J. Parallel Distrib. Comput. 120 (2018) 251–266

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

A scalable algorithm for simulating the structural plasticity of the brain

Sebastian Rinke [a,*], Markus Butz-Ostendorf [b], Marc-André Hermanns [c], Mikaël Naveau [d,1], Felix Wolf [a]

[a] Technische Universität Darmstadt, Germany
[b] Biomax Informatics AG, Germany
[c] Jülich Aachen Research Alliance, Section JARA-HPC, Forschungszentrum Jülich, Germany
[d] Simulation Laboratory Neuroscience, Jülich Aachen Research Alliance, Forschungszentrum Jülich, Germany

HIGHLIGHTS

- A scalable algorithm from particle physics can be adapted to solve large-scale problems in neuroscience.
- The approximation underlying the algorithm does not adversely affect the quality of the results.
- The scalable algorithm can simulate structural plasticity in the brain with 10^9 neurons.
- Performance extrapolations suggest that the algorithm could in principle simulate neuron counts as found in the human brain (10^{11}).

ARTICLE INFO

Article history:
Received 24 April 2017
Received in revised form 20 September 2017
Accepted 30 November 2017
Available online 7 December 2017

Keywords:
Brain
Connectome
Dynamics
Simulation
Barnes–Hut
Large-scale

ABSTRACT

The neural network in the brain is not hard-wired. Even in the mature brain, new connections between neurons are formed and existing ones are deleted, which is called structural plasticity. The dynamics of the connectome is key to understanding how learning, memory, and healing after lesions such as stroke work. However, with current experimental techniques even the creation of an exact static connectivity map, which is required for various brain simulations, is very difficult. One alternative is to use network models to simulate the evolution of synapses between neurons based on their specified activity targets. This is particularly useful as experimental measurements of the spiking frequency of neurons are more easily accessible and reliable than biological connectivity data. The Model of Structural Plasticity (MSP) by Butz and van Ooyen is an example of this approach. However, to predict which neurons connect to each other, the current MSP model computes probabilities for all pairs of neurons, resulting in a complexity $O(n^2)$. To enable large-scale simulations with millions of neurons and beyond, this quadratic term is prohibitive. Inspired by hierarchical methods for solving n-body problems in particle physics, we propose a scalable approximation algorithm for MSP that reduces the complexity to $O(n \log^2 n)$ without any notable impact on the quality of the results. We show that an MPI-based parallel implementation of our scalable algorithm can simulate the structural plasticity of up to 10^9 neurons—four orders of magnitude more than the naïve $O(n^2)$ version.

1. Introduction

The brain is not as hard-wired as traditionally thought. Neurons are connected to each other in a dynamically changing biological network of synapses, also known as the connectome. Even in the mature brain, new connections between neurons (i.e., synapses) are continuously created and existing ones are deleted, which can be described as structural plasticity. Studying the dynamics of connectivity in the brain is fundamental to understanding how learning, memory, and healing after lesions in the brain such as strokes work. Unfortunately, accurately observing the connectome and its evolution empirically is very hard. Limiting factors are, for example, the resolution of sensors and restricted access to the brain areas of interest [14]. Thus, even creating an exact connectivity map of a small region of the brain is extremely challenging. However, it is exactly such a connectivity map that is needed as the basis of state-of-the-art brain simulations [2,16].

An alternative to acquiring biological connectivity data is to determine the connections between neurons using a network model.

* Corresponding author.
E-mail addresses: rinke@cs.tu-darmstadt.de (S. Rinke), wolf@cs.tu-darmstadt.de (F. Wolf).
[1] Present affiliation: UNICAEN, CNRS, CHU, UMS3408, GIP CYCERON, France.

https://doi.org/10.1016/j.jpdc.2017.11.019
0743-7315/© 2017 Elsevier Inc. All rights reserved.

For example, when the spiking frequency of a neuron is too low, it starts to form more synapses, with the aim of increasing its electrical activity. Conversely, synapses are deleted when the electrical activity of a participating neuron is too high. One big advantage of the spiking frequency is that it is easier to observe experimentally than the connectome itself. In addition to generating static connectivity maps, such a network model can also help investigate the dynamics of connectivity, such as (i) structural plasticity in a cell-type-dependent manner [13], (ii) the creation of structures due to external stimuli [19], and (iii) functional reorganization and restructuring after a lesion [23,34].

The Model of Structural Plasticity (MSP) by Butz and van Ooyen [9] is a network model with activity-dependent dynamic creation and deletion of synapses. In traditional models, connectivity is fixed while plasticity merely arises from changes in the strength of existing synapses, typically modeled as weight factors. MSP, in contrast, is suitable for simulating the reorganization of the connectome. Instead of representing a synapse by a weight factor, MSP models a synapse as a connection between an axonal "plug" and a dendritic "socket". These synaptic elements grow and shrink independently on each neuron. When an axonal element of one neuron connects to the dendritic element of another neuron, a new synapse is formed. Conversely, when a synaptic element bound in a synapse retracts, the corresponding synapse is removed. The governing idea of the model is that plasticity in cortical networks is driven by the need of individual neurons to homeostatically maintain their average electrical activity. Consequently, neurons form new synaptic elements if their activity is below a desired threshold, and remove elements if it exceeds the threshold. As empirical observation shows, MSP lets networks of neurons robustly grow towards a stable homeostatic equilibrium of activity and connectivity. It was shown that this structural-plasticity rule can account for network rewiring after a partial loss of external input (deafferentation) [9]. The simulation results exhibited strong similarities with biological data from network rewiring in the primary visual cortex after focal retinal lesions [23,34]. To make MSP available to a larger community and combine its capabilities with the features of a state-of-the-art brain simulator, a simplified version of the model was recently integrated [14] into the NEST neural network simulator [16].

In contrast to the original MSP, this simplified version does not consider the different distances between neurons for synapse creation, which makes it less computationally demanding at the expense of accuracy. The largest published structural plasticity simulations of the simplified MSP in NEST contained 10^5 neurons [14]. However, the computational complexity of the original MSP in terms of the number of neurons seriously limits its scalability. To decide which pairs of axonal and dendritic elements will form a synapse, MSP follows a probabilistic approach. It considers all pairs of neurons with a vacant axonal element on one side of the pair and a vacant dendritic element on the other, and calculates the probability of them establishing a connection between them. The shorter their distance, the higher this probability becomes. Given that every neuron creates a certain amount of both axonal and dendritic elements (limited by a constant due to biological restrictions), ultimately all pairs of neurons have to be considered. Thus, the cost grows quadratically ($O(n^2)$) with the number of neurons. However, as soon as we start investigating the connectivity across individual brain regions and the number of neurons involved rises above a hundred thousand, this cost becomes prohibitive. Note that the human brain has about 10^{11} neurons [3]. For this reason, we urgently need a scalable algorithm for MSP.

A similar challenge arises in *n*-body problems, where pairs of bodies have to be considered for force calculations. To improve the scalability of the force calculations, powerful approximation methods have been developed [4,18]. They are based on the observation that particles sufficiently far away from a target particle do not need to be considered individually. It is our goal to leverage their underlying ideas and adapt them to the problem of structural brain plasticity. The most influential algorithms are Barnes–Hut [4] and the Fast Multipole Method [18] (FMM). However, they cannot be applied to our problem directly. They calculate the force exerted on (Barnes–Hut) or the potential of (FMM) each body, whereas we need to select pairs of neurons (bodies) for synapse creation. Moreover, *n*-body simulations continuously subject each particle to force calculations. In the brain, after an initial network creation phase, only a small subset of neurons exhibits vacant axonal elements. Thus, vacant dendrites only have to be found for this smaller subset.

In this paper, we present a scalable approximation method for simulating structural plasticity based on MSP. Our algorithm, an adaptation of Barnes–Hut, reduces the complexity of MSP from $O(n^2)$ to $O(n \log^2 n)$. We further show that the approximations of our method are still precise enough to resemble neural networks created by the original MSP. An MPI-based parallel implementation of our scalable algorithm is the first to enable the model-based creation of neural networks consisting of up to 10^9 neurons—with the potential even for far greater problem sizes.

This work is an extension of our recent previous work [29]. Notable enhancements include the distribution of the octree across the processes, which makes the implementation of our algorithm far more scalable. In comparison to our previous work, the new implementation substantially reduces the memory consumption per process, which allows the simulation size to be increased from 10^7 to 10^9 neurons. Finally, we use performance models to extrapolate the execution times to the full scale of the human brain (10^{11}), showing that simulating structural plasticity at this size is a realistic mid-term target. Overall, this paper makes the following contributions:

- The insight that a scalable algorithm from particle physics can be adapted to solve large-scale problems in neuroscience.
- The actual adaptation, which provides a scalable solution for the simulation of structural plasticity in the brain with a time complexity of $O(n \log^2 n)$ instead of $O(n^2)$.
- The evidence that the scalable algorithm can simulate structural plasticity in the brain with 10^9 neurons.
- The experimental validation that the approximation underlying the algorithm does not adversely affect the quality of the results.

The remainder of this article is organized as follows. After reviewing related work in the next section, we describe the Model of Structural Plasticity in Section 3. In Section 4, we present our scalable algorithm, followed by our scalable implementations in Section 5. Section 6 establishes the algorithm's accuracy and discusses performance results. Finally, Section 7 concludes the paper, highlighting the potential our work offers to future research.

2. Related work

Today's largest brain simulations contain about 10^9 neurons. C2 [2] and NEST [24] are examples of state-of-the-art brain simulators able to reach such a large scale. Both require the user to describe the connectivity between neurons before the simulation starts. During the simulation, the connectivity map remains static. However, the strength of the synapses it defines may change over time. Well-known models that strive to capture structural plasticity include the compensation model by Dammasch et al. [12] and the activity-dependent neurite outgrowth model by van Ooyen et al. [31]. However, while the compensation model ignores topology altogether, van Ooyen's model is too restrictive in that neurons always connect to their direct neighbors before connecting to more distant neurons. These limitations are addressed in the Model of Structural Plasticity [9], the subject of this paper, where synapses are randomly created in a distance-dependent way.

An example for using n-body simulation in brain research has been presented by Prasad et al. [28], where cortical brain regions are represented as particles with mass proportional to the region's volume. Particles attract each other with a force proportional to the strength of the connectivity between the regions they represent. The connectivity between regions was derived from diffusion imaging data from patients with Alzheimer's disease and healthy subjects. Based on these parameters, the authors use a gravitational n-body simulation to obtain a connectivity matrix between brain regions. This matrix is then examined with the goal of distinguishing between patients and healthy subjects. Our work differs from this approach in that we do not perform an n-body simulation. Instead, we adopt ideas of hierarchical n-body methods to reduce the complexity of a structural plasticity model.

The concepts of force between particles and distance-dependent probability for pairs of neurons are similar enough to make the adaptation of n-body methods [4,18] to our problem a realistic option. Another motivation is that the data locality and approximation of advanced n-body methods seem to better mimic biological behavior in the brain. In particular, while "actively" trying to find a vacant dendrite, a neuron's vacant axon has only partial knowledge of other available neurons. Our choice of n-body methods for adaptation is the Barnes–Hut algorithm [4], a decision we will outline in Section 4. An example showing the good scalability of the Barnes–Hut approach for n-body problems is the PEPC code [33], which has already been used to efficiently simulate systems with about $6.4 \cdot 10^{10}$ bodies on 458,752 cores of an IBM Blue Gene/Q system.

3. The MSP model of structural plasticity

This section describes the Model of Structural Plasticity [9], which consists of three basic steps to simulate network connectivity in an activity-dependent fashion: (i) update of electrical activity, (ii) update of synaptic elements, and (iii) update of connectivity.

(1) *Update of electrical activity.* The electrical activity of each neuron is continuously calculated on a millisecond timescale. Intracellular calcium concentration is updated according to the electrical activity. As calcium concentration and average firing rate are linearly proportional, the model uses calcium concentration to guide the growth dynamics of the synaptic elements.

We use a Poisson spiking neuron model to determine when a neuron generates an electrical signal (spike). The firing rate of a neuron decreases exponentially over time by a constant decay factor until it reaches a specified minimum. Spikes received through synapses from neighboring neurons (synaptic input) affect the firing rate. In particular, a spike from an excitatory neuron increases whereas a spike from an inhibitory neuron decreases the firing rate. Based on its firing rate r, a neuron fires in a time step and sends a spike to all its neighbors with probability $r \cdot dt$, where time step size $dt = 1$ ms. After generating a spike, the neuron enters a refractory phase for 4 ms, in which it cannot fire anymore. Based on electrical activity, a neuron's intracellular calcium concentration is calculated as follows:

$$\frac{d\mathrm{Ca}}{dt} = \begin{cases} -\dfrac{\mathrm{Ca}(t)}{\tau} + \beta & \text{if neuron fires} \\ -\dfrac{\mathrm{Ca}(t)}{\tau} & \text{else} \end{cases} \tag{1}$$

τ is the calcium decay constant and β is the calcium intake constant denoting how much calcium is accumulated every time the neuron fires. The calcium concentration is then used to guide the growth of synaptic elements.

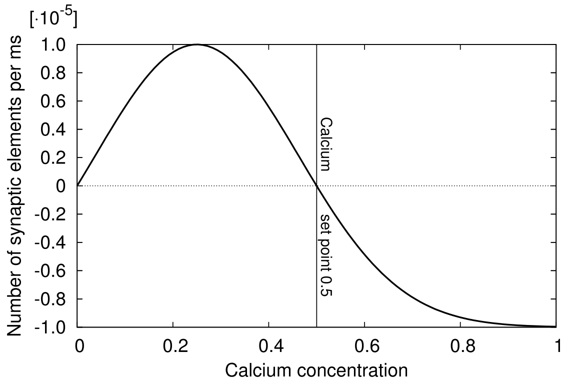


Fig. 1. Homeostatic growth curve which determines the creation and deletion of synaptic elements based on a neuron's calcium concentration. The curve reflects the parameters in our simulations. The desired calcium concentration (set point) is 0.5. A neuron grows synaptic elements until the set point is reached. When calcium concentration exceeds the set point, synaptic elements are deleted. Creation and deletion is a continuous process, where a neuron's number of synaptic elements is represented by a real number x. The actual number of synaptic elements available is the greatest integer $\lfloor x \rfloor$ that is less than or equal to x.

(2) *Update of synaptic elements.* The detailed morphology of synaptic elements is abstracted and represented only by the number of synaptic contacts on axons (axonal boutons) and dendrites (dendritic spines). We call these contacts collectively synaptic elements. A homeostatic rule determines for each neuron when axonal and dendritic synaptic elements are created or deleted. If the calcium concentration is below the desired set point, they are created. If it is above the set point, they are deleted. Creation or deletion proceeds until the desired level of electrical activity has been reached. The homeostatic rule is described through a Gaussian-shaped growth curve. Fig. 1 depicts the growth curve in our simulations.

(3) *Update of connectivity.* At discrete points in time, existing synapses are deleted and new synapses are formed, depending on the current number of synaptic elements. A synapse is deleted after either the participating axonal or dendritic element has been removed during the update of synaptic elements. If a synapse is removed, a synaptic element that was previously bound in this synapse becomes vacant again. If a *source* neuron with a vacant axonal element is assumed, then the *target* neuron which the axonal element will try to connect to is determined by considering every neuron as a potential target and calculating the probability of establishing a connection. The probability depends on the distance between source and target, and the number of unbound dendritic elements available at the target. Given the three-dimensional position (x, y, z) of a source neuron j and a target neuron candidate i, we can evaluate a three-dimensional Gaussian-shaped kernel:

$$K_{ij} = \exp\left(-\frac{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}{\sigma^2}\right) \tag{2}$$

σ is a simulation parameter that controls the width of the curve. To avoid creating autapses (i.e., source neuron connecting to itself), we set $K_{ij} = 0$ for $i = j$. K_{ij} is then weighted by a factor w_i denoting the number of vacant dendritic elements at the target neuron. This yields $|Neurons|$ (number of neurons) values of the form $\{w_i \cdot K_{ij} \mid j \text{ is source neuron} \wedge i \in Neurons\}$ for the source neuron j. The sum of the elements in this set is not necessarily 1. To construct probabilities, all the elements are finally scaled so that their sum equals 1. Finally, a random number in the interval [0, 1] selects the target neuron out of all candidates. Constructing the probabilities in this way ensures that the closer the two neurons are, and the more dendrites the target neuron candidate offers,

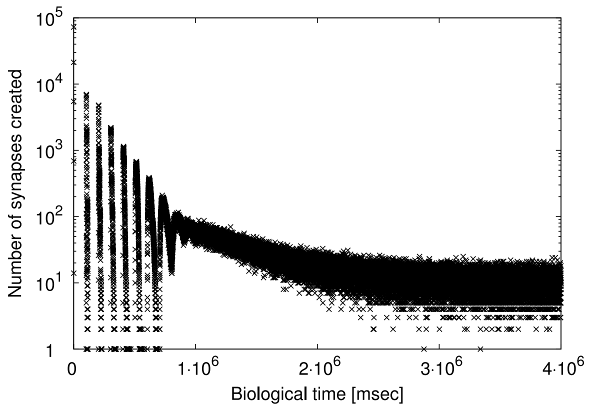


Fig. 2. The number of synapses MSP creates over time for 10^5 neurons. To save simulation time, every neuron is initialized with one vacant axonal element and two vacant dendritic elements. Otherwise, we would have to wait until synaptic elements have grown to form synapses. At the beginning, no synapses exist and the network is empty. Neurons start forming synapses to reach their desired level of electrical activity. From $3 \cdot 10^6$ ms on, the neurons enter equilibrium and only a few synapses are being formed. Note that some synapses are also deleted as the simulation progresses, which is not shown here.

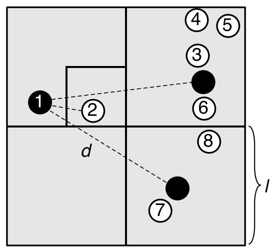


Fig. 3. A two-dimensional example of grouping neurons. Neurons 3–6 and 7–8 are in the same respective squares. They form two groups. The length of their squares is denoted by *l*, which can be seen as the spatial extent of the group. The two groups are represented by a virtual neuron (black solid circle). The source neuron is neuron 1. Instead of considering all individual neurons as target neuron candidates, grouping reduces the work at this stage to considering only two virtual neurons and two normal neurons (neuron 2 and the source neuron itself). To avoid autapses, the source neuron's probability is set to zero. A dashed line depicts the distance *d* from the source neuron to the other neurons under consideration, which are neuron 2 and the two virtual neurons.

the higher the probability for the target neuron candidate to be chosen for synapse formation is. During the connectivity update, every vacant axonal element selects a target neuron, as described above. Note that multiple axonal elements may try to connect to the same target neuron. If the neuron does not have sufficient vacant dendritic elements, some of the axonal elements are rejected. In this case, they try to find another target neuron during the next connectivity update. The extent of changes in the neural network is determined by the update of synaptic elements. As these elements grow rather slowly, the connectivity update occurs only infrequently.

Performance considerations. Steps (1) and (2) consider every neuron and thus run in $O(n)$. In step (3), every neuron is examined to decide which of its synapses have to be deleted. As the number of synapses per neuron is limited through a constant (due to biological reasons), synapse deletion runs in $O(n)$. However, for synapse creation in step (3), probabilities are calculated for all pairs of neurons, which takes $O(n^2)$. This worst case occurs in the early phase of network creation where no synapses exist yet and all neurons still have vacant axonal elements available. Fig. 2 depicts this situation. Note that the number of synapses created during the first $5 \cdot 10^5$ ms is in the order of the number of neurons and much higher than during the remainder of the simulation. The peaks in the plot appear because we apply the same growth curve to all synaptic elements. That is, after all axonal elements have been bound in synapses during connectivity updates, new axonal elements grow and become available for all neurons at about the same time. Especially if major structural changes occur, that is, at the beginning or after introducing lesions, synapse creation prevents MSP from being scaled to large neural networks. Thus, reducing the complexity of MSP's synapse creation is a prerequisite for simulating larger portions of the brain.

4. A scalable algorithm for MSP

We shall now describe our scalable approximation algorithm for MSP, an adaptation of the Barnes–Hut n-body method to our specific problem. Although the $O(n)$ complexity of FMM is lower than the $O(n \log n)$ complexity of Barnes–Hut, we chose Barnes–Hut because FMM is harder to tailor to our needs and to implement. One noteworthy difference is that FMM groups not only source but also target particles and calculates interactions between groups. However, synapse creation is initiated by an individual source neuron with a vacant axon. When forming a group of source neurons, it must be ensured that this individual source neuron can use the group's probabilities of establishing connections with other (groups of) target neuron candidates. Recalculating these probabilities when the source neuron is finally processed, or storing them for re-use, could harm the scalability of the FMM. On the other hand, the Barnes–Hut method groups only target neurons, which more closely matches our problem. Moreover, it was unclear whether adapting hierarchical n-body methods for MSP would provide approximation techniques which are accurate enough to resemble exact networks of MSP. The FMM appeared too complex for this exploration. Finally, it was unsure whether the superior scalability and thus the additional complexity of FMM may ultimately be needed, given that the number of neurons in the human brain (10^{11}) is the largest problem that our algorithm will ever be required to handle efficiently. For these reasons, we believe that following the design philosophy of the simpler Barnes–Hut algorithm is a reasonable choice.

Calculating the probabilities for creating synapses is the most time consuming part of MSP. Similarly to Barnes–Hut, we combine distant neurons into groups whenever possible instead of considering them individually. Neurons in the same group have a similar distance to the source neuron. We represent a group of neurons through a virtual neuron whose position is a linear combination of the positions of the group members. Weight factors position the virtual neuron closer to neurons with many vacant dendritic elements. The number of vacant dendritic elements of the virtual neuron is the sum of vacant dendritic elements present in the entire group. This approach resembles the concept of the center of mass in gravitational versions of the Barnes–Hut algorithm. Only neurons close to the source neuron are considered individually because they differ more in their relative distance to the source neuron. Otherwise, the probability error, caused by using the averaged position of the virtual neuron, could become too large. Fig. 3 shows an example. Neuron 2 is too close to the source neuron and hence not considered as part of a group. Below, we explain the three steps of our algorithm: (i) tree construction, (ii) tree update, and (iii) target neuron selection.

Tree construction. Similarly to the Barnes–Hut algorithm, we start by forming a tree of neuron groups. However, compared to particles in an n-body simulation, our neurons do not move. Thus, the tree is created only once at the start of the simulation. The tree construction proceeds as follows: Given a cube that contains all

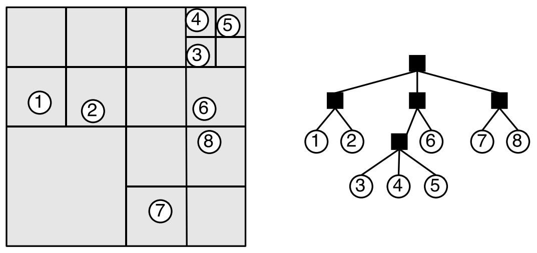


Fig. 4. A two-dimensional tree-construction example. On the left, we see the subdivision of the simulation domain. On the right, we see the resulting tree. Neurons are numbered from 1 to 8. The inner nodes in the tree are virtual neurons, while the leaves are real neurons belonging to their subdomains.

neurons in our simulation domain, we create a spatial tree representation of the domain step-by-step. If the domain contains more than one neuron, we subdivide it into eight cubical subdomains of the same size. Each of the eight subdomains is then recursively subdivided if it contains more than one neuron. The recursion ends when every subdomain contains at most one neuron. As a result, we obtain an octree (i.e., a tree with at most eight children per node) with the root representing the cube containing all neurons. Its children are the eight subdomains that it was divided into and so on. Every leaf in the tree represents a single neuron, every inner node represents a virtual neuron in its subdomain. This hierarchy of subdomains defines the groups of neurons that we need. For ease of illustration, we have used two-dimensional examples in our figures. In comparison to three dimensions, we have subdivided a domain into four squares and the resulting tree is a quadtree (i.e., a tree with at most four children per node). Fig. 4 shows the final subdivision of such a domain and the resulting tree.

The depth of the tree depends on the distribution of the neurons. The closer neurons are located to each other, the more domain subdivisions are required and the tree depth could in principle grow indefinitely [1]. Fortunately, biological constraints ensure that neurons are not positioned at purely arbitrary distances from each other within the brain. For example, the diameter of the soma (cell body) of neurons determines the minimum distance between neurons. In general, neuron densities vary depending on the brain region and the cortical layer. In this work, we consider a high density of neurons such as in layer 5A of the rat cortex [25], where the average distance between neurons is about 26 µm. Let us analyze how this translates into the tree depth. We assume that a cube with edge length L contains all neurons of the simulation. Let the smallest distance between any two neurons be s. How often must the cube be recursively subdivided until all neurons belong to separate subdomains? The smallest subdomain containing two neurons with minimum distance s has edge length $s/\sqrt{3}$. This follows from calculating the distance between the closest neurons by

$$\sqrt{\left(s/\sqrt{3}\right)^2 + \left(s/\sqrt{3}\right)^2 + \left(s/\sqrt{3}\right)^2} = \sqrt{3} \cdot s/\sqrt{3} = s. \tag{3}$$

Now, we can rephrase the question: How often must the cube be recursively halved until subdomains with edge length less than $s/\sqrt{3}$ are obtained? To answer this question, we need to find the smallest k which satisfies $L/2^k < s/\sqrt{3}$. Solving for k yields $L\sqrt{3}/s < 2^k$, and finally $k = \left\lceil \log_2 \frac{L\sqrt{3}}{s} \right\rceil$. Note that k grows when the length L of the cube increases, or the minimum distance s between neurons decreases. For example, the average length of the human brain is about 20 cm. We believe that the average distance of 26 µm between densely packed neurons in the rat cortex is also valid for the human brain. Using these biological constraints and assigning $L = 20$ cm and $s = 26$ µm yields:

$$k = \left\lceil \log_2 \frac{20 \cdot 10^{-2} \cdot \sqrt{3}}{26 \cdot 10^{-6}} \right\rceil \leq 14 \tag{4}$$

The result tells us that a cube containing all neurons of the human brain is at most 14 times recursively subdivided until every neuron belongs to a separate subdomain. Consequently, the maximum depth of the tree would be 14. In practice, the maximum length of any simulation domain will most likely not exceed the length of the human brain. Moreover, the smallest distance between neurons in the human brain is not expected to be much smaller than $s = 10$ µm [11], as the lower bound is the diameter of the neuron's cell body. For $s = 10$ µm, tree depth $k \leq 16$. From these observations follows that $\frac{L\sqrt{3}}{s} \approx \frac{20 \cdot 10^{-2} \cdot \sqrt{3}}{10 \cdot 10^{-6}} \approx 34,642 = O(n)$. Finally, we can provide an asymptotic upper bound for the tree depth k:

$$k = \left\lceil \log_2 \frac{L\sqrt{3}}{s} \right\rceil \leq \log_2 O(n) \leq O(\log n) \tag{5}$$

For this reason, we conclude that the depth of the tree is $O(\log n)$. As in Barnes–Hut, we create the tree by successively inserting all neurons into the tree. Since our tree is of depth $O(\log n)$, tree creation takes $O(n \log n)$.

Tree update. Neurons do not move, but the number of their vacant dendritic elements is subject to change. For this reason, the tree has to be updated before creating new synapses. For every leaf (i.e., real neuron), we store the current number of vacant dendritic elements. For every inner node (virtual neuron), we not only update the number of vacant elements but also the position of its virtual neuron. The number of vacant dendritic elements is simply the sum of those available on its (direct) children. Let v be a virtual neuron. Then the number of vacant dendritic elements is $D_v = \sum_{i \in Children} D_i$. The position is a linear combination of the positions of its children and their vacant dendritic elements. After updating its vacant element count, the x-coordinate of the virtual neuron v is calculated as $x_v = 1/D_v \sum_{i \in Children} x_i D_i$. The y- and z-coordinates are obtained in a similar way. We update the information in the tree bottom-up from the leaves to the root via postorder traversal, which takes time $O(n)$.

Target neuron selection. After a tree update, we form new synapses by finding a target neuron for every vacant axonal element. To minimize the number of probability calculations, we already decide at the coarser level of neuron groups which neurons the source neuron will not connect to and which we therefore do not need to consider any further. If the source neuron decides to connect to a virtual neuron, we unfold the group it represents. This makes all its (virtual) constituent neurons visible, from which we again select one. Every group selection decreases the number of target neuron candidates. The recursion ends once a single real target neuron has been selected. Selecting a target neuron for a given source neuron means choosing a path from the root to a leaf. To decide which subdomains we consider as a whole on the path down the tree, we use the acceptance criterion (AC) of the Barnes–Hut method. Let d be the distance from the source neuron to the virtual neuron. Let l denote the length of the virtual neuron's subdomain. If $l/d < \theta$, we calculate a single connection probability for the entire subdomain. Otherwise, we unfold it and recursively apply the AC to its constituent subdomains. Here, $\theta \in [0, 1]$ is a configurable precision parameter that ensures that subdomains for which we calculate probabilities are distant enough from the source neuron in relation to their size. Note that a subdomain can be unfolded for two reasons, either because it has been selected to form a connection or because it does not satisfy the AC.

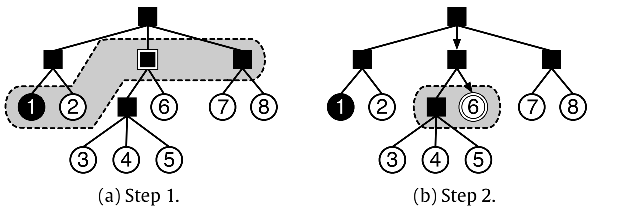


Fig. 5. A two-dimensional example of target neuron selection in two steps (a) and (b). Neuron 1 is the source neuron. Areas shaded in gray identify the set of (virtual) neurons from which one must be selected. The selection is framed. In (b), arrows indicate the path from the root to the target neuron 6.

The reason for using the Barnes–Hut AC is as follows. Neurons in a subdomain that satisfies the criterion have similar distances to the source neuron. Thus, the distance of their virtual neuron seems to properly approximate their individual distances to the source neuron. On the other hand, the neurons in a subdomain that does not meet the AC may show greater relative differences in their distance to the source neuron. Consequently, their probabilities differ more and a single virtual neuron would not properly represent all neurons in the group.

Fig. 5 continues our previous example. Starting from the root, the AC is applied. Because the root does not satisfy $l/d < \theta$ for its domain, we need to unfold it. The same is true for its first child. At this point, the recursion stops because both its children are leaves. The other two children of the root satisfy the AC and remain closed for now. The gray shaded area marks the first set of nodes for which we now calculate probabilities, as described in Section 3. The difference to the original MSP is that we only consider a subset of neurons, with some of them being virtual. Based on their probabilities, we select one neuron from this subset. It is the second child of the root (framed square), a virtual neuron. In the next step (Fig. 5b), we unfold the subdomain of the selected neuron and apply the AC to one virtual neuron and neuron 6. Both are accepted (gray area). Note that a real neuron is trivially accepted since unfolding is not possible. We calculate the connection probabilities and select one neuron, which is neuron 6. It is a real neuron and thus the target neuron for source neuron 1. Now, the selection terminates at this point.

The complexity of the target neuron selection depends on the number of nodes to consider. With $\theta = 0$, all subdomains are unfolded and we consider every neuron for a given source neuron. That is, the algorithm is exact and behaves as the original MSP with complexity $O(n^2)$. For $\theta > 0$, we start considering groups of neurons. Here, the complexity depends on the depth of the tree, which we assume to be $O(\log n)$, as previously stated. When randomly selecting nodes on the path down from the root to the target neuron, after every selection the depth of the remaining subtree is reduced in the worst case by one only. That is, we have to perform $O(\log n)$ steps (depth of the tree) until we find a target neuron. To determine the complexity of each step, we follow the argument of Barnes and Hut for homogeneously distributed particles [4]. In particular, when increasing the number of neurons, the new additional subdomains not containing the source neuron incur a certain amount of extra probability calculations. This amount depends on θ but not on the number of neurons. Consequently, increasing the number of neurons by a constant factor only increases the number of probability calculations by an additive constant (for θ). That is, the complexity of each step is $O(\log n)$. Therefore, it takes time proportional to $O(\log^2 n)$ to find a target neuron for one source neuron. Under the biologically motivated assumption that the tree is of depth $O(\log n)$, the complexity of finding a target neuron for every neuron in one connectivity update is therefore $O(n \log^2 n)$.

4.1. Error analysis

In Fig. 5, the chain of subdomains that contain the source neuron (root and its first child) is completely unfolded until the source neuron is encountered. This is very helpful for avoiding autapses, as is usually desirable in brain simulations. To avoid autapses, MSP sets the probability of a neuron to connect to itself to zero. However, during the recursive descent, every virtual neuron's probability, which depends on the position and number of dendrites, is based on all neurons in its subdomain. As we have no information about whether or not the source neuron is included in a particular virtual neuron's domain, we cannot exclude the source neuron's contribution from the virtual neuron. Consequently, the probability assigned to a virtual neuron whose group contains the source neuron is too high. This is because the zero probability of the source neuron applies only if it is considered directly and not through a virtual neuron. As a side effect, the selection also becomes biased towards other neurons in the same domain through the inflated probability of the source neuron. To eliminate this bias, we define the AC in such a way that the source neuron can never become part of a virtual neuron's domain during probability calculations. We accomplish this by setting $\theta \le 1/\sqrt{3}$.

According to the AC, every subdomain with $l/d \ge \theta$ is unfolded, with d again being the distance from the source neuron and l the edge length of the subdomain. The ratio l/d decreases with increasing distance d between the source neuron and the subdomain's virtual neuron. This is also true for subdomains of different sizes l that contain the source neuron. For those, $d = \sqrt{l^2 + l^2 + l^2} = l\sqrt{3}$ is the greatest possible distance between the source neuron and the virtual neuron (in three dimensions). After substituting $l\sqrt{3}$ for d, $l/d \ge \frac{l}{l\sqrt{3}} = 1/\sqrt{3}$ becomes the smallest possible ratio between l and d. As a result, setting $\theta \le 1/\sqrt{3}$ and defining the AC as $l/d < \theta$ unfolds all subdomains containing the source neuron. Note that $1/\sqrt{3} > 0.5$. That is, setting $\theta \le 0.5$ is a practical usage guideline for our algorithm. To achieve the same behavior in two dimensions, we need to set $\theta \le 1/\sqrt{2}$.

Let us now discuss how a vacant axon of the source neuron selects a target neuron for synapse creation in the tree. We run a multistage probability experiment in which, starting from the root, virtual neurons are randomly selected until a real neuron is chosen. Intuitively, this corresponds to following a path from the root to a leaf as indicated by arrows in Fig. 5b. The sample space S (set of all possible outcomes) of the experiment comprises all possible paths from the root to the leaves (real neurons), one path per neuron. Tree nodes which do not satisfy the acceptance criterion AC for the source neuron are unfolded and thus do not occur on the paths. Since the AC depends on the source neuron's position, every source neuron has its own sample space.

The tree diagram in Fig. 6 illustrates the probability experiment with multiple stages for our example. Neuron 1 is the source neuron whose vacant axon tries to find a target neuron. As can be seen, the diagram omits the virtual parent neuron of the neurons 1 and 2 because it does not satisfy the AC. The corresponding sample space is $S = \{1, 2, AC3, AC4, AC5, A6, B7, B8\}$, where each element is the concatenation of the nodes along the path. For example, $AC4$ is the path between the nodes A-C-4. Given that a path is selected only when all nodes on the path have been selected successively, the probability of choosing path A-C-4 (event $AC4$) in the example is $P(AC4) = P(A \cap C \cap 4)$. Note that a node's likelihood of being selected in a stage depends on the nodes selected in previous stages. In our example, node 4 can only be selected in stage three if nodes A and C were selected in stages one to two. Hence, $P(A \cap C \cap 4) = P(A) \cdot P(C \mid A) \cdot P(4 \mid A \cap C)$.

In general, before a target neuron t_k at depth k in the tree is selected for synapse creation, all virtual neurons $\{v_i \mid 0 \le i < k \wedge v_i \text{ satisfies AC}\}$ on the path to the target neuron are selected.

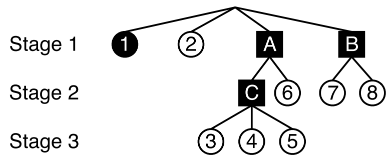


Fig. 6. Tree diagram of the multistage probability experiment for neuron 1. Real neurons are numbers, virtual neurons are letters. A vacant axon of source neuron 1 tries to find a target neuron for synapse creation. In every stage, a (virtual) neuron is selected. The experiment terminates once a real neuron is chosen. The children of the node selected in the current stage are the set of nodes to chose from in the next stage.

```
 1: Create empty network without synapses                      ▷ $O(n)$
 2: Initialize number synaptic elements per neuron             ▷ $O(n)$
 3: Construct tree from domain                                 ▷ $O(n \log n)$
 4: while Desired avg. calcium concentration not reached do
 5:     UpdateElectricalActivity                               ▷ $O(n)$
 6:     UpdateSynapticElements                                 ▷ $O(n)$
 7:     if Connectivity time step completed then
 8:         UpdateConnectivity {                               ▷ $O(n \log^2 n)$
 9:             Delete synapses & update network               ▷ $O(n)$
10:             Create synapses {
11:                 Update tree                                ▷ $O(n)$
12:                 Find target neuron for every
13:                   vacant axonal element                    ▷ $O(n \log^2 n)$
14:                 Update network                             ▷ $O(n)$
15:             }
16:         }
17:     end if
18: end while
```

Fig. 7. Sequential scalable algorithm: the simulation flow of MSP when using our hierarchical algorithm. The runtime complexity of each step is shown on the right. Simulation parameters appear in *italics*. n denotes the number of neurons. The simulation terminates when the neurons reach the *desired average calcium concentration*. The connectivity update is only executed when a *connectivity time step* is completed.

This path is of length k. The probability of selecting t_k depends on the depth k, the precision parameter θ, and t_k's number of vacant dendrites. While depth and precision are constant, the number of vacant dendrites changes in the update of synaptic elements and thus t_k's chance of being selected for synapse creation, too. Based on the concatenated notation of the sample space above, the probability of connecting to t_k is

$$\begin{aligned} P(v_0 \dots v_{k-1} t_k) = P(v_0) \cdot P(v_1 \mid v_0) \cdot P(v_2 \mid v_0 \cap v_1) \cdot \dots \\ \cdot P(t_k \mid v_0 \cap \dots \cap v_{k-1}). \end{aligned} \quad (6)$$

With $\theta = 0$, our approximation method uses the same probabilities as the exact MSP for selecting a target neuron. In particular, the multistage probability experiment shrinks to one stage where probabilities are calculated for all neurons directly without virtual neurons in between. Here, the error of our approximation becomes zero, while the complexity becomes quadratic. Of course, an error analysis is supposed to indicate the extent to which the approximated probabilities differ from the exact MSP. However, it is difficult to determine analytically which effect the error has on the actual structure of the resulting neural network. Hence, we run simulations with different precision parameters and analyze their effect on the resulting networks in Section 6.

4.2. Summary

The execution flow of our scalable MSP algorithm is depicted in Fig. 7. We call this version *sequential scalable algorithm*. The steps within the while loop are executed in every iteration. The algorithm terminates when the neurons reach the desired average level of calcium concentration (line 4). Note that termination does not depend on changes in the structure of the neural network, as this is the result that we are investigating. Moreover, depending on simulation parameters such as calcium increase when a neuron fires, growth curve of synaptic elements, and time between connectivity updates, network dynamics might not even reach equilibrium [14]. During the update of electrical activity (line 5), three steps are performed for every neuron: (i) receive spikes (electrical signals) from other neurons connected to it through synapses, (ii) calculate electrical activity and calcium concentration based on the spikes received, and (iii) determine if neuron fires a spike by considering its updated electrical activity and send the spike to its directly connected neighbors. Spikes generated in the current iteration are received by the neurons in the next iteration. During the update of synaptic elements (line 6), every neuron creates or deletes synaptic elements based on the updated calcium concentration. As can be seen, the connectivity update (line 8), which depends on the number of synaptic elements, has the greatest complexity. Given that synaptic elements grow much slower compared to the frequency at which neurons fire, the connectivity update is run only when a connectivity time step is completed (line 7) and thus not in every iteration. Note that this parameter should not be too large as this might lead to high oscillations in connectivity and thus electrical activity of the neurons [14]. However, as our experimental results in Section 6 show, even a single connectivity update using the exact MSP takes as long as about 40 min for 10^6 neurons, which makes it the clear bottleneck. This is why we must update the connectivity more quickly. Our algorithm brings the complexity from $O(n^2)$ down to $O(n \log^2 n)$, which now becomes the overall computational complexity of the approximated MSP.

5. Implementations of the scalable algorithm

In this section, we present two MPI-based parallel implementations of the scalable algorithm. Our first approach enabled us to simulate neuron counts which are already two orders of magnitude larger compared to the largest possible simulations of the original MSP (see Section 6). However, the scalability of this simple approach is limited because of the high memory consumption per process. By eliminating this bottleneck, our second implementation is a truly scalable implementation of our scalable approximation algorithm of MSP.

5.1. Simple approach

In our first approach, every process simulates n/p neurons, where p is the total number of processes. Although distributing the work of the neurons over all processes is easily achieved, an efficient distribution of the tree is more complicated (see Section 5.2). Hence, our simple implementation stores the complete tree and thus all neurons on every process. This simple approach enabled us already to simulate 10^7 neurons with our approximation algorithm of MSP. So far, the largest published simulations of a simplified version of MSP contained 10^5 neurons [14] (see Section 1). Fig. 8 lists the steps performed by every process with their complexity. We call this version *parallel scalable algorithm with replicated tree*. In the following, we focus on those steps which require a more detailed description. Every process maintains a subgraph of the entire neural network with the incoming and outgoing edges (synapses) of its own neurons. Our corresponding graph data structure can be initialized to the empty network in constant time $O(1)$ as a graph node is only added when it is required for creating an edge (line 1). Line 3 constructs the tree, which contains all neurons. Similarly to the sequential scalable algorithm (Fig. 7), every process executes the update of electrical activity (line 5) and the update of synaptic elements (line 6) for all its

```
1: Create empty network without synapses                ▹ $O(1)$
2: Initialize number synaptic elements                  ▹ $O(n/p)$
3: Construct tree with all neurons                      ▹ $O(n \log n)$
4: while Desired avg. calcium concentration not reached do
5:     UpdateElectricalActivity                         ▹ $O(n/p)$
6:     UpdateSynapticElements                           ▹ $O(n/p)$
7:     if Connectivity time step completed then
8:         UpdateConnectivity {                         ▹ $O(n/p \log^2 n + n)$
9:             Delete synapses & update network         ▹ $O(n/p)$
10:            Create synapses {
11:                Gather all neurons                   ▹ $O(n)$
12:                Update tree of all neurons           ▹ $O(n)$
13:                Find target neurons for
14:                  vacant axonal elements             ▹ $O(n/p \log^2 n)$
15:                Update network                       ▹ $O(n/p)$
16:            }
17:        }
18:    end if
19: end while
```

Fig. 8. Parallel scalable algorithm with replicated tree: the simulation flow of MSP for each process when using our hierarchical algorithm with the replicated tree. The time complexity of each step is shown on the right. Simulation parameters appear in *italics*. n denotes the number of neurons. The simulation terminates when the neurons reach the *desired average calcium concentration*. The connectivity update is only executed when a *connectivity time step* is completed. p denotes the number of processes. It is explicitly stated when a step is performed for all neurons of the simulation. Otherwise, the step processes own neurons only.

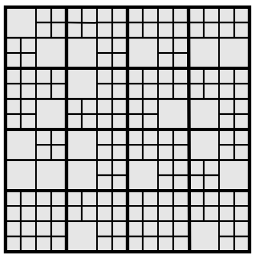


Fig. 9. Two-dimensional example of a domain decomposition into blocks and its subdivision obtained during tree construction. The blocks of the domain decomposition (regular coarse-grained grid of thick lines) are aligned with the subdivisions of the tree construction.

neurons. However, while updating synaptic elements does not require interprocess communication, processes need to exchange spikes for updating the electrical activity because a neuron which receives spikes is not necessarily located on the same process as its firing neighbors. Before a process can update the positions and the number of vacant dendrites of all (virtual) neurons in the tree (line 12), it needs to gather this information about the neurons belonging to the other processes (line 11). The time complexity of the initialization is $O(n \log n)$, which is done once before the simulation starts. One iteration of a complete simulation step takes time $O(n/p \log^2 n + n)$. In practice, this is the dominating term as a typical simulation executes thousands of iterations. Given that every process stores the complete tree and thus all neurons, the space complexity per process is $O(n)$. This clearly limits the total number of neurons to the memory available per compute node. To address this limitation, the tree needs to be distributed over all processes so that every process stores only a portion of the neurons.

5.2. Distribution of the tree

Let us now present our implementation with a distributed tree. The main components of the implementation are: (i) domain decomposition, (ii) assignment of the decomposed domain to the processes, (iii) the distributed tree, and (iv) access to remote tree nodes.

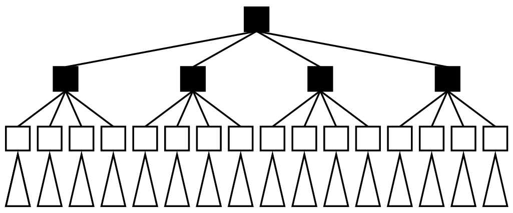


Fig. 10. Tree corresponding to the subdivision of the domain in Fig. 9. Tree nodes are only depicted up to the level of subdomains which correspond to the regular coarse-grained grid of blocks of the domain decomposition in Fig. 9. Every triangle denotes the subtree below the tree node (branch node) to which it is attached. Branch nodes are drawn as empty squares.

5.2.1. Domain decomposition

With the aim of distributing the neurons of the simulation domain over all processes, we decompose the domain by recursively dividing it into subdomains (blocks). For a two-dimensional domain, the blocks are squares, whereas blocks are cubes for three dimensions. These blocks are then assigned to the processes. All neurons inside the same block are stored on and simulated by the same process. Note that this decomposition is not the same subdivision as obtained during the tree construction phase of the scalable algorithm (Section 4). However, the resulting blocks are aligned with the boundaries of the subdomains created during the tree construction. Fig. 9 depicts a two-dimensional example of a simulation domain's decomposition and its subdivision obtained during the tree construction. The decomposition is a regular subdivision into blocks of the same size, whereas the tree construction yields subdomains of different sizes, due to differing distances between neurons. Note that the blocks of the decomposition resemble a grid on top of the subdomains of the tree. To create a grid which is aligned with the tree structure, we decompose the domain into a power of eight blocks for three dimensions (power of four for two dimensions). This ensures that by assigning a block to a process, this process contains all the neurons of the corresponding subdomain in the scalable algorithm's tree. Fig. 10 illustrates the tree created for the example (example domain in Fig. 9). The tree nodes are only depicted up to the level of subdomains which correspond to the blocks of the domain decomposition. The Barnes–Hut literature [32] calls the tree nodes at this level *branch nodes*. A branch node is a node whose children are completely available on the same process. The role of the branch nodes is discussed in more detail below.

Our domain decomposition exploits that, although organized in layers, neurons are relatively homogeneously distributed in the brain. This ensures that every block contains about the same number of neurons. With the goal of equally distributing neurons and thus the work over processes, blocks are assigned to processes. Nevertheless, block sizes and thus the smallest unit of work which can be assigned could be too large. For example, decomposing the domain into few large blocks could lead to load imbalance bottlenecks between processes whose number of blocks differs by only one. Reducing the block size and thus increasing the number of blocks can help to achieve more balanced load between processes. Although simple, our domain decomposition is powerful enough to enable simulations of 10^9 neurons, as we will show in Section 6. The Barnes–Hut literature proposes further load-balancing schemes for particle simulations [15,17,32] which could also be adapted to our brain simulation.

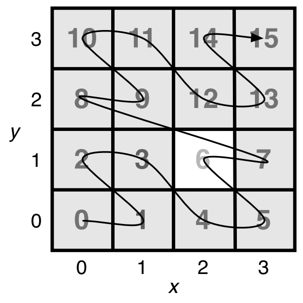


Fig. 11. Morton curve which passes through the blocks of the domain decomposition. A number inside every block denotes the block's position on the Morton curve. Block six is marked for explanations in the text. The numbers along the edges of the domain represent x- and y-coordinates of the blocks. The curve starts at block $(0, 0)$ and ends at block $(3, 3)$.

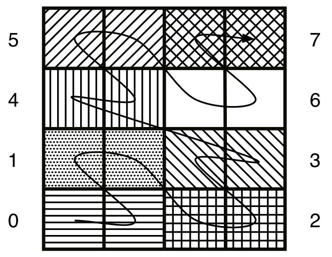


Fig. 12. Assignment of the blocks of the domain decomposition in Fig. 9 to eight processes. Each pattern type marks the blocks of one process. The number next to each pattern refers to the rank of the process which owns the blocks with this pattern. Neighboring process ranks are assigned neighboring blocks on the Morton curve.

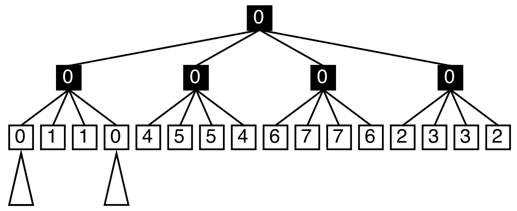


Fig. 13. Partial tree of process 0. Empty squares denote branch nodes. Every triangle is the subtree with all (virtual) neurons below the corresponding branch node. Numbers denote the process ranks of the owners of the tree nodes.

5.2.2. Assignment of the blocks of the decomposed domain

After decomposing the domain into blocks, we assign a set of blocks to every process. This assignment is subject to equally distributing the number of blocks over the processes while maintaining spatial locality per process. The equal distribution accounts for balancing the work between processes. The spatial locality ensures that the blocks of each process are located as close as possible to each other in the simulation domain. Locality is motivated by the scalable algorithm's acceptance criterion, according to which a source neuron calculates probabilities for all target neuron candidates in its close neighborhood. Neurons outside of this neighborhood are grouped and groups are approximated through virtual neurons, which reduces the number of calculations for distant neurons. Hence, assigning neighboring blocks to the same process minimizes the number of costly requests to other processes for retrieving the position and the number of dendrites of non-local (virtual) neurons.

To determine which blocks are close to each other, we "draw" a Morton curve [27] through the domain. Neighboring blocks on this curve are often neighbors in the simulation domain, as well. Fig. 11 continues the two-dimensional example and shows the Morton curve through the blocks of the domain decomposition. The Morton curve is a space-filling curve, which provides us a mapping from a block's three-dimensional (two-dimensional) position in the domain to its one-dimensional position on the curve. The corresponding mapping function achieves this by interleaving the binary coordinate values of the block. For example, let us assume that the three-dimensional position of a block in binary notation has the form $(x_2x_1x_0, y_2y_1y_0, z_2z_1z_0)$, where subscripts denote bit positions. The mapping position on the Morton curve is $(z_2y_2x_2z_1y_1x_1z_0y_0x_0)$. The two-dimensional position $(2, 1)$ of the block marked in Fig. 11 is in binary notation $(x_1x_0, y_1y_0) = (10, 01)$. The Morton-curve mapping yields $(y_1x_1y_0x_0) = (0110)$, which is position 6 on the curve.

To assign blocks to processes, we cut the Morton curve into equally-sized sections and assign each section to one process. Using the inverse Morton-curve mapping, every process can then quickly determine the three-dimensional position of its blocks in the simulation domain. This achieves balanced load while preserving spatial locality. Fig. 12 shows the assignment of the blocks to eight processes for our example.

Note that the Morton-curve contains jumps which impairs its spatial locality property, as visible between position 7 and 8 in Fig. 11. To reduce a negative impact on the spatial locality of our block-to-process assignment, those jumps should coincide with the cuts of the curve. Alternatively, more complex space-filling curves with better spatial locality such as the Hilbert curve [26] may improve this. Exploring their suitability is left for future work.

5.2.3. The distributed tree

By distributing the domain over the processes, every process owns only a subset of the neurons in the simulation domain. Consequently, processes store only part of the scalable algorithm's original tree. We call this part *partial tree*. Every process' partial tree contains the nodes of the original tree from the root down to including the branch nodes. That is, this upper part is replicated on all processes. In addition, each process expands its own partial tree by adding all children (i.e., the subtree) of those branch nodes which correspond to the blocks of the domain assigned to it. Fig. 13 shows the partial tree on process 0 for our example. Merging the partial trees of all processes would yield the original tree again. Now, every process has only incomplete knowledge of the simulation domain. This resembles the real brain where neurons have only partial knowledge of their environment.

As a result, when a process unfolds a branch node in its partial tree and is not the owner of this branch node, it needs to know which other process (the owner) to contact to retrieve the branch node's direct children. We provide this contact information by storing the MPI rank of the owner in every tree node. Note that only branch nodes owned by remote processes are labeled with MPI ranks that differ from the own rank. All the remaining nodes in the partial tree carry the same local rank. In our example, Fig. 13 shows process 0's partial tree augmented with the MPI ranks of the owner of each node.

5.2.4. Access to remote tree nodes

To retrieve tree nodes from other processes, we use MPI's one-sided communication routines. They are able to combine the send call on the sender and the receive call on the receiver of traditional two-sided communication into one call. In this approach, which is called remote memory access (RMA), the calling process (*origin*) specifies both the send buffer and the receive buffer. The process at the other end of the RMA communication is called *target*. We use MPI RMA passive-target synchronization as it does not require the target to call any MPI routines. Our choice of MPI's one-sided communication model is motivated by the properties of our scalable algorithm: First, a process does not need to be actively involved when other processes fetch tree nodes from it, as the corresponding data is already available in memory and does not need to be "prepared" for transmission (e.g., by copying into a contiguous send buffer). Of course, this assumes an appropriate data layout in memory. Second, MSP and thus our algorithm focuses on the mature brain in which neurons do not move, as opposed to moving particles in a particle simulation, for example. Consequently, restructuring the tree is not needed and thus the owners (RMA target ranks) of tree nodes do not change.

From the programmer's productivity perspective, using RMA for fetching tree nodes from a remote process avoids additional code complexity. In particular, a process needs to answer an a priori unknown number of incoming requests for tree nodes and, at the same time, find target neurons for its own local neurons. To avoid deadlocks and increase performance, answering remote requests and processing local neurons should ideally overlap. For example, the Barnes–Hut code PEPC implements this approach with a dedicated communication thread [33]. With our RMA operations, this additional communication thread is not needed, as the MPI library transparently handles the incoming memory accesses.

Finally, MPI RMA operations have also potential to improve performance over two-sided communication by eliminating the need to match send and receive calls. This reduces the amount of synchronization between processes and allows them to progress more independently of each other. For example, combining send and receive in one RMA call such as MPI_Get avoids scenarios where the sender has to wait for the receiver or vice versa. However, RMA operations strongly depend on the progress these operations make in the MPI library after they have been initiated. For example, an optimized RMA implementation could ensure progress through a dedicated progress thread internally. Moreover, the network hardware's support for remote direct memory access (RDMA) operations, such as available with InfiniBand, could be exploited. The MPI library on the Blue Gene/Q system on which we evaluate our scalable algorithm's implementation with the distributed tree offers a progress thread.

In addition to the target rank, the origin of the RMA call needs to specify the memory location of the tree node to be fetched. We use the absolute memory addresses (pointers) that every tree node stores of its children (at most eight in a quadtree). That is, a tree node labeled with its process' own rank contains pointers to locations in local memory, while a tree node labeled with a remote rank contains pointers to locations in the remote process' memory. However, how does a process get the "remote" pointers? As shown in Fig. 13, every branch node is the root of a subtree which is completely owned by one process. This is why every process broadcasts the branch nodes that it owns (i.e., labeled with its own rank) to the other processes. Finally, the received branch nodes contain the remote pointers.

When a process unfolds a tree node marked as remote, it fetches all children during the same RMA *access epoch*. MPI_Win_lock and MPI_Win_unlock respectively start and complete the access epoch during which we issue up to eight MPI_Get calls to fetch the children. Only after MPI_Win_unlock returns, the children are guaranteed to be locally available. For this reason, we have to wait for all children before we can evaluate the acceptance criterion for any of them and, if necessary, fetch their children, too.

Note that for finding target neurons, often the same remote tree nodes are used for probability calculations by several source neurons on the same process. Instead of retrieving the same remote nodes repeatedly, we store them in a tree-node cache and only issue RMA operations on cache misses. This helps to reduce the time spent in communication.

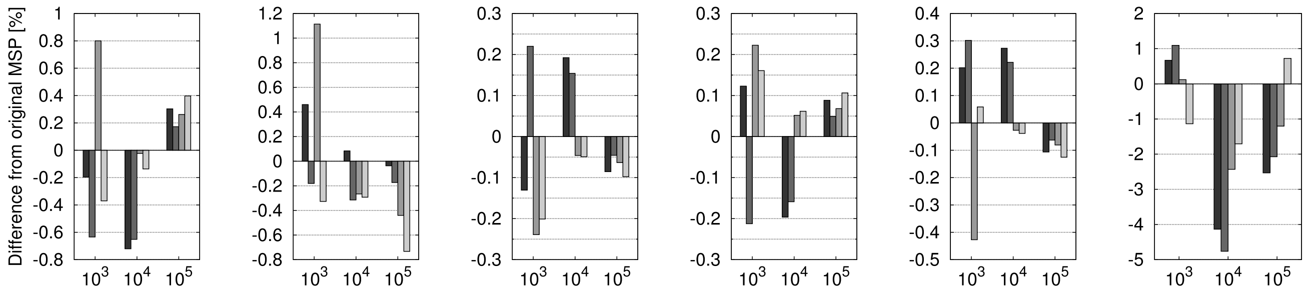
To the best of our knowledge, the FLY [5,6] code for cosmological simulations is the only Barnes–Hut code at the time of writing which uses MPI RMA. In contrast to our approach, it also uses RMA operations for the tree construction. In particular, all processes cooperate and build a single tree structure one level after the other. During the force calculation of the particles, FLY uses an RMA-based work-stealing approach to dynamically balance the load between processes. The authors of FLY state that the code contains about 110 calls to MPI_Win_lock and MPI_Win_unlock and 60 calls to MPI_Put and MPI_Get [6]. Obviously, our approach of using MPI RMA in brain simulation is more lightweight and strives to reduce code complexity while improving performance at the same time.

```
 1: Create empty network without synapses                          ▷ $O(1)$
 2: Initialize number synaptic elements                            ▷ $O(n/p)$
 3: Insert local neurons into partial tree                         ▷ $O(n/p \log n)$
 4: while Desired avg. calcium concentration not reached do
 5:     UpdateElectricalActivity                                   ▷ $O(n/p)$
 6:     UpdateSynapticElements                                     ▷ $O(n/p)$
 7:     if Connectivity time step completed then
 8:         UpdateConnectivity {                                   ▷ $O(n/p \log^2 n + p)$
 9:             Delete synapses & update network                   ▷ $O(n/p)$
10:             Create synapses {
11:                 Update lower partial tree                      ▷ $O(n/p)$
12:                 Gather branch nodes &
13:                  update them in the partial tree               ▷ $O(p)$
14:                 Update upper partial tree                      ▷ $O(p)$
15:                 Find target neurons for
16:                  vacant axonal elements                        ▷ $O(n/p \log^2 n)$
17:                 Update network                                 ▷ $O(n/p)$
18:             }
19:         }
20:     end if
21: end while
```

Fig. 14. Parallel scalable algorithm with distributed tree: the simulation flow of MSP for each process when using our hierarchical algorithm with the distributed tree. The time complexity of each step is shown on the right. Simulation parameters appear in *italics*. n denotes the number of neurons. The simulation terminates when the neurons reach the *desired average calcium concentration*. The connectivity update is only executed when a *connectivity time step* is completed. p denotes the number of processes.

5.2.5. Summary

Let us now give an overview of our scalable algorithm's implementation using the distributed tree as described above. We call this version *parallel scalable algorithm with distributed tree*. Every process owns about the same number of neurons n/p and performs all the steps for them. Fig. 14 lists the individual steps with their complexity. Instead of discussing every single step, we focus on those steps which require further explanation. In line 11, every process updates the positions and the number of vacant dendrites of the (virtual) neurons in its partial tree from bottom to top. Note that after this step, the upper tree levels above the branch nodes do still not reflect the current state of the simulation. The reason is that only the own branch nodes are up-to-date. Branch nodes owned by other processes need to be updated with the current information from these processes. This is why in the branch-node exchange (line 12), every process sends its own branch nodes and receives those from the other processes. The branch nodes received from other processes replace their outdated counterparts in the partial tree (line 13). Now, all branch nodes contain the correct current position and number of vacant dendrites of their corresponding virtual neuron. Finally, the remaining upper tree levels are updated up to the root (line 14). At this point, every process' partial tree reflects the current state of the neurons in the simulation. The total time of a single simulation step is $O(n/p \log^2 n + p)$. With our design of the distributed tree, the space complexity per process drops from $O(n)$ to $O(n/p+p)$. Note that the



(a) Number of edges. (b) Average Euclidean distance. (c) Average shortest path length. (d) Global efficiency. (e) Average betweenness centrality. (f) Average clustering coefficient.

Fig. 15. Network comparison between the original MSP and our algorithm for 10^3, 10^4, 10^5 neurons. The bars at each cluster correspond from left to right to $\theta = 0.1, 0.2, 0.3, 0.4$.

additive term p stems from the branch node exchange between all the processes. However, our practical results in Section 6 do not indicate memory capacity issues even for 256k processes.

6. Results

Let us now evaluate our algorithm in terms of impact of the approximation on the result quality and performance. Our simulation parameters correspond to layer 5A of the rat cortex [25], which is about 500 μm thick with a density of 54,500 neurons per mm^3. With this density, the average distance between neurons in all three dimensions is about 26 μm. Of the neurons, 20% are inhibitory, the remaining 80% are excitatory. Initially, no synapses exist. Every neuron is initialized with vacant synaptic elements: one excitatory and one inhibitory dendritic element plus one axonal element. The type of the axonal element depends on the type of the neuron. Excitatory neurons form excitatory axonal elements while inhibitory neurons form inhibitory axonal elements. However, both neuron populations grow excitatory and inhibitory dendritic elements. Synapses are only possible between synaptic elements of the same type. That is, excitatory axonal elements only connect to excitatory dendritic elements. A similar rule applies to inhibitory synaptic elements. During all simulations, we run the connectivity update every 100 ms biological time, which equals 100 simulation steps. In our experiments, we consider an MPI-based version of the original MSP algorithm and our two implementations of the scalable MSP approximation algorithm (Section 5).

6.1. Accuracy

To validate the accuracy of our algorithm, we compare the neural networks it generates to those of the original MSP. We consider a neural network as a weighted directed graph where neurons are the vertices and synapses are the directed edges pointing from a source neuron to a target neuron. The number of synapses reaching from the source to the target is the edge weight. Our comparison is based on the following graph topology metrics that Butz et al. [8] use to describe the structure of neural networks: (i) number of edges, (ii) average Euclidean distance, (iii) average shortest path length, (iv) global efficiency, (v) average betweenness centrality, and (vi) average clustering coefficient. We compare networks of the original MSP and our algorithm with 10^3, 10^4, and 10^5 neurons. The quadratic computational complexity of some of the graph metrics prevents us from evaluating them for larger neuron counts. We randomly distribute the neurons in a volume of height 500 μm and let the other two dimensions grow with the number of neurons. We start with an empty network and initially vacant synaptic elements. The biological simulation time is $6 \cdot 10^6$ ms, which allows 60,000 connectivity updates. At this time, the neuronal electrical activity (desired average calcium concentration) and the network have reached their equilibrium and change only insignificantly. Note that equilibrium is in fact already reached after about 50,580 updates of connectivity in our experiments. However, even during equilibrium small numbers of neurons are rewired to maintain the desired electrical activity. Continuing the simulation until 60,000 connectivity updates captures additional minor structural changes which could affect the accuracy of the approximated networks of our algorithm.

Fig. 15 shows the metrics of the networks generated by our algorithm relative to those of the original MSP. Except for the average clustering coefficient, the networks produced by our algorithm differ only by about 1% from the original MSP. This is even true for low precisions. However, for 10^4 neurons the average clustering coefficient differs by about 5% with even a small $\theta = 0.2$. To investigate this case further, we calculate the average and standard deviation of the metrics over 11 simulations for both algorithms (Fig. 16). We use a different random number seed for every run. Interestingly, even for the exact MSP, individual average clustering coefficients vary by about 5% across measurements. That is, the 5% difference can also be found between different runs of the exact MSP. Fig. 16 gives an overview of how the results vary. It can be seen that the difference between the average of the exact MSP and our algorithm is below 1% for all metrics. Also the standard deviations are similar except for the average Euclidean distance. However, although the "spread" around the average is different for this metric, the actual average of our method only differs by 0.14%. Hence, we do not consider the difference in the standard deviation as significant.

These experiments support our claim that our approximated networks are still precise enough to represent neural networks of the exact MSP. Nevertheless, to account for the variation of the results due to MSP's probabilistic approach, several simulation runs are necessary to reliably capture the essential structure of a neural network at the end of the simulation. This is true for both, the exact MSP and our approximation of it. Thus, the scalability of the MSP algorithm is even more critical.

6.2. Performance

Our performance evaluation consists of three parts. First, we investigate the performance of our parallel scalable algorithm with replicated tree and compare it to the original MSP algorithm. Second, to analyze the scalability of our approximation algorithm, we run large-scale simulations with up to 10^9 neurons using the parallel scalable algorithm with distributed tree. Finally, we extrapolate the performance of the scalable algorithm to 10^{11}, the number of neurons in the human brain.

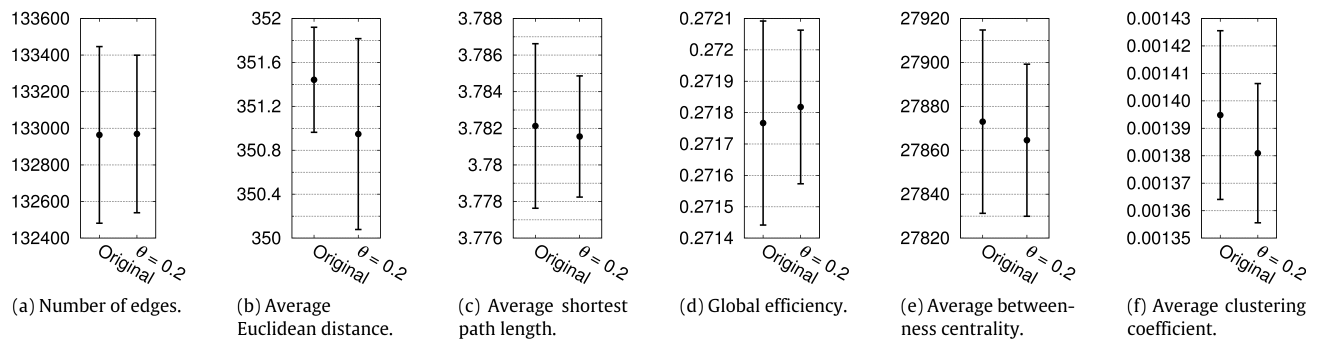


Fig. 16. Network comparison between the original MSP and our algorithm with $\theta = 0.2$ for 10^4 neurons. The results per method are based on 11 simulations with a different random number seed for each run. Each dot denotes the average of the 11 simulations. Bars depict the standard deviations.

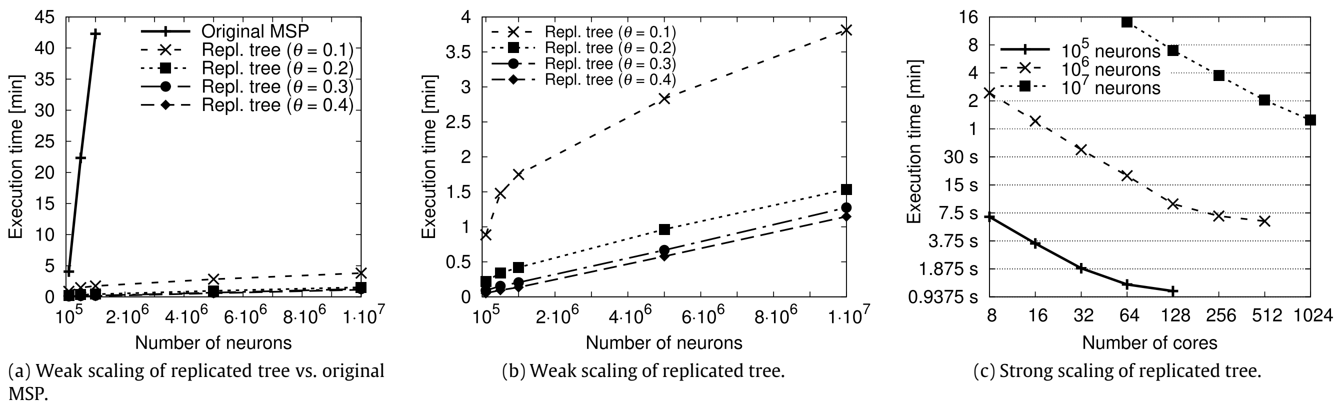


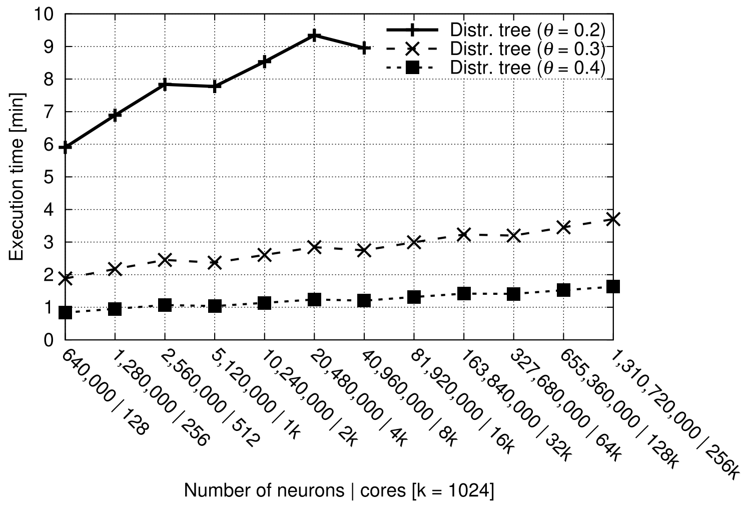
Fig. 17. Scaling results. (a–b) Weak scaling execution times with different precision parameters. The number of neurons per core (MPI process) is 10^4. The numbers of neurons (cores) are: 10^5 (10), $5 \cdot 10^5$ (50), 10^6 (100), $5 \cdot 10^6$ (500), 10^7 (1000). (c) Strong scaling execution time with $\theta = 0.3$.

6.2.1. Simple approach vs. original MSP

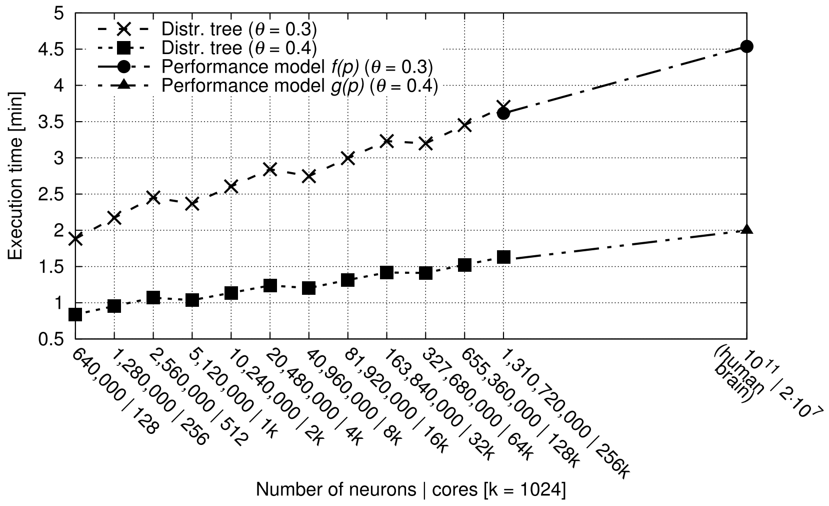
We run the experiments for the original MSP algorithm and the scalable algorithm with the replicated tree on the Lichtenberg cluster at TU Darmstadt. Our compute nodes are equipped with two Intel Xeon E5-2670 (8 cores each) and 32 GiB RAM. The cluster network is InfiniBand FDR10. In both experiments, we randomly distribute the neurons in a volume of height 500 μm and let the other two dimensions grow with the number of neurons. Every process simulates the same number of neurons n/p. Similarly to the algorithm with replicated tree, the original MSP stores all neurons on every process as every neuron considers all the other neurons for synapse creation. Due to the high memory consumption, we only run two processes per compute node in the experiments. In contrast to the original MSP, the algorithm with replicated tree performs tree construction and tree update (Fig. 8, lines 3 and 12). Nevertheless, even for 10^7 neurons tree construction does not exceed 2 min. We do not include this time in our measurements as it is a one-off expense. Another difference is the synapse formation, which is accelerated via approximation in the replicated tree algorithm (Fig. 8, lines 13–14). Fig. 17a and 17b show weak scaling results for one connectivity update for the original MSP and the replicated tree algorithm. We simulate the very first connectivity update where no synapses yet exist and every neuron has vacant elements, as described above. That is, every neuron is trying to find a target neuron for its vacant axonal element. The original MSP needs about 40 min for 10^6 neurons, while the algorithm with replicated tree terminates in 2 min. On the other hand, even for high precision with $\theta = 0.1$, the replicated tree algorithm is still in the range of 5 min for 10^7 neurons. This large number of neurons is practically out of reach for the original MSP. The results also show that reducing the precision of our method from 0.1 to 0.2 can help to further reduce the execution time by at least a factor of 2.5. Fig. 17c depicts strong scaling results for the replicated tree. The results exhibit good scalability given that successively doubling the number of cores respectively reduces the execution time by a factor of two. Note that the timings presented above partly differ from our previous recent publication of the scalable algorithm [29]. The reason is that previous measurements were partially affected by interference with other jobs running on our compute nodes.

6.2.2. Distribution of the tree

To investigate the scalability of our approximation algorithm, we perform large-scale structural plasticity simulations on the IBM Blue Gene/Q system JUQUEEN [22] at Forschungszentrum Jülich. The system houses 28,672 compute nodes where each node is equipped with one IBM PowerPC A2 (16 cores) and 16 GiB RAM. The network is a 5d-torus custom design. To improve RMA performance, we enable the progress thread of JUQUEEN's MPI library during our measurements. In our experiments, all neurons are randomly distributed in a cube where the average distance between neurons is 26 μm. Note that some experiments exceed the number of neurons in the rat brain ($2 \cdot 10^8$) [20]. However, we believe that the parameter of 26 μm average distance between neurons is also a valid assumption in the human brain, which is the final scalability target of our algorithm. As in the previous experiments, we simulate the very first connectivity update where no synapses yet exist and every neuron has vacant elements, as described above. Similarly, we do not include the initializations (e.g., inserting local neurons into the partial trees) in our timings, since they are executed only once before simulation start. However, for all experiments, initialization is below 2 s. We run simulations for numbers of processes (cores) which are powers



(a) Weak scaling of distributed tree.



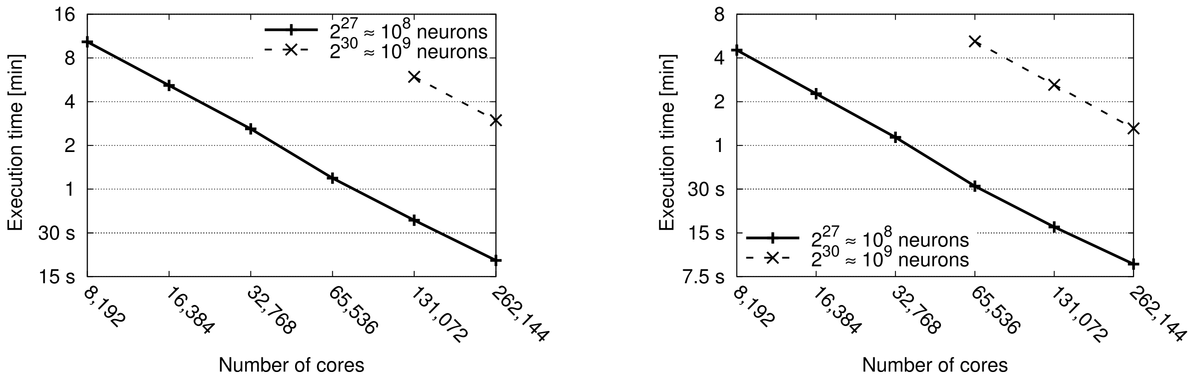
(b) Weak scaling of distributed tree with extrapolation.

Fig. 18. Weak scaling results of the distributed tree algorithm and extrapolation to the human brain (10^{11} neurons). (a) Weak scaling execution times with different precision parameters. The number of neurons per core (MPI process) is 5,000. The numbers of cores start from 128 and are doubled until 262,144. (b) Scaling results from (a) and extrapolation of the execution time to a human-brain sized simulation. See Eq. (7) for the performance models $f(p)$ and $g(p)$.

of two. Given that the simulation domain is decomposed into a power of eight blocks, every process is assigned one, two, or four blocks. We execute one process on every core, which amounts to 16 processes per compute node.

Fig. 18a shows weak scaling results of the distributed tree algorithm for one connectivity update for up to 10^9 neurons and different precisions. Every process (core) contains 5,000 neurons. Each timing result is the average of 5 runs. For all results, the standard deviation is always below 1 s. As can be seen, the precision parameter θ clearly determines the slope of the execution times. While the graphs of the functions for $\theta = 0.3$ and 0.4 are closer together, $\theta = 0.2$ shows a much larger increase. A similar pattern is also visible for the replicated tree for $\theta = 0.1$ and 0.2 (Fig. 17b). Note that this behavior can also be observed for Barnes–Hut due to the non-linear relationship between the work and precision θ [7,21,30]. In particular, work is proportional to $1/\theta^3$. Nevertheless, all our weak scaling results scale logarithmically with the number of neurons. Doubling the number of neurons increases execution time by an additive constant only. Consequently, our asymptotic upper bound of $O(n/p \log^2 n + p)$ for the execution time is confirmed by our experimental results.

Another remarkable result is the fluctuation of execution times. Although less pronounced for $\theta = 0.3$ and 0.4, all timings exhibit the same fluctuation pattern which consist of groups of three increasing execution times. More precisely, in Fig. 18a, the first group contains 128, 256, and 512 processes, the second group contains the next three process counts, and so on. The main reason for the timing fluctuations is the number of tree levels that are replicated on every process. Note that the more levels a process stores locally, the fewer virtual neurons must be fetched remotely. Given that every process is assigned at least one of 8^k blocks of the decomposed domain, the number of blocks must grow with the number of processes. Decomposing the domain into more blocks (i.e., 8^{k+1}) of smaller size, increments the depth of the replicated portion of the tree. Consequently, every process replicates $k + 1$ instead of k levels. For example, for 512 processes in Fig. 18a, every process owns one of $512 = 8^3$ blocks. To assign at least one block to each of the 1,024 processes, we divide the domain into the next larger power of eight $8^4 = 4 \cdot 1{,}024$ blocks. Now, the depth of the replicated portion increases by one and every process is assigned 4 blocks of the domain. The corresponding execution time for 1,024 processes decreased slightly. Interestingly, timings vary less with



(a) Strong scaling of distributed tree with $\theta = 0.3$. (b) Strong scaling of distributed tree with $\theta = 0.4$.

Fig. 19. Strong scaling results of the distributed tree algorithm. (a) Strong scaling execution time with $\theta = 0.3$. (b) Strong scaling execution time with $\theta = 0.4$.

decreasing precision (i.e., larger θ). The reason is that the Barnes-Hut acceptance criterion is already satisfied by nodes at smaller depth in the tree, and thus deep excursions towards the leaves are avoided. Similarly, replicating more tree levels on every process lowers the timing benefit of one additional level because many nodes are already locally available. At the expense of $O(n)$ space complexity per process, our algorithm with the replicated tree eliminates these fluctuations. The replicated and distributed tree implementations of our scalable algorithm illustrate the common trade-off between performance and memory consumption. In Section 6.1, we saw that using $\theta = 0.3$ or 0.4 still yields neural networks of the same quality as $\theta = 0.2$ or even 0.1. This is why $\theta = 0.3$ or 0.4 are appropriate parameters for large-scale structural plasticity simulations with our scalable approximation algorithm.

The results of the strong scaling experiments are depicted in Fig. 19. For $\theta = 0.3$ and 0.4, we simulate $2^{27} \approx 10^8$ and $2^{30} \approx 10^9$ neurons on up to 262,144 cores. The largest number of neurons per core (process) is 16,384 for both neuron counts. The smallest number of neurons per core is 512 for 10^8 neurons, and 4,096 for 10^9 neurons. We observe an almost perfect strong scaling for all experiments. Unfortunately, the JUQUEEN system does not contain 524,288 cores which is the reason why we stopped the experiments at 262,144 cores.

Our weak and strong scaling results show that our scalable approximation algorithm is able to simulate the dynamics of neural networks of up to 10^9 neurons and it can also efficiently use more compute cores to speed up the simulation. Note that 10^9 neurons correspond to today's largest brain simulations with fixed connectivity [2,24]. That is, our approximation algorithm of MSP could extend state-of-the-art brain simulators with the ability to create large-scale neural networks from scratch and to rewire their neurons during simulation, and thus help to deepen the understanding of the brain.

6.2.3. *Extrapolation*

Despite good scaling behavior of our distributed tree algorithm, we are still not able to run full-scale simulations of the human brain. The reason is that to reach 10^{11} neurons, we still need to increase the size of our largest simulation of 10^9 neurons by two orders of magnitude. Currently, as in our weak scaling experiments, every process (core) with 5,000 neurons occupies about 390 MiB RAM on JUQUEEN. In these 390 MiB, there is still free memory for about 1.2 million tree nodes which have been preallocated with MPI_Alloc_mem for access via RMA. Under the assumption that the memory footprint per process does not exceed 500 MiB, a simulation of 10^{11} neurons would require $10^{11}/5{,}000 = 2 \cdot 10^7$ processes (cores) with 500 MiB each. This amounts to 9.3 PiB RAM in total. Today's largest supercomputers in the TOP500 list (November 2016) are equipped with at most 2 PiB memory and contain at most 10^7 cores. Nevertheless, with reference to exascale computing, we would like to be able to "forecast" the execution time expected for such a simulation if a system with sufficient compute resources was available. For this reason, we create performance models based on our weak scaling timings on JUQUEEN. We use the performance model generator Extra-P [10] to obtain a function for the execution time. The input parameter to this function is the number of processes (cores) p. Fig. 18b illustrates our weak scaling results together with the performance models for $\theta = 0.3$ and 0.4. For the sake of clarity, we start to draw the models behind the timings obtained experimentally. The models $f(p)$ for $\theta = 0.3$ and $g(p)$ for $\theta = 0.4$ are:

$$
\begin{aligned}
f(p) &= 0.961461 + 0.14743 \cdot \log_2 p \\
g(p) &= 0.415784 + 0.0652235 \cdot \log_2 p
\end{aligned}
\qquad (7)
$$

The statistical quality measures of the model for $\theta = 0.3$ are (i) residual sum of squares (RSS) $= 0.1018$ and (ii) adjusted $R^2 = 0.9682$. The RSS is less than 4% of the average of the timings. For $\theta = 0.4$, RSS $= 0.0169$ and adjusted $R^2 = 0.9728$. Here, the RSS is less than 2% of the average of the timings. Note that both models have an adjusted R^2 close to 1, where 1 indicates a perfect fit between model and data. Given these measures, we consider our models to be good representations of the timings. The corresponding plots show us an estimate of the execution time of the first connectivity update of MSP with 10^{11} neurons. In particular, the extrapolated time is about 4.5 min for $\theta = 0.3$ and 2 min for $\theta = 0.4$.

Of course, given that these forecasts are based on scaling behavior for smaller neuron counts, they should be considered with a healthy degree of skepticism. Nevertheless, they can still provide us a reasonable lower bound of the execution time on a potential exascale computer with similar performance characteristics as the Blue Gene/Q system JUQUEEN. Based on this extrapolation, we believe that simulating structural plasticity of the full human brain with our scalable approximation of MSP could be feasible on an exascale supercomputer. However, given that our algorithm performs only a modest amount of floating point operations, the tremendous amount of computing power of an exascale machine might not even be needed. In contrast, a scalable special-purpose system with modest floating point performance and with focus on low latency communication might be an even more practical solution for brain simulation. Such a neuromorphic computing system is also expected to consume only a fraction of the power of a fully-featured exascale supercomputer.

7. Conclusion

The quadratic time complexity of MSP in terms of the number of neurons limits the largest possible structural-plasticity simulations to networks with 10^5 neurons, which is less than one finds in the brain of a mouse. For example, one connectivity update for 10^6 neurons takes about 40 min with the original MSP, which is prohibitive considering the frequency of such updates. Our approximation algorithm for MSP reduces the complexity to $O(n \log^2 n)$. As a result, the scalable implementation of our algorithm with the replicated tree reduces the execution time required for one connectivity update of 10^6 neurons using the same number of processors already by a factor of 20. Using more processors allows even further speedup.

For the first time, it will now be possible to simulate structural plasticity at the scale of a rat brain ($2 \cdot 10^8$ neurons) and beyond. Encouraging performance results demonstrating the practical feasibility of simulating 10^9 neurons have already been presented in this study. With this, neuroscientists can now more easily create connectivity maps for large-scale brain simulations. Overall, our results will allow more realistic large-scale brain simulations that, for the first time, account for the dynamics of the connectome. This is of utmost importance to investigating the mechanisms behind learning and healing.

Finally, our performance models predict that even with today's technology a full-scale simulation of the dynamics of the connectome in the human brain is possible in principle. While this would require a machine with about twice the number of compute cores and about five times the memory capacity of today's most powerful supercomputer, a system that meets these requirements might emerge quite soon.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 720270 (HBP SGA1). This work was also partially supported by the Helmholtz Association through the Helmholtz Portfolio Theme "Supercomputing and Modeling for the Human Brain" and by the Jülich Aachen Research Alliance (JARA) with funding from the Excellence Initiative of the German federal and state governments. The authors gratefully acknowledge the computing time granted by the JARA-HPC Vergabegremium and provided on the JARA-HPC Partition part of the supercomputer JUQUEEN [22] at Forschungszentrum Jülich.

References

[1] S. Aluru, J. Gustafson, G. Prabhu, F. Sevilgen, Distribution-independent hierarchical algorithms for the N-body problem, J. Supercomput. 12 (4) (1998). http://dx.doi.org/10.1023/A:1008047806690.

[2] R. Ananthanarayanan, S. Esser, H. Simon, D. Modha, The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses, in: Proc. of SC'2009, 2009. http://dx.doi.org/10.1145/1654059.1654124.

[3] F.A.C. Azevedo, L.R.B. Carvalho, L.T. Grinberg, J.M. Farfel, R.E.L. Ferretti, R.E.P. Leite, W.J. Filho, R. Lent, S. Herculano-Houzel, Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain, J. Comp. Neurol. 513 (5) (2009) 532–541. http://dx.doi.org/10.1002/cne.21974.

[4] J. Barnes, P. Hut, A hierarchical O(N log N) force-calculation algorithm, Nature 324 (6096) (1986). http://dx.doi.org/10.1038/324446a0.

[5] U. Becciani, V. Antonuccio-Delogu, Are you ready to FLY in the universe? A multi-platform N-body tree code for parallel supercomputers, Comput. Phys. Comm. 136 (1–2) (2001) 54–63. http://dx.doi.org/10.1016/S0010-4655(00)00253-8. arXiv:0101148v1.

[6] U. Becciani, V. Antonuccio-Delogu, M. Comparato, FLY: MPI-2 high resolution code for LSS cosmological simulations, Comput. Phys. Comm. 176 (3) (2007) 211–217. http://dx.doi.org/10.1016/j.cpc.2006.10.001. arXiv:0703526.

[7] G. Blelloch, G. Narlikar, A practical comparison of *n*-body algorithms, in: Parallel Algorithms, in: Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1997.

[8] M. Butz, I. Steenbuck, A. van Ooyen, Homeostatic structural plasticity increases the efficiency of small-world networks, Front. Syn. Neurosci. 6 (7) (2014). http://dx.doi.org/10.3389/fnsyn.2014.00007.

[9] M. Butz, A. van Ooyen, A simple rule for dendritic spine and axonal bouton formation can account for cortical reorganization after focal retinal lesions, PLoS Comput. Biol. 9 (10) (2013). http://dx.doi.org/10.1371/journal.pcbi.1003259.

[10] A. Calotoiu, T. Hoefler, M. Poke, F. Wolf, Using automated performance modeling to find scalability bugs in complex codes, in: Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA, ACM, 2013, pp. 1–12. http://dx.doi.org/10.1145/2503210.2503277.

[11] M.F. Casanova, L. De Zeeuw, A. Switala, P. Kreczmanski, H. Korr, N. Ulfig, H. Heinsen, H.W.M. Steinbusch, C. Schmitz, Mean cell spacing abnormalities in the neocortex of patients with schizophrenia, Psychiatry Res. 133 (1) (2005) 1–12. http://dx.doi.org/10.1016/j.psychres.2004.11.004.

[12] I. Dammasch, G. Wagner, J. Wolff, Self-stabilization of neuronal networks, Biol. Cybernet. 54 (4) (1986). http://dx.doi.org/10.1007/BF00318417.

[13] V. De Paola, A. Holtmaat, G. Knott, S. Song, L. Wilbrecht, P. Caroni, K. Svoboda, Cell type-specific structural plasticity of axonal branches and boutons in the adult neocortex, Neuron 49 (6) (2006). http://dx.doi.org/10.1016/j.neuron.2006.02.017.

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

[15] J. Dubinski, A parallel tree code, New Astron. 1 (2) (1996) 133–147. http://dx.doi.org/10.1016/S1384-1076(96)00009-7. arXiv:9603097v1.

[16] M. Gewaltig, M. Diesmann, NEST (NEural Simulation Tool), Scholarpedia 2 (4) (2007). http://dx.doi.org/10.4249/scholarpedia.1430.

[17] A. Grama, V. Kumar, A. Sameh, Scalable parallel formulations of the Barnes-Hut method for n-body simulations, Parallel Comput. 24 (5) (1998) 797–822. http://dx.doi.org/10.1016/S0167-8191(98)00011-8.

[18] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, J. Comput. Phys. 73 (2) (1987). http://dx.doi.org/10.1016/0021-9991(87)90140-9.

[19] T. Hensch, Critical period plasticity in local cortical circuits, Nat. Rev. Neurosci. 6 (11) (2005). http://dx.doi.org/10.1038/nrn1787.

[20] S. Herculano-Houzel, The human brain in numbers: a linearly scaled-up primate brain., Front. Hum. Neurosci. 3 (November) (2009) 31. http://dx.doi.org/10.3389/neuro.09.031.2009. URL http://journal.frontiersin.org/article/10.3389/neuro.09.031.2009/full.

[21] L. Hernquist, Performance characteristics of tree codes, Astrophys. J. Suppl. Ser. 64 (1987) 715–734. http://dx.doi.org/10.1086/191215. URL http://www.sciencedirect.com/science/article/pii/S0021999184710503.

[22] Jülich Supercomputing Centre, JUQUEEN: IBM Blue Gene/Q supercomputer system at the Jülich supercomputing centre, J. Large-Scale Res. Facil. 1 (A1) (2015). http://dx.doi.org/10.17815/jlsrf-1-18.

[23] T. Keck, T. Mrsic-Flogel, M. Vaz Afonso, U. Eysel, T. Bonhoeffer, M. Hübener, Massive restructuring of neuronal circuits during functional reorganization of adult visual cortex, Nat. Neurosci. 11 (10) (2008). http://dx.doi.org/10.1038/nn.2181.

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

[25] H. Meyer, D. Schwarz, V. Wimmer, A. Schmitt, J. Kerr, B. Sakmann, M. Helmstaedter, Inhibitory interneurons in a cortical column form hot zones of inhibition in layers 2 and 5A, Proc. of Nat. Acad. of Sci. 108 (40) (2011). http://dx.doi.org/10.1073/pnas.1113648108.

[26] B. Moon, H.V. Jagadish, C. Faloutsos, J.H. Saltz, Analysis of the clustering properties of the hilbert space-filling curve, IEEE Trans. Knowl. Data Eng. 13 (1) (2001) 124–141. http://dx.doi.org/10.1109/69.908985.

[27] Morton, A computer oriented geodetic data base and a new technique in file sequencing, Tech. Rep., IBM Ltd., Ottawa, Ontario, Canada, 1966.

[28] G. Prasad, J. Burkart, S. Joshi, T. Nir, A. Toga, P. Thompson, A dynamical clustering model of brain connectivity inspired by the n-body problem, in: Proc. of Int'l Workshop on Multimodal Brain Image Analysis (MBIA), 2013. http://dx.doi.org/10.1007/978-3-319-02126-3_13.

[29] S. Rinke, M. Butz-Ostendorf, M.-A. Hermanns, M. Naveau, F. Wolf, A scalable algorithm for simulating the structural plasticity of the brain, in: Proc. of the 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Los Angeles, CA, USA, 2016, pp. 1–8. http://dx.doi/org/10.1109/SBAC-PAD.2016.9.

[30] J.K. Salmon, M.S. Warren, Skeletons from the treecode closet, J. Comput. Phys. 111 (1) (1994) 136–155. http://dx.doi.org/10.1006/jcph.1994.1050. URL http://www.sciencedirect.com/science/article/pii/S0021999184710503.

[31] A. van Ooyen, J. van Pelt, Activity-dependent Neurite Outgrowth and Neural Network Development, in: Progr. in Brain Res., vol. 102, 1994. http://dx.doi.org/10.1016/S0079-6123(08)60544-0.

[32] M.S. Warren, J.K. Salmon, A portable parallel particle program, Comput. Phys. Comm. 87 (1–2) (1995) 266–290. http://dx.doi.org/10.1016/0010-4655(94)00177-4.

[33] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, P. Gibbon, A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations, Comput. Phys. Comm. 183 (4) (2012) 880–889. http://dx.doi.org/10.1016/j.cpc.2011.12.013.

[34] H. Yamahachi, S. Marik, J. McManus, W. Denk, C. Gilbert, Rapid axonal sprouting and pruning accompany functional reorganization in primary visual cortex, Neuron 64 (5) (2009). http://dx.doi.org/10.1016/j.neuron.2009.11.026.

Sebastian Rinke is a research associate at the Laboratory for Parallel Programming at TU Darmstadt in Germany. He received his Masters degree in computer science from TU Chemnitz in 2009. After graduation, he investigated novel computer architectures for supercomputing at the Jülich Supercomputing Centre of Forschungszentrum Jülich. Later, he worked on programming models for network-attached accelerators and scalable algorithms for brain simulation at RWTH Aachen University. In 2015, he moved to TU Darmstadt and began to explore large-scale natural language processing on supercomputers. Currently, he pursues a doctoral degree in the field of scalable algorithms in neuroscience.

Markus Butz-Ostendorf studied informatics and biology and holds a Ph.D. in neuroanatomy. He did several postdocs e.g. at the Bernstein Center for Computational Neuroscience Göttingen, the Neuroscience Campus VU Universiteit Amsterdam and the Forschungszentrum Jülich. His research focus is on modeling structural plasticity in the healthy and diseased brain. He phrased a computational theory on the driving forces for homeostatic structural plasticity following brain lesions. The underlying algorithms are freely available in the modeling framework for large-scale spiking neuronal networks NEST. He recently edited Frontiers Research Topic "Anatomy and plasticity in large-scale neuronal networks" and is editor of the first book on modeling structural plasticity entitled "The Rewiring Brain-A Computational Approach to Structural Plasiticity in the Adult Brain", Academic Press June 2017.

Marc-André Hermanns is a research assistant at the Jülich Supercomputing Centre of Forschungszentrum Jülich. He specializes in design and implementation of performance analysis tools for one-sided communication. Further research interests include scientific software development processes and visual performance analytics. He received his Master's degree in computer science in 2008 from the University of Hagen. Since 2008, he has also been an active part of the MPI Forum working group for tools interfaces. He is the author and co-author of several publications on performance tools and related topics. He is currently pursuing his Ph.D. studies on the scalable performance analysis of one-sided communication.

Mikaël Naveau is a Research Engineer of the CYCERON imaging platform. He received his M.Sc. degree in Bioinformatics from Paris Diderot University (France) and a Ph.D. in Neurosciences from Caen University (France). His research interests cover various aspects of biological data simulation and analysis including brain functional networks from the cellular to the macroscopic scales, multimodal imaging of the brain as well as applications of high performance computing tools to the management and analysis of large clinical and preclinical datasets.

Felix Wolf is a full professor of parallel programming at the Department of Computer Science of TU Darmstadt in Germany. He specializes in software and tools for parallel computers. After receiving his Ph.D. degree from RWTH Aachen University in 2003, he worked more than two years as a postdoc at the Innovative Computing Laboratory of the University of Tennessee. In 2005, he was appointed research group leader at the Jülich Supercomputing Center. From 2009 until recently, he was head of Parallel Programming at the German Research School for Simulation Sciences. Prof. Wolf has published more than a hundred refereed articles on parallel computing, several of which have received awards.