Parallel Computing 113 (2022) 102952

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Routing brain traffic through the von Neumann bottleneck: Efficient cache usage in spiking neural network simulation code on general purpose computers

J. Pronold [a,b,*], J. Jordan [c], B.J.N. Wylie [d], I. Kitayama [e], M. Diesmann [a,f,g], S. Kunkel [h,**]

[a] 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
[b] RWTH Aachen University, Aachen, Germany
[c] Department of Physiology, University of Bern, Bern, Switzerland
[d] Jülich Supercomputing Centre, Jülich Research Centre, Jülich, Germany
[e] RIKEN Center for Computational Science, Kobe, Japan
[f] Department of Physics, Faculty 1, RWTH Aachen University, Aachen, Germany
[g] Department of Psychiatry, Psychotherapy and Psychosomatics, Medical Faculty, RWTH Aachen University, Aachen, Germany
[h] Faculty of Science and Technology, Norwegian University of Life Sciences, Ås, Norway

ARTICLE INFO

Keywords:
Spiking neural networks
Large-scale simulation
Cache performance
Distributed computing
Parallel computing
Memory access bottleneck

ABSTRACT

Simulation is a third pillar next to experiment and theory in the study of complex dynamic systems such as biological neural networks. Contemporary brain-scale networks correspond to directed random graphs of a few million nodes, each with an in-degree and out-degree of several thousands of edges, where nodes and edges correspond to the fundamental biological units, neurons and synapses, respectively. The activity in neuronal networks is also sparse. Each neuron occasionally transmits a brief signal, called spike, via its outgoing synapses to the corresponding target neurons. In distributed computing these targets are scattered across thousands of parallel processes. The spatial and temporal sparsity represents an inherent bottleneck for simulations on conventional computers: irregular memory-access patterns cause poor cache utilization. Using an established neuronal network simulation code as a reference implementation, we investigate how common techniques to recover cache performance such as software-induced prefetching and software pipelining can benefit a real-world application. The algorithmic changes reduce simulation time by up to 50%. The study exemplifies that many-core systems assigned with an intrinsically parallel computational problem can alleviate the von Neumann bottleneck of conventional computer architectures.

1. Introduction

Irregular access to large amounts of memory challenges the von Neumann computer architecture. Distributed applications typically make use of systems with hybrid parallelization, using message passing libraries for the communication between compute nodes and multi-threading to employ the computational cores in each node. In this contribution, we investigate as an extreme real-world example application simulation code for biological neural networks.

Such networks correspond to graphs. The graph representing the neurons as nodes and their contact points, called synapses, as directed edges is sparse and complex. In the mammalian brain a neuron establishes several thousands or even more than ten-thousand of incoming synapses and the number of outgoing synapses is of the same order. This corresponds to a graph, where both in-degree and out-degree of each node are of the order of 1000 or even 10,000.

The brain mantle, called cerebral cortex, contains the neuronal cell bodies. Tangential to the cortical surface, the probability of two cortical neurons to establish a contact is approximately 0.1 within a distance of one millimeter, but it declines rapidly for longer distances. Half of a neuron's outgoing connections are not local but connect to neurons at distant locations forming a hierarchically organized architecture [for an example see 1]. Due to the sheer number of neurons in the brain, the probability of any pair sharing an edge is vanishingly small.

* Corresponding author at: 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.
** Corresponding author.
E-mail addresses: j.pronold@fz-juelich.de (J. Pronold), susanne.kunkel@nmbu.no (S. Kunkel).

https://doi.org/10.1016/j.parco.2022.102952
Received 28 September 2021; Received in revised form 5 May 2022; Accepted 7 July 2022
Available online 13 July 2022

The interaction between neurons is mediated through synapses by point-like events, called spikes. Spike events are infrequent; neurons emit a single or few spikes per second while the time constants of single-neuron dynamics are in the range of milliseconds. In addition, the behavior is organized by the brain on a sub-second time scale.

Synapses outnumber the neurons by three to four orders of magnitude and thereby consume a significant part of computer memory in simulations of spiking neural networks. The strength of the interaction mediated by a synapse can change over time depending on the activities of the presynaptic and the postsynaptic neurons and third factors like neuromodulators and the membrane potential of the postsynaptic neuron. These dynamics, called synaptic plasticity, are a key mechanism of system-level learning. For the purpose of this investigation, we assume that each synapse maintains a state variable representing the coupling strength. We refer to this as synaptic weight, but plastic processes are not considered. Moreover, synaptic transmission of spikes involves a delay, which is the time interval between the presynaptic neuron emitting a spike and the spike affecting the postsynaptic neuron. Depending on the spatial distance between presynaptic and postsynaptic neuron, the delay can be shorter than 0.1 ms or longer than 10.0 ms. This study considers homogeneous delays of 1.5 ms.

The spikes emitted by model neurons represent the sharp voltage transients of biological neurons, called action potentials. Models describing neuronal networks at the level of resolution of neurons and synapses represent individual neurons by a small system of differential equations. Often the system is linear and all non-linearity is condensed in a threshold operation on the state vector generating the point-like event. In our case the subthreshold dynamics can be integrated exactly, limiting the workload in terms of floating point operations.

Over the past two decades simulation tools in computational neuroscience have increasingly embraced a conceptual separation of generic simulation engines and specific models of neuronal networks [2]. Many different models can thus be simulated with the same simulation engine. This enables the community to separate the life cycle of a simulation engine from those of specific models and to maintain and further develop simulation engines as an infrastructure. Furthermore, the separation facilitates the cross-validation of simulation engines.

Before the dynamical state of a model of a neuronal network can be propagated, an instance of the model needs to be created in computer memory. Often network models are concisely defined by probabilistic construction rules rather than explicit adjacency lists. Therefore, in simulations of neuronal networks we distinguish between the phase of network construction and the actual simulation phase, where state propagation takes place. The former is a research topic on its own [3]. The present work concentrates on the simulation phase. Network construction may take relevant amounts of wall-clock time, but unlike simulation time, it does not scale with the biological time span covered by the model. Propagating the dynamical state of the network in time involves three repeating phases [4,5]. The first, termed update, advances the state of the neurons by a time interval corresponding to the minimal synaptic delay in the system, where even smaller update steps at the level of individual neurons are possible. The second, communication, is concerned with distributing the spikes that have occurred in this time interval to the compute nodes and threads hosting the respective target neurons. The third, spike-delivery, routes the spikes arriving at a compute node via the representations of the corresponding synapses to their target neurons. Our investigation concentrates on this final phase of the cycle.

The combination of irregular spiking activity and sparse connectivity, leads to a practically random memory-access pattern during spike delivery. Seemingly this is a worst case scenario for the von Neumann architecture where for any computation the content of a respective memory unit has to be transported to the central processing unit and the result needs to be transported back. Other disciplines, such as graph processing [6] and main memory database systems [7,8] suffer from frequent and unpredictable main memory access as well.

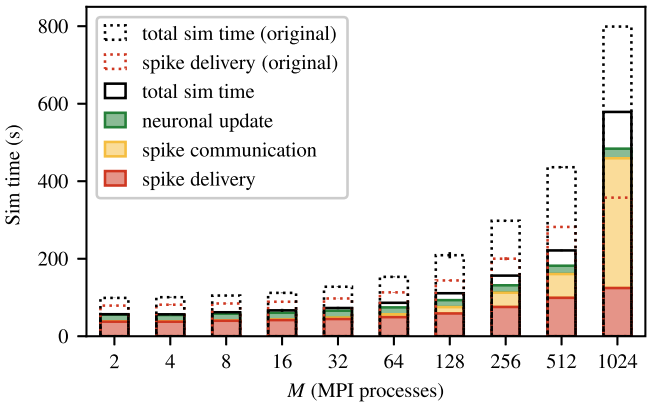

Fig. 1. Main achievement of this study: reduced simulation time (sim time) for spiking neural network simulations in comparison to the situation prior to optimization. Weak-scaling experiment running 2 MPI processes per compute node and 12 threads per MPI process, with a workload of 125,000 neurons per MPI process (network model see Section 2.2). The network dynamics is simulated for 1 s of biological time; spikes are communicated across MPI processes every 1.5 ms. Solid bars show total sim time (black outline), time spent on spike delivery (red), communication of spike data (yellow), and neuronal update (green) after improving parallelization of spike delivery [9] and using a combination of the cache optimizations discussed in this study (bwRB*, bwTS). Dashed bars show total sim time (black dashed) and time spent on spike delivery (red dashed) of the original code (**Algorithm ORI** in [9]); original communication and update times (Figure 3 of [9]) omitted for clarity. Error bars (visible only at 128 MPI processes for the original sim time) indicate standard deviation over three repetitions. Timings obtained via manual instrumentation of the source code; measured on JURECA CM (Section 2.3).

Let us consider a concrete example to summarize the main results of the present study. In weak scaling of simulations with the same number of neurons per MPI process, spike delivery dominates simulation time independent of the number of MPI processes employed (Fig. 1). In the regime from 2 to 512 MPI processes, the time required for spike delivery almost quadruples (factor of 3.9). The improved parallelization described in a technical companion paper [9] and the cache optimizations discussed in this article reduce the dependence of spike delivery on the number of MPI processes. Simulation time is reduced by 43% for low numbers of MPI processes, by 49% for 512 MPI processes, and by 28% for 1024 MPI processes. Even below 1024 MPI processes, the major cause for the loss in performance now is the increase in communication time. A second qualitative change sets in at 1024 MPI processes, where the absolute simulation time is no longer dominated by spike delivery but by communication time. Hence, the new algorithm overcomes the previously reported barrier of spike delivery limiting the performance on supercomputers (Figure 12 in [10]). The neuronal update and communication phase are not affected by these changes to the original spike-delivery algorithm (cf. Figure 3 of [9]). Additionally, the absolute time for neuronal update remains unchanged throughout as the number of neurons per MPI process is fixed. The absolute simulation times of Fig. 1 also illustrate the practical relevance of these improvements. The simulations are slower than real time by at least a factor of one hundred, but in neuroscience, observation times of 100 s of biological time are commonly required to gather statistics. Even minutes and hours can be needed to investigate plastic processes.

All simulation code analyzed in this study already provides optimizations reducing the number of spikes that MPI processes need to exchange in small to medium scale simulations (Section 3.2; see Section 3.3 in 10). The code exploits the degree of distribution across processes in this regime where the average number of outgoing synapses per neuron outnumbers the total number of MPI processes. This reduces both communication times and spike-delivery times, which in turn results in shorter overall simulation times (Fig. 1). The present work addresses this intermediate regime, where the effect of the optimizations

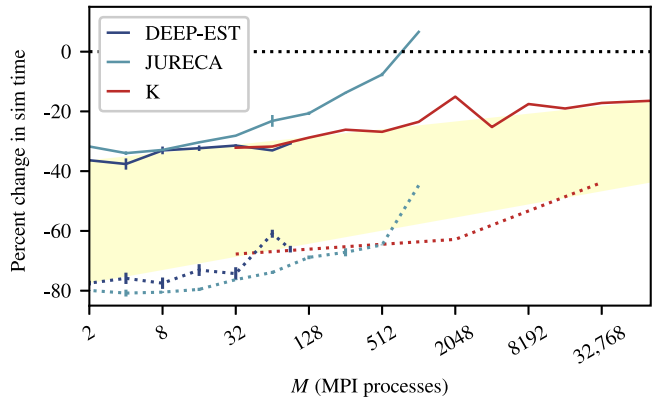

Fig. 2. Potential reduction of simulation time by cache optimization of spike delivery after preparatory improvement of parallelization. Black dotted line at zero indicates performance of original code (**ORI** in [9]). Solid curves show relative change in simulation time due to improved parallelization by prior work (REF) as a function of the number of MPI processes M. Weak scaling of benchmark network model (Section 2.2) in linear-log representation. Same configuration as in Fig. 1 for DEEP-EST CM (blue) and JURECA CM (turquoise) systems; error bars show standard deviation based on 3 repetitions. Data of K computer (red) for 1 MPI process per compute node, 8 threads per MPI process, and 18,000 neurons per MPI process (systems identified in Section 2.3). Dotted curves with corresponding colors indicate hypothetical limit to the decrease in sim time defined by the contribution of spike-delivery time. Yellow band in background indicates maximal regime accessible to cache optimization.

gradually diminishes with the degree of distribution across processes, which impacts scalability. The technical companion paper [9] derives an analytical expression for the transition to a fully distributed network under the constraints of weak scaling.

The success of an optimization needs to be evaluated in the light of potential gain. The maximal gain by an optimization to a specific part of code is limited by the contribution of this part of code to the overall runtime. Conversely, if an unrelated component of the code section dominates the fraction of runtime, the optimization may go unnoticed. Therefore, in a preparatory study, we reviewed the existing spike-delivery code [10] for any algorithmic improvements not related to explicit cache optimization techniques. The resulting spike-delivery algorithm that serves as a reference for the present (Section 3.2) is documented in the technical companion paper [9]. While in the original algorithm all threads examine all the spikes arriving at a compute node to find those for which they harbor target neurons, the revised algorithm first divides the spikes equally among the threads and performs a parallel sorting according to the thread responsible for the target neuron. In addition, the revision removes indirections such that a spike has access to the parameters of a synapse and the relevant target data structure on the same level of the call stack. The changes significantly reduce simulation time (Fig. 2) showing different effectiveness on the tested systems (Section 2.3). However, over the full range there is room for further improvement to spike delivery (highlighted in figure) by at least the same amount.

Recently, Cremonesi et al. made major progress by deriving a detailed analytic performance model [11,12]. The model reveals specific algorithmic and hardware bottlenecks for three classes of neuronal network models within the field of computational neuroscience. Latency of the communication network and memory bandwidth are identified as the most severe bottlenecks in large-scale simulations. The authors found that in models of the type discussed here the spike-delivery phase is the most expensive phase once the data do not fit into the cache hierarchy anymore. The most severe hardware constraint is the saturation of memory bandwidth which is driven by the memory-latency effect of the irregular access patterns. While the CPU is capable of issuing many memory accesses in advance to partially hide latency of processing spikes, the authors expect that two established strategies, pipelining and prefetching, cannot effectively hide the latency introduced by the non-contiguous data accesses.

Prefetching attempts to hide cache misses and memory stalls [13] by overlapping memory access with computation. In general, two types can be distinguished: hardware-induced and software-induced prefetching. The former relies on the underlying hardware to detect patterns in memory access such that the hardware can take care of getting the data into the cache just when it is needed. The latter relies on hints in the source code via prefetch instructions indicating what data should be loaded into cache. This is a promising technique when hardware predictions fail but the access pattern can still be known by the developer a priori.

In the past, several techniques for prefetching have been investigated. Especially pointer-based data structures which suffer from "pointer chasing", as for example large graphs [14,15] and databases [16–18], profit from prefetching. The main challenge is to issue the prefetch instructions early enough such that the data is loaded timely into cache, but also late enough such that it does not clog the cache.

Recent work has focused on introducing code stages to deal with dynamic memory access and uncertainty in the number of lookups. This is achieved either by manually implementing state machines [19] or usage of coroutines [16–18]. Coroutines are resumable functions that can suspend their current execution. In both cases a prefetching instruction is inserted just before the memory stalls. After the prefetch instruction the program saves its current state and continues with other work. When entering the function the next time, the required piece of memory is available and operation can resume where it left off. These techniques are promising for cases where irregularity, such as variable-length pointer chains and potential early loop exits, prevent usage of simpler prefetching techniques.

If the number of pointer dereferences is known ahead of time and is constant across lookups, a promising technique to employ is group prefetching [20]. Group prefetching is a loop-transformation method which breaks a single for-loop into an outer and several inner loops allowing batchwise processing of code stages and critical data to be prefetched.

Another technique capable of hiding cache misses is software pipelining [21–23]. Here loops are transformed such that the instructions inside of the loop are carried out with an offset and overlapped with each other (details in Section 4.2). Thus, memory accesses and arithmetic operations inside the loop are no longer mutually dependent. As recent CPUs are superscalar, independent memory accesses and arithmetic operations can be executed in parallel. This increases the number of instructions completed per cycle. As the modification of the for loop increases the number of instructions, the increase in instructions per cycle has to be greater than the increase in number of instructions to improve the overall performance.

The present study begins with a description of our setting in terms of hardware and software as well as the profiling framework (Section 2). We use the open-source community simulation code NEST (Section 2.1) to obtain performance data from a real-world application and as a framework for reference implementations of the optimization techniques discussed. Absolute performance data require a concrete neuronal network model close to the ones used in neuroscientific studies. The network model introduced in Section 2.2 is prototypical for a wide class of models in neuroscience, it is scalable, and it has been used in a number of previous studies. A scientific community code needs to be developed and maintained over decades. It is important that new algorithms do not improve the performance on one architecture while making it impossible to adapt to the next generation of systems. Therefore, we assess the performance of a recent mainstream architecture, a common but older high-end cluster, and a dedicated supercomputer (Section 2.3). Section 3 describes the data structures representing neurons and synapses and how spikes travel through these data structures from arrival at the compute node to their ultimate delivery at the target neurons. Based on this, the subsequent two sections

form the core of the investigation. Section 4 proposes new algorithms based on latency-hiding techniques indicated above for different phases of spike delivery. Section 5 presents a quantitative analysis of the effects of the new algorithms and their combination on different hardware architectures. Finally, Section 6 derives a combination of algorithms delivering a robust overall performance gain across problem sizes and hardware architectures. The study concludes by considering the results in the context of applications with essentially random memory access patterns and discusses those implications for the interpretation of the von Neumann bottleneck and the design of neuromorphic computing systems.

Source code, simulation and analysis scripts are openly available [24] at Zenodo.[1] The presented conceptual and algorithmic work is part of our long-term collaborative project to provide the technology for neural systems simulations [25]. Preliminary results have been presented in abstract form [26].

2. Benchmarking framework

2.1. Simulation engine

In the present work, we evaluate the concepts and new algorithms in the framework of the simulation code NEST[2] [25], a widely used engine for spiking neuronal networks of natural density at the resolution of individual nerve cells and synapses. NEST is an open source project governed by the public society NEST Initiative[3] and a component of the ICT infrastructure[4] created by the European Human Brain Project (HBP). The development is managed via GitHub where contributions undergo a formal code review and consistency is ensured by continuous integration using automated style checks and testing. The kernel of the simulator is written in C++ and uses MT coequal OpenMP [27] threads for parallelization which are arranged into M MPI [28] processes harboring T threads each. The differential equations and state transitions defining neuron and synapse models are expressed in the domain specific language NESTML [29,30], which generates the required C++ code for dynamic loading into the simulation engine. The configuration of simulation experiments, including neuron and synapse models, network structure and recorded data, are specified interactively via Python using PyNEST [31,32]. Neurons are distributed across parallel resources together with their incoming synapses in a round-robin fashion [4]. This distribution implements a simple load balancing scheme as it minimizes the number of neurons from the same population, which may exhibit similar activity patterns, on the same thread. The work is based on commit 059fe89 of the development version leading to release 2.18 using the system `malloc()`.

NEST uses a globally time-driven simulation scheme [4], where neurons are typically updated every 0.1 ms and spike times are constrained to this time grid. There is a biophysical delay between the emission of a spike by the source neuron and the arrival at the target neuron. Therefore, it suffices to exchange spike data between threads in intervals of the minimal delay in the neuronal network [5]. Consequently, the simulation cycle propagating the dynamical state of the network is divided into three phases: update neurons, communicate spikes between threads, and deliver spikes to target neurons including the propagation of synaptic dynamics (see Fig. 1). Different neuron models require solvers of different computational load, ranging from precalculated exact propagator matrices for linear neuron models [33] to generic solvers for non-linear differential equations with adaptive time-stepping. Similarly the workload of synapses depends on the chosen model. Static synapses are stateless, whereas for plastic synapses the state may depend on the activities of the pre- and the postsynaptic neuron [34]. Over the years the scalability of NEST has been demonstrated on a range of supercomputers [10,35–37]. Recent revisions of the code employ MPI_Alltoall to send spikes only to MPI ranks where they have targets [10].

2.2. Network model

To measure and compare the suggested algorithmic improvements, we use a balanced random network model [38] as a benchmark similar to the one used in previous studies on neuronal network simulation technology [3,10,35–37,39,40]. The parameters for this benchmark model are specified in the parameter tables 1, 2, and 3 in Jordan et al. [10]. Note that in contrast to previous studies, where excitatory–excitatory connections exhibit spike-timing dependent plasticity, the model considered here uses static synapses, which do not exhibit any dynamics and consequently have a fixed weight.

We consider the model a scalable version of a typical neuronal network simulation as the neuronal activity exhibits an asynchronous irregular spike pattern and does not depend significantly on the model's size. Furthermore, the random connectivity of the network model represents a worst-case scenario in terms of network structure: Local connection patterns cannot, not even in principle, be exploited by representing subnetworks on a subset of available nodes as a single neuron connects with equal probability with any other neuron in the network. In the intermediate regime of network sizes between the local microcircuit and the brain scale addressed here, the synapses on an MPI process have a maximum number of different sources: at the lower end, many synapses have the same source neurons while at the upper end each synapse has a unique source neuron (cf. Fig. 1 of [9]).

All measurements of runtime in this study refer to the actual simulation time, or in short "sim time", where the network state is propagated; measurements of network-construction time and initialization time are not part of this study.

2.3. Systems

The detailed specifications of the computer systems are given in [9]. JURECA CM [41] consists of 1872 compute nodes (dual Intel Xeon E5-2680 v3 Haswell 12-core CPUs at 2.5 GHz), DEEP-EST[5] has 50 nodes (dual Intel Xeon Gold 6146 Skylake 12-core CPUs at 3.2 GHz), and the K computer [42] houses 82,944 nodes (8-core Fujitsu SPARC64 VIIIfx CPU at 2 GHz).

2.4. Measurements of runtime and profiling

For systematic benchmarking, we rely on the Jülich Benchmarking Environment (JUBE)[6] [43], which is a software suite actively developed by the Jülich Supercomputing Centre. For measuring the time consumption of different parts of the code, we rely on the `Stopwatch` class distributed with NEST. This class acts as a wrapper around `gettimeofday()` which is part of the header `<sys/time.h>` of the C POSIX library.[7]

We use the Microarchitecture Exploration analysis mode of the Intel VTune Profiler,[8] which provides detailed information on hardware usage. By specifying the option `uarch-exploration`, the running process is periodically interrupted enabling the sampling of hardware events from the processor. These events are used for calculating predefined ratios, which are reported once the program has finished. The study collects data only for the first 64 MPI processes and restricts

[1] https://www.zenodo.org.
[2] https://www.nest-simulator.org.
[3] https://www.nest-initiative.org.
[4] https://www.ebrains.eu.
[5] https://www.deep-projects.eu.
[6] https://www.fz-juelich.de/jsc/jube.
[7] https://pubs.opengroup.org/onlinepubs/9699919799/idx/head.html.
[8] https://software.intel.com/vtune.

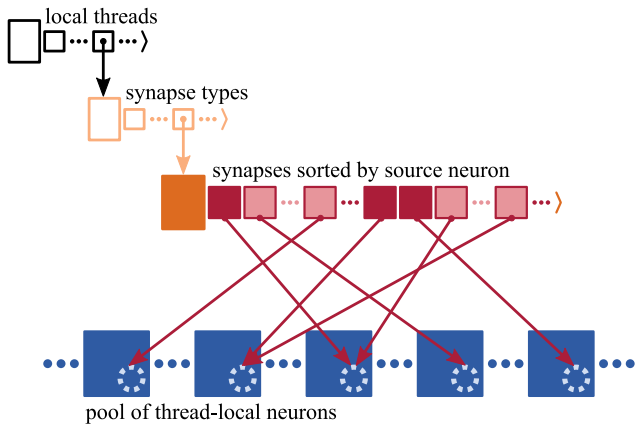

Fig. 3. Memory layout of synapses and neurons on each MPI process. Each process stores the local synapses (pink filled squares) in a three-dimensional resizable array sorted by hosting thread and synapse type. At the innermost level, synapses are arranged in source-specific target segments (dark pink: first synapse; light pink: subsequent targets); only one innermost array shown for simplicity. Target neurons (blue filled squares) are stored in neuron-type and thread-specific memory pools; only one pool shown for simplicity. Each neuron maintains a spike ring buffer (dotted light blue circles). Synapses have access to their target neurons' spike ring buffers through pointers (dark pink arrows).
Source: Adapted from Figure 1 of Pronold et al. [9].

measurements to the spike-delivery phase. Specifically, we focus on the Clockticks per Instructions Retired (CPI) event ratio. The CPI measure is calculated via dividing the number of unhalted processor cycles (clockticks) by the number of instructions retired. CPI indicates to what extent latency affects the application's execution, with smaller values corresponding to smaller latencies.

3. Memory access during spike delivery

In spiking neuronal network simulation code, the temporally sparse event-based communication between neurons presents a challenging performance bottleneck for modern architectures optimized for dense data but also presents an opportunity for optimization. Due to the connection data structures and the spike-delivery algorithm of our reference implementation NEST (Section 2.1), delivery of spikes to their targets involves frequent access to essentially random memory locations. Therefore, memory access is difficult to predict automatically, leading to long data access times due to ineffective use of caches. The following two sections describe the connection data structures and the spike-delivery algorithm that serves as a starting point for the latency-hiding techniques investigated in this study (Section 4) and as a reference in the quantitative analysis (Section 5).

3.1. Memory layout of synapses and neurons

Each synapse is represented on the same MPI process and thread as its target neuron [4]. A model synapse has a memory footprint of few tens of Bytes, whereas a model neuron easily consumes a Kilobyte or more. Each MPI process makes use of a three-dimensional resizable array to store the process-local synapses sorted by hosting thread and synapse type (Fig. 3). The data structure takes into account that each neuron typically connects to many target neurons (out-degree): In the innermost arrays, synapses are sorted by source neuron and thereby arranged in target segments, each consisting of at least one target synapse potentially followed by subsequent targets. The number of unique source neurons and hence the average length of target segments depends on the distribution of synapses across MPI processes and threads as well as the number of synapse types. In the limit of sparsity where the number of neurons in the network exceeds the number of thread-local synapses, the length of target segments approaches one.

REF: Reference algorithm delivering spikes to local targets. **TS** marks iteration over a synaptic target segment. **SYN** marks access to an individual target synapse; **RB** marks access to the spike ring buffer of the corresponding target neuron. Variables typeset in italics, functions in typewriter. See Pronold et al. [9] for a more detailed presentation.

```
    Data: spike_reg, synapses
    foreach spike in spike_reg do
        lcid ← spike.lcid
        subsq ← true
TS      while subsq do
SYN         (subsq, target_rb, delay, weight) ←
              synapses[lcid].Send()
            lcid ← lcid + 1
RB          target_rb.AddValue(delay, weight)
```

In order to account for synaptic transmission delays, each neuron maintains a ring buffer that accommodates the incoming spikes until they are due to take effect on the neuronal dynamics [4]. In the reference algorithm, a synapse has access to its target neuron's spike ring buffer through a pointer [cf. Section 4.2,9].

Neurons of the same type that are also hosted by the same thread belong to the same memory pool. The memory pool consists of multiple chunks that allow for contiguous storage of many objects. As synapses from many different source neurons converge on a target neuron (in-degree), the memory locations of target neurons cannot be ordered according to source neurons and are hence independent of the order of synapses in the target segments.

3.2. Spike-delivery algorithm

The distributed time-driven simulation of spiking neuronal networks proceeds in a cycle of updates to all neurons, communication of all recent spikes across MPI processes, and delivery of the spikes to their process-local synaptic and neuronal targets. After every spike communication, each MPI process holds a receive buffer filled with spike data that need to be dispatched to the process-local targets. Each spike entry addresses an entire target segment of synapses (Section 3.1). To this end, the spike entry needs to be able to locate the beginning of the target segment within the three-dimensional data structure. This data structure is storing the process-local synapses (Fig. 3), that is, the first synapse of the target segment. Therefore, each spike entry conveys identifiers for the hosting thread and the type of the synapse, as well as the synapse's index within the innermost resizable array.

In the reference algorithm, the delivery of spikes from the MPI receive buffer to the local targets hosted by different threads is a two-step process allowing for an entirely thread-parallel delivery with a single synchronization point [cf. Section 4.1, 9]. First, the threads sort the spike entries by hosting thread and synapse type in parallel using a dedicated intermediate data structure, called spike-receive register, only then the threads dispatch the spikes, now exclusively reading relevant entries.

Starting with the first synaptic target, the hosting thread subsequently processes all synapses of a spike's target segment. The number of spike entries in the receive buffer depends on the degree of distribution of synapses across MPI processes and threads as well as the number of synapse types. Arranging synapses in source-specific target segments such that only one spike needs to be communicated to address an entire segment is an effective optimization for small to medium-scale simulations (see Section 3.3 in 10). However, when increasing the network size in a weak-scaling experiment, ever more source neurons have ever fewer thread-local targets, or conversely, ever more thread-local synapses originate from different source neurons such that they

bwRB*: Delivery of spikes in batches of size B_{RB}, including group prefetching of spike ring buffers. **TS** marks iteration over a synaptic target segment. **SYN** marks access to an individual target synapse; **RB** marks access to the spike ring buffer of the corresponding target neuron; **RB*** marks group prefetching prior to access. Based on REF.

```
    Data: spike_reg, synapses
    create arrays target_rb, delay, weight of size B_RB
    i ← 0
    foreach spike in spike_reg do
        lcid ← spike.lcid
        subsq ← true
TS      while subsq do
SYN         (subsq, target_rb[i], delay[i], weight[i]) ←
              synapses[lcid].Send()
            i ← i + 1
            lcid ← lcid + 1
            if i == B_RB then
                for i ← 0 to B_RB − 1 do
RB*                 prefetch target_rb[i]
                for i ← 0 to B_RB − 1 do
RB                  target_rb[i].AddValue(delay[i], weight[i])
                i ← 0
    process remaining entries in target_rb
```

lagRB: Delivery of spikes introducing a lagged access to ring buffers with respect to the corresponding target synapses; lag is B_{RB}. **TS** marks iteration over a synaptic target segment. **SYN** marks access to an individual target synapse; **RB** marks access to the ring buffer of the target neuron. Based on REF.

```
    Data: spike_reg, synapses
    create arrays target_rb, delay, weight of size B_RB +1
    i ← 0
    j ← 0
    foreach spike in spike_reg do
        lcid ← spike.lcid
        is_init ← true
        subsq ← true
TS      while subsq do
SYN         (subsq, target_rb[i], delay[i], weight[i]) ←
              synapses[lcid].Send()
            i ← i + 1
            if i == B_RB + 1 then
                i ← 0
            lcid ← lcid + 1
            if is_init and i == B_RB then
                is_init ← false
            else
RB              target_rb[j].AddValue(delay[j], weight[j])
                j ← j + 1
                if j == B_RB + 1 then
                    j ← 0
    process remaining entries in target_rb
```

can no longer be combined into long target segments addressed by a single source-specific spike entry. This is the limit of sparsity discussed in Section 3.1.

The synapse object stores all information relevant for the subsequent delivery to the target neuron, foremost a pointer to the target neuron's spike ring buffer. During delivery of the spike to the target neuron, the algorithm retrieves the pointer from the synapse as well as synaptic properties such as delay and weight, which define time and amplitude of the spike's impact on the neuron, respectively. Taking into account the delay, the algorithm adds the weight of the incoming spike to the correct position in the neuronal spike ring buffer.

For each relevant spike entry the hosting thread accesses target synapses and neurons in an alternating fashion. All target synapses of a spike entry are in contiguous locations in memory as they are part of the same target segment, but the corresponding target neurons are in nonadjacent memory locations. Furthermore, with every spike entry, the thread proceeds to a different synaptic target segment, most likely not in a proximate memory location. With increasing sparsity of the network in a weak-scaling experiment, such switches between target segments become more frequent as more spike entries need to be delivered to ever shorter target segments. In the sparse limit, both accessing target synapses and the corresponding target neurons requires the hosting thread to jump to random memory locations. The pseudocode of the reference algorithm (REF, illustration Fig. 4A) presents this memory bottleneck of the spike-delivery algorithm in an abstract way. It omits intricacies caused by support for multi-threading and different synapse types.

4. Latency-hiding techniques

Based on the reference algorithm (REF), we investigate three techniques to hide memory fetch latency and reduce the number of cache misses during spike delivery. To assist the reader in following the algorithmic changes, we introduce each of the potential adaptations and a combination of two adaptations in its own subsection below where the titles define acronyms for later reference to the respective pseudo codes.

The algorithms bwRB* (Section 4.1) and lagRB (Section 4.2) are competing adaptations of the reference algorithm REF, whereas bwTS (Section 4.3) is an adaptation that can be combined with either of the former as exemplified by bwTSRB* (Section 4.4).

The pseudocode is reduced to the essential elements of the spike delivery; in particular, we apply the following simplifications: the code does not take into account multiple threads or synapse types and the initial thread-parallel transfer of spike entries from the MPI receive buffer to the spike-receive register is omitted (Section 3.2). The performance of the algorithms is analyzed in Section 5.

The reference algorithm has access to a resizable array of thread-local *synapses* and to a spike register (*spike_reg*), which contains all spike entries that need to be delivered [see Section 4.1 of 9, for details]. For each spike entry, the location of the first target synapse is extracted and assigned to the variable *lcid*, which is then used in the enclosed while loop to iterate over the spike's entire synaptic target segment within the *synapses* array (**TS**). Each synapse stores an indicator (*subsq*) of whether the target segment continues or not. To deliver a spike to the target synapse at position *lcid*, the synapse member function `Send()` is called on *synapses*[*lcid*] returning the indicator *subsq*, the pointer to the spike ring buffer, and the synaptic delay and weight (**SYN**). The pointer is then used to call `AddValue()`, a member function of the neuronal spike ring buffer requiring the delay and the weight (**RB**). Taking into account the delay, the spike ring buffer adds the weight of the incoming spike to the correct position. This implements the delivery of the spike to the target neuron. Note the dependency of the algorithmic step **RB** on step **SYN**, which is readily visible as a result of the preparatory study [Section 4.2 of 9].

4.1. Batchwise access to spike ring buffers (bwRB*)

We next consider a loop-transformation method that allows for group prefetching [20]. The method breaks a single for loop of L

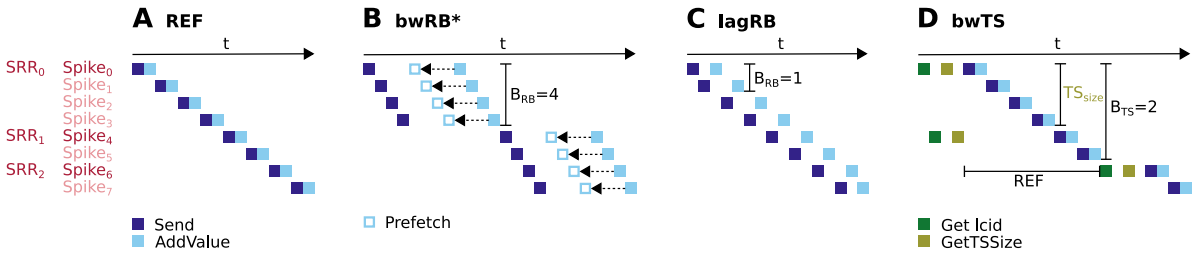

Fig. 4. Graphical representations of latency hiding algorithms (horizontal temporal axes t). Example of eight spike entries for three target segments in a spike-receive register (SRR): initial spike in each segment in dark pink, subsequent spikes in light pink. (A) REF. In the reference algorithm spikes are delivered by alternating between two operations: first the pointer to the target neuron's spike ring buffer and synaptic properties are retrieved (dark blue squares: Send), then the weights are added to the spike ring buffers (light blue squares: AddValue). (B) bwRB*. Delivery operations are organized in code stages of size B_{RB} (here $B_{\mathrm{RB}} = 4$). First, B_{RB} synapses are accessed to retrieve pointers and synaptic properties. Second, the B_{RB} spike ring buffers are prefetched into the cache (light blue unfilled squares and dotted arrows indicating prefetching). Third, weights are added to the B_{RB} buffers. Once one group has been processed the algorithm moves on to the next group. (C) lagRB. Delivery operations are reformed such that Send–AddValue sequences are carried out with an offset of B_{RB} (here $B_{\mathrm{RB}} = 1$) and overlapped. (D) bwTS. The first code stage extracts the locations of the first target synapses of the corresponding target segments from B_{TS} spike entries (dark green squares: Get lcid, here $B_{\mathrm{TS}} = 2$). The second code stage uses the B_{TS} locations to retrieve the corresponding target-segment sizes (yellow–green squares: GetTSSize). Finally spike delivery is carried out as in the reference algorithm (horizontal segment labeled REF).

iterations containing several code stages into an outer and several inner loops of size L_B and B, respectively, where $L = L_B B$. For example, an original loop $\{X_i Y_i\}^L$ over two operations X_i and Y_i depending on running index i is transformed into $\{X_{(j-1)B+1} \dots X_{jB} Y_{(j-1)B+1} \dots Y_{jB}\}^{L_B}$. The code stages of the original for loop are hence processed in a batchwise manner. All code stages are handled B times before moving to the next step of the outer loop. By introducing code stages of size B, a prior loop prefetching all critical data can be inserted, making use of memory-level parallelism.

We adapt the reference algorithm such that access to synapses and access to spike ring buffers is carried out in batches of size B_{RB} (bwRB*, illustration Fig. 4B). First, B_{RB} synapses are accessed to retrieve the pointers to the corresponding target neurons' spike ring buffers as well as the synaptic weights and delays (**SYN**). The retrieved synaptic information is temporarily buffered in three auxiliary arrays (*target_rb*, *delay*, *weight*) of size B_{RB}. Each time B_{RB} synapses have been accessed such that the auxiliary buffers are filled, batchwise access to the B_{RB} collected target spike ring buffers is triggered to add the corresponding weights to the correct buffer positions (**RB**). An optional prior step is the software-induced prefetching of the B_{RB} spike ring buffers into the cache (**RB***). To this end, in the NEST implementation (Section 2.1) we make use of the GCC built-in function `__builtin_prefetch(const void *addr, . . . )`[9]; the prefetching can be enabled at compile time. The method is referred to as group prefetching as it consecutively prefetches data from an entire batch of B_{RB} individual memory addresses before operating on the group of data instead of using a per-memory-address approach alternating between prefetching and processing. In the quantitative analysis (Section 5), we refer to this set of optimizations as either bwRB or bwRB*, where the asterisk indicates prefetching.

The batchwise progression is agnostic with regard to the boundaries of synaptic target segments (Section 3.1), which means that in case of short target segments, it can take several iterations of the for loop over spike entries to process B_{RB} synaptic targets. Leaving and re-entering of the enclosed while loop that is traversing the synaptic target segment of a specific spike entry (**TS**) does not affect the progression. Eventually, all spike entries in the register are processed and either delivered to their synaptic targets or added to the auxiliary array. After the loop over the spike register exits, the algorithm delivers the remaining entries in the auxiliary arrays to their targets.

4.2. Lagging access to spike ring buffers (lagRB)

This optimization exploits the idea of software pipelining for spike delivery. In software pipelining [21–23], loops are reformed in such a way that the instructions inside of the loop are carried out with an offset of $B \geq 1$ and overlap with each other. For example, an original loop of L iterations $\{X_i Y_i\}^L$ over two operations X_i and Y_i depending on index i is transformed into $\{X_i\}^B \{X_i Y_{i-B}\}_{B+1}^{L} \{Y\}_{L-B+1}^{L}$. By doing so, the operation X inside of the central loop is from a different iteration than Y.

We adapt the reference algorithm such that access to synapses and access to spike ring buffers is still carried out in an alternating fashion but algorithmic progression is always B_{RB} synapses ahead of spike ring buffers (lagRB, illustration Fig. 4C). This means when a ring buffer is accessed the synaptic information and the respective pointers for the next B_{RB} spike ring buffers are already available in local arrays. To this end, B_{RB} synapses are initially accessed to retrieve the pointers to the corresponding target neurons' spike ring buffers as well as the synaptic weights and delays (**SYN**). The retrieved synaptic information is temporarily buffered in auxiliary arrays (*target_rb*, *delay*, *weight*) of size $B_{\mathrm{RB}} + 1$. Once B_{RB} synapses have been accessed such that the auxiliary buffers are almost full (`if`-condition is met), the initialization phase ends and alternating access to synaptic targets (**SYN**) and neuronal spike ring buffers (**RB**, `else`-clause) starts, using the auxiliary arrays as ring buffers and the indices i and j for writing to the arrays and reading from the arrays, respectively. In the quantitative analysis (Section 5) we refer to this optimization as lagRB.

As for bwRB*, leaving and re-entering of the enclosed while loop that is traversing the synaptic target segment of a specific spike entry (**TS**) does not interrupt the cyclic processing of the auxiliary arrays. Eventually, all spike entries in the register are processed and delivered to their synaptic targets, but the auxiliary arrays still contain synaptic information that needs to be delivered to the spike ring buffers.

4.3. Batchwise access to target segments (bwTS)

The previous two algorithms, bwRB* and lagRB, are concerned with disentangling access to synapses and neuronal spike ring buffers. They implicitly make hopping from one synaptic target segment to the next more seamless as they advance the writing and reading of the auxiliary arrays (*target_rb*, *delay*, *weight*) irrespective of the boundaries of single target segments. The optimizations of the present algorithm, bwTS (illustration Fig. 4D), directly target the iteration over spike entries in the spike receive register (*spike_reg*) in the outermost loop and the concomitant processing of synaptic target segments of different lengths in different memory locations. The quantitative analysis (Section 5) refers to this set of optimizations as bwTS.

We adapt the reference algorithm to process spike entries in the spike receive register in batches of size B_{TS}, which in turn entails batchwise access to B_{TS} distinct target segments (bwTS). Like in bwRB*, this

[9] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html.

bwTS: Delivery of spikes with processing of spike entries and corresponding target segments in batches of size B_{TS}. **TS** marks iteration over a synaptic target segment using a fixed count loop according to the target-segment size *ts_size*. **SYN** marks access to a target synapse; **RB** marks access to the spike ring buffer of the corresponding target neuron. Based on REF.

```
    Data: spike_reg, synapses
    create arrays lcid, ts_size of size B_TS
    l ← 0
    repeat spike_reg.Size() / B_TS times
        for k ← 0 to B_TS − 1 do
            lcid[k] ← spike_reg[l + k].lcid
        l ← l + B_TS
        for k ← 0 to B_TS − 1 do
            ts_size[k] ← synapses[lcid[k]].GetTSSize()
        for k ← 0 to B_TS − 1 do
TS          repeat ts_size[k] times
SYN             (target_rb, delay, weight) ←
                  synapses[lcid[k]].Send()
                lcid[k] ← lcid[k] + 1
RB              target_rb.AddValue(delay, weight)
    process remaining entries in spike_reg
```

optimization is based on a loop-transformation method (Section 4.1) but targets a different part of the algorithm. Moreover, we replace the while loop iterating over every target synapse of a specific target segment with a fixed count loop (**TS**). This requires, however, that the length of the target segment is available when entering the fixed count loop. Here, we decide on a straightforward solution: the information is provided by the first target synapse of each target segment. To this end, we extend synapse objects with a member variable to store the target-segment size, which needs to be determined just once when all synapses have been created (not shown in the algorithm); the public member function `GetTSSize()` returns the size. Note that while the algorithm only requires this capability for the first synapse of each target segment, all synapse objects are equipped with the extra member variable as they are stored in a container for homogeneous objects (Section 3.1). In our reference implementation (Section 2.1), we ensure that the extra member variable does not increase the per-synapse memory usage by reducing the storage size of another synaptic member variable, namely the synaptic delay. This does not affect the precision of results for our benchmark network model as the reduced storage size is sufficient to fully represent the homogeneous and relatively short delays of the model (Section 2.2). This solution does, however, not generalize to all types of network models as, for example, in case of longer delays, an increase in per-synapse memory usage due to the additional member variable might be inevitable.

For each batch of B_{TS} spike entries, the algorithm carries out three consecutive for loops each with a fixed number of B_{TS} iterations. In the first for loop iterating over B_{TS} spike entries, the locations of the first target synapses of the corresponding target segments are extracted from the spike entries and buffered in the auxiliary array *lcid*. In the subsequent for loop the B_{TS} locations are used to access the first target synapses to retrieve the corresponding target-segment sizes, which are buffered in the auxiliary array *ts_size*. In the final for loop and the enclosed fixed count loop, alternating access to synaptic targets (**SYN**) and neuronal spike ring buffers (**RB**) is carried out like in the reference algorithm. If the number of spike entries in the spike receive register is divisible by B_{TS} with a remainder, the remaining spike entries are processed in a similar fashion using three consecutive for loops, where the fixed number of iterations is given by the number of remaining entries.

bwTSRB*: Delivery of spikes combining processing of spike entries and corresponding target segments in batches of size B_{TS} (bwTS) with access to target synapses and neuronal ring buffers in batches of size B_{RB}, including group prefetching of spike ring buffers (bwRB*). **TS** marks iteration over a synaptic target segment using a fixed count loop according to the target-segment size *ts_size*. **SYN** marks access to an individual target synapse; **RB** marks access to the ring buffer of the corresponding target neuron; **RB*** marks group prefetching prior to access. Based on REF.

```
    Data: spike_reg, synapses
    create arrays lcid, ts_size of size B_TS
    create arrays target_rb, delay, weight of size B_RB
    l ← 0
    i ← 0
    repeat spike_reg.Size() / B_TS times
        for k ← 0 to B_TS − 1 do
            lcid[k] ← spike_reg[l + k].lcid
        l ← l + B_TS
        for k ← 0 to B_TS − 1 do
            ts_size[k] ← synapses[lcid[k]].GetTSSize()
        for k ← 0 to B_TS − 1 do
TS          repeat ts_size[k] times
SYN             (target_rb[i], delay[i], weight[i]) ←
                  synapses[lcid[k]].Send()
                i ← i + 1
                lcid[k] ← lcid[k] + 1
                if i == B_RB then
                    for i ← 0 to B_RB − 1 do
RB*                     prefetch target_rb[i]
                    for i ← 0 to B_RB − 1 do
RB                      target_rb[i].AddValue(delay[i], weight[i])
                    i ← 0
    process remaining entries in spike_reg and target_rb
```

4.4. Combined batchwise access to target segments and spike ring buffers (bwTSRB*)

The combined algorithm bwTSRB* adopts the loop structure of bwTS, but the instructions inside the fixed count loop iterating over a specific synaptic target segment (**TS**) are adapted according to bwRB*. Fig. 4 illustrates that this is possible because REF (Fig. 4A) reappears as a component of bwTS (Fig. 4D). The combined algorithm requires all auxiliary arrays of the two individual algorithms, where indices l and k and index i are used as in bwTS and bwRB*, respectively.

5. Results

We quantitatively evaluate the effect of the three different techniques of latency hiding bwRB*, lagRB, and bwTS on simulation time relative to the code after preparatory improvement of parallelization (Section 3.2), where the optimization bwTS can be combined with either bwRB* or lagRB because they modify different parts of the code. As the combined optimizations may either support each other or in the worst case reduce the impact of the best individual optimization, we also present performance data for a combined scenario of bwTS and bwRB*. The section concludes by demonstrating that the improved performance is indeed a result of a reduction of clock ticks per instruction retired (Section 5.1).

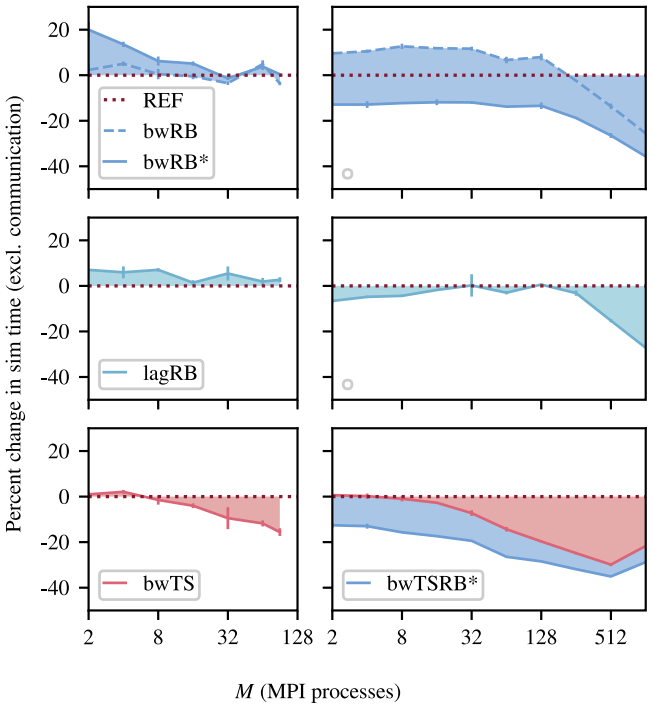

Fig. 5. Cumulative change in simulation time relative to the preparatory code disregarding communication as a function of number of MPI processes. Left column DEEP-EST CM and right column JURECA CM: linear-log representation for number of MPI processes $M \in \{2; 4; 8; 16; 32; 64; 90\}$ and $M \in \{2; 4; 8; 16; 32; 64; 128; 256; 512; 1,024\}$, respectively. Dark carmine red dotted line at zero percent (REF, Section 3.2) indicates performance of reference code. First row, batchwise access to spike ring buffers (Section 4.1) with batches of size 16 without group prefetching (bwRB, dashed light blue curve) and with prefetching (bwRB*, solid light blue curve). Second row, software pipelining with a lag of 16 (lagRB, turquoise curve). Third row, batchwise access to target segments with batches of size 16 (bwTS, solid coral red curve), for JURECA CM (right) further combined with batchwise access to spike ring buffers using prefetching (bwRB*, same coloring as in top row: solid light blue curve; see bwTSRB* for combined pseudocode). Shadings fill area to respective reference. Weak scaling of benchmark network model as in Fig. 1.

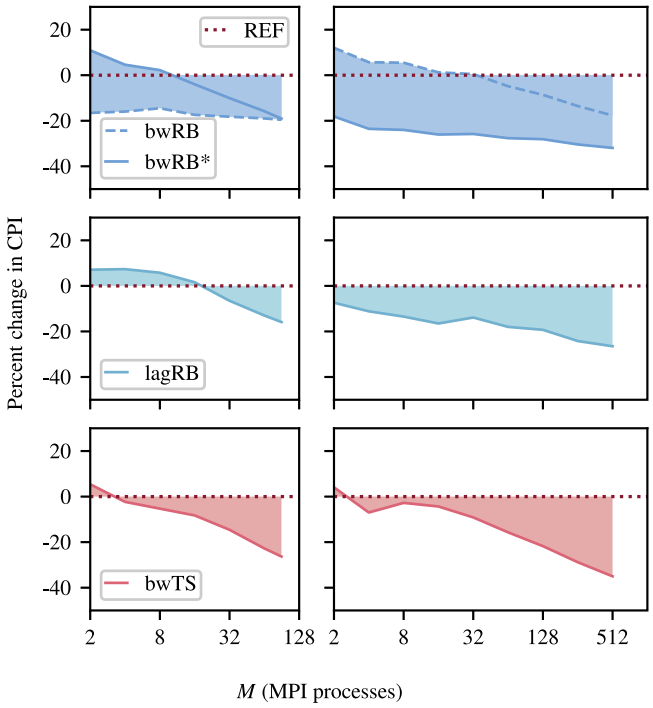

Fig. 6. Relative change in clock ticks per instruction retired (CPI) during spike delivery as a function of the number of MPI processes. Same arrangement of panels and labeling as in Fig. 5.

Batchwise access to spike ring buffers (Fig. 5, top row) without group prefetching (bwRB, Section 4.1) has no effect on the performance on the DEEP-EST CM and enabling prefetching (bwRB*) actually worsens the situation for small numbers of MPI processes. The situation is entirely different on JURECA CM, which uses an older generation of processors. Here, plain batchwise access has a negative effect on the performance, but group prefetching leads to an overall performance improvement. At larger numbers of MPI processes already the batchwise processing increasingly improves performance and the additional gain by group prefetching remains constant. For tested batch sizes between 1 and 64, we observe the least decline in performance for batch sizes of 8 or larger on DEEP-EST CM and the best performance for batch sizes of 16 or larger on JURECA CM (data not shown). However, a comprehensive analysis is outside the scope of this study; we do not claim that this observation can be generalized to other architectures.

Lagged access to ring buffers (Fig. 5, middle row) is an alternative latency-hiding technique for the same part of the code (lagRB, Section 4.2), where we use a lag of 16. The algorithm thus retrieves information from a target synapse 16 steps ahead from the position in the target segment where it currently accesses the corresponding ring buffer, thereby decoupling these operations. Again we observe almost no effect on the DEEP-EST CM. JURECA CM exhibits an increasing gain only for large numbers of MPI processes. For tested lags between 1 and 16, we observe little differences on DEEP-EST CM and the best performance for lags of 2 and larger on JURECA CM (data not shown).

Batchwise processing of the spike-receive register [9] and the synaptic target segments (Fig. 5, bottom row) has an increasing gain reaching 20% at 90 MPI processes on the DEEP-EST CM, and the total gain reaches more than 40% on JURECA CM (bwTS, Section 4.3). For tested batch sizes between 1 and 64, we observe little differences on DEEP-EST CM and the best performance for batch sizes of up to 16 on JURECA CM (data not shown).

Finally, we investigate a combined implementation of batchwise access to target segments and batchwise access to spike ring buffers including group prefetching (bwTSRB*) as the two techniques modify different parts of the code. We do not further consider lagged access to ring buffers (Section 4.2), as lagRB is only effective for large numbers of MPI processes, while its alternative bwRB* exhibits a sustained gain over the full range. The lower right panel of Fig. 5 shows that on JURECA CM bwRB* is effective already at small numbers of MPI processes but continues to improve the performance of bwTS, leading to a combined gain of 50% at larger numbers of MPI processes. As expected from batchwise access to target segments and spike ring buffers alone, on DEEP-EST CM their combination has no positive effect (data not shown).

The improved parallelization described in Pronold et al. [9] and the latency-hiding techniques investigated in this study jointly reduce simulation time with respect to the original code (Fig. 1). While the former is most effective for small numbers of MPI processes where synaptic target segments are long (Fig. 2), the latter is most effective for larger numbers of MPI processes where target segments significantly shorten. The differential behavior of the combined technologies leads to a sustained performance gain of 30% to 50% over the whole range of investigated MPI processes.

5.1. Dependence of effectiveness on microarchitecure

The latency-hiding techniques introduced in Section 4 promise to speed up the overall code by reducing the latency of memory access. Indeed our measurements show that our particular application becomes

substantially faster, despite additional lines of code required to implement the techniques. Profiling tools (Section 2.4) provide direct access to the latency that processor instructions experience. This allows us to investigate whether the new algorithms are faster due to the reduction of latency. Specifically, we are investigating clock ticks per instruction retired (CPI), a metric available in the tool VTune (Section 2.4). All algorithms exhibit a decrease in CPI compared to REF (Fig. 6), except for less than 32 MPI processes bwRB* on DEEP-EST CM and bwRB for JURECA CM. Larger networks show larger gains and lagRB and bwTS show similar behavior on both machines. The dependence on network size is particularly pronounced for bwTS. This directly relates to the decreasing length of target segments, diminishing the algorithmic potential for ordering synaptic data for sequential processing. As bwTS addresses the bottleneck of hopping from one target segment to the next, its effect increases as target-segment lengths decrease. The characteristics of bwRB (Section 4.1) and bwRB* on DEEP-EST and JURECA appear inverted. This hints that the processor in DEEP-EST CM, in contrast to JURECA CM, automatically prefetches the necessary data on time. Adding additional explicit prefetching instructions may interfere with the automatic prefetching and hence degrade performance. While the changes in simulation time (Fig. 5) follow the changes in CPI, the reduction in sim time does not fully reflect the success observed in CPI. All latency-hiding techniques come with additional lines of source code. This may lead to an increased number of processor instructions that the reduction in CPI has to make up for, but the compiler may now also find more opportunities for reducing the number of instructions.

6. Discussion

At the outset of this investigation, we observed that for a typical neuronal network the spike-delivery phase dominates total simulation time in state-of-the-art simulation code and that the time required for spike delivery increases under weak scaling such that it still dominates at 1024 MPI processes (Fig. 1). This article explores techniques to rearrange the elementary algorithmic steps required to deliver the incoming essentially random spike data to the thread-local targets such that they can be more efficiently processed by conventional computer hardware. Each of the techniques under consideration is characterized individually, but some address different steps of spike delivery. Combining the most promising techniques achieves a significant performance boost.

The combined spike-delivery algorithm starts with the thread-parallel sorting of the spike data from the MPI receive buffer into a thread-specific data structure called spike receive register (SRR). Here the sorting is performed according to hosting thread and synapse type of each spike's synaptic targets. This preparatory work improves parallelization but is not related to the optimization of cache performance on the level of the individual threads. The technical companion paper [9] provides the detailed description of the SRR together with pseudo code. After sorting into the spike receive register, the thread-parallel processing of the SRR in fixed-size batches cycle through three stages (bwTS, Section 4.3): for each batch of spike entries, in the first stage, the location and, in the second stage, the length of the spike's synaptic target segment are collected in separate arrays before continuing with the actual delivery as a third stage. In the third stage, synapses and corresponding neuronal spike ring buffers are also processed in batches (bwRB*, Section 4.1) . First separate arrays are collected for each batch of synapses before adding the weights to the buffer, irrespective of target-segment boundaries, weight, delay, and pointer to ring buffer.

As a result of the redesign, the time required for spike delivery is halved (reduced by factor of 2.1) for two MPI processes compared to the original algorithm and reduced to a third (by a factor of 2.9) for 1024 MPI processes. In the original algorithm, the increased network sparsity at larger numbers of MPI processes causes an increase in spike-delivery time by a factor of 4.5 between two and 1024 MPI processes. The new algorithm limits this dependence to a factor of 3.3 and substantially reduces overall simulation time. The measurements were obtained using ParaStationMPI v5.2.2-1. Preliminary data indicate that the more recent version v5.4 achieves a significant reduction in communication time (data not shown). However, following up on this observation is not within the scope of this manuscript due to JURECA CM being decommissioned. In summary, with the new algorithm, spike delivery still dominates the simulation time below 1024 MPI processes under weak scaling, but it increases less rapidly such that now even below 1024 MPI processes the major cause for the loss in performance is the increase in communication time. A second qualitative change sets in at 1024 MPI processes, where the absolute simulation time is no longer dominated by spike delivery but by communication time. Hence, the new algorithm overcomes the previously reported barrier of spike delivery limiting the performance on supercomputers (Figure 12 in [10]). Now progress can be made by optimizing communication, for example, by exploiting the spatial organization of neuronal networks. Cortical neuronal networks are characterized by a local coupling with a space constant of a few hundred micrometers and delays in the range of a tenth of a millisecond, combined with long distance coupling between brain areas and delays beyond a millisecond. If cortical areas were represented on one or a few compute nodes, the communication between nodes hosting different areas could be reduced to much larger intervals than required between nodes hosting neurons of the same area. Topology-aware distribution of neuronal networks across compute nodes has been exploited in other simulation codes [44,45] but without taking into account different delay categories.

Performance profiling of the algorithms bwRB*, lagRB, and bwTS indicates that they indeed reduce latency (Section 5.1). In accordance with the measurements of sim times (Section 5), the techniques are more successful on JURECA CM (Intel Haswell) than on DEEP-EST CM (Intel Skylake). The relative reduction in the performance metric clock ticks per instruction retired (CPI) is also more pronounced on JURECA CM than on DEEP-EST CM. We speculate that the improved cache utilization of the newer generation processor (Skylake) renders bwRB* and lagRB ineffective or even detrimental, at least in the regime up to 90 MPI processes investigated here. On JURECA CM, the effect on sim time increases beyond 128 MPI processes, which is beyond the current capacity of DEEP-EST CM. Therefore, emulating simulations on more than 128 MPI processes on DEEP-EST CM using the NEST dry-run mode [40] may provide further insights. A comprehensive comparison between the two generations of processors based on microbenchmarks as presented in [46] for the related microarchitectures Intel Broadwell and Intel Cascade Lake is out of the scope of this study. It is our hope though that our efforts to present the changes to code in an abstract fashion make this study relevant for a broader computer science community and might even inspire the definition of future microbenchmarks. The present work focuses on the level of source code only. It may be illuminating to explore in future studies the effect of algorithmic changes on the level of the resulting assembler code.

The incoming spike events of a compute node specify the thread that hosts the target neuron as well as the location of the synaptic targets. But these spikes are unsorted with respect to the hosting thread and synapse type. Nevertheless, the technical companion paper [9] shows that the processing of spikes can be completely parallelized. This process requires only a single synchronization between the threads at the point where the spikes have been sorted according to target threads and synapse types. This is when all spikes have been transferred from the MPI receive buffer into the novel spike receive register. This suggests that spike delivery can fully profit from a further increase in number of threads per compute node. Due to the large number of outgoing synapses per neuron, early work on distributed simulations of spiking neuronal networks employing tens of compute nodes was concerned with the problem of efficiently delivering each spike emitted by a specific source neuron on a specific MPI process to many targets on all MPI processes [4]. With increasing parallelization using more compute

nodes and threads, however, each neuron has ever fewer thread-local targets resulting in ever shorter synaptic target segments in the data structure storing the local synapses. Therefore, later work concentrated on reducing the memory overhead caused by the increasing number of shorter target segments [37]. However, shorter target segments are also a burden computationally because memory locations need to be switched often leaving little opportunity for vectorization. The present study overcomes the problem of short segments by reorganizing the spike delivery algorithm such that it operates across the boundaries of target segments. In this way becomes more independent of the degree of parallelization. Spikes are routed to their thread-local targets in a batchwise fashion using the technique of loop transformation. This explicit declaration of virtually independent code blocks apparently helps compilers to generate efficient machine code.

Due to the simplicity of the neuron model of the benchmark and the lack of synaptic plasticity, the application has little workload in terms of the propagation of neuron and synapse dynamics and thus exposes bottlenecks in the delivery of spikes to local targets and the communication between compute nodes. The synaptic delay of about a millisecond assumed in the present benchmark model is used in prominent neuronal network models of the balanced random class [38] and representative for the connectivity at the brain scale [1]. In models of the local network below a square millimeter of cortical surface [cf.47], the minimal delay (Section 2.1) is an order of magnitude smaller than the delay considered in this study. Consequently, the simulation of such models requires a ten-fold increase in the number of communication calls, and therefore, the communication phase takes up a larger fraction of the total simulation time. Albers et al. [48] introduce different optimizations to the communication phase for this type of models. Nevertheless, for a given spike rate, the amount of spikes that need to be delivered in a given stretch of biological time is independent of the number of communication calls. Therefore, we expect the optimizations discussed in the present work to still be effective, while their impact on total simulation time will be lower. Neuroscientists have started to investigate models combining the local structure of the brain with its organization over long distances [49]. The minimal delay in such a brain-scale network is the minimal delay in the local structure. Unless the placement of neurons on compute nodes respects the architecture of the network, global communication in intervals is required, like it is for the local network. This limits the success of optimizations of spike delivery.

Our analysis is restricted to static neuronal networks. In such networks, all synaptic connections and their weights are determined at the time of network construction. In nature, synaptic efficacies change over time: the phenomenon called synaptic plasticity. A wide class of models of synaptic plasticity can be formulated as a scheme driven by the presynaptic spike events in which the variables tracing the state of a synapse are only updated for those times where the synaptic weight becomes visible for the postsynaptic neuron [39]. In this way, the computations become part of the spike delivery phase and contribute substantially to its duration. Depending on the particular plasticity model, alongside the location of the ring buffer, additional information on the target neuron needs to be accessed. This may be the time of the last spike, the full record of spikes since the last presynaptic spike, or the membrane potential [50]. Our companion paper [9] discusses the resulting tradeoff between memory consumption and speed. A domain specific language like NESTML [29] could avoid cluttering the model pool with solutions for different optimization goals by providing directives influencing the balance between the generation of code with lower memory usage or higher speed depending on the needs. How the more complex data handling for plastic synapses and the floating point operations on the synaptic weights interact with the batchwise processing needs to be evaluated. The same holds for a further mechanism, removing and creating synapses, called structural plasticity [51].

The present study analyses spike-delivery times and overall simulation times in a weak-scaling scenario. Neuroscientists and industry may, however, also be interested in reducing the simulation time of an application for a fixed size network. For example, there may be interest in networks studied over long stretches of biological time, as required for system-level learning or for the capability to interact with the real world in robotics. Therefore strong-scaling scenarios are of interest. With an increasing number of threads, the number of target neurons and therefore synapses a thread needs to take care of decreases. Consequently the amount of memory a thread operates on decreases. This suggests that it becomes easier for the core running the thread to predict memory access, and if the amount of cache memory per core decreases proportionally to the number of cores, faster execution is expected. In this way, the combination of a highly parallel spike-delivery algorithm with a many-core architecture alleviates the von Neumann bottleneck of conventional applications. Indeed there is recent data for a many-core compute node showing its competitiveness with respect to neuromorphic and GPU systems [52]. Thus our findings reconsidering the spike-delivery algorithm support a more positive view on the prospects of improving the performance of neuronal network simulations by prefetching and pipelining than the analytical results of [11,12]. It remains to be seen whether the optimizations by batchwise processing explored in the present work become less relevant in situations where each core needs to manage a small amount of memory anyway.

The study removes several bottlenecks in routing spikes in a compute node to ever more distributed targets. The spike-delivery phase of a simulation code for spiking neuronal network remains a specific sorting problem in space and time, but our study exposes that its fully parallelizable nature will certainly benefit from future fine-grained processing hardware.

Funding

Partly supported by the European Union's Horizon 2020 (H2020) funding framework under grant no. 785907 (Human Brain Project, HBP SGA2), no. 945539 (HBP SGA3), and no. 754304 (DEEP-EST), as well as the Helmholtz Association Initiative and Networking Fund project no. SO-092 (Advanced Computing Architectures, ACA), as well as the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 368482240/GRK2416 and for open access 49111148. Use of the JURECA supercomputer in Jülich was made possible through VSR computation time grant JINB33. This research used resources of K computer at the RIKEN Center for Computational Science (R-CCS). Supported by the project Exploratory Challenge on Post-K Computer (Understanding the neural mechanisms of thoughts and its applications to AI) of the Ministry of Education, Culture, Sports, Science and Technology (MEXT).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We are grateful to Mitsuhisa Sato for his guidance, which helped us shape the project, to Johanna Senk and Dennis Terhorst for fruitful discussions and joint efforts at the HPC Optimization and Scaling Workshop 2019 at the Jülich Supercomputing Centre, to Sebastian Lührs for help with JUBE, to Laurent Montigny and Hans-Christian Hoppe for insights on the Intel microarchitectures, to Jessica Mitchell for proofreading the manuscript, to our colleagues in the Simulation and Data Laboratory Neuroscience of the Jülich Supercomputing Centre for continuous collaboration, and to the NEST development community for the concepts and implementation of NEST (http://www.nest-simulator.org).

References

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

[2] G.T. Einevoll, A. Destexhe, M. Diesmann, S. Grün, V. Jirsa, M. de Kamps, M. Migliore, T.V. Ness, H.E. Plesser, F. Schürmann, The scientific case for brain simulations, Neuron 102 (2019) 735–744, http://dx.doi.org/10.1016/j.neuron.2019.03.027.

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

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

[5] A. Morrison, M. Diesmann, in: P.b. Graben (Ed.), Maintaining causality in discrete time neuronal network simulations, Berlin, Heidelberg, 2008, pp. 267–278, http://dx.doi.org/10.1007/978-3-540-73159-7_10.

[6] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, Challenges in parallel graph processing, Parallel Process. Lett. 17 (2007) 5–20, http://dx.doi.org/10.1142/S0129626407002843.

[7] A. Ailamaki, D.J. DeWitt, M.D. Hill, D.A. Wood, DBMSs on a modern processor: Where does time go? in: Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999, pp. 266–277, https://dl.acm.org/doi/10.5555/645925.671662.

[8] S. Manegold, P.A. Boncz, M.L. Kersten, Optimizing database architecture for the new bottleneck: memory access, VLDBJ 9 (2000) 231–246, http://dx.doi.org/10.1007/s007780000031.

[9] J. Pronold, J. Jordan, B.J.N. Wylie, I. Kitayama, M. Diesmann, S. Kunkel, Routing brain traffic through the von neumann bottleneck: Parallel sorting and refactoring, Front. Neuroinform. 15 (2022) http://dx.doi.org/10.3389/fninf.2021.785068.

[10] J. Jordan, T. Ippen, M. Helias, I. Kitayama, M. Sato, J. Igarashi, M. Diesmann, S. Kunkel, Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers, Front. Neuroinform. 12 (2018) 2, http://dx.doi.org/10.3389/fninf.2018.00002.

[11] F. Cremonesi, G. Hager, G. Wellein, F. Schürmann, Analytic performance modeling and analysis of detailed neuron simulations, Int. J. High. Perform. Comput. Appl. 34 (2020) 428–449, http://dx.doi.org/10.1177/1094342020912528.

[12] F. Cremonesi, F. Schürmann, Understanding computational costs of cellular-level brain tissue simulations through analytical performance models, Neuroinformatics 18 (2020) 407–428, http://dx.doi.org/10.1007/s12021-019-09451-w.

[13] S. Mittal, A survey of recent prefetching techniques for processor caches, ACM Comput. Surv. 49 (2016) 1–35, http://dx.doi.org/10.1145/2907071.

[14] S. Ainsworth, T.M. Jones, Graph prefetching using data structure knowledge, in: Proceedings of the 2016 International Conference on Supercomputing, ICS '16, Association for Computing Machinery, New York, NY, USA, 2016, http://dx.doi.org/10.1145/2925426.2926254.

[15] S. Ainsworth, T.M. Jones, Software prefetching for indirect memory accesses, in: 2017 IEEE/ACM International Symposium on Code Generation and Optimization, CGO, 2017, pp. 305–317, http://dx.doi.org/10.1109/CGO.2017.7863749.

[16] C. Jonathan, U.F. Minhas, J. Hunter, J. Levandoski, G. Nishanov, Exploiting coroutines to attack the killer nanoseconds, Proc. VLDB Endow. 11 (2018) 1702–1714, http://dx.doi.org/10.14778/3236187.3236216.

[17] G. Psaropoulos, T. Legler, N. May, A. Ailamaki, Interleaving with coroutines: A practical approach for robust index joins, Proc. VLDB Endow. 11 (2017) 230–242, http://dx.doi.org/10.14778/3149193.3149202.

[18] G. Psaropoulos, T. Legler, N. May, A. Ailamaki, Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins, VLDBJ 28 (2019) 451–471, http://dx.doi.org/10.1007/s00778-018-0533-6.

[19] O. Kocberber, B. Falsafi, B. Grot, Asynchronous memory access chaining, Proc. VLDB Endow. 9 (2015) 252–263, http://dx.doi.org/10.14778/2856318.2856321.

[20] S. Chen, A. Ailamaki, P.B. Gibbons, T.C. Mowry, Improving hash join performance through prefetching, ACM Trans. Database Syst. 32 (2007) http://dx.doi.org/10.1145/1272743.1272747, 17–es.

[21] M. Lam, Software pipelining: An effective scheduling technique for VLIW machines, ACM SIGPLAN Not. 23 (1988) 318–328, http://dx.doi.org/10.1145/960116.54022.

[22] V.H. Allan, R.B. Jones, R.M. Lee, S.J. Allan, Software pipelining, ACM Comput. Surv. 27 (1995) 367–432, http://dx.doi.org/10.1145/212094.212131.

[23] H. Watanabe, K.M. Nakagawa, SIMD vectorization for the lennard-jones potential with AVX2 and AVX-512 instructions, Comput. Phys. Comm. 237 (2019) 1–7, http://dx.doi.org/10.1016/j.cpc.2018.10.028.

[24] J. Pronold, J. Jordan, B. Wylie, I. Kitayama, M. Sato, M. Diesmann, S. Kunkel, Code for routing brain traffic through the von Neumann bottleneck: Efficient cache usage in spiking neural network simulation code on general purpose computers, 2021, http://dx.doi.org/10.5281/zenodo.4564078.

[25] M.-O. Gewaltig, M. Diesmann, NEST (NEural simulation tool), Scholarpedia J. 2 (2007) 1430, http://dx.doi.org/10.4249/scholarpedia.1430.

[26] S. Kunkel, Routing brain traffic through the bottlenecks of general purpose computers: Challenges for spiking neural network simulation code, in: 33rd ISC High Performance Conference, Frankfurt, Germany, 2019, pp. 16–20.

[27] OpenM.P. Architecture Review Board, Openmp application program interface, 2008, http://www.openmp.org/mp-documents/spec30.pdf. (Accessed 27 September2016).

[28] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 2.2, Technical Report, Knoxville, Tennessee, 2009, http://www.mpi-forum.org/docs/.

[29] D. Plotnikov, I. Blundell, T. Ippen, J.M. Eppler, B. Rumpe, A. Morrison, Lecture Notes in Informatics (LNI), P-254, Gesellschaft für Informatik e.V. (GI), 2016, pp. 93–108, http://juser.fz-juelich.de/record/826510.

[30] C. Linssen, B. Rumpe, S. Berns, T. Fardet, K. Perun, T. Schulte to Brinke, D. Terhorst, J.M. Eppler, A. Morrison, Nestml 3.1, 2020, http://dx.doi.org/10.5281/zenodo.3697733.

[31] J.M. Eppler, M. Helias, E. Muller, M. Diesmann, M. Gewaltig, PyNEST: a convenient interface to the NEST simulator, Front. Neuroinform. 2 (2009) 12, http://dx.doi.org/10.3389/neuro.11.012.2008.

[32] Y.V. Zaytsev, A. Morrison, CyNEST: a maintainable cython-based interface for the NEST simulator, Front. Neuroinform. 8 (2014) http://dx.doi.org/10.3389/fninf.2014.00023.

[33] S. Rotter, M. Diesmann, Exact digital simulation of time-invariant linear systems with applications to neuronal modeling, Biol. Cybernet. 81 (1999) 381–402, http://dx.doi.org/10.1007/s004220050570.

[34] A. Morrison, M. Diesmann, W. Gerstner, Phenomenological models of synaptic plasticity based on spike-timing, Biol. Cybernet. 98 (2008) 459–478, http://dx.doi.org/10.1007/s00422-008-0233-1.

[35] M. Helias, S. Kunkel, G. Masumoto, J. Igarashi, J.M. Eppler, S. Ishii, T. Fukai, A. Morrison, M. Diesmann, Supercomputers ready for use as discovery machines for neuroscience, Front. Neuroinform. 6 (2012) 26, http://dx.doi.org/10.3389/fninf.2012.00026.

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

[37] S. Kunkel, M. Schmidt, J.M. Eppler, G. Masumoto, J. Igarashi, S. Ishii, T. Fukai, A. Morrison, M. Diesmann, M. Helias, Spiking network simulation code for petascale computers, Front. Neuroinform. 8 (2014) 78, http://dx.doi.org/10.3389/fninf.2014.00078.

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

[39] A. Morrison, A. Aertsen, M. Diesmann, Spike-timing dependent plasticity in balanced random networks, Neural Comput. 19 (2007) 1437–1467, http://dx.doi.org/10.1162/neco.2007.19.6.1437.

[40] S. Kunkel, W. Schenck, The NEST dry-run mode: Efficient dynamic analysis of neuronal network simulation code, Front. Neuroinform. 11 (2017) 40, http://dx.doi.org/10.3389/fninf.2017.00040.

[41] D. Krause, P. Thörnig, JURECA: Modular supercomputer at Jülich supercomputing centre, 4 (2018) A132, http://dx.doi.org/10.17815/jlsrf-4-121-1.

[42] H. Miyazaki, Y. Kusano, N. Shinjou, S. Fumiyoshi, M. Yokokawa, T. Watanabe, Overview of the k computer system, Fujitsu Sci. Tech. J. 48 (2012) 255–265.

[43] S. Lührs, D. Rohe, A. Schnurpfeil, K. Thust, W. Frings, Flexible and generic workflow management, in: Parallel Computing: On the Road To Exascale, in: Advances in parallel computing, vol. 27, IOS Press, Amsterdam, 2016, pp. 431–438, http://dx.doi.org/10.3233/978-1-61499-621-7-431.

[44] J. Kozloski, J. Wagner, An ultrascalable solution to large-scale neural tissue simulation, Front. Neuroinform. 5 (2011) http://dx.doi.org/10.3389/fninf.2011.00015.

[45] J. Igarashi, H. Yamaura, T. Yamazaki, Large-scale simulation of a layered cortical sheet of spiking network model using a tile partitioning method, Front. Neuroinform. 13 (2019) 71, http://dx.doi.org/10.3389/fninf.2019.00071.

[46] C.L. Alappat, J. Hofmann, G. Hager, H. Fehske, A.R. Bishop, G. Wellein, Understanding HPC benchmark performance on intel broadwell and cascade lake processors, in: High Performance Computing, Springer International Publishing, Cham, 2020, pp. 412–433, http://dx.doi.org/10.1007/978-3-030-50743-5_21.

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

[48] J. Albers, J. Pronold, A.C. Kurth, S. Vennemo, K. Haghighi Mood, A. Patronis, D. Terhorst, J. Jordan, S. Kunkel, T. Tetzlaff, M. Diesmann, J. Senk, A modular workflow for performance benchmarking of neuronal network simulations, Front. Neuroinform. 16 (2022) 837549, http://dx.doi.org/10.3389/fninf.2022.837549.

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

[50] J. Stapmanns, J. Hahne, M. Helias, M. Bolten, M. Diesmann, D. Dahmen, Event-based update of synapses in voltage-based learning rules, Front. Neuroinform. 15 (2021) 609147, http://dx.doi.org/10.3389/fninf.2021.609147.

[51] S. Diaz-Pier, M. Naveau, M. Butz-Ostendorf, A. Morrison, Automatic generation of connectivity for large-scale neuronal network models through structural plasticity, Front. Neuroanat. 10 (2016) 57, http://dx.doi.org/10.3389/fnana.2016.00057.

[52] A.C. Kurth, J. Senk, D. Terhorst, J. Finnerty, M. Diesmann, Sub-realtime simulation of a neuronal network of natural density, Neuromorphic Comput. Eng. 2 (2022) 021001, http://dx.doi.org/10.1088/2634-4386/ac55fc.