High-Performance and Scalable Agent-Based Simulation with BioDynaMo

Breitwieser, Lukas; Hesam, Ahmad; Rademakers, Fons; Luna, Juan Gómez; Mutlu, Onur

DOI
10.1145/3572848.3577480

Publication date
2023

Document Version
Final published version

Published in
PPoPP 2023 - Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming

High-Performance and Scalable Agent-Based Simulation with BioDynaMo

Lukas Breitwieser*
CERN, Switzerland
ETH Zurich, Switzerland

Ahmad Hesam
Delft University of Technology, The Netherlands

Fons Rademakers
CERN, Switzerland

Juan Gómez Luna
ETH Zurich, Switzerland

Onur Mutlu†
ETH Zurich, Switzerland

Abstract

Agent-based modeling plays an essential role in gaining insights into biology, sociology, economics, and other fields. However, many existing agent-based simulation platforms are not suitable for large-scale studies due to the low performance of the underlying simulation engines. To overcome this limitation, we present a novel high-performance simulation engine.

We identify three key challenges for which we present the following solutions. First, to maximize parallelization, we present an optimized grid to search for neighbors and parallelize the merging of thread-local results. Second, we reduce the memory access latency with a NUMA-aware agent iterator, agent sorting with a space-filling curve, and a custom heap memory allocator. Third, we present a mechanism to omit the collision force calculation under certain conditions.

Our evaluation shows an order of magnitude improvement over Biocellion, three orders of magnitude speedup over Cortex3D and NetLogo, and the ability to simulate 1.72 billion agents on a single server.

Supplementary Materials, including instructions to reproduce the results, are available at: https://doi.org/10.5281/zenodo.6463816

CCS Concepts: • **Computing methodologies** → **Massively parallel and high-performance simulations**; **Agent / discrete models**; **Parallel algorithms**; • **Software and its engineering** → **Software performance**.

Keywords: agent-based modeling, high-performance simulation, HPC, parallel computing, scalability, performance optimization, performance evaluation, space-filling curve, memory layout optimization, memory allocation, NUMA

*lukas.breitwieser@gmail.com
†omutlu@gmail.com

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577480

1 Introduction

Agent-based modeling (ABM) allows to simulate complex dynamics in a wide range of research fields. ABM has been used to answer research questions in biology [28, 41, 71], sociology [18], economics [63], technology [48], business [54], and more fields [37]. *Agents* are individual entities that, among others, can represent subcellular structures to simulate the growth of a neuron, a cell to investigate cancer development, or a person to simulate the spread of infectious diseases [10]. The actions of an agent are defined through instances of class *behavior*. Possible behaviors are neurite bifurcation, uncontrolled cell division, or infection.

Agent-based models are developed in an iterative way, during which an initial model is increasingly refined until it matches with observed data [56, 65]. Model parameters that cannot be derived from the literature are determined through optimization. An optimization algorithm generates a parameter set, executes the model, and evaluates the error with respect to observed data until the error converges to a local or global minimum. This loop might also contain an uncertainty analysis to evaluate the robustness of a solution [38]. Consequently, the model must be simulated many times.

The simulation engine's performance limits the scale of the model and determines how often the model can be simulated. Thus, performance is a key issue for simulating models on extreme scales that might one day be able to simulate all 86 billion neurons in the brain [7]. It is also crucial for smaller-scale simulations to explore vast parameter space, analyze parameter uncertainty, repeat the simulation often enough to reach statistical significance, and develop models rapidly.

To achieve these goals, we present a novel simulation engine called BioDynaMo, which is optimized for high performance and scalability. During its development, we identify the following three main performance challenges.

Challenge 1: To fully utilize systems with high processor core counts, the parallel part of the simulation engine has to be maximized (see Amdahl's law [3]). Although it is easy to parallelize the loop over all agents (Algorithm 1), our benchmarks revealed two operations whose level of parallelization has a significant performance impact. First, building the environment index, which is used to determine the neighbors of an agent. The literature describes various radial-neighbor

search algorithms with different design trade offs between build and search performance. Second, combining thread-local results at the end of each iteration.

Challenge 2: ABMs are predominantly memory-bound due to two reasons. First, the behavior of agents often has low arithmetic intensity. Second, ABM can be very dynamic. During a simulation, agents move through space, change their behavior, and are created and destroyed. Consequently, the neighborhood of an agent changes continuously, leading to an irregular memory access pattern and poor cache utilization. This results in large data movement between the main memory and the processor cores.

Challenge 3: Under certain conditions, the expensive calculation of mechanical forces between agents is redundant (Section 5). These forces are for example used in tissue models to determine the displacement of agents. The challenge is identifying those agents for which the pairwise force calculation can be safely omitted.

BioDynaMo addresses these challenges with the following new optimizations. To maximize the parallelization (Challenge 1), we develop an optimized uniform grid to search for agent neighbors and fully parallelize the addition and removal of agents. We address the data movement bottleneck (Challenge 2) in software by (i) optimizing the iteration over all agents on systems with non-uniform memory architecture, (ii) sorting agents and their neighbors to improve the cache hit rate and minimize access to remote DRAM, and (iii) introducing a pool memory allocator. To avoid redundant mechanical force calculations (Challenge 3), we add a mechanism to detect agents for which we can guarantee that the resulting force will not move the agent.

These mechanisms make BioDynaMo nearly an order of magnitude more efficient than Biocellion and three orders of magnitude faster than Cortex3D and NetLogo. The performance improvements account for a median speedup of 159× compared to BioDynaMo's standard implementation with all optimizations turned off. As a result, BioDynaMo is able to simulate 1.72 billion agents on one server. The main contributions of this paper are as follows.

- We present a novel high-performance agent-based simulation engine. The engine can be used in many domains due to its modular software design and features a specialization for neuroscience, capable of simulating the development of neurons.
- We present six optimizations to maximize performance (Section 3–5). These insights are transferable and can be used to improve the performance of other agent-based simulators.
- We present an in-depth evaluation of BioDynaMo's performance using five different simulations (Section 6). This comprehensive analysis provides insights for users of BioDynaMo into which parameters yield the best performance and hints for developers of future agent-based simulation tools.

2 BioDynaMo's Simulation Engine

This Section gives an overview of BioDynaMo and its components. BioDynaMo is written in C++, uses OpenMP [51] for shared-memory parallelism, and is available under the Apache 2.0 open-source license.

Breitwieser et al. [10] describe the user-facing features of the BioDynaMo platform and detail its modular software design and ease-of-use by means of three use cases in the domains of neuroscience, epidemiology, and oncology.

BioDynaMo is a hybrid framework able to utilize multi-core CPUs and GPUs. This paper focuses on the CPU version, which has two major advantages. First, the CPU version can simulate many more agents than a GPU version. The reason is that GPUs have typically significantly smaller memory than CPUs. For example, our benchmark hardware has 12× more memory than the current flagship GPU from NVidia, the A100 [50]. Second, the CPU version improves the usability and flexibility for our broad user community, who often only have a Matlab [29] or R [62] coding background. In BioDynaMo, users create simulations by writing C++ code. A GPU-only version would require users to write CUDA code to define new agents, behaviors, and other user-defined components. Therefore, BioDynaMo only offloads computations to the GPU, transparently to the user [25].

The main objects in agent-based simulations are agents, behaviors, and operations. Agents (e.g., a cancer cell) have attributes that are updated through behaviors and operations. Behaviors (e.g., uncontrolled cell division) are functions that can be assigned and removed from an agent and give users fine-grained control over the actions of an agent. In contrast, *Agent operations* are executed for each agent. For example, to calculate the mechanical forces between agents and execute all individual behaviors of an agent. The second type of operation, called *standalone operation* is executed once per iteration to perform a specific task (e.g., visualization). A characteristic property of agent-based simulation is local interaction. BioDynaMo provides a common interface for different neighbor search algorithms called *environment*. Besides the uniform grid detailed in Section 3.1, BioDynaMo features a kd-tree based on nanoflann [9] and octree based on the publication of Behley et al. [8].

The agent-based simulation algorithm (Algorithm 1) comprises two steps. First, users have to define the starting condition of the model (L1) in which agents, behaviors, operations, and any other resource are created. Second, the simulation engine executes this model for a number of iterations (L2–19). The engine executes all agent operations for each agent (L7–12) and all standalone operations (L12–14). Standalone operations can be further separated into operations that must be executed at the beginning of the iteration (e.g., to update the environment index [L3-5]) or the end (e.g., visualization [L16-18]). There are two barriers synchronizing threads (L6 and L15).

Algorithm 1: Simulation algorithm

```
1  ModelInitialization()
2  for i ∈ iterations do
3      for op ∈ pre_standalone_operations do
4          op();
5      end
6      wait()
7      parallel for a ∈ agents do
8          for op ∈ agent_operations do
9              op(a);
10         end
11     end
12     for op ∈ standalone_operations do
13         op();
14     end
15     wait()
16     for op ∈ post_standalone_operations do
17         op();
18     end
19 end
```

3 Maximize Parallelization

3.1 Grid-based Neighbor Search

Determining the neighbors of an agent is a pre-condition for all agent interactions. For example, the infection behavior in an epidemiological model requires information if any of the immediate neighbors is infected. In this context, it is essential to find neighbors fast and efficiently and minimize the build time of the required index. Building an index in every iteration has a high cost, as shown in the evaluation section. We exploit the fact that the interaction radius is known at the beginning of the iteration. For this fixed-radius search problem, a grid-based solution is a good choice because the box of an agent can be determined in constant time using the agent's position [8]. This is confirmed by our evaluation in Section 6.9. The build stage in which all agents are assigned to a box can be easily parallelized. In the search stage, the grid determines all neighbors by iterating over all agents in the same box and the surrounding boxes. In 3D space, we consider the 3x3x3 cube of boxes surrounding and including the query box. All agents inside a box are stored in an array-based linked list. The box only needs to store the start index and the number of elements it contains. To avoid zeroing all boxes at the beginning of the build stage, we add a timestamp attribute to each box, updated whenever an agent is added. Consequently, we can determine that a box is empty if the simulation and box timestamp is different. Therefore, we can build the grid in $O(\#agents)$ time instead of $O(\#agents + \#boxes)$, which is relevant for large simulation spaces that are not fully populated.

The array-based linked list uses the same agent indices as in the `ResourceManager`. The `ResourceManager`, an essential class in the simulation engine, stores raw agent pointers and offers functions to add, remove, get, and iterate over agents. Thus, it also benefits from the memory layout optimization presented in Section 4.2. This optimization reduces the distance in memory of agents that are close in space. Consequently, linked list elements will be closer to each other, improving the cache hit rate of traversing the linked list during the search stage of the grid. The described grid implementation can be found in the class `UniformGridEnvironment`.

3.2 Adding and Removing Agents in Parallel

To maximize the theoretically achievable speedup described in Amdahl's law [3], we maximize the parallel part of the simulation by parallelizing the addition and removal of agents. By default, BioDynaMo stores a thread-local copy of additions and removals and commits them to the `ResourceManager` at the end of each iteration.

Additions are trivial; the engine determines the total number of additions, grows the data structures in the `ResourceManager`, and adds the agent pointers in parallel. In contrast, the parallelization of removals is a more elaborate process because we disallow empty vector elements in the `ResourceManager`. If the simulation engine has to remove an agent stored in the middle of the vector, it must swap it with the last element before shrinking it. The following algorithm aims at performing the necessary swaps and updates in dependent data structures in parallel. Figure 1 illustrates the parallelized algorithm simplified for a single NUMA domain. This example assumes a simulation with seven agents represented with identifier 1–7 and two threads, which remove three agents from the simulation. These agents are highlighted with a grey background. The other colors serve as a visual aid to track the agents that must be swapped.

The algorithm comprises five main steps. First, the algorithm determines the total number of removed agents, calculates the new size of the vector ($old\_size - removed\_agents$), and initializes two auxiliary arrays. The size of the auxiliary arrays equals the number of removed agents. The new vector size is indicated by the vertical line between indexes three and four in Figure 1.

Second, each thread iterates over its vector of removed agents and fills the auxiliary arrays. If the agent is stored to the left of the new size index, it must be moved to the right. Therefore the algorithm inserts the element index into the array to_right. If the agent is stored to the right of the new size, we insert a one into array not_to_left at the index: $idx - new\_size$. The maximum index used to access elements in the auxiliary array is smaller than the number of removed agents and is independent of the number of remaining agents.

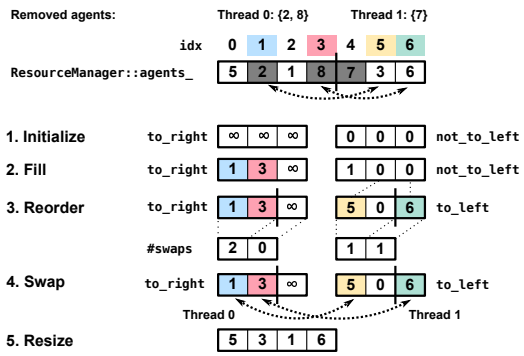


Figure 1. Parallel agent removal mechanism

Third, we partition the two auxiliary arrays into blocks corresponding to the total number of threads. A thread iterates over its auxiliary block, moves entries to the beginning if they indicate a swap, and stores a counter in the #swaps array. For the *to_right* array, the algorithm skips or overwrites elements with the value *UINT_MAX*. In this step, the semantic of the not_to_left array changes to to_left. Thus, the algorithm looks for all zeros in the array, replaces them with the value $array\_index + new\_size$, and moves them to the beginning of the block.

In the fourth step, we can finally perform the swaps. The algorithm calculates the prefix sum of the two *#swaps* arrays and partitions the swaps among all threads. Each thread can determine the indices based on the prefix sum of *#swaps*.

Lastly, the algorithm completes the removal by shrinking the vector to *new_size*. This algorithm requires $O(removed\_agents)$ time and space and parallelizes steps 1–4.

4 Optimize Memory Layout

4.1 NUMA-Aware Iteration

BioDynaMo supports systems with multiple NUMA domains. We add a mechanism to match threads with agents from the same NUMA domain to minimize the traffic to remote DRAM because OpenMP does not provide this functionality.

Figure 2 shows a server with two NUMA domains (`ND0`, `ND1`) corresponding to two CPUs with two threads each (T0 & T1, T2 & T3). The CPUs have a local DRAM with shorter memory access latency than the remote DRAM. The agents in the simulation are balanced between the two NUMA domains (see Section 4.2). The `ResourceManager` maintains a vector of agent pointers for each NUMA domain ❶. To iterate over agents in a NUMA-aware way, BioDynaMo first partitions these vectors into blocks of agent pointers of the same size ❷. Second, these blocks are partitioned among the threads from the matching NUMA domain ❸. Threads process the assigned blocks in parallel. Figure 2 shows processed blocks with a background color of the corresponding thread.

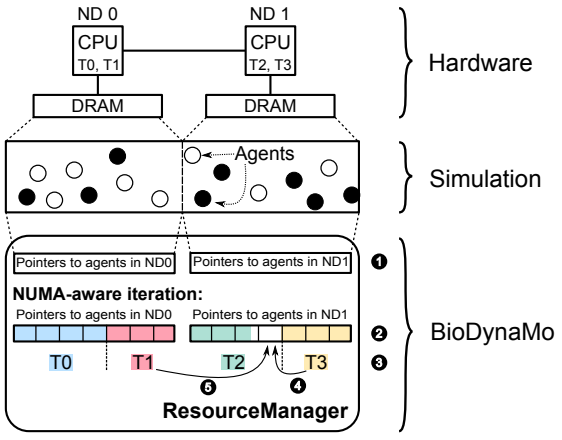


Figure 2. NUMA-aware iteration

We implement a two-level work-stealing mechanism to avoid imbalanced execution times across threads. First, a thread can steal a block from a different thread from the *same* NUMA domain (e.g., ❹). Second, if the thread's NUMA domain has already finished all work, the thread can steal work from a different NUMA domain (e.g., ❺).

4.2 Agent Sorting and Balancing

To accelerate the memory-bound simulations, we must increase the cache hit ratio and load balance the agents among NUMA domains to minimize remote DRAM accesses. In Section 4.1 and Figure 2, we assume this is already the case. This section presents an efficient algorithm to achieve this goal by sorting the agents' memory locations and preserving the neighborhood relations in 3D.

Preserving the neighborhood relation and reducing the dimensionality is the main characteristic of space-filling curves (e.g., Morton order [44] or Hilbert curve [26]). We compared the performance of the Morton order with the Hilbert curve using an oncological simulation [10] and observed a negligible performance improvement of 0.54% from using the Hilbert curve. Higher costs to decode the Hilbert curve offset small gains for the agent operations. Therefore, we use the Morton order because it results in simpler code.

Figure 3 and the following description present the algorithm in 2D space, but the same principles apply in 3D. In BioDynaMo, the neighborhood information is stored in the implementation of the environment interface. Since the uniform grid environment performs best, as shown in the evaluation section (Section 6.9), we utilize its characteristics to achieve fast sorting and balancing. We assume the following

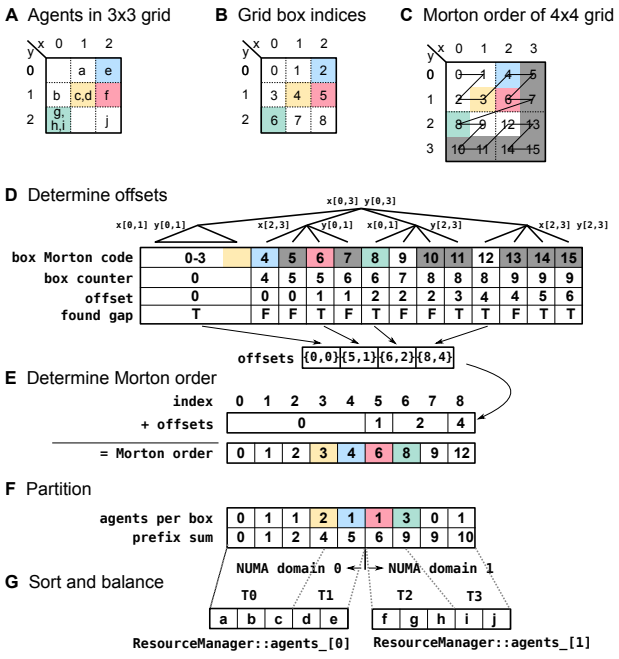


Figure 3. Agent sorting and balancing mechanism

scenario. Agents are stored in a 3×3 uniform grid (A). The simulation runs on a system with two NUMA domains and two threads per domain, resulting in four threads. The grid boxes are stored in a flattened array. Figure 3B shows the box indices for x and y coordinates. We apply a Morton order space-filling curve [44] on the uniform grid (C). Our goal is to sort the agents in the nine boxes in increasing Morton order. The Morton order is only contiguous for quadratic simulation spaces where the length is a power of two. Therefore, C shows a 4×4 grid. For the 3×3 simulation space, there are gaps between Morton code four and six, six and eight, and nine and twelve.

The algorithm comprises three main steps. First, the algorithm determines the sequence of boxes in Morton order (D, E). Second, the algorithm partitions the boxes into segments that balance agents among NUMA domains and threads (F). Third, the algorithm stores the agents in their new position in the resource manager (G). In Figure 3, we selected four boxes and colored them in red, green, blue, and yellow to quickly find the corresponding entries for contained agents, box index, and Morton code. The boxes outside the simulation space have a grey background.

In the first main step (D, E), we determine all gaps between grid boxes in simulation space (D) to avoid a costly sorting operation or iteration over all $N \times N$ boxes, where N is the next higher power of two of $max(x, y)$. We exploit the fact that the Morton order corresponds to a depth-first traversal of a quadtree, in which each box is a leaf in the tree. Leaves whose boxes are outside the simulation space are considered empty. Similarly, an inner node is empty if all of its corresponding leaves are outside the simulation space. If all the corresponding boxes of an inner node are inside the simulation space (i.e., the inner node has a perfect subtree), we say the inner node is complete. The quadtree is only an abstraction and does not need to be constructed. It is only necessary to store the current traversal path, which requires $O(log(\#boxes))$ space.

The mechanism in D uses three auxiliary variables: box counter, offsets, and `found_gap`, which are initialized to zero, zero, and true. The matrix in Figure 3D shows the three variables before the update of the current traversal step. The algorithm traverses the tree depth-first and continues to the next deeper tree level only if the current node is neither complete nor empty. In this case, the variables are not changed. If a complete inner node or leave inside the simulation space is found and the `found_gap` variable is true, the algorithm adds an entry with the current box counter and offset values in the offsets array and clears `found_gap`. Afterward, the box counter variable is incremented by the number of leaves in its subtree or one if it was a leave, irrespectively of the former value of `found_gap`. Empty nodes or leaves are handled similarly. The offset variable is incremented by the number of empty leaves in the subtree of an empty node or one if it is an empty leave. Additionally, the `found_gap` variable is set to true. The algorithm keeps track of the x and y intervals to calculate (in constant time) the number of leaves in a subtree and determine if an inner node is entirely, partially, or not inside the simulation space.

With the already sorted offsets array, the Morton order can be determined in linear time by iterating over all indices and adding the corresponding offset (E).

In the second main step (F), the algorithm iterates over all boxes in Morton order and fills an auxiliary array with the number of agents in each box. Afterward, the algorithm calculates the prefix sum of the auxiliary array in a parallel work-efficient manner [34] and partitions the total number of agents in the simulation such that each NUMA domain receives a share corresponding to its number of threads. Inside a NUMA domain, the agents are further partitioned such that each thread in this domain receives an equal share.

In the third main step (G), the threads copy the agents and store the pointer in the new position in the resource manager. The simulation engine can immediately free obsolete agents' memory or delete all old copies after the step is finished. The latter requires more memory but might improve performance due to a more optimal memory layout in conjunction with the BioDynaMo memory allocator (Section 4.3). The presented algorithm runs in $O(\#agents + \#boxes)$ time and space and parallelizes steps E–G.

4.3 BioDynaMo Memory Allocator

To improve the performance of the simulation engine, we present a custom dynamic memory allocator which improves the memory layout of the most frequently allocated objects: agents and behaviors. Our solution builds upon pool allocators due to their constant time allocation performance. Pool allocators divide a memory block into equal-sized elements and store pointers to free elements in a linked list.

We create multiple instances of these allocators because they can only return memory elements of one size. As a result, agents and behaviors with distinct sizes are separated and stored in a columnar way. We separate the pool allocator into multiple NUMA domains (class `NumaPoolAllocator`) to fully control where memory is allocated. The `NumaPoolAllocator` has a central free-list and thread-private ones to minimize synchronization between threads. List nodes, which correspond to free memory locations, can be migrated between thread private and the central list, which is essential to avoid memory leaks. Migrations are triggered if a thread-private list exceeds a specific memory threshold. Lists minimize these migrations, and thus thread synchronization, by maintaining additional skip lists. These skip lists support migrations of a large number of elements in constant time.

Memory is allocated in large blocks with exponentially increasing sizes controlled by the parameter `mem_mgr_growth_rate`. The initialization of these memory blocks, which includes list node generation, is performed on-demand in smaller segments to minimize the worst-case allocation time.

Every allocated memory block is divided into N-page aligned segments (Figure 4A), where N can be set with parameter `mem_mgr_aligned_pages_shift`. The linked list nodes are stored inside free memory elements and do not require extra space. At the beginning of each N-aligned segment, we write the pointer to the corresponding `NumaPoolAllocator` instance. Therefore, allocated memory elements can obtain this pointer in constant time, based on their memory address. This solution enables constant time deallocations (see Figure 4b) but wastes memory in three ways. First, memory blocks are allocated using `numa_alloc_onnode` (libnuma [5]). This function does not return N-page aligned pointers and causes unusable regions at the beginning and the end, which sum up to $N * page\_size$ bytes. Second, there might not be enough space to place a whole element at the end of an N-page aligned segment. Elements must not cross N-page aligned borders because it would overwrite the necessary metadata. Third, the metadata requires the size of a pointer, which is eight bytes on 64-bit hardware. The amount of wasted memory is bounded by the following equation: $N * page\_size + element\_size + metadata\_size$. Despite this memory overhead, our performance evaluation shows that the BioDynaMo pool allocator uses on average less memory than ptmalloc2 and jemalloc [19]. Another side effect of this design choice is that the allocation size is limited by $N * page\_size - metadata\_size$.

5 Omit Collision Force Calculation

The most time-consuming operation in the tissue models presented in Section 6.1 is the calculation of the displacement of agents based on all mechanical forces. For this purpose, the simulation engine has to calculate pairwise collision forces between agents and their neighbors implemented in the class `InteractionForce`. By default, BioDynaMo uses the force calculation method detailed in the Cortex3D paper [70]. We observe that simulations can contain a significant amount of regions where agents do not move. Neural development simulations, for example, might only have an active growth front, while the remaining part of the neuron is unchanged.

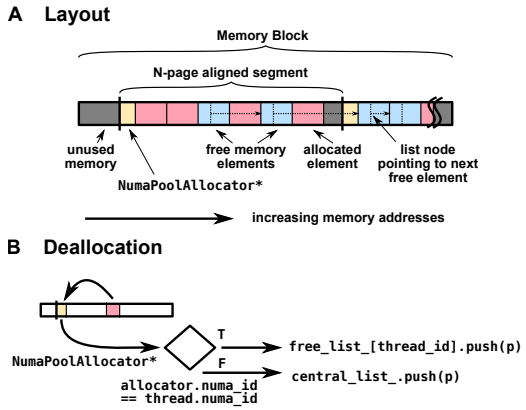


Figure 4. BioDynaMo's memory allocator

Therefore, we present a mechanism to detect agents for which it is safe to skip the expensive force calculation. We call these agents static. The following four conditions must be fulfilled in the last iteration: (i) the agent and none of its neighbors moved (ii) the agent's and neighbors' attributes did not change in a way that could increase the pairwise force (e.g., larger diameter), or the resulting displacement, (iii) new agents were not added within the interaction radius of the agent, and (iv) there is maximum one neighbor force which is non-zero. The detection mechanism is closely tied to the `InteractionForce` implementation (see [10]), as condition two implies, and might have to be adjusted if a different force implementation is used.

Condition four is needed because we want to allow agents to shrink and to be removed from the simulation without setting the agents in this region to not static. Consequently, we have to ensure that two or more neighbor forces did not cancel each other out in the previous iteration.

The simulation engine monitors if any of the conditions are violated for each agent and sets the affected agents to not static. In this process, a distinction has to be made whether the changed attribute affects only the current agent or also its neighbors. If, for example, a static agent moves, the agent and all its neighbors will be affected, while a change to the agent's force threshold, which must be exceeded to move the agent, only affects itself.

6 Evaluation

6.1 Benchmark Simulations

We use five simulations to evaluate the performance of the simulation engine: cell proliferation, cell clustering, and use cases in the domains of epidemiology, neuroscience, and oncology. These simulations use double-precision floating point variables and are described in detail in [10]. Table 1 shows that these simulations cover a broad spectrum of performance-related simulation characteristics and contain information about the number of agents, diffusion volumes, and iterations executed. We set the number of agents between two and 12.6 million to keep the total execution time of all benchmarks manageable. This is necessary due to the slow execution of the various baselines. In addition, Section 6.4 shows a benchmark in which each simulation is executed with one billion agents. Also the comparison with Biocellion contains a benchmark with 1.72 billion cells.

6.2 Experimental Setup and Reproducibility

All tests were executed in a Docker container with an Ubuntu 20.04 based image. Table 2 gives an overview of the main parameters of the three servers we used to evaluate the performance of BioDynaMo. If it is not explicitly mentioned, assume that System A was used to execute a benchmark.

We provide additional evaluations, detailed instructions to reproduce the results, all code, a self-contained docker image, and raw measurement data in the supplementary materials (https://doi.org/10.5281/zenodo.6463816).

Table 1. Performance-relevant simulation characteristics.

Characteristic	Cell prolif.	Cell clustering	Epidemiology	Neuroscience	Oncology
Create new agents during simulation	✗			✗	✗
Delete agents during simulation					✗
Agents modify neighbors				✗	
Load imbalance			✗	✗	
Agents move randomly			✗		✗
Simulation uses diffusion		✗		✗	
Simulation has static regions				✗	
Number of iterations	500	1000	1000	500	288
Number of agents (in millions)	12.6	2	10	9	10
Number of diffusion volumes	0	54m	0	65k	0

Table 2. Benchmark hardware

System	Memory	CPU	OS
A	504 GB	Four Intel(R) Xeon(R) E7-8890 v3 CPUs @ 2.50GHz with a total of 72 physical cores, two threads per core and four NUMA domains.	CentOS 7.9.2009
B	1008 GB		
C	62 GB	Two Intel(R) Xeon(R) E5-2683 v3 CPUs @ 2.00GHz with a total of 28 physical cores, two threads per core and two NUMA domains.	CentOS Stream 8

6.3 General Performance Metrics

We characterize the agent-based simulation workload by breaking down the operation's execution time, and exploring microarchitecture inefficiencies.

The following benchmarks were performed with all optimizations enabled. Figure 5 shows a breakdown of all operations in our benchmark simulations. The majority of the runtime is spent in agent operations (median: 76.3%) which subsumes, among others, the execution of behaviors, calculation of mechanical forces, discretization, and detection of static regions. Rebuilding the uniform grid environment at every time step is the second biggest runtime contributor, 4.09–36.5% (median: 18.0%). The epidemiology use case considers a wider environment that manifests itself in an increased update time. The average cost of agent sorting in its optimal setting (Figure 12) is 0.180%–6.33%. Since adding and removing agents is parallelized, iterations' setup and tear down consume only 2.66% (max) of the execution time.

In the microarchitecture analysis, we observe that the benchmark simulations are primarily memory-bound. We lose between 31.8 and 47.2% of processor pipeline slots because the operands are not available.

6.4 Runtime and Space Complexity

We analyze the runtime and memory consumption of BioDynaMo on System B by increasing the number of agents from 10^3 to 10^9 for each simulation (Figure 6). With one thousand agents, the execution time for one iteration is on average 1.21 ms and increases only slightly until 10^5 agents (2.80ms). From there on, runtime increases *linearly* to one billion agents in which one iteration takes between 6.41 and 38.1 seconds to execute. A similar trend can be observed

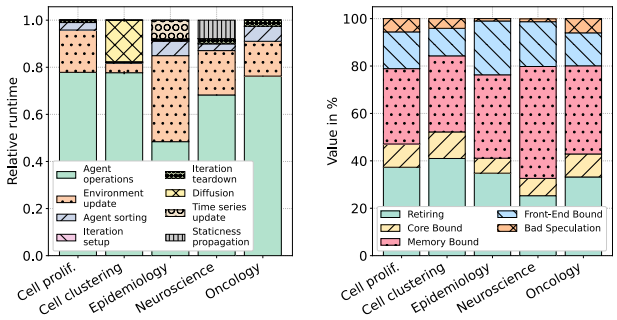


Figure 5. Operation runtime breakdown (left) and microarchitecture analysis (right)

for the memory consumption of BioDynaMo (using double-precision floating point values), which remains below 1.60 GB until 10^6 agents and increases *linearly* to a maximum between 245 and 564 GB.

The number of agents that BioDynaMo can simulate is *not* fundamentally limited to one billion. The maximum depends only on the available memory of the underlying hardware and the tolerable execution time.

6.5 Comparison with Biocellion

We compare BioDynaMo with Biocellion [31], an agent-based framework for tissue models optimized for performance. We implement the cell sorting simulation presented in the Biocellion paper (Section 3.1) in BioDynaMo and use identical model parameters. The visualization of the BioDynaMo simulation with 50k cells (Figure 7a) demonstrates a good agreement with the Biocellion results in Fig. 3a in [31].

Since we do not have access to the Biocellion code, because it is proprietary software, we compare BioDynaMo to the performance results provided in [31]. First, we replicate the benchmark with 26.8 million agents using 16 CPU cores. For Biocellion, Khang et al. [31] used a system with two Intel Xeon E5-2670 CPUs with 2.6 GHz. We execute BioDynaMo on System C with a comparable CPU and limit the number of CPU cores to 16 to ensure a fair comparison. We observe that BioDynaMo is 4.14× faster than Biocellion. BioDynaMo executes one iteration in 1.80s (averaged over 500 iterations), while Biocellion requires 7.48s.

Second, we consider the Biocellion benchmark in which Kang et al. executed 1.72 billion cells on a cluster with 4096 CPU cores (128 nodes with two AMD Opteron 6271 Interlago 2.1 GHz CPUs per node). We execute the BioDynaMo simulation with the same number of cells on a *single* node (System B). Although BioDynaMo requires 26.3s per iteration, which is 5.90× slower than Biocellion, BioDynaMo uses 56.9× fewer CPU cores. Therefore, we conclude that the performance per CPU core of BioDynaMo is 9.64× more efficient than Biocellion. We repeat the experiment with 281.4 million cells to verify the last observation. Biocellion requires 4.37s per iteration (extracted from Figure 3b in [31]) using 21

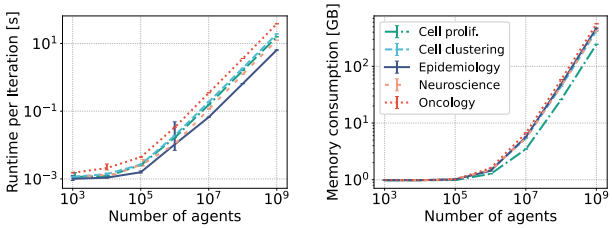


Figure 6. Runtime per iteration and memory consumption analysis as the number of agents varies from 10^3 to 10^9

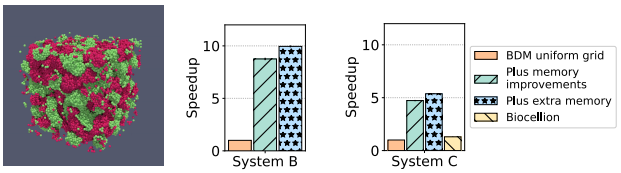


Figure 7. (a) Final simulation state after executing the Biocellion cell sorting model on BioDynaMo. (b) Performance evaluation of the BioDynaMo optimizations with a model with 28.6 million cells on System B (left) and System C limited to 16 physical CPU cores (right). The Biocellion paper [31] provides only a measurement for the latter benchmark.

nodes with a total of 672 CPU cores. The BioDynaMo simulation on System B with 72 CPU cores runs in almost identical 4.24s per iteration. This result confirms our observation that BioDynaMo is an order of magnitude more efficient than Biocellion.

We evaluate the impact of our optimizations to provide insights into the question of why BioDynaMo processes 4.14× more agents per CPU core in the first benchmark and 9.64× in the second. Therefore, we execute the relevant optimizations with 26.8 million cells on System C limited to 16 CPU cores and System B with 72 CPU cores. Figure 7b shows that the difference can largely be explained by the memory optimizations having a more significant impact on machines with higher CPU core count.

6.6 Comparison with Cortex3D and NetLogo

We also compare with capable single-thread tools to evaluate the parallel overhead of the BioDynaMo implementation [40]. We choose Cortex3D [70] due to its similarity with the neuroscience features of BioDynaMo and select NetLogo [68] as a representative for an easy-to-use general-purpose tool. We extend the experiments from [10] by analyzing the impact of the presented performance improvements and comparing the memory consumption. This benchmark uses different simulation parameters for agents, diffusion volumes, and iterations, than shown in Table 1. The first four benchmarks in Figure 8 are small-scale benchmarks using between 2k and 30k agents and 0–128k diffusion volumes. These benchmarks run for 100–1000 iterations and only use one thread because Cortex3D and NetLogo are not parallelized. The "epidemiology (medium-scale)" benchmark contains 100k agents and uses 144 threads. NetLogo only benefits from parallel garbage collection in this scenario. In the "BioDynaMo standard implementation", all optimizations are turned off, and the kd-tree environment is used.

We make the following observations. For the small-scale simulations using one thread, BioDynaMo achieves a speedup of up to 78.8× while using 2.49× less memory. We observe three orders of magnitude speedup and two orders of magnitude reduction in memory consumption for the medium-scale benchmark in which all threads were used.

The median speedup of the BioDynaMo standard implementation is 15.5×. The optimized uniform grid of BioDynaMo boosts performance in all benchmarks (median: 2.18×) but has the most significant impact if parallelization is used (45.5×). Memory layout optimizations improve the runtime of medium-scale simulations by 26.2%, but not for small-scale ones. The memory layout optimizations comprise the NUMA-aware iteration (Section 4.1), agent sorting and balancing (Section 4.2), and memory allocator (Section 4.3). Due to the interdependency between these individual optimizations, we subsumed them into one category. Similarly, extra memory usage during the agent sorting and balancing stage (Section 4.2) has only a slight performance impact (median speedup: 4.82%). However, the static region optimization dramatically improves the performance in the neuroscience use case (speedup 9.22×). Although the mechanism's overhead reduces the speedup for simulations without static regions, this is not problematic. The modeler usually knows this characteristic a priori and only enables the mechanism if static regions are expected (see parameter `detect_static_agents`).

6.7 Optimization Overview

We assess the performance of the presented optimizations using larger-scale simulations (Table 1) by enabling optimizations step-by-step (Figure 9). The baseline in this comparison is the BioDynaMo standard implementation introduced in Section 6.6. We make the following observations.

The BioDynaMo optimizations improve overall performance between 33.1× and 524× (median: 159×). These benchmarks confirm the speedup of BioDynaMo's optimized uniform grid that we observed in comparison with Cortex3D and NetLogo. For these larger-scale simulations, the magnitude of the speedup increases up to 184× with a median of 27.4×. A similar observation can be made for the static region detection mechanism, albeit with reduced magnitude (speedup: 3.22×). The main difference between the comparison with Cortex3D and NetLogo and this benchmark is the impact of the memory layout optimizations of agents and behaviors and the usage of extra memory during agent sorting. The maximum speedup is up to 5.30× (median: 2.96×) and up to 2.07× (median: 1.09×), respectively. Only the Biocellion benchmark in Figure 7b shows a bigger impact.

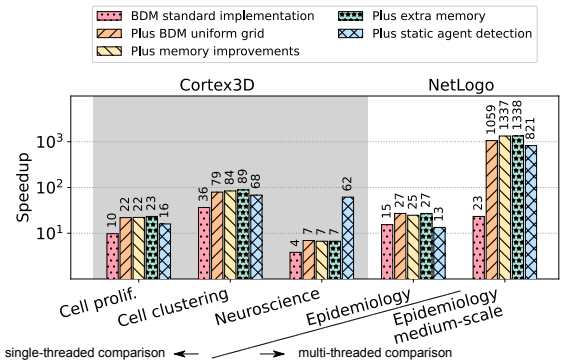


Figure 8. Performance comparison with Cortex3D and NetLogo after the optimizations are progressively switched on.

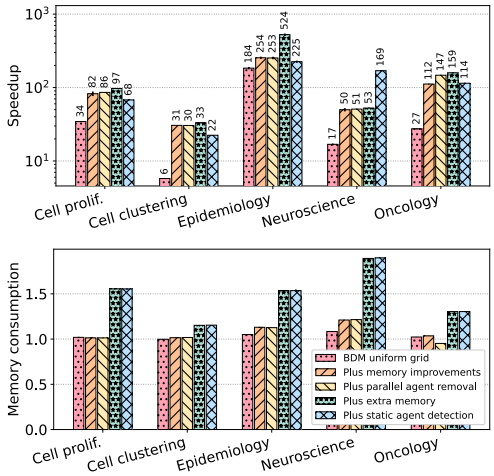


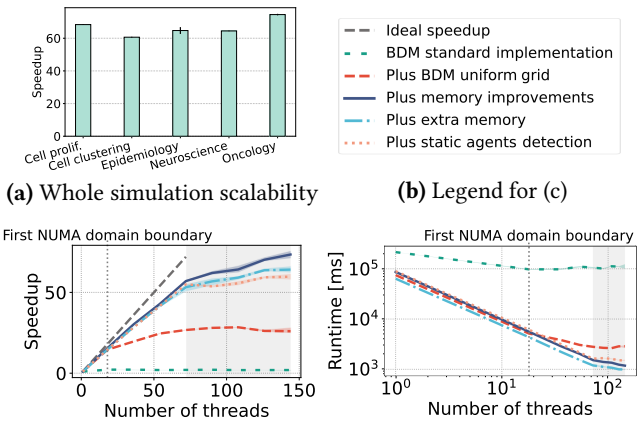
Figure 9. Speedup (top) and memory consumption (bottom) compared with the BioDynaMo standard implementation after the optimizations are progressively switched on.

The simulation time of the oncology use case, the only benchmark that removes agents from the simulation is reduced by 31.7% using the "parallel removal" optimization described in Section 3.2. The optimizations increase the median memory consumption by a mere 1.77%, which increases to 55.6% by enabling the use of extra memory during sorting.

6.8 Scalability

We evaluate the scalability of BioDynaMo using the complete simulations lasting between 288 and 1000 iterations and perform a strong scaling analysis with different optimizations enabled. The strong scaling analysis is performed with ten iterations.

Figure 10a illustrates the excellent scalability of BioDynaMo for complete simulations (i.e., executing all iterations). The speedup using 72 physical cores with hyperthreading enabled is between 60.7× and 74.0× (median 64.7×) compared to serial execution. Section 6.6 shows that BioDynaMo with one CPU core is more than 23× faster than Cortex3D. If we combine this result with the scalability analysis, which



(a) Whole simulation scalability **(b)** Legend for (c)

(c) Cell proliferation

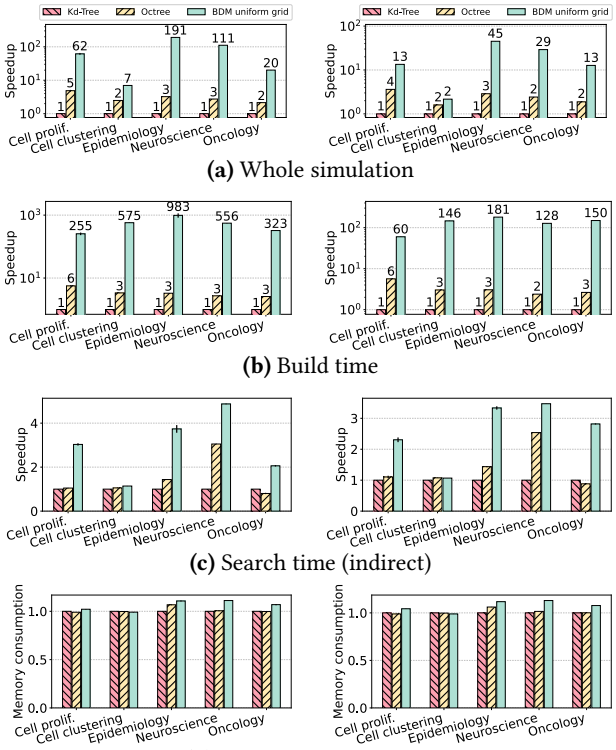
Figure 10. (a) Simulation scalability using the whole simulation. (b–c) Detailed strong scaling analysis using only ten time steps. The left plot shows the speedup with respect to a single-thread execution, while the right plot presents the total runtime. Plots for the remaining simulations can be found in the supplementary materials.

shows that BioDynaMo with 72 CPU cores is more than 60× faster than one CPU core, we can conclude that BioDynaMo is up to three orders of magnitude faster than Cortex3D.

Figure 10c shows the strong scaling analysis for the cell proliferation simulation with ten iterations after progressively switching on the presented optimizations. The left plot shows the speedup with respect to a single-thread execution, and the right plot presents the average runtime in milliseconds to highlight the absolute differences between various optimizations and the reduction in runtime with increasing threads. We make the following observations. The BioDynaMo standard implementation scales poorly due to the serial build of the kd-tree environment, which is improved considerably by using BioDynaMo's optimized uniform grid (Section 3.1). The memory optimizations (Section 4) fully achieve their desired effect and allow BioDynaMo to scale across NUMA domains and high CPU-core counts.

6.9 Neighbor Search Algorithm Comparison

Figure 11 compares three different neighbor search algorithms: BioDynaMo's uniform grid, UniBN's octree [8], and the kd-tree from nanoflann [9]. To ensure a fair comparison, we turned off agent sorting for all algorithms because it is currently only implemented for the uniform grid. We validate our choice for the octree bucket size and nanoflann depth parameter and observe that the used parameters are within 4.20% of the optimum runtime. The left column in Figure 11 shows the result for four NUMA domains and 144 threads, while the right column shows results for one NUMA domain and 18 threads. We analyzed four properties of these radial neighbor search methods: runtime impact on the whole simulation, build and search time of the index, and memory consumption (Figure 11). We measure the search



(a) Whole simulation

(b) Build time

(c) Search time (indirect)

(d) Memory consumption

Figure 11. Neighbor search algorithm comparison (left column: four NUMA domains and 144 threads, right column: one NUMA domain and 18 threads).

time indirectly by comparing the agent operation runtimes. This operation contains the initial neighbor searches and thus provides information on how fast searches are executed.

The BioDynaMo uniform grid implementation shows its benefits not only in the pure build time comparison but also in the full simulation analysis. Although a significant build time difference in comparison to the kd-tree and octree is expected (because the build process is serial), the magnitude between 255 and 983× on four NUMA domains is surprising. The uniform grid outperforms the other algorithms also during the search stage for all simulations.

Simulations using BioDynaMo's uniform grid implementation are up to 191× faster than the kd-tree implementation while consuming only 11% more memory in the worst case.

6.10 Agent Sorting and Balancing

This section evaluates the impact of agent sorting and balancing (Section 4.2) on the simulation runtime for one and four NUMA domains. To this extent, we perform a parameter study with varying agent sorting frequencies for each simulation. Figure 12 shows the speedup for four NUMA domains (left) and one NUMA domain (right). The baselines in both cases are simulations without agent sorting. An agent sorting frequency of one means that the operation is executed in every iteration; similarly, a frequency of ten would mean that the operation is executed every ten iterations.

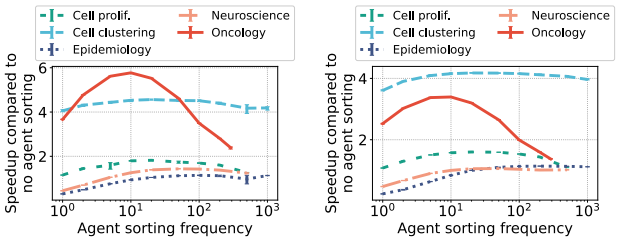


Figure 12. Agent sorting and balancing speedup for different execution frequencies (left: four NUMA domains and 144 threads, right: one NUMA domain and 18 threads).

Load balancing of agents among NUMA domains greatly impacts performance even on systems without NUMA architecture. This stems from the fact that agent sorting aligns agents that are close in space also in memory.

The oncology and cell clustering simulations benefit most of this performance improvement (peak speedup of 5.77 and 4.56× for four NUMA domains). Both simulations are initialized with a random distribution of agents. Although the epidemiology simulation is also initialized randomly, its agents also move randomly with large distances between iterations. This behavior reduces the alignment improvements significantly (peak speedup 1.14× for four NUMA domains). The cell proliferation simulation is initialized with a 3D grid of cells, which improves the alignment compared to the worst-case random initialization. Therefore, the maximum obtained speedup is reduced to 1.82× (four NUMA domains). Suppose we change the initialization of the cell proliferation simulation to random, the maximum speedup increases to 4.68×. This optimization performs below average for the neuroscience simulation. This simulation only has an active growth front, while the remaining part remains static. The static agent detection mechanism exploits this fact and avoids calculating mechanical forces for the static regions. Therefore, the number of neighbor accesses is significantly reduced, and thus the benefits of aligned agents. If static region detection is disabled, agent sorting and balancing improve the runtime by 3.80× at a frequency of 20.

7 Related Work

ABM tools: To our knowledge, this is the first paper to present an agent-based simulation engine capable of simulating neuroscientific models with billions of agents. Biocellion [31] and Timothy [14] can also simulate one billion agents, but neither of them support simulations in the demanding computational neuroscience domain. Our performance evaluation shows that BioDynaMo is 9.64× more efficient than Biocellion [31] for a simulation with 1.72 billion agents. Furthermore, BioDynaMo is three orders of magnitude faster than the popular neuroscientific simulator Cortex3D [70], and the general-purpose tool NetLogo [68] on our benchmark hardware. Other tools [20, 42, 55] also support large-scale models, but only show simulations of up to 10^6 agents.

Memory layout optimizations: Data movement between main memory and processor cores is a fundamental bottleneck in today's computing systems [46]. Recent research in computer architecture explores new approaches to address this bottleneck, such as processing-in-memory, i.e., placing compute capability closer to the data [2, 23]. Agent-based simulation tools are also negatively impacted by the data movement bottleneck. We address this problem in software with a better memory layout resulting in more efficient bandwidth utilization and data reuse in caches. Space-filling curves [26, 44, 52] can improve the memory layout by aligning objects that are close in 3D space. Therefore, these curves are frequently used to optimize geometric data structures [6, 8] and molecular dynamics simulations [4, 21, 47]. To our knowledge, none of the other agent-based simulation frameworks (e.g., [11, 13, 15, 17, 20, 31, 32, 35, 36, 39, 42, 49, 55, 59, 61, 66, 68, 70]) use space-filling curves to improve the cache hit rate and minimize the amount of remote DRAM accesses. We introduce this proven technique to the agent-based workload and present a mechanism to determine the Morton order of a non-cubic grid in linear time.

Neighbor search: Agent-based simulation platforms use various neighbor search algorithms: Delaunay triangulation [70], octree [24], and grid-based approaches [31, 55, 68]. Grids are also commonly used on the GPU [1, 8, 16, 22, 27]. Depending on the dataset and specific search query ([fixed-]radius neighbor search or k-nearest neighbors), the literature recommends different algorithms [8, 67]. Our contribution lies in the efficient implementation and integration of the uniform grid into the simulation engine and in providing insights into the performance differences for the agent-based workload.

Performance evaluation: To our knowledge, this paper presents the most comprehensive performance analysis of an agent-based simulation platform. Existing platforms report only limited performance results, including simulation execution times and occasionally scalability analyses [12, 20, 31, 36, 42, 70]. Performance data can also be found in model papers [45, 60] and in works that focus on hardware accelerators [69]. We improve upon these works by providing an in-depth analysis of each performance-relevant component. Efforts in the direction of a standard agent-based benchmark have been made by Moreno et al. [43] and Rousset et al. [57]. However, these synthetic benchmarks fall short of representing a realistic range of agent-based simulations by over-simplifying memory access patterns and assuming that agents always move randomly. Compared to these, our benchmark simulations cover a broader spectrum of performance relevant simulation metrics (see Table 1).

Comparison outside the ABM field: Other particle-based applications, such as molecular dynamics (MD) [33, 53, 64], astrophysics (AP) [58], or computational fluid dynamic (CFD) [30] simulations often face similar computational challenges to improve the performance of large-scale simulations. LAMMPS [64], for example, also uses a grid-based structure to determine neighbors. While LAMMPS stores neighbor lists for each atom, which according to Thompson et al. [64] "consumes the most memory of any data structure in LAMMPS", BioDynaMo does not need these lists and therefore saves memory. BioDynaMo improves over LAMMPS and VASP [33] by sorting agents using a space-filling curve (Section 4.2) and using a custom memory allocator (Section 4.3) to reduce the memory access latency. A NUMA-aware thread allocation mechanism, as the one used in BioDynaMo (Section 4.1), is not needed in LAMMPS or VASP because both tools support distributed parallelism with MPI. In this work, we identify several computational challenges in ABM, which we tackle by using methods inspired by MD, AP, and CFD. The main difference between ABM and other particle-based applications is that the computations can vary significantly from each other in terms of arithmetic intensity, the number of considered neighbors, data access patterns, and more, thus posing diverse computational challenges.

8 Conclusion and Future Work

This paper presents a novel agent-based simulation engine optimized for high performance and scalability. BioDynaMo enables not only larger-scale simulations, but also helps researchers of small scale studies with accelerated parameter space exploration, and faster iterative development.

We identify general agent-based performance challenges and provide six solutions to maximize parallelization, reduce memory access latency and data transfers, and avoid unnecessary work. These solutions are transferable and can be used to accelerate other agent-based simulation tools.

We present a comprehensive performance analysis to provide insights into the agent-based workload and to give our users a better understanding of BioDynaMo's capabilities. We find that on our system, the presented optimizations improve performance up to 524× (median 159×) and allow BioDynaMo to scale to 72 physical processor cores with a parallel efficiency of 91.7%. A comparison with state-of-the-art tools shows that BioDynaMo is up to three orders of magnitude faster. These performance characteristics enable simulations with billions of agents, as demonstrated.

Our performance optimizations, which are effective on machines with one or more NUMA domains, are an important stepping stone towards a distributed simulation engine with a hybrid MPI/OpenMP design. Ongoing work focuses on realizing this distributed simulation engine capable of dividing the computation among multiple nodes to push the boundaries of agent-based simulation even further.

Acknowledgments

We want to thank the CERN Knowledge Transfer office for supporting this work. We acknowledge the support provided to the SAFARI Research Group by our industrial partners, including Huawei, Intel, Microsoft, and VMware.

References

[1] Brandon G. Aaby, Kalyan S. Perumalla, and Sudip K. Seal. 2010. Efficient simulation of agent-based models on multi-GPU and multi-core clusters. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools '10)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 1–10. https://doi.org/10.4108/ICST.SIMUTOOLS2010.8822

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 336–348. https://doi.org/10.1145/2749469.2750385

[3] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '67 (Spring))*. ACM, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[4] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* 227, 10 (May 2008), 5342–5359. https://doi.org/10.1016/j.jcp.2008.01.047

[5] Andi Kleen. 2007. libnuma. https://www.man7.org/linux/man-pages/man3/numa.3.html.

[6] Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welzl, and Peter Widmayer. 1997. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science* 181, 1 (July 1997), 3–15. https://doi.org/10.1016/S0304-3975(96)00259-9

[7] Frederico A. C. Azevedo, Ludmila R. B. Carvalho, Lea T. Grinberg, José Marcelo Farfel, Renata E. L. Ferretti, Renata E. P. Leite, Wilson Jacob Filho, Roberto Lent, and Suzana Herculano-Houzel. 2009. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *The Journal of Comparative Neurology* 513, 5 (April 2009), 532–541. https://doi.org/10.1002/cne.21974

[8] Jens Behley, Volker Steinhage, and Armin B. Cremers. 2015. Efficient radius neighbor search in three-dimensional point clouds. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, New York, NY, USA, 3625–3630. https://doi.org/10.1109/ICRA.2015.7139702

[9] Jose Luis Blanco and Pranjal Kumar Rai. 2014. nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann.

[10] Lukas Breitwieser, Ahmad Hesam, Jean de Montigny, Vasileios Vavourakis, Alexandros Iosif, Jack Jennings, Marcus Kaiser, Marco Manca, Alberto Di Meglio, Zaid Al-Ars, Fons Rademakers, Onur Mutlu, and Roman Bauer. 2021. BioDynaMo: a modular platform for high-performance agent-based simulation. *Bioinformatics* 38, 2 (09 2021), 453–460. https://doi.org/10.1093/bioinformatics/btab649

[11] Nicholson Collier and Michael North. 2011. Repast HPC: A Platform for Large-Scale Agent-Based Modeling. In *Large-Scale Computing*. John Wiley & Sons, Inc., New York, NY, USA, 81–109. https://doi.org/10.1002/9781118130506.ch5 Section: 5 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118130506.ch5.

[12] Nicholson Collier and Michael North. 2013. Parallel agent-based simulation with Repast for High Performance Computing. *SIMULATION* 89, 10 (Oct. 2013), 1215–1235. https://doi.org/10.1177/0037549712462620 Publisher: SAGE Publications Ltd STM.

[13] Gennaro Cordasco, Carmine Spagnuolo, and Vittorio Scarano. 2016. Toward the New Version of D-MASON: Efficiency, Effectiveness and Correctness in Parallel and Distributed Agent-Based Simulations. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, New York, NY, USA, 1803–1812. https://doi.org/10.1109/IPDPSW.2016.52

[14] M. Cytowski and Z. Szymanska. 2014. Large-Scale Parallel Simulations of 3D Cell Colony Dynamics. *Computing in Science Engineering* 16, 5 (Sept. 2014), 86–95. https://doi.org/10.1109/MCSE.2014.2

[15] M. Cytowski and Z. Szymanska. 2015. Large-Scale Parallel Simulations of 3D Cell Colony Dynamics: The Cellular Environment. *Computing in Science Engineering* 17, 5 (Sept. 2015), 44–48. https://doi.org/10.1109/MCSE.2015.66

[16] Roshan M. D'Souza, Mikola Lysenko, Simeone Marino, and Denise Kirschner. 2009. Data-Parallel Algorithms for Agent-Based Model Simulation of Tuberculosis on Graphics Processing Units. In *Proceedings of the 2009 Spring Simulation Multiconference* (San Diego, California) *(SpringSim '09)*. Society for Computer Simulation International, San Diego, CA, USA, Article 21, 12 pages.

[17] Thierry Emonet, Charles M. Macal, Michael J. North, Charles E. Wickersham, and Philippe Cluzel. 2005. AgentCell: a digital single-cell assay for bacterial chemotaxis. *Bioinformatics* 21, 11 (June 2005), 2714–2721. https://doi.org/10.1093/bioinformatics/bti391

[18] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up*. Brookings Institution Press, Washington D.C., USA. Google-Books-ID: xXvelSs2caQC.

[19] Jason Evans. 2011. Scalable memory allocation using jemalloc. http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919.

[20] Ahmadreza Ghaffarizadeh, Randy Heiland, Samuel H. Friedman, Shannon M. Mumenthaler, and Paul Macklin. 2018. PhysiCell: An open source physics-based cell simulator for 3-D multicellular systems. *PLOS Computational Biology* 14, 2 (Feb. 2018), e1005991. https://doi.org/10.1371/journal.pcbi.1005991

[21] John. M. A. Grime and Gregory A. Voth. 2014. Highly Scalable and Memory Efficient Ultra-Coarse-Grained Molecular Dynamics Simulations. *Journal of Chemical Theory and Computation* 10, 1 (Jan. 2014), 423–431. https://doi.org/10.1021/ct400727q Publisher: American Chemical Society.

[22] Julian Gross, Marcel Köster, and Antonio Krüger. 2019. Fast and Efficient Nearest Neighbor Search for Particle Simulations. In *Computer Graphics and Visual Computing (CGVC)*, Franck P. Vidal, Gary K. L. Tam, and Jonathan C. Roberts (Eds.). The Eurographics Association, Eindhoven, The Netherlands, 55–63. https://doi.org/10.2312/cgvc.20191258

[23] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022), 52565–52608. https://doi.org/10.1109/ACCESS.2022.3174101

[24] Andreas Hauri. 2013. *Self-construction in the context of cortical growth*. Ph. D. Dissertation. ETH Zurich. https://doi.org/10.3929/ethz-a-009997273

[25] Ahmad Hesam, Lukas Breitwieser, Fons Rademakers, and Zaid Al-Ars. 2021. GPU Acceleration of 3D Agent-Based Biological Simulations. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, New York, NY, USA, 210–217. https://doi.org/10.1109/IPDPSW52791.2021.00040

[26] David Hilbert. 1891. Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Math. Ann.* 38, 3 (Sept. 1891), 459–460. https://doi.org/10.1007/BF01199431

[27] Rama C Hoetzlein. 2014. Fast fixed-radius nearest neighbors: interactive million-particle fluids. https://raw.githubusercontent.com/binaryfoundry/nnsearch/master/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf. Accessed: November 30, 2022.

[28] Elizabeth Hunter, Brian Mac Namee, and John D. Kelleher. 2017. A Taxonomy for Agent-Based Models in Human Infectious Disease Epidemiology. *Journal of Artificial Societies and Social Simulation* 20, 3 (2017), 2.

[29] The MathWorks Inc. 2022. MATLAB. https://www.mathworks.com/products/matlab.html. Accessed: November 30, 2022.

[30] Hrvoje Jasak. 2009. OpenFOAM: Open source CFD in research and industry. *International Journal of Naval Architecture and Ocean Engineering* 1, 2 (Dec. 2009), 89–94. https://doi.org/10.2478/IJNAOE-2013-0011

[31] Seunghwa Kang, Simon Kahan, Jason McDermott, Nicholas Flann, and Ilya Shmulevich. 2014. Biocellion: accelerating computer simulation of multicellular biological system models. *Bioinformatics* 30, 21 (Nov. 2014), 3101–3108. https://doi.org/10.1093/bioinformatics/btu498

[32] Randal A. Koene, Betty Tijms, Peter van Hees, Frank Postma, Alexander de Ridder, Ger J. A. Ramakers, Jaap van Pelt, and Arjen van Ooyen. 2009. NETMORPH: A Framework for the Stochastic Generation of Large Scale Neuronal Networks With Realistic Neuron Morphologies. *Neuroinformatics* 7, 3 (Sept. 2009), 195–210. https://doi.org/10.1007/s12021-009-9052-3

[33] G. Kresse and J. Hafner. 1993. Ab initio molecular dynamics for open-shell transition metals. *Phys. Rev. B* 48 (Nov 1993), 13115–13118. Issue 17. https://doi.org/10.1103/PhysRevB.48.13115

[34] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (Oct. 1980), 831–838. https://doi.org/10.1145/322217.322232

[35] Laurent A. Lardon, Brian V. Merkey, Sónia Martins, Andreas Dötsch, Cristian Picioreanu, Jan-Ulrich Kreft, and Barth F. Smets. 2011. iDynoMiCS: next-generation individual-based modelling of biofilms. *Environmental Microbiology* 13, 9 (Sept. 2011), 2416–2434. https://doi.org/10.1111/j.1462-2920.2011.02414.x

[36] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. 2005. MASON: A Multiagent Simulation Environment. *SIMULATION* 81, 7 (July 2005), 517–527. https://doi.org/10.1177/0037549705058073 Publisher: SAGE Publications Ltd STM.

[37] C. Macal and M. North. 2014. Introductory tutorial: Agent-based modeling and simulation. In *Proceedings of the Winter Simulation Conference 2014*. IEEE, New York, NY, USA, 6–20. https://doi.org/10.1109/WSC.2014.7019874 ISSN: 1558-4305.

[38] Simeone Marino, Ian B. Hogue, Christian J. Ray, and Denise E. Kirschner. 2008. A methodology for performing global uncertainty and sensitivity analysis in systems biology. *Journal of Theoretical Biology* 254, 1 (2008), 178–196. https://doi.org/10.1016/j.jtbi.2008.04.011

[39] Antoni Matyjaszkiewicz, Gianfranco Fiore, Fabio Annunziata, Claire S. Grierson, Nigel J. Savery, Lucia Marucci, and Mario di Bernardo. 2017. BSim 2.0: An Advanced Agent-Based Cell Simulator. *ACS Synthetic Biology* 6 (June 2017), 1969–1972. https://doi.org/10.1021/acssynbio.7b00121

[40] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[41] John Metzcar, Yafei Wang, Randy Heiland, and Paul Macklin. 2019. A Review of Cell-Based Computational Modeling in Cancer Biology. *JCO Clinical Cancer Informatics* 3, 3 (Feb. 2019), 1–13. https://doi.org/10.1200/CCI.18.00069 Publisher: Wolters Kluwer.

[42] Gary R. Mirams, Christopher J. Arthurs, Miguel O. Bernabeu, Rafel Bordas, Jonathan Cooper, Alberto Corrias, Yohan Davit, Sara-Jane Dunn, Alexander G. Fletcher, Daniel G. Harvey, Megan E. Marsh, James M. Osborne, Pras Pathmanathan, Joe Pitt-Francis, James Southern, Nejib Zemzemi, and David J. Gavaghan. 2013. Chaste: An Open Source C++ Library for Computational Physiology and Biology. *PLOS Computational Biology* 9, 3 (March 2013). https://doi.org/10.1371/journal.pcbi.1002970

[43] Andreu Moreno, Juan J. Rodríguez, Daniel Beltrán, Anna Sikora, Josep Jorba, and Eduardo César. 2019. Designing a benchmark for the performance evaluation of agent-based simulation applications on HPC. *The Journal of Supercomputing* 75, 3 (March 2019), 1524–1550. https://doi.org/10.1007/s11227-018-2688-8

[44] Guy M Morton. 1966. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York.

[45] John T. Murphy, Elif Seyma Bayrak, Mustafa Cagdas Ozturk, and Ali Cinar. 2016. Simulating 3-D bone tissue growth using repast HPC: Initial simulation design and performance results. In *2016 Winter Simulation Conference (WSC)*. IEEE, New York, NY, USA, 2087–2098. https://doi.org/10.1109/WSC.2016.7822252 ISSN: 1558-4305.

[46] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems* 67 (June 2019), 28–41. https://doi.org/10.1016/j.micpro.2019.01.009

[47] A. Nakano, R.K. Kalia, and P. Vashishta. 1999. Scalable molecular-dynamics, visualization, and data management algorithms for materials simulations. *Computing in Science Engineering* 1, 5 (Sept. 1999), 39–47. https://doi.org/10.1109/5992.790586 Conference Name: Computing in Science Engineering.

[48] Muaz Niazi and Amir Hussain. 2009. Agent-based tools for modeling and simulation of self-organization in peer-to-peer, ad hoc, and other complex networks. *IEEE Communications Magazine* 47, 3 (March 2009), 166–173. https://doi.org/10.1109/MCOM.2009.4804403 Conference Name: IEEE Communications Magazine.

[49] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling* 1, 1 (March 2013), 3. https://doi.org/10.1186/2194-3206-1-3

[50] Nvidia. 2020. A100 Tensor Core GPU architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. Accessed: November 30, 2022.

[51] OpenMP Architecture Review Board. 2015. OpenMP Application Program Interface Version 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[52] G. Peano. 1890. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.* 36, 1 (March 1890), 157–160. https://doi.org/10.1007/BF01199438

[53] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. 2005. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 26, 16 (Dec. 2005), 1781–1802. https://doi.org/10.1002/jcc.20289

[54] William Rand and Roland T. Rust. 2011. Agent-based modeling in marketing: Guidelines for rigor. *International Journal of Research in Marketing* 28, 3 (Sept. 2011), 181–193. https://doi.org/10.1016/j.ijresmar.2011.04.002

[55] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. 2010. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics* 11, 3 (May 2010), 334–347. https://doi.org/10.1093/bib/bbp073

[56] Paul A. Roberts, Eamonn A. Gaffney, Philip J. Luthert, Alexander J. E. Foss, and Helen M. Byrne. 2016. Mathematical and computational models of the retina in health, development and disease. *Progress in Retinal and Eye Research* 53 (July 2016), 48–69. https://doi.org/10.1016/j.preteyeres.2016.04.001

[57] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. 2016. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review* 22 (Nov. 2016), 27–46. https://doi.org/10.1016/j.cosrev.2016.08.001

[58] Victor-Lucian Spiridon and Emil-Ioan Slusanschi. 2013. N-Body Simulations with GADGET-2. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, New York, NY, USA, 526–533. https://doi.org/10.1109/SYNASC.2013.75

[59] Jörn Starruß, Walter de Back, Lutz Brusch, and Andreas Deutsch. 2014. Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology. *Bioinformatics* 30, 9 (May 2014), 1331–1332. https://doi.org/10.1093/bioinformatics/btt772

[60] Peter Strazdins and Markus Hegland. 2011. Performance Analysis of a Cardiac Simulation Code Using IPM. In *Proceedings of the Second Workshop on Scalable Algorithms for Large-scale Systems (ScalA '11)*. ACM, New York, NY, USA, 29–32. https://doi.org/10.1145/2133173.2133186

[61] Maciej H. Swat, Gilberto L. Thomas, Julio M. Belmonte, Abbas Shirinifard, Dimitrij Hmeljak, and James A. Glazier. 2012. Chapter 13 - Multi-Scale Modeling of Tissues Using CompuCell3D. In *Methods in Cell Biology*, Anand R. Asthagiri and Adam P. Arkin (Eds.). Computational Methods in Cell Biology, Vol. 110. Academic Press, Amsterdam, The Netherlands, 325–366. https://doi.org/10.1016/B978-0-12-388403-9.00013-8

[62] R Core Team. 2013. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. http://www.r-project.org/. Accessed: November 30, 2022.

[63] Leigh Tesfatsion. 2006. Chapter 16 Agent-Based Computational Economics: A Constructive Approach to Economic Theory. In *Handbook of Computational Economics*, L. Tesfatsion and K. L. Judd (Eds.). Vol. 2. Elsevier, Amsterdam, The Netherlands, 831–880. https://doi.org/10.1016/S1574-0021(05)02016-2

[64] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. 2022. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications* 271 (Feb. 2022), 108–171. https://doi.org/10.1016/j.cpc.2021.108171

[65] Bryan C. Thorne, Alexander M. Bailey, and Shayn M. Peirce. 2007. Combining experiments with multi-cell agent-based modeling to study biological tissue patterning. *Briefings in Bioinformatics* 8, 4 (July 2007), 245–257. https://doi.org/10.1093/bib/bbm024

[66] Benjamin Torben-Nielsen and Erik De Schutter. 2014. Context-aware modeling of neuronal morphologies. *Frontiers in Neuroanatomy* 8 (Sept. 2014). https://doi.org/10.3389/fnana.2014.00092

[67] Jordi L. Vermeulen, Arne Hillebrand, and Roland Geraerts. 2017. A comparative study of k-nearest neighbour techniques in crowd simulation. *Computer Animation and Virtual Worlds* 28, 3-4 (2017), e1775. https://doi.org/10.1002/cav.1775

[68] U. Wilensky. 1999. NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-based Modeling, Northwestern University, Evanston, IL..

[69] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2019. A Survey on Agent-based Simulation Using Hardware Accelerators. *Comput. Surveys* 51, 6 (Jan. 2019), 131:1–131:35. https://doi.org/10.1145/3291048

[70] Frederic Zubler and Rodney Douglas. 2009. A framework for modeling the growth and development of neurons and networks. *Frontiers in computational neuroscience* 3 (2009), 25. https://doi.org/10.3389/neuro.10.025.2009

[71] Frederic Zubler, Andreas Hauri, Sabina Pfister, Roman Bauer, John C. Anderson, Adrian M. Whatley, and Rodney J. Douglas. 2013. Simulating Cortical Development as a Self Constructing Process: A Novel Multi-Scale Approach Combining Molecular and Physical Aspects. *PLOS Computational Biology* 9, 8 (Aug. 2013), e1003173. https://doi.org/10.1371/journal.pcbi.1003173 Publisher: Public Library of Science.

A Artifact Description

This appendix contains a short summary of the instructions to reproduce the results in the paper. The whole process is fully automated and generates all plots and visualizations shown. The complete instructions can be found in the `SF1-readme.pdf` file located at https://doi.org/10.5281/zenodo.6463816 and https://github.com/CMU-SAFARI/BioDynaMo.

List of Files: We provide the following supplementary files on Zenodo (https://doi.org/10.5281/zenodo.6463816) as well as the SAFARI Research Group's Github page https://github.com/CMU-SAFARI/BioDynaMo:

- `SF0-additional-evaluations.pdf`: This document complements the main paper by providing additional performance evaluations.
- `SF1-readme.pdf`: This document provides detailed documentation, step-by-step instructions to set up the systems and execute the benchmark scripts, and ideas on how to reuse and repurpose this artifact.
- `SF2-code.tar.gz`: This archive contains all the necessary code to produce all results shown in the paper.
- `SF3-bdm-publication-image.tar.gz`: This archive contains a self-contained docker image to simplify executing our benchmarks and aid long-term reproducibility.
- `SF4-raw-results.tar.gz`: This archive contains the raw results we obtained when we executed the benchmarks on our systems.

A.1 Getting Started

Setting up a new system requires four steps. More details can be found in Section 2 in `SF1-readme.pdf`.

1. Download and extract the code archive in Supplementary File SF2.
2. Install the following software packages on the host machine: Docker (version >= 19.0.3), Intel Vtune sampling driver (version 2022.2.0), and the Linux `screen` command.
3. Load the docker image provided by Supplementary File SF3.
4. Verify the setup by executing the following command in the `bdm-paper-examples` directory: `docker/run.sh ./run-functional-evaluation.sh`.

A.2 Reproducing Results

We separate the benchmarks into multiple scripts with different memory and disk space requirements to allow researchers with less powerful hardware to execute a subset of benchmarks. To execute one of the scripts below, change into the `bdm-paper-examples` directory, and pass the script as parameter to the command `docker/run.sh`. More details can be found in Section 4 in `SF1-readme.pdf`.

- `run-main.sh`: This script executes the majority of benchmarks described in the evaluation section of the paper and outputs Figure 5 (left) and Figures 9–12.
- `run-comparison-with-others.sh`: This script executes the comparison of BioDynaMo with Cortex3D and NetLogo and outputs Figure 8.
- `run-runtime-complexity.sh`: This script analyses the runtime and memory consumption of BioDynaMo, with the number of agents increasing from 10^3 to 10^9, and generates Figure 6.
- `run-profiling.sh`: This script performs the microarchitecture analysis for all simulations in Table 1 and generates Figure 5 (right).
- `run-biocellion-cmprsn-single-node.sh`: This script executes the small-scale comparison with Biocellion and the optimization analysis in Figure 7b (right).
- `run-biocellion-cmprsn-cluster.sh`: This script executes the large-scale comparison with Biocellion, the optimization analysis in Figure 7b (left), and renders the visualization in Figure 7a.

A.3 Reusing and Repurposing the Artifact

Besides reproducing the results, researchers can build upon our artifact in numerous ways. A non-exhaustive list of possibilities with detailed instructions is given in Section 5 in `SF1-readme.pdf`. These possibilities include:

- Add additional benchmarks
- Evaluate the effectiveness of additional optimizations
- Evaluate BioDynaMo's performance for additional simulations

A.4 Contact

Please contact us with any feedback or if you need any help building on BioDynaMo. You can reach us at lukas.breitwieser@gmail.com and omutlu@gmail.com.