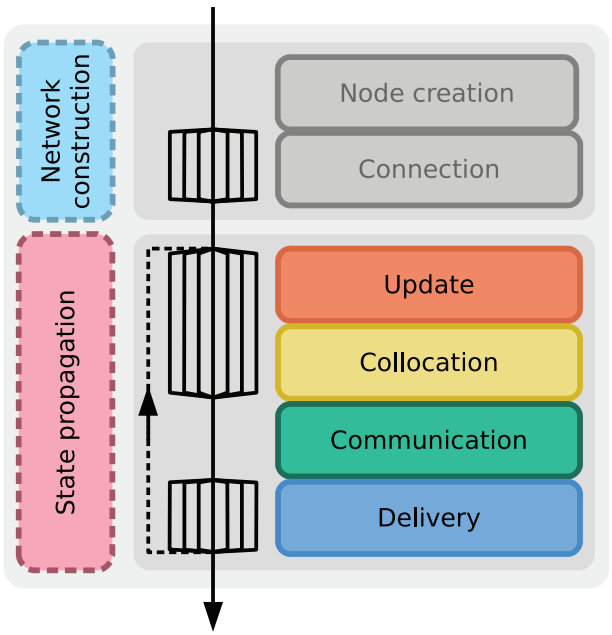

FIGURE 3 | Instrumentation to measure time-to-solution. Successive phases of a NEST simulation; time is indicated by top-down arrow. Fanning arrows denote parallel operation of multiple threads. The main phases network construction (cyan) and state propagation (pink) are captured by external timers on the Python level. Built-in NEST timers on the C++ level measure sub-phases: node creation and connection (both gray, not used in benchmark plots); update (orange), collocation (yellow), communication (green), and delivery (blue). The sub-phases of the state propagation are repeated until the simulation is finished as shown by the dashed arrow connecting delivery and update.

In NEST, the postsynaptic connection infrastructure is established during the "connection" phase. However, the presynaptic counterpart is typically only set up at the beginning of the state propagation phase (see Jordan et al., 2018, for details). In this work, we trigger this step deliberately and include it in our measurement of network-construction time rather than state-propagation time. Besides, it is common practice to introduce a short pre-simulation before the actual simulation to give the network dynamics time to level out; the state propagation phase is only recorded when potential startup transients have decayed (Rhodes et al., 2019). The model time for pre-simulation can be configured via a parameter in beNNch. For simplicity, **Figure 3** does not show this pre-simulation phase.

2.2.4. beNNch-models

We instantiate the "model" module with the repository beNNch-models[8] which contains a collection of PyNEST neuronal

[6]https://github.com/INM-6/Builder
[7]https://modules.readthedocs.io and http://lmod.readthedocs.io
[8]https://github.com/INM-6/beNNch-models

network models, i.e., models that can be simulated using the Python interface of NEST (Eppler et al., 2009). In principle, any such model can be used in conjunction with beNNch; only a few adaptations are required concerning the interfacing. On the input side, the framework needs to be able to set relevant model parameters. For recording the performance data, the required Python timers (Section 2.2.3) must be incorporated. On the output side, the model description is required to include instructions to store the recorded performance data and metadata in a standardized format. Finally, if a network model is benchmarked with different NEST versions that require different syntax, as is the case for switching between `NEST 2.X` and `NEST 3.X`, the model description also needs to be adjusted accordingly. Which model version is used in a simulation can thereby be deduced from knowing which simulator version was tested. For fine-grained version tracking, we additionally record the commit hash of `beNNch-models` and attach it as metadata to the results. Instructions on how to adapt existing models are provided in the documentation of `beNNch-models`.

The current version of beNNch provides benchmark versions of three widely studied spiking neuronal network models: the two-population HPC-benchmark model[9], the microcircuit model[10] by Potjans and Diesmann (2014) representing 1 mm^2 of cortical surface with realistic neuron and synapse densities, and the multi-area model[11] by Schmidt et al. (2018a,b) consisting of 32 microcircuit-like interconnected networks representing different areas of visual cortex of macaque monkey. The model versions used for this study employ the required modifications described above.

2.2.5. config files

When executing benchmarks, the main user interaction with beNNch consists of defining the characteristic parameters. We separate this from the executable code by providing `yaml`-based templates for "config files" to be customized by the user. Thereby, the information that defines a benchmark experiment is kept short and well arranged, limiting the number of files a user needs to touch and reducing the risk of user errors on the input side. **Listing 1** presents an excerpt from such a config file which has distinct sections to specify model, machine, and software parameters. While some parameters are model specific, standardized variable names are defined for parameters that are shared between models.

```yaml
parameterset:

    - name: model_parameters
      parameter:
       # can be either "metastable" or "ground"
       - {name: network_state, type: string, _: "metastable"}
       # biological model time to be simulated in ms
       - {name: model_time_sim, type: float, _: "10000."}
       # "weak" or "strong" scaling
       - {name: scaling_type, _: "strong"}

    - name: machine_parameters
      parameter:
       # number of compute nodes
       - {name: num_nodes, type: int, _: "4,8,12,16,24,32"}
       # number of MPI tasks per node
       - {name: tasks_per_node, type: int, _: "8"}
       # number of OpenMP threads per task
       - {name: threads_per_task, type: int, _: "16"}

    - name: software_parameters
      parameter:
       # simulator used for executing benchmarks
       - {name: simulator, _: "nest-simulator"}
       # simulator version
       - {name: version, _: "3.0"}
```

Listing 1: Excerpt of a config file in `yaml`-format for setting model, machine, and software parameters for benchmarking the multi-area model. When giving a list (e.g., for `num_nodes`), a job for each entry of the list is created. **Model parameters**: `network_state` describes particular model choices that induce different dynamical fixed points; `model_time_sim` defines the total model simulation time in ms; `scaling_type` sets up the simulation for either a weak- or a strong-scaling experiment. The former scales the number of neurons linearly with the used resources which might be ill-defined for anatomically constrained models. **Machine parameters**: `num_nodes` defines the number of nodes over which the scaling experiment shall be performed; `tasks_per_node` and `threads_per_task` specify the number of MPI tasks per node and threads per MPI task respectively. **Software parameters**: `simulator` and `version` describe which version of which simulator to use (and to install if not yet available on the machine).

2.2.6. JUBE

At this point, the first segment of the benchmarking workflow (**Figure 2**) is complete and hence all necessary requirements are set up: the software deployment provides the underlying simulator (here: NEST with built-in instrumentation), the models define the simulation, and the configuration specifies the benchmark parameters. This information is now processed by the core element of the framework: generating and submitting simulation jobs and gathering and unifying the obtained performance data. We construct this component of beNNch around the `xml`-based JUBE[4] software tool using its `yaml` interface. Built around the idea of benchmarking, JUBE can fulfill the role of creating job scripts from the experiment, user and machine configuration, their subsequent submission, as well as gathering and unifying of the raw data output. Here, we focus on the prevalent scheduling software SLURM (Yoo et al., 2003), but extensions to allow for other workload managers would be straightforward to implement. Our approach aims at high code re-usability. Model specific code is kept to a minimum, and where necessary, written in a similar way across models. Adhering to a common interface between JUBE scripts and models facilitates the integration of new models, starting from existing ones as a reference. Since JUBE can execute arbitrary code, we use it to also record metadata in

[9]Original repository: https://github.com/nest/nest-simulator/blob/master/pynest/examples/hpc_benchmark.py.
[10]Original repository: https://github.com/nest/nest-simulator/tree/master/examples/nest/Potjans_2014.
[11]Original repository: https://github.com/INM-6/multi-area-model.

conjunction with each simulation. This includes specifications of the hardware architecture as well as parameters detailing the run and model configuration.

2.2.7. git-annex

Without a mature strategy for sharing benchmark results, communication can be a major obstacle. Typically, each researcher has their preferred workflow, thus results are shared over different means of communication, for example, via email attachments, cloud-based storage options, or `git` repositories. This makes it difficult to maintain an overview of all results, especially if researchers from different labs are involved. Ideally, results would be stored in a decentralized fashion that allows for tracking the history of files while allowing on-demand access for collaborators. To this end, we use `git-annex`[12] as a versatile base technology; it synchronizes file information in a standard `git` repository while keeping the content of large files in a separate object store, thereby keeping the repository size at a minimum. `git-annex` is supported by the GIN platform[13] which we employ for organizing our benchmark results. In addition, it allows for metadata annotation: instead of relying on separate files that store the metadata, `git-annex` can directly attach them to the data files, thereby implementing the "metadata annotation" module. Previously this needed to be cataloged by hand, whereas now the framework allows for an automatic annotation, reducing the workload on researchers and thus probability of human mistakes. A downside of following this approach is a limitation to command line-based interaction. Furthermore, `git-annex` is not supported by some of the more widely used `git` repository hosting services such as GitHub or GitLab in favor of `Git LFS`.

A difficult task when scaling up the usage of the framework and, by extension, handling large amounts of results, is providing an efficient way of dealing with user queries for specific benchmark results. Attaching the metadata directly to the performance data not only reduces the visible complexity of the repository, but also provides an efficient solution: `git-annex` implements a native way of selecting values for metadata keys via `git-annex` "views", automatically and flexibly reordering the results in folders and sub-folders accordingly. For example, consider the case of a user specifying the `NEST` version to be `3.0`, the `tasks_per_node` to be either `4` or `8`, and the `network_state` to be either `metastable` or `ground`. First, `git-annex` filters out metadata keys for which only a single value is given; in our example, only benchmarks conducted with `NEST` version `3.0` remain. Second, a hierarchy of directories is constructed with a level for each metadata key for which multiple options are given. Here, the top level contains the folders "4" and "8", each containing sub-folders `metastable` and `ground` where the corresponding results reside. However, it may be difficult to judge exactly what metadata is important to collect; oftentimes, it is only visible in hindsight that certain metadata is relevant for the simulation performance. Therefore, recording as much metadata as possible would be ideal, allowing for retrospective investigations if certain metadata becomes relevant after run time. Importantly, a balance needs to be struck between recording large amounts of metadata and keeping the volume of annotations manageable. In our implementation, we choose to solve this issue by recording detailed metadata about the system, software, and benchmarks, but only attaching what we currently deem relevant for performance to the data. The remaining metadata is archived and stored alongside the data, thereby sacrificing ease of availability for a compact format. This way, if future studies discover that a certain hardware feature or software parameter is indeed relevant for performance, the information remains accessible also for previously simulated benchmarks while staying relatively hidden otherwise. Furthermore, using `git` as a base technology allows to collect data sets provided by different researchers in a curated fashion by using well-established mechanisms like branches and merge-request reviews. This use of `git-annex` thereby implements the "metadata based filtering and sorting" module of **Figure 2**.

2.2.8. beNNch-plot

To enable a comparison between plots of benchmark results across the dimensions illustrated in **Figure 1** it is paramount to use the same plotting style. To this end, we have developed the standalone plotting package `beNNch-plot`[14] based on `matplotlib` (Hunter, 2007). Here, we define a set of tools to create individual plot styles that can be combined flexibly by the user. The standardized definitions of performance measures employed by `beNNch` directly plug into this package. In addition, `beNNch-plot` includes default plot styles that can be readily used, and provides a platform for creating and sharing new ones. `beNNch` utilizes the default plot styles of `beNNch-plot` for both initial validation—a preliminary plot offering a quick glance at the results, thereby enabling a swift judgement whether any problems occurred during simulation—and visualization of the final results.

2.2.9. Flip-Book

When devising a method of presenting benchmark results we found the following aspects to be of crucial relevance for our purposes. First, it should be possible to navigate the results such that plots are always at the same screen position and have the same dimensions, thereby minimizing the effort to visually compare results. To achieve such a format, we decided to create a flip-book in which each slide presents the results of one experiment. Second, relevant metadata should be displayed right next to the plots. This can include similarities across the runs, but more importantly should highlight the differences. As each user might be interested in different comparisons, we let the user decide which kind of metadata should be shown. Third, it should be easy to select only the benchmarks of interest in order to keep the number of plots small. This is already handled by the filtering provided by `git-annex` views as described in Section 2.2.7. As an underlying technology for programmatically creating `HTML` slides we use

[12]https://git-annex.branchable.com
[13]https://gin.g-node.org
[14]https://github.com/INM-6/beNNch-plot

TABLE 1 | Shorthand notation and description of NEST versions used in this work.

Shorthand notation of NEST version	Description
2.20.2	Official 2.20.2 release (Fardet et al., 2021)
3.0rc	Release candidate for 3.0
3.0rc+ShrinkBuff	3.0rc plus shrinking MPI buffers
3.0rc+ShrinkBuff+SpikeComp	3.0rc+ShrinkBuff plus spike compression
3.0	Official 3.0 release (Hahne et al., 2021) = 3.0rc+ShrinkBuff+SpikeComp plus neuronal input buffers with multiple channels

jupyter notebooks[15] in conjunction with the open source HTML presentation framework reveal.js[16]. An exemplary flip-book containing the NEST performance results described in this work is published alongside the beNNch repository[17]. By respecting these considerations, our proposed solution offers a way of sharing benchmarking insights between researchers that is both scalable and flexible.

2.2.10. Exchanging External Modules

beNNch is written in a modular fashion; as such, it is possible to exchange certain modules without compromising the functionality of the framework. In particular, the "external modules" (see **Figure 2**) are implemented such that an exchange is straight-forward to implement. This section presents a recipe to exchange the "job execution" module, i.e., the simulator, along with necessary changes in "data generation" and "model" that follow:

First, the simulator to be substituted instead of NEST needs to be properly installed. Builder—our implementation of the "software deployment" module—provides the flexibility to install any software as well as make it loadable via a module system. Thus, a plan file specifying dependencies as well as source code location and installation flags needs to be created for the new simulator.

Second, models compatible with the new simulator need to be added. On the framework side, the execute commands may need to be adapted. Required adaptations to the models are the same as for PyNEST models and are described in Section 2.2.4.

Third, the instrumentation needs to be changed. As NEST has built-in instrumentation, only superficial timing measurements are necessary on the model level. Depending on the new simulator's existing ability to measure performance, timing measurements might need to be implemented on the simulator or model level. If different measurements than implemented are of interest, a simple addition to an existing list in beNNch suffices to add the recorded data to the csv-format result file.

[15]https://jupyter.org
[16]https://github.com/hakimel/reveal.js
[17]https://inm-6.github.io/beNNch

2.3. Using beNNch for Simulator Development

For the development of simulation software with the goal to optimize its performance, it is vital to focus efforts on those parts of the simulation loop that take the most time to complete. Benchmarking can help in identifying performance bottlenecks and testing the effect of algorithmic adaptations. However, the dimensions of benchmarking identified in **Figure 1** make this difficult: to guarantee that observed differences in performance are caused by changes in the simulator code, many variables need to be controlled for, such as hardware and software configurations as well as simulator versions. General-purpose simulators also need to be tested with respect to different settings and applications to ensure that a performance improvement in one case does not lead to a crucial decline in another case. Neuronal network simulators are one such example as they should exhibit reasonable performance for a variety of different models with different resource demands. A systematic assessment of the scaling performance covering the relevant scenarios is therefore a substantial component of the iterative simulator development.

beNNch, as an implementation of the workflow outlined in Section 2.1, provides a platform to handle the complexity of benchmarking while staying configurable on a low level. The following suggests how beNNch can support the process of detecting and tackling performance issues of a simulator. In a first step, exploration is necessary to identify the performance bottlenecks of the current version of the simulation engine. As many software and model parameters need to be explored, the centralized location of configuration parameters built into beNNch helps in maintaining an overview of conducted experiments. Neuronal network simulations can usually be decomposed into separate stages, such as neuronal update and spike communication. The instrumentation and visualization of these stages is part of beNNch and points the researcher to the respective sections in the code. If a potential bottleneck for a certain model is identified, tests with other models provide the basis for deciding whether these are model- and scale-specific or are present across models, hinting at long-reaching issues of the simulator. beNNch's native support for handling the benchmarking of multiple models alleviates the researchers from operating a different code base for every model. In the process of solving the simulator issue, running further benchmarks and directly comparing new results can assist in deciding which adaptations bear fruit. The standardized visualization tools of beNNch support spotting differences in performance plots. Finally, an ongoing development of a neuronal network simulator should respect the value of insights gained by resource-intensive benchmarks. To this end, beNNch implements a decentralized storage of standardized performance results. In addition to preserving information for the long term, this also helps in communicating between researchers working on the simulator's development.

2.3.1. Use Case: NEST Development

This section illustrates the relevance of performance benchmarks for the development of neuronal network simulators with the

example of recent changes to the NEST code base; for historical context see Section 4.1.1. We use beNNch to outline crucial steps of the development from the release candidate NEST 3.0rc to the final NEST 3.0 and also discuss improvements compared to the latest NEST 2 version (NEST 2.20.2, Fardet et al., 2021). **Table 1** summarizes the NEST versions employed in this study.

Regarding the dimensions of HPC benchmarking in **Figure 1**, this use case primarily addresses the "Simulators" dimension by testing different NEST versions and the "Models and parameters" dimension by testing different network models; the approach can be extended similarly to the other dimensions.

Our starting point is the weak-scaling experiments of the HPC-benchmark model (Jordan et al., 2018); the times for network construction and state propagation as well as the memory usage remain almost constant with the newly introduced 5g kernel (see their **Figures 7**, **8**). **Figure 4** shows similar benchmarks of the same network model conducted with beNNch using the release candidate in **Figure 4A** and the final release in **Figure 4B**. The graph design used here corresponds to the one used in the flip-book format by the framework. A flip-book version of the results shown in this work can be accessed via the GitHub Pages instance of the beNNch repository[17]. While the release candidate in **Figure 4A** exhibits growing state-propagation times when increasing the number of nodes, network-construction times stay constant and are, for $T_{\text{model}} = 1\,\text{s}$, small, making up less than 10% of the total simulation time. The phases "delivery" and "communication" both contribute significantly to the state-propagation time. Jordan et al. (2018) report real-time factors of about 500 (e.g., their **Figure 7C**) in contrast to values smaller than 40 shown here and their simulations are by far dominated by the delivery phase (see their Figure 12). A comparison of our data and the data of Jordan et al. (2018) is not straightforward due to the inherent complexity of benchmarking and we will here emphasize a few concurring aspects: first, Jordan et al. (2018) run their benchmarks on the dedicated supercomputers JUQUEEN (Jülich Supercomputing Centre, 2015) and K Computer (Miyazaki et al., 2012) while our benchmarks use the recent cluster JURECA-DC (Thörnig and von St. Vieth, 2021). Each compute node of the BlueGene/Q system JUQUEEN is equipped with a 16-core IBM PowerPC A2 processor running at 1.6 GHz and each node of the K Computer has an 8-core SPARC64 VIIIfx processor operating at 2 GHz; both systems provide 16 GB RAM per node. In contrast, the JURECA-DC cluster employs compute nodes consisting of two sockets, each housing a 64-core AMD EPYC Rome 7742 processor clocked at 2.2 GHz, that are equipped with 512 GB of DDR4 RAM. Here, nodes are connected via an InfiniBand HDR100/HDR network. Second, Jordan et al. (2018) use 1 MPI process per node and 8 threads per process while our simulations are performed throughout this study with 8 MPI processes per node and 16 threads per process. Third, Jordan et al. (2018) simulate 18,000 neurons per MPI process while we only simulate 11,250 neurons per process. This list of differences is not complete and only aims to illustrate that potential discrepancies in benchmarking results may be explained by differences in hardware, software, simulation and model configuration, and other aspects exemplified in **Figure 1**.

Having demonstrated that beNNch can perform weak-scaling experiments of the HPC-benchmark model as done in previous publications, we next turn to strong-scaling benchmarks of the multi-area model (Schmidt et al., 2018a). To fulfill the memory requirements of the model, at least three compute nodes of JURECA-DC are needed; here, we choose to demonstrate the scaling on four to 32 nodes. Initially, we compare the latest NEST 2 version (**Figure 5A**) with the release candidate for NEST 3.0 (**Figure 5B**). The improved parameter handling implemented in NEST 3.0rc reduces the network-construction time. However, the communication phase here largely dominates state propagation in both NEST versions shown; both use the original 5g kernel. Previous simulations of the HPC-benchmark model have not identified the communication phase as a bottleneck (Jordan et al., 2018, Figure 12). Communication only becomes an issue when then smallest delay in the network is of the same order as the computation step size because NEST uses the smallest delay as the communication interval for MPI. While the HPC-benchmark model uses 1.5 ms for all connections—which is a good estimate for inter-area connections—the multi-area model and microcircuit use distributed delays with a lower bound of 0.1 ms leading to a fifteen-fold increase in the number MPI communication steps.

The following identifies and eliminates the main cause for the large communication time in case of the multi-area model, thus introducing the first out of three performance-improving developments applied to NEST 3.0rc. Cross-node communication, handled in NEST by MPI, needs to strike a balance between the amount of messages to transfer and the size of each message. The size of the MPI buffer limits the amount of data that fits into a single message, and is therefore the main parameter controlling this balance. Ideally, each buffer would fit exactly the right amount of information by storing all spikes of the process relevant for the respective communication step. Due to overhead attached to operating on additional vectors, a scheme in which the buffer size adapts precisely for each MPI process for each communication step can be highly inefficient. Therefore, in cases where communication is roughly homogeneous, it is advantageous to keep the exchanged buffer between all processes the same size, as is implemented in NEST 3.0rc. While buffer sizes are constant across processes, NEST does adapt them over time to minimize the number of MPI communications. Concretely, whenever the spike information that a process needs to send exceeds what fits into one buffer, the buffer size for the next communication step is increased. However, the original 5g kernel of NEST does not shrink buffer sizes. In networks such as the multi-area model, the firing is not stationary over time; transients of high activity propagate through the network (Schmidt et al., 2018a). In general, startup transients may cause high spike rates only in the beginning of a simulation unless the network is carefully initialized (Rhodes et al., 2019). If the rates decrease, the spiking information becomes much smaller than the available space in the MPI buffer. Consequently, the original 5g kernel preserves unnecessarily large buffer sizes which results in the communication of useless data. To address

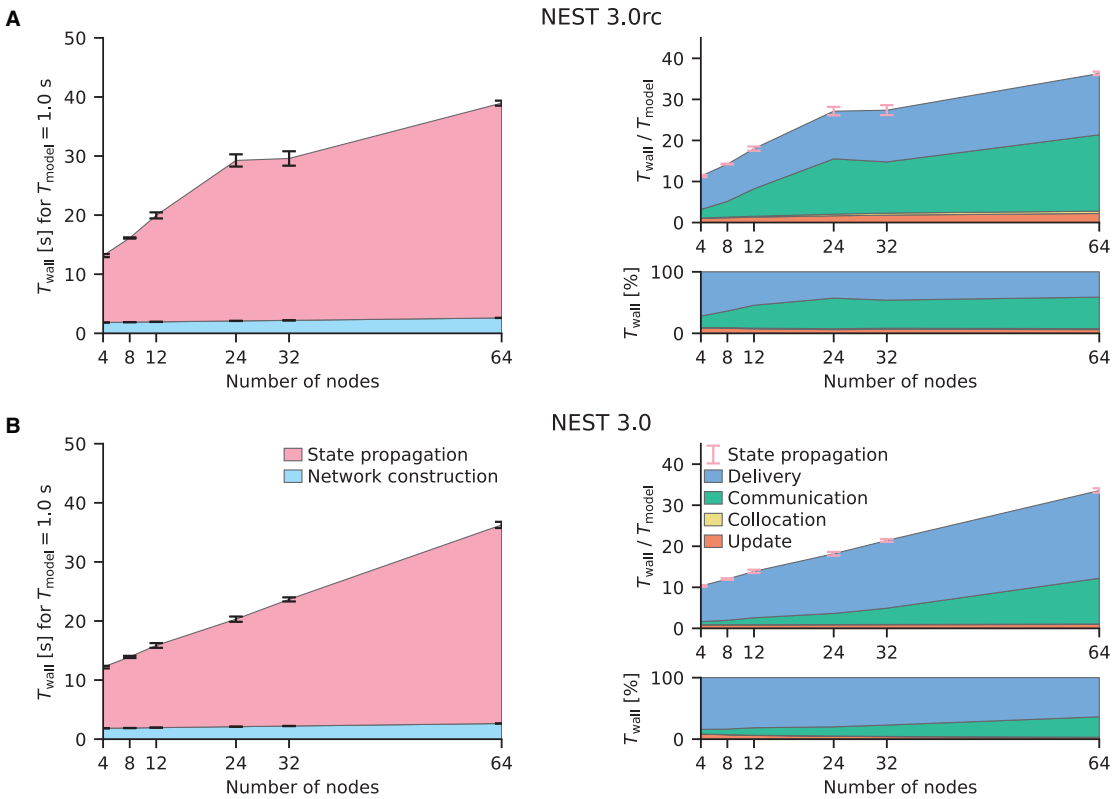

FIGURE 4 | Weak-scaling performance of the HPC-benchmark model on JURECA-DC. **(A)** `NEST 3.0rc`. The left graph shows the absolute wall-clock time T_{wall} measured with Python-level timers for both network construction and state propagation [legend in **(B)**]; the model time is $T_{model} = 1$ s. Error bars indicate variability across three simulation repeats with different random seeds. The top right graph displays the real-time factor defined as wall-clock time normalized by the model time. Built-in timers resolve four different phases of the state propagation [legend in **(B)**]: update, collocation, communication, and delivery. Pink error bars show the same variability of state propagation as the left graph. The lower right graph shows the relative contribution of these phases to the state-propagation time. Same colors used for phases as in **Figure 3**. **(B)** `NEST 3.0`. Same display as **(A)**.

this issue, a mechanism for automatically shrinking the buffer sizes has been introduced. For details see Section 4.1.2. The release candidate with the implementation of shrinking MPI buffers (`NEST 3.0rc+ShrinkBuff`) approximately halves the time spent in the communication phase compared to the original implementation (compare **Figures 5B,C**).

Next, we assess the strong-scaling performance of the microcircuit model (Potjans and Diesmann, 2014). The model size is similar to the size of one of the 32 areas of the multi-area model. The microcircuit therefore requires fewer resources. We show results of the model run on one to six compute nodes; for a detailed analysis of NEST's thread scaling performance on the example of this model refer to Kurth et al. (2021). Using `NEST 3.0rc`, the microcircuit is simulated faster than the HPC-benchmark and the multi-area models and achieves approximately real time ($T_{wall}/T_{model} \approx 1$, **Figure 6A**). The finer resolution of the vertical axis of the top-right graph reveals a small gap between the state propagation measured with Python timers and the sum of the phases timed on the C++ level which is not visible for the other models. The state-propagation time of the microcircuit is also dominated by the communication phase similarly to the respective benchmarks with the multi-area model (**Figure 5B**) and even increases with the number of nodes used. However, shrinking MPI buffers does not reduce communication significantly (data not shown), indicating that we face a different bottleneck with the microcircuit model. With on the order of 10^3 outgoing connections per neuron, a single neuron of this model has multiple targets on each MPI process and, in particular, on multiple threads of a given process. Since the 5g kernel is designed to send out a separate copy of a neuron's spiking information to each target thread, multiple copies of identical information about the activity of a presynaptic neuron may be sent to the same process, causing unnecessary communication load. To tackle this, we devise a spike compression algorithm which only requires transmitting the spiking information once to each MPI process where it is locally routed to the receiving threads. For details see Section 4.1.3. This algorithm leads to a significant reduction in communication time for the microcircuit model (compare **Figures 6A,B**).

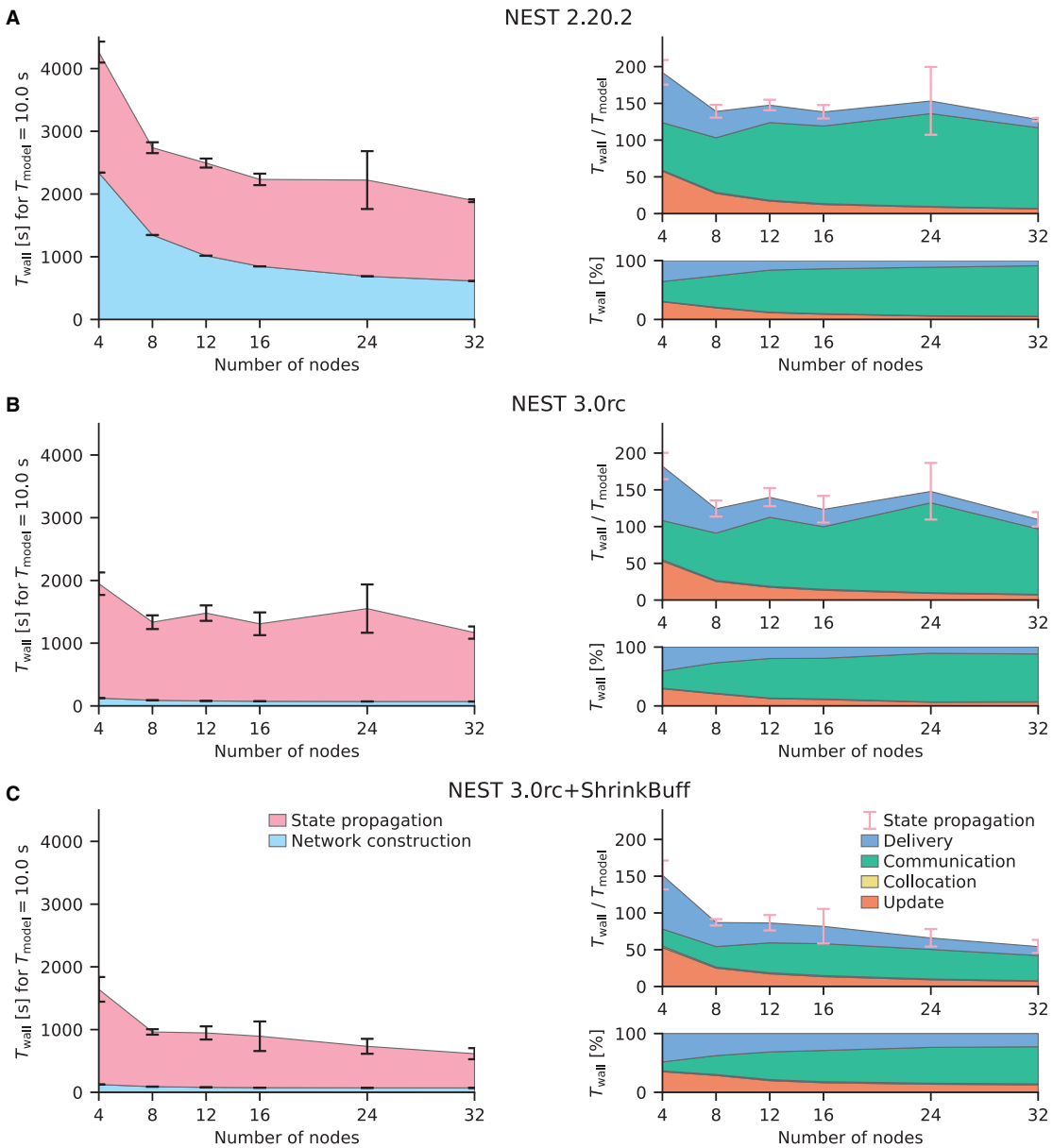

FIGURE 5 | Strong-scaling performance of the multi-area model on JURECA-DC. Same display as in **Figure 4**. The multi-area model is simulated in its meta-stable state leading to a high amount of spikes that are communicated. The model time is $T_{model} = 10$ s. Simulations are repeated for 10 different random seeds. **(A)** `NEST 2.20.2` (latest `NEST 2` release). **(B)** `NEST 3.0` release candidate. **(C)** `NEST 3.0` release candidate with shrinking MPI buffers.

The microcircuit model easily fits within the main memory of one compute node of JURECA-DC. Due to the simplicity of the employed model neurons and the absence of synaptic plasticity mechanisms, the network model causes little workload during update and delivery in a strong-scaling experiment—real-time simulation is already possible with a single compute node. Consequently, communication naturally starts to dominate the state-propagation time at a few compute nodes even with the spike-compression optimization described above. While increasing the number of compute nodes from one to two still results in a fair reduction of state-propagation time, scaling is already sublinear, and increasing the number of compute nodes further hardly results in further improvement. Therefore, simulation phases other than the so far discussed communication become important if the objective of the optimizations is, for example, achieving real-time performance with even fewer resources. In the following we highlight an algorithm adaptation that concentrates on the update phase. A redesign of the neuronal input buffers prevents neurons from retrieving the input values for different channels, for example, excitatory and

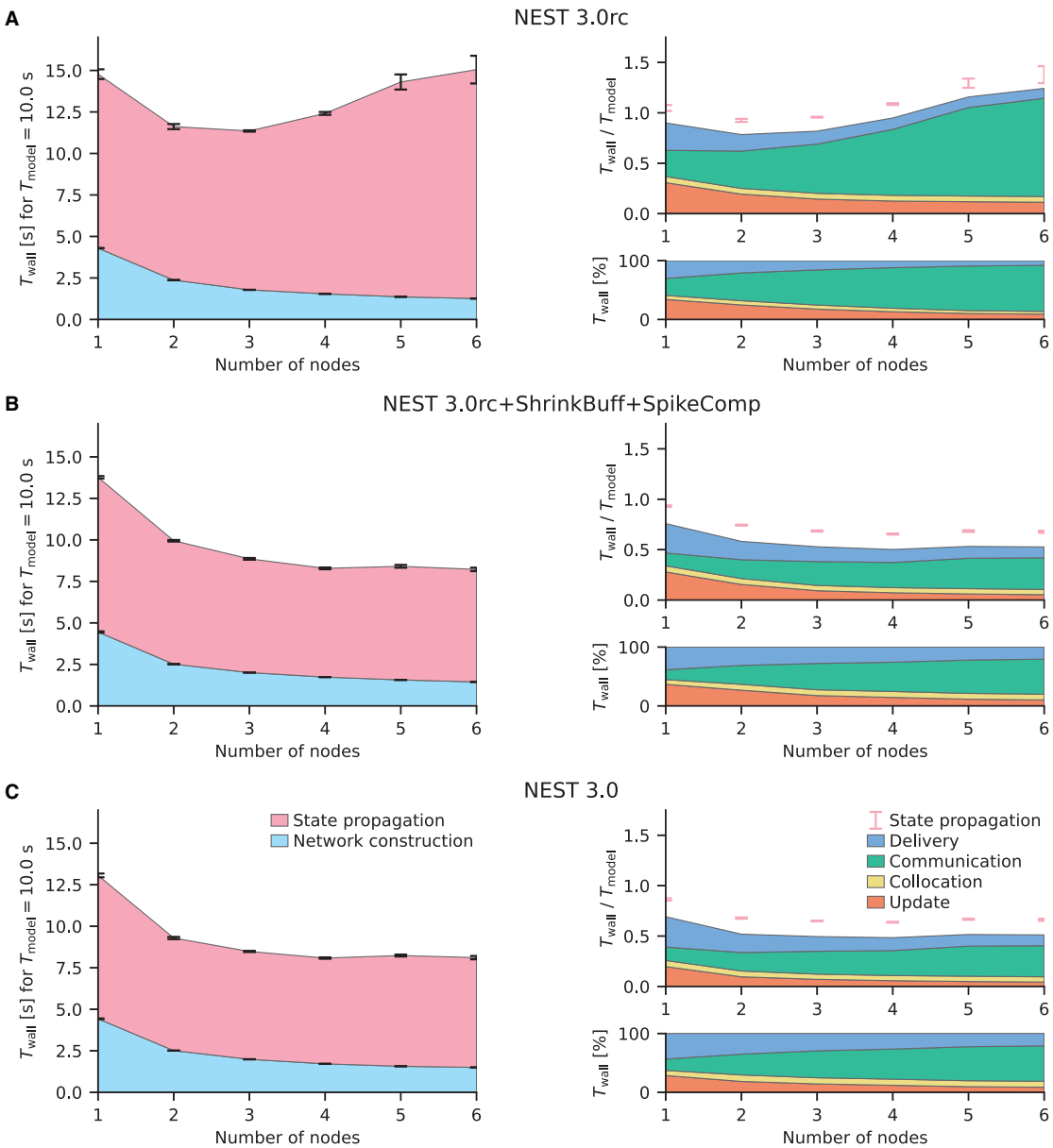

FIGURE 6 | Strong-scaling performance of the microcircuit model on JURECA-DC. Same display as in **Figure 4**. The model time is $T_{model} = 10$ s. Simulations are repeated for 10 different random seeds. **(A)** `NEST 3.0` release candidate. **(B)** `NEST 3.0` release candidate with spike compression and shrinking MPI buffers. **(C)** `NEST 3.0`.

inhibitory, from separate locations in memory. Thereby, the cache can be better utilized during neuronal updates. Instead of maintaining separate buffers for the input channels as in the original 5g kernel, neurons maintain a single buffer with all inputs for a particular simulation time step stored contiguously in memory. For details see Section 4.1.4. This adaptation is most effective for network models with short minimum synaptic delays; both the microcircuit and the multi-area model use 0.1 ms. **Figure 6C** shows the resulting decrease in update time for few compute nodes.

In summary, the analysis with beNNch exposes the communication phase as a major performance bottleneck in microcircuit and multi-area model simulations with the release candidate `NEST 3.0rc`. The underlying problem is, however, a different one for each of the two models, and they are rectified with different adaptations to the code: the shrinking MPI buffers (Section 4.1.2) improve the performance of the multi-area model while spike compression (Section 4.1.3) increases simulation speed of the microcircuit model. Notably, none of the adaptations introduce performance regressions for

the respective other model (data not shown). In addition, the update phase is improved by introducing neuronal input buffers with multiple channels (Section 4.1.4). Returning to the HPC-benchmark model, **Figure 4B** shows that the kernel adaptations are not detrimental to the originally tested model; the overall state-propagation time is preserved with the final NEST 3.0 release. However, the reduced communication and update times here come at the cost of increased delivery times due to an additional indirection introduced with spike compression. Ongoing work targets the delivery phase (Pronold et al., 2021, 2022) and gives a perspective for performance improvements in future NEST releases.

3. DISCUSSION

Benchmarking studies in the field of neuronal network simulations are often hard to reproduce and compare. To overcome this problem, we propose a unified and modular workflow for defining, running, and analyzing benchmark simulations. We identify five dimensions spanning the space of the benchmarking endeavor, and work out their specific challenges: hardware configuration, software configuration, simulators, models and parameters, and researcher communication. The benchmarking concept developed in this study encompasses all five dimensions and proposes solutions for the posed challenges in the form of self-contained and interacting modules. Each module contributes to one of the main workflow segments: configuration and preparation, actual benchmarking, data- and metadata handling, and data presentation. As a proof of concept, we provide a reference implementation of the framework (beNNch), describe the concrete underlying technologies, and apply it to a specific use case: assessing and comparing the performance of different versions of the neuronal network simulator NEST for three different network models. The reference implementation goes beyond existing benchmarking environment software such as JUBE: it adds an interface to models, installs and deploys simulation software, automates data and metadata annotation, and implements storage and presentation of results. The use case illustrates how the framework helps to focus simulator development by detecting performance bottlenecks, and demonstrates the relevance of an accessible and comprehensive benchmarking setup. The software is ready to use, not only for developers of simulation technology, but also for researchers seeking to find optimal performance configurations for their models.

The proposed workflow is generic and not restricted to benchmarking neuronal network simulations with NEST. The reference implementation, however, still faces limitations and open problems. First, it is *a priori* unclear what parameters, configurations or external influences may possibly contribute to differences in the performance of complex software systems such as simulation engines. beNNch seeks to address this problem by employing a metadata archive which—in addition to the selection of metadata directly attached to the performance results—tracks further metadata that are seemingly insignificant at the time of simulation but may become relevant in future investigations. Exhaustiveness, however, can not be claimed. For the exploration and presentation of benchmarking data, the reference implementation uses metadata to filter benchmark results and to highlight differences in a flip-book format. However, even if all relevant metadata were tracked, selecting sensible metadata keys for filtering and highlighting is a hard problem. In the current implementation, this requires knowledge about existing results and, therefore, human input. Future solutions could, for example, categorize and hierarchically structure metadata keys to facilitate and semi-automatize these steps. Second, the network model specifications, expressed in the PyNEST set of commands for the Python language, require adaptations to interface with the benchmarking framework. These include accepting parameters passed by JUBE benchmarking files, adjusting the model specification to work with different versions of the simulation engine, and storing recorded metadata and performance measures such as the duration of simulation phases. At the moment, it is a manual task to keep the benchmarking model version up to date with the original model version, which is error prone. We use rigorous version control of the code, automatic checking for errors (via exceptions), and continuous testing for correct simulation outcome to reduce the risk of errors. This strategy could be automatized further in the future by finding methods to automatically inject respective instrumentation into the executable model descriptions. To mitigate the additional overhead, we keep the necessary changes as minimal as possible, thereby lowering the entry barrier for new models. Third, the reference implementation makes concrete choices on the employed tools. Alternatives, however, may be viable. For example, the required software for the simulations is installed with Builder which can be integrated with other package management systems or replaced by a different solution. Our strategy exploits the native software environment available on a compute cluster which is typically specifically configured for the underlying hardware. An alternative is to use containerized systems such as Docker[18] or Singularity[19]. Replacing NEST by a different simulator requires adapting the model implementations. Expressing the models in the simulator-independent language PyNN (Davison et al., 2009) instead of PyNEST would avoid this. However, additional layers of complexity such as PyNN may have an impact on performance, making it more challenging to pinpoint bottlenecks in the simulator backend. JUBE as an environment to manage jobs on compute clusters could be substituted by tools such as ecFlow[20], AiiDA[21] (Huber et al., 2020), or cylc (Oliver et al., 2021). Further, one could replace git-annex with, e.g., DataLad[22] which is based on the same technology but extends its functionality and provides slightly different metadata handling. The flip-book-style presentation of results could also

[18]https://www.docker.com
[19]https://sylabs.io
[20]https://confluence.ecmwf.int/display/ECFLOW
[21]https://www.aiida.net
[22]https://www.datalad.org

be replaced or supplemented with other approaches, for example an automatically generated overview figure showing results from multiple benchmarking runs together, similar to **Figures 4**–**6** in this article. Furthermore, tools like Rust Compiler Performance Monitoring and Benchmarking[23] or Sacred[24] cover multiple aspects of the workflow and can be a source of inspiration for further development of beNNch. Fourth, beNNch presently focuses on a single performance measure: the time-to-solution. However, different performance aspects, such as energy-to-solution and memory consumption, may also be of interest. Energy-to-solution, for example, combines power consumption and time-to-solution. Monitoring both power consumption and time-to-solution enables researchers to determine an optimal number of compute nodes balancing speed and energy consumption (van Albada et al., 2018). The memory consumption of the simulation dictates, for instance, the smallest number of nodes required to simulate a network of a given size, or the largest network size possible to simulate on a given machine. Reducing memory requirements was a major driving force behind the improvements to the NEST kernel (Helias et al., 2012; Kunkel et al., 2012, 2014; Jordan et al., 2018) in the past decade. The spike compression introduced here reduces the time-to-solution (communication phase, **Figures 4**, **6**). However, this code change directly affects the memory consumption. Assuming that the number of postsynaptic targets per neuron is fixed, the memory overhead is negligible if the number of MPI processes is small. But in the limit of a large number of MPI processes, i.e., when each neuron has at most one target on each process, the effective size of each synapse is increased by 8 byte. In this limit, users thus are encouraged to actively deactivate the "spike compression" feature. This example illustrates that performance optimizations often have to find a balance between acceptable solutions for different measures. Due to its modular structure, beNNch is ready to include further performance measures.

To achieve long term sustainability, organized and openly available communication on development is essential. Adhering to this guideline, we have developed beNNch as an open source software project from the start, making use of a public issue tracker, suggestions via pull requests, public code reviews, and detailed documentation. This approach facilitates constructive communication between users and developers which enables a targeted progression of the framework by demand. While the concrete application of NEST benchmarks of neuronal network models shaped our specific implementations, the modular structure allows for adaptation to other use cases. In certain domains of software development, it is already common practice to verify each code change on the basis of syntax, results, and other unit tests. The proposed automated approach to execute performance benchmarks creates the opportunity to integrate an aspect of validation directly into the development cycle. This way, performance regressions of algorithm adaptations are immediately exposed, while positive effects can readily be demonstrated. For high-performance software, however, comprehensive checks for scaling performance are particularly costly because they require compute time on state-of-the-art clusters and supercomputers. Therefore, it is important to conduct the performance benchmarks purposefully and with care. By organizing benchmarking results and keeping track of metadata, beNNch helps to avoid redundant benchmark repeats and instead encourages a direct comparison with previous results.

It has long been recognized that software development in science underlies different constraints and needs to fulfill different requirements as compared to industry (Diesmann and Gewaltig, 2002). The software crisis in neuroscience at the beginning of the century led to the foundation of the International Neuroinformatics Coordinating Facility (INCF) in 2005. A first INCF report in 2006 addresses the software challenges of large-scale modeling in neuroscience (INCF Secretariat et al., 2018) and recommends establishing a common set of benchmark models and a corresponding framework for assessing accuracy and efficiency. Furthermore, the report advocates benchmarking neuroscientifically relevant published models rather than network models constructed specifically for the purpose of benchmarking only. In 2007, the community made a first effort in verifying simulation codes by using a number of simple network models (Brette et al., 2007). Executable model descriptions are, in part, already expressed in the simulator independent language PyNN (Davison et al., 2009), but there is no support by a common benchmarking framework, and a focus is set on correctness rather than computational efficiency. The emerging field of Research Software Engineering (RSE) is studying how, in the scientific setting, reliable and sustainable software can be developed, developers can be educated for this purpose, and science organizations and politics can be made aware of its strategic relevance (Manifesto[25] and Akhmerov et al., 2019). Obvious differences to software engineering in the industrial setting include research code being developed by scientists rather than experienced software developers, the time-constrained and thesis-bound nature of scientific projects, and the continuous integration of new contributors into the development process. Our study contributes to RSE conceptually by identifying the dimensions of benchmarking simulation technology and proposing a general workflow capable of coping with the complexity, and practically by developing a reference implementation of a benchmarking framework which can be used to test and refine the concepts. It is too early to tell quantitatively whether the benchmarking framework improves the collaboration in a joint project and the communication between researchers in the community.

The proposed framework enables benchmarking of research software to evolve from one-off tasks of individual researchers to a collaborative routine effort, thereby increasing the benchmarking capacity and reducing its susceptibility to errors. Making beNNch accessible to the community as an open-source software puts the concept to the test. We are looking forward to learn how the current implementation of the framework's components are received and adapted to other applications. Due to the conceptual foundation and modular

[23] https://github.com/rust-lang/rustc-perf
[24] https://github.com/IDSIA/sacred
[25] https://www.software.ac.uk/about/manifesto

structure, we hope that beNNch can adjust to future requirements and ultimately help increase the complexity and explanatory scope of brain models. The benchmarking concepts developed in this work are not limited to neuroscience and can be transferred to other types of simulation research.

4. MATERIALS AND METHODS

4.1. NEST Developments

4.1.1. Brief History of NEST

The series of NEST 2.X releases includes enhancements, bug fixes, and contributions to maintenance with only marginal effects on the PyNEST user interface (Eppler et al., 2009). Performance-related updates to the simulation kernel are accomplished under the hood. The 3g kernel (Helias et al., 2012; Kunkel et al., 2012) is in use from NEST 2.2.0 (van Albada et al., 2015a). NEST 2.12.0 (Kunkel et al., 2017) introduces the 4g kernel (Kunkel et al., 2014) which implements novel data structures allowing for an efficient and flexible representation of sparse network connectivity on highly distributed computing systems such as supercomputers. The 5g kernel (Jordan et al., 2018) in NEST 2.16.0 (Linssen et al., 2018) continues this direction of development toward an optimal usage of HPC systems for large-scale simulations by disentangling the memory usage per compute node from the total network size. The transition from NEST 2 to NEST 3 corresponds to a refurbishment of the simulator code which also breaks the backwards compatibility of the user interface. While improved high-level functionality and parameter handling are the primary goals of this transition, the 5g kernel is supposed to remain. In the past, performance changes due to kernel updates have been predominantly assessed using the HPC-benchmark model. The performance of the NEST 3.0 release candidate ("3.0rc"), however, is in addition evaluated with the microcircuit and multi-area model which exhibit a more complex connectivity structure and a different distribution of synaptic delays. In this way, so far undetected performance bottlenecks are discovered and subsequently resolved, leading to the official release NEST 3.0 (Hahne et al., 2021).

4.1.2. Shrinking MPI Buffers

Motivated by reducing the memory footprint of the postsynaptic infrastructure—necessary to deliver spikes to their process-local targets—the 5g kernel of NEST 3.0rc prepares a separate part of the MPI send buffer for each target process and only includes the relevant spikes. Thus, each process is responsible for sending the spikes of its neurons to all target processes for each communication time step. NEST 3.0rc implements a homogeneous buffer size across processes to avoid overhead introduced by variable buffer sizes; in the latter case, each process would need to complete two rounds of communication, one for transmitting the size, and one for the actual spiking information. Similarly, transmitting a certain amount of information via sending MPI buffers is more efficient when fewer buffers—each carrying more information—are sent. NEST 3.0rc consequently aims to reduce the number of needed MPI buffers to only 1 by dynamically increasing the global buffer size whenever a process

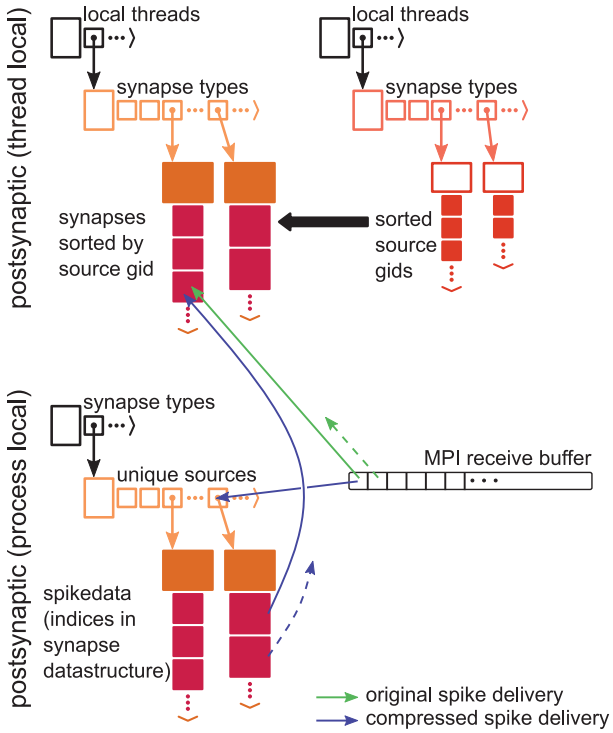

FIGURE 7 | Spike compression adds an additional indirection to post-synaptic spike routing. Green arrow denotes original spike delivery introduced with the 5g kernel (Jordan et al., 2018, same display as their **Figure 4A**). Blue arrow illustrates additional indirection with compressed spike delivery. Dashed arrows indicate spikes from the same source neuron with target on a different thread.

cannot fit all spikes into the buffer. Specifically, every time more than a single buffer needs to be sent by a process, NEST increases the buffer size of the following communication step by a factor of 1.5. In this scheme, a reduction of buffer sizes is not implemented, meaning that buffer sizes can only increase or stay constant. The kernel of NEST 3.0rc+ShrinkBuff addresses this by introducing the following algorithm for shrinking the global buffer size. In each communication round in which only a single send buffer is required, the buffer for the following round decreases by a factor of 1.1. Even though this implementation leads to an oscillation of buffer size for constant spiking activity, tests show that this simple mechanism only introduces negligible cost while being robust.

4.1.3. Spike Compression

NEST's 5g kernel (Jordan et al., 2018) introduces a two-tier connection infrastructure for routing spikes. The connection infrastructure consists of data structures on the presynaptic side (the MPI process of the sending neuron) and the postsynaptic side (the MPI process of the receiving neuron), cf. Section 2.2.3. Communication of spikes is organized as follows: when a neuron becomes active, its targets are retrieved from the local presynaptic data structure. These targets represent indices of synapses in the "thread-local" post-synaptic data structure through which spikes are routed to the target neurons. The presynaptic side then creates MPI buffers containing collections of such indices

which are subsequently communicated to the postsynaptic side via the MPI `Alltoall` function. To deliver spikes on the postsynaptic side, each thread uses the received spikes to index its local postsynaptic data structure and register a spike in the corresponding synapse (**Figure 7**, "original spike delivery"). If a presynaptic neuron has targets on multiple threads of a process, it hence has to send multiple spikes, i.e., indices in different thread-local data structures, to the target process.

Here, we adapt this infrastructure as follows. We introduce an additional data structure on the post-synaptic side which is shared across threads ("process local"). This data structure contains, arranged by source neuron, the indices of all process-local synapses. While the pre-synaptic part of communicating spikes remains essentially identical, the postsynaptic part incurs an additional indirection: each entry in the MPI receive buffer now represents an index in the new process-local postsynaptic data structure. Using this index, each thread can retrieve the indices of thread-local targets, to which it can then deliver spikes as previously (**Figure 7**, "compressed spike delivery"; note that the origin of the dashed arrow changes). In contrast to the previous implementation, each presynaptic neuron thus sends at most one spike to each process.

In `NEST 3.0`, spike compression is turned on by default, but the previous 5g behavior can be recovered by setting:

```
nest.SetKernelStatus({"use_compressed_spikes": False})
```

4.1.4. Neuronal Input Buffers With Multiple Channels

Simulation technology for spiking neuronal networks requires techniques to handle synaptic transmission delays. The reference simulation code (Section 2.2.2) follows a globally time-driven approach: spikes are constrained to a time grid and regularly exchanged between MPI processes using collective communication. The time grid defines the simulation time step for neuronal updates, whereas the minimum synaptic delay d_{min} in the network model defines the communication interval (Morrison et al., 2005a), which comprises at least one simulation time step. In the microcircuit model and the multi-area model used in this study the minimum delay is 0.1 ms (i.e., $d_{\text{min}} = 1$ simulation time step) and in the HPC-benchmark model it is 1.5 ms (i.e., $d_{\text{min}} = 15$ simulation time steps). While communication and subsequent process-local delivery of spikes define interaction points between neurons, within a communication interval each neuron independently updates its state for all time steps without interruption. Hence, a simulation cycle of neuronal update, spike-communication, and spike-delivery phase propagates the network state by one communication interval, but within each update phase neurons propagate their state in potentially shorter simulation time steps. All spikes emitted by the process-local neurons during such an update are immediately transmitted during the subsequent communication and on the receiver side delivered to their target neurons. Hence, to account for synaptic delays, neurons cannot immediately integrate the incoming spikes into their dynamics, but they need to buffer the inputs until the corresponding delays elapse. To this end, neurons maintain input buffers of $d_{\text{min}} + d_{\text{max}}$ time slots, where d_{max} denotes the maximum synaptic delay in the network (**Figure 8A**). The relative time origin S defining the time slots from which to retrieve inputs during update and the time slots for adding inputs during spike delivery advances by d_{min} time slots at the end of every simulation cycle. In this way, the time slots that were read and reset during the update of the current cycle become available for adding new inputs during the spike delivery in the next cycle. For cases where the communication interval comprises multiple simulation time steps (e.g., HPC-benchmark model), input retrieval is most costly for the first step as the corresponding buffer entry needs to be loaded into cache, but then benefits from the already cached subsequent buffer entries in the subsequent steps of the communication interval. If, however, the communication interval consists of only one simulation step due to a very short minimal synaptic delay (e.g., microcircuit and multi-area model), input retrieval is costly for every simulation step as each step is handled in a separate simulation cycle, and hence caching of relevant input buffer entries is rendered ineffective during the spike communication and delivery that follows each neuronal update phase.

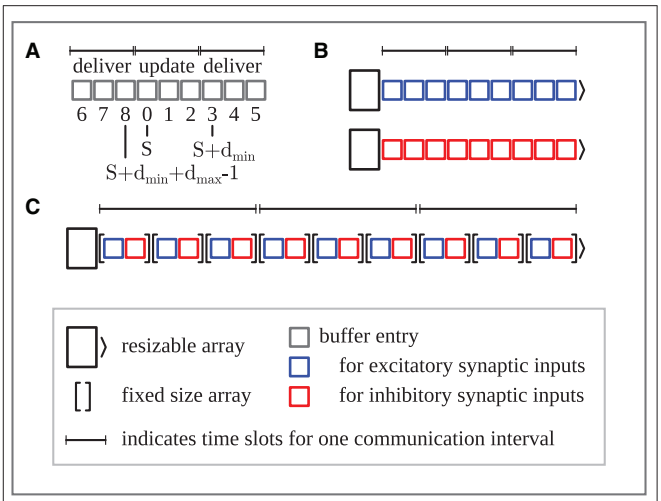

FIGURE 8 | Neuronal input buffers accounting for synaptic delays in simulations of spiking neuronal networks. **(A)** Structure of neuronal input buffers assuming a minimum synaptic delay d_{min} of three simulation time steps and a maximum delay $d_{\text{max}} = 2d_{\text{min}}$. To buffer upcoming inputs during simulation a total buffer size of $d_{\text{min}} + d_{\text{max}}$ time slots is required, which corresponds to three communication intervals of three simulation time steps each. After every spike communication and subsequent spike delivery to local targets, simulation time is advanced, meaning that the relative time origin S of the neuronal input buffers advances by d_{min} time slots with a wrap-around at the buffer end. A pre-calculated and continuously updated look-up table maps the index relative to S to the actual buffer index. Example: The relative time origin S is located at the fourth time slot. Synaptic delays of the inputs of the middle buffer segment elapse with the upcoming three simulation time steps; the neuron integrates these inputs updating its state. Spikes are then communicated and new inputs delivered to the neuron are added to the time slots in the last or first buffer segment depending on the delay, which is at least d_{min} and at most d_{max}. Relative time origin S then advances to the seventh buffer slot (not shown). **(B)** Original neuronal spike buffers for two input channels (e.g., excitatory and inhibitory synaptic inputs). For each channel a separate resizable array buffers the inputs for the upcoming time slots. **(C)** Multi-channel input buffer for two input channels. A single resizable array stores the inputs for the upcoming time slots, where for each time slot a fixed size array holds the inputs sorted by channel.

Most neuron models need to distinguish between input channels to treat the corresponding inputs dynamically differently, as for example, excitatory and inhibitory synaptic inputs causing different post-synaptic responses. The original input-buffer design required a separate resizable array per channel storing the channel's input values per time slot (**Figure 8B**). This entailed retrieval of the input values for a particular time step from separate locations in memory, which amplifies the cache inefficiency during update for network models with short minimum delays described above. To alleviate this issue, the newly introduced input buffer allows storing the input values for multiple channels per time slot contiguously in fixed size arrays in a single resizable array (**Figure 8C**). Thus, neurons now retrieve all input values for a particular time step by accessing subsequent locations in memory in one pass.

DATA AVAILABILITY STATEMENT

The benchmarking framework is publicly available under https://github.com/INM-6/beNNch. Benchmarks for this study were performed with version `1.0` of `beNNch` which is available as a release on GitHub and on Zenodo (https://doi.org/10.5281/zenodo.6092768, Albers et al., 2022). The data sets generated and analyzed for this study as well as the code to reproduce all figures of this paper are available on Zenodo (https://doi.org/10.5281/zenodo.5784633). An exemplary flip-book containing the results shown in this work can be accessed under https://inm-6.github.io/beNNch.

AUTHOR CONTRIBUTIONS

JA, JP, ACK, SBV, KHM, AP, DT, TT, MD, and JS: study design. JA, JP, ACK, SBV, KHM, DT, and JS: implementation of `beNNch`. JA: execution and analysis of benchmarks. JA, JP, SK, and JJ: figures. AP and JA: implementation of shrinking MPI buffers. JJ and JS: implementation of spike compression. SK: implementation of neuronal input buffers with multiple channels. All authors contributed to the writing of the manuscript and approved it for publication.

FUNDING

This project has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreement No. 945539 (Human Brain Project SGA3) and No. 754304 (DEEP-EST); the Helmholtz Association Initiative and Networking Fund under project number SO-092 (Advanced Computing Architectures, ACA); the Joint Lab Supercomputing and Modeling for the Human Brain; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—368482240/GRK2416 and 491111487; the Helmholtz Metadata Collaboration (HMC), an incubator-platform of the Helmholtz Association within the framework of the Information and Data Science strategic initiative, under the funding ZT-I-PF-3-026.

ACKNOWLEDGMENTS

We thank the members of the NEST development community for their contributions to the concepts and implementation of the NEST simulator, and our colleagues in the Simulation and Data Laboratory Neuroscience of the Jülich Supercomputing Centre for continuous collaboration. We gratefully acknowledge the computing time granted by the JARA Vergabegremium and provided on the JARA Partition part of the supercomputer JURECA at Forschungszentrum Jülich (computation grant JINB33). We acknowledge the use of Fenix Infrastructure resources, which are partially funded from the European Union's Horizon 2020 research and innovation programme through the ICEI project under the grant agreement No. 800858.

REFERENCES

Akar, N. A., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). "Arbor — a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia: IEEE), 274–282.

Akhmerov, A., Cruz, M., Drost, N., Hof, C., Knapen, T., Kuzak, M., et al. (2019). Raising the profile of research software. doi: 10.5281/zenodo.3378572

Albers, J., Pronold, J., Kurth, A. C., Vennemo, S. B., Haghighi, M. K., Patronis, A., et al. (2022). beNNch. Version 1.0. *Zenedo*. doi: 10.5281/zenodo.6092768

Beyeler, M., Carlson, K. D., Chou, T.-S., Dutt, N., and Krichmar, J. L. (2015). "CARLsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks," in *2015 International Joint Conference on Neural Networks (IJCNN)* (Killarney: IEEE).

Bhalla, U., Bilitch, D., and Bower, J. M. (1992). Rallpacks: A set of benchmarks for neuronal simulators. *Trends Neurosci.* 15, 453–458. doi: 10.1016/0166-2236(92)90009-w

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398. doi: 10.1007/s10827-007-0038-6

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208. doi: 10.1023/a:1008925309027

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

Chou, T.-S., Kashyap, H. J., Xing, J., Listopad, S., Rounds, E. L., Beyeler, M., et al. (2018). "CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in *2018 International Joint Conference on Neural Networks (IJCNN)*, Rio de Janeiro.

Crook, S. M., Davison, A. P., and Plesser, H. E. (2013). "Learning from the past: approaches for reproducibility in computational neuroscience," in *20 Years of Computational Neuroscience*, Springer Series in Computational Neuroscience, ed J. Bower (New York, NY: Springer), 73–102.

Dai, W., and Berleant, D. (2019). "Benchmarking contemporary deep learning hardware and frameworks: a survey of qualitative metrics," in *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)* (Los Angeles, CA).

Davison, A., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2:10. doi: 10.3389/neuro.11.011.2008

Diesmann, M., and Gewaltig, M.-O. (2002). "NEST: an environment for neural systems simulations," in *Forschung und Wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, eds T. Plesser and V. Macho (Göttingen: Ges. für Wiss. Datenverarbeitung), 43–70.

Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK benchmark: past, present and future. *Concurr. Comput.* 15, 803–820. doi: 10.1002/cpe.728

Einevoll, G. T., Destexhe, A., Diesmann, M., Grün, S., Jirsa, V., de Kamps, M., et al. (2019). The scientific case for brain simulations. *Neuron* 102, 735–744. doi: 10.1016/j.neuron.2019.03.027

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M. (2009). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2:12. doi: 10.3389/neuro.11.012.2008

Fardet, T., Vennemo, S. B., Mitchell, J., Mork, H., Graber, S., Hahne, J., et al. (2021). NEST 2.20.2, Version 2.20.2. *Zenedo*. doi: 10.5281/zenodo.5242954

Furber, S., Galluppi, F., Temple, S., and Plana, L. (2014). The SpiNNaker Project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., et al. (2015). "The spack package manager," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (ACM)*. doi: 10.1145/2807591.2807623

Geimer, M., Hoste, K., and McLay, R. (2014). "Modern scientific software management using EasyBuild and lmod," in *2014 First International Workshop on HPC User Support Tools* (New Orleans, LA,).

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural simulation tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Gleeson, P., Cantarelli, M., Marin, B., Quintana, A., Earnshaw, M., Sadeh, S., et al. (2019). Open source brain: a collaborative resource for visualizing, analyzing, simulating, and developing standardized models of neurons and circuits. *Neuron* 103, 395–411.e5. doi: 10.1016/j.neuron.2019.05.019

Golosio, B., Tiddia, G., Luca, C. D., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021). Fast simulations of highly-connected spiking cortical models using GPUs. *Front. Comput. Neurosci.* 15:627620. doi: 10.3389/fncom.2021.627620

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Front. Neuroinform.* 2:8. doi: 10.3389/neuro.11.005.2008

Gutzen, R., von Papen, M., Trensch, G., Quaglio, P., Grün, S., and Denker, M. (2018). Reproducible neural network simulations: statistical methods for model validation on the level of network activity data. *Front. Neuroinform.* 12:90. doi: 10.3389/fninf.2018.00090

Hager, G., and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers, 1st Edn.* New York, NY: CRC Press.

Hahne, J., Diaz, S., Patronis, A., Schenck, W., Peyser, A., Graber, S., et al. (2021). NEST 3.0. *Zenedo*. doi: 10.5281/zenodo.4739103

Helias, M., Kunkel, S., Masumoto, G., Igarashi, J., Eppler, J. M., Ishii, S., et al. (2012). Supercomputers ready for use as discovery machines for neuroscience. *Front. Neuroinform.* 6:26. doi: 10.3389/fninf.2012.00026

Huber, S. P., Zoupanos, S., Uhrin, M., Talirz, L., Kahle, L., Häuselmann, R., et al. (2020). Aiida 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Sci. Data* 7, 1–18. doi: 10.1038/s41597-020-00638-4

Hunter, J. D. (2007). Matplotlib: a 2D graphics environment. *Comput. Sci. Eng.* 9, 90–95. doi: 10.1109/MCSE.2007.55

INCF Secretariat, Djurfeldt, M., and Lansner, A. (2018). "*1st INCF Workshop on Large-Scale Modeling of the Nervous System*." Stockholm: F1000 Research Limited. doi: 10.7490/F1000RESEARCH.1116028.1

Ippen, T., Eppler, J. M., Plesser, H. E., and Diesmann, M. (2017). Constructing neuronal network models in massively parallel environments. *Front. Neuroinform.* 11:30. doi: 10.3389/fninf.2017.00030

Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Trans. Neural Netw.* 14, 1569–1572. doi: 10.1109/TNN.2003.820440

Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., et al. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Front. Neuroinform.* 12:2. doi: 10.3389/fninf.2018.00002

Jülich Supercomputing Centre (2015). *JUQUEEN: IBM Blue Gene/Q® Supercomputer System.* Jülich Supercomputing Centre.

Knight, J. C., Komissarov, A., and Nowotny, T. (2021). PyGeNN: A python library for GPU-enhanced neural networks. *Front. Neuroinform.* 15:5. doi: 10.3389/fninf.2021.659005

Knight, J. C., and Nowotny, T. (2018). GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Front. Neurosci.* 12:941. doi: 10.3389/fnins.2018.00941

Knight, J. C., and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nat. Comput. Sci.* 1, 136–142. doi: 10.1038/s43588-020-00022-7

Kunkel, S., Morrison, A., Weidel, P., Eppler, J. M., Sinha, A., Schenck, W., et al. (2017). NEST 2.12.0. *Zenedo*. doi: 10.5281/zenodo.259534

Kunkel, S., Potjans, T. C., Eppler, J. M., Plesser, H. E., Morrison, A., and Diesmann, M. (2012). Meeting the memory challenges of brain-scale simulation. *Front. Neuroinform.* 5:35. doi: 10.3389/fninf.2011.00035

Kunkel, S., and Schenck, W. (2017). The NEST dry-run mode: efficient dynamic analysis of neuronal network simulation code. *Front. Neuroinform.* 11:40. doi: 10.3389/fninf.2017.00040

Kunkel, S., Schmidt, M., Eppler, J. M., Masumoto, G., Igarashi, J., Ishii, S., et al. (2014). Spiking network simulation code for petascale computers. *Front. Neuroinform.* 8:78. doi: 10.3389/fninf.2014.00078

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2021). Sub-realtime simulation of a neuronal network of natural density. *Neural. Comput. Eng.* doi: 10.1088/2634-4386/ac55fc

Linssen, C., Lepperod, M. E., Mitchell, J., Pronold, J., Eppler, J. M., Keup, C., et al. (2018). NEST 2.16.0. *Zenedo*. doi: 10.5281/zenodo.1400175

Lytton, W. W., Seidenstein, A. H., Dura-Bernal, S., McDougal, R. A., Schürmann, F., and Hines, M. L. (2016). Simulation neurotechnologies for advancing brain research: parallelizing large networks in NEURON. *Neural Comput.* 28, 2063–2090. doi: 10.1162/neco_a_00876

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., et al. (2020). "MLPerf training benchmark," in *Proceedings of Machine Learning and Systems,* Vol. 2, eds I. Dhillon, D. Papailiopoulos, and V. Sze, 336–349. Available online at: https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf

McDougal, R. A., Bulanova, A. S., and Lytton, W. W. (2016). Reproducibility in computational neuroscience models and simulations. *IEEE Trans. Biomed. Eng.* 63, 2021–2035. doi: 10.1109/TBME.2016.2539602

Message Passing Interface Forum (2009). *MPI: A Message-Passing Interface Standard, Version 2.2*. Technical Report, Knoxville, TN, United States.

Migliore, M., Cannia, C., Lytton, W. W., Markram, H., and Hines, M. (2006). Parallel network simulations with NEURON. *J. Comput. Neurosci.* 21, 119–223. doi: 10.1007/s10827-006-7949-5

Miyazaki, H., Kusano, Y., Shinjou, N., Fumiyoshi, S., Yokokawa, M., and Watanabe, T. (2012). Overview of the K computer System. *Fujitsu Sci. Techn. J.* 48, 255–265.

Monteforte, M., and Wolf, F. (2010). Dynamical entropy production in spiking neuron networks in the balanced state. *Phys. Rev. Lett.* 105:268104. doi: 10.1103/PhysRevLett.105.268104

Morrison, A., Hake, J., Straube, S., Plesser, H. E., and Diesmann, M. (2005a). "Precise spike timing with exact subthreshold integration in discrete time network simulations," in *Proceedings of the 30th Göttingen Neurobiology Conference*, 205B, Göttingen.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005b). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801. doi: 10.1162/0899766054026648

Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., and Veidenbaum, A. V. (2009). A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Netw.* 22, 791–800. doi: 10.1016/j.neunet.2009.06.028

Oliver, H. J., Shin, M., Sanders, O., Fitzpatrick, B., Clark, A., Dutta, R., et al. (2021). cylc/cylc-flow: cylc-flow-8.0b3. *Zenedo*. doi: 10.5281/zenodo.5668823

OpenMP Architecture Review Board (2008). *OpenMP Application Program Interface.* Available online at: http://www.openmp.org/mp-documents/spec30.pdf (accessed September 27, 2016).

Ostrau, C., Klarhorst, C., Thies, M., and Rückert, U. (2020). "Benchmarking of neuromorphic hardware systems," in *NICE '20: Proceedings of the Neuro-inspired Computational Elements Workshop*, Heidelberg. doi: 10.1145/3381755.3381772

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing polychronization: a guide to maximizing the reproducibility of spiking network models. *Front. Neuroinform.* 12:46. doi: 10.3389/fninf.2018.00046

Pfeil, T., Grübl, A., Jeltsch, S., Müller, E., Müller, P., Petrovici, M. A., et al. (2013). Six networks on a universal neuromorphic computing substrate. *Front. Neurosci.* 7:11. doi: 10.3389/fnins.2013.00011

Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., and Gewaltig, M.-O. (2007). "Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers," in *Euro-Par 2007: Parallel Processing, Vol. 4641 of Lecture Notes in Computer Science*, eds A.-M. Kermarrec, L. Bougé, and T. Priol (Berlin: Springer-Verlag), 672–681.

Potjans, T. C., and Diesmann, M. (2014). The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cereb. Cortex* 24, 785–806. doi: 10.1093/cercor/bhs358

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2021). Routing brain traffic through the von Neumann bottleneck: Efficient cache usage in spiking neural network simulation code on general purpose computers. *arXiv [Preprint]*. arXiv: 2109.12855. Available online at: https://arxiv.org/pdf/2109.12855.pdf (accessed March 11, 2022).

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2022). Routing brain traffic through the Von Neumann bottleneck: Parallel sorting and refactoring. *Front. Neuroinform.* 15:785068. doi: 10.3389/fninf.2021.785068

Rhodes, O., Peres, L., Rowley, A. G. D., Gait, A., Plana, L. A., Brenninkmeijer, C., et al. (2019). Real-time cortical simulation on neuromorphic hardware. *Philos. Trans. R. Soc. A* 378:20190160. doi: 10.1098/rsta.2019.0160

Richert, M., Nageswaran, J. M., Dutt, N., and Krichmar, J. L. (2011). An efficient simulation environment for modeling large-scale cortical processing. *Front. Neuroinform.* 5:19. doi: 10.3389/fninf.2011.00019

Rougier, N. P., Hinsen, K., Alexandre, F., Arildsen, T., Barba, L. A., Benureau, F. C., et al. (2017). Sustainable computational science: the ReScience initiative. *PeerJ Comput. Sci.* 3:e142. doi: 10.7717/peerj-cs.142

Schemmel, J., Brüderle, D., Grübl, A., Hock, M., Meier, K., and Millner, S. (2010). "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proceedings of the 2010 International Symposium on Circuits and Systems (ISCAS)* (Paris: IEEE Press), 1947–1950.

Schmidt, M., Bakker, R., Hilgetag, C. C., Diesmann, M., and van Albada, S. J. (2018a). Multi-scale account of the network structure of macaque visual cortex. *Brain Struct Funct.* 223, 1409–1435. doi: 10.1007/s00429-017-1554-4

Schmidt, M., Bakker, R., Shen, K., Bezgin, G., Diesmann, M., and van Albada, S. J. (2018b). A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLOS Comput. Biol.* 14:e1006359. doi: 10.1371/journal.pcbi.1006359

Senk, J., Yegenoglu, A., Amblet, O., Brukau, Y., Davison, A., Lester, D. R., et al. (2017). "A collaborative simulation-analysis workflow for computational neuroscience using HPC," in *High-Performance Scientific Computing, JHPCS 2016, Vol. 10164 of Lecture Notes in Computer Science*, eds E. Di Napoli, M.-A. Hermanns, H. Iliev, A. Lintermann, and A. Peyser (Cham: Springer), 243–256. doi: 10.1007/978-3-319-53862-4_21

Sompolinsky, H., Crisanti, A., and Sommers, H. J. (1988). Chaos in random neural networks. *Phys. Rev. Lett.* 61, 259–262. doi: 10.1103/PhysRevLett.61.259

Stimberg, M., Brette, R., and Goodman, D. F. (2019). Brian 2, an intuitive and efficient neural simulator. *eLife* 8:47314. doi: 10.7554/elife.47314

Stimberg, M., Goodman, D. F. M., and Nowotny, T. (2020). Brian2GeNN: accelerating spiking neural network simulations with graphics hardware. *Sci. Rep.* 10:410. doi: 10.1038/s41598-019-54957-7

Thörnig, P., and von St. Vieth, B. (2021). JURECA: data centric and booster modules implementing the modular supercomputing architecture at Jülich supercomputing centre. *J. Large Scale Res. Facil.* 7:A182. doi: 10.17815/jlsrf-7-182

van Albada, S., Chindemi, G., Deger, M., Diesmann, M., Djurfeldt, M., Enger, H., et al. (2015a). NEST 2.2.0. *Zenedo.* doi: 10.5281/zenodo.5772624

van Albada, S. J., Helias, M., and Diesmann, M. (2015b). Scalability of asynchronous networks is limited by one-to-one mapping between effective connectivity and correlations. *PLOS Comput. Biol.* 11:e1004490. doi: 10.1371/journal.pcbi.1004490

van Albada, S. J., Kunkel, S., Morrison, A., and Diesmann, M. (2014). "Integrating brain structure and dynamics on supercomputers," in *Brain-Inspired Computing*, eds L. Grandinetti, T. Lippert, and N. Petkov (Cham: Springer), 22–32.

van Albada, S. J., Pronold, J., van Meegen, A., and Diesmann, M. (2021). "Usage and scaling of an open-source spiking multi-area model of monkey cortex," in *Brain-Inspired Computing*. Lecture Notes in Computer Science (Cetraro: Springer International Publishing), 47–59.

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model. *Front. Neurosci.* 12:291. doi: 10.3389/fnins.2018.00291

van Vreeswijk, C., and Sompolinsky, H. (1998). Chaotic balanced state in a model of cortical circuits. *Neural Comput.* 10, 1321–1371. doi: 10.1162/089976698300017214

Yavuz, E., Turner, J., and Nowotny, T. (2016). GeNN: a code generation framework for accelerated brain simulations. *Sci. Rep.* 6:18854. doi: 10.1038/srep18854

Yoo, A. B., Jette, M. A., and Grondona, M. (2003). "Slurm: simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, eds D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Berlin; Heidelberg: Springer Berlin Heidelberg), 44–60.

Zaytsev, Y. V., and Morrison, A. (2014). CyNEST: a maintainable Cython-based interface for the NEST simulator. *Front. Neuroinform.* 8:23. doi: 10.3389/fninf.2014.00023

Conflict of Interest: 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.