Fully-Asynchronous Cache-Efficient Simulation of Detailed Neural Networks

Bruno R. C. Magalhães[1], Thomas Sterling[2], Michael Hines[3], and Felix Schürmann[1]

[1] Blue Brain Project, École polytechnique fédérale de Lausanne Biotech Campus, 1202 Geneva, Switzerland

[2] CREST - Center for Research in Extreme Scale Technologies, Indiana University, Bloomington, 47404 IN

[3] Department of Neuroscience, Yale University, New Haven, 06510 CT

Abstract. Modern asynchronous runtime systems allow the re-thinking of large-scale scientific applications. With the example of a simulator of morphologically detailed neural networks, we show how detaching from the commonly used bulk-synchronous parallel (BSP) execution allows for the increase of prefetching capabilities, better cache locality, and a overlap of computation and communication, consequently leading to a lower time to solution. Our strategy removes the operation of collective synchronization of ODEs' coupling information, and takes advantage of the pairwise time dependency between equations, leading to a fully-asynchronous exhaustive yet not speculative stepping model. Combined with fully linear data structures, communication reduce at compute node level, and an earliest equation steps first scheduler, we perform an acceleration at the cache level that reduces communication and time to solution by maximizing the number of timesteps taken per neuron at each iteration.

Our methods were implemented on the core kernel of the NEURON scientific application. Asynchronicity and distributed memory space are provided by the HPX runtime system for the ParalleX execution model. Benchmark results demonstrate a superlinear speed-up that leads to a reduced runtime compared to the bulk synchronous execution, yielding a speed-up between 25% to 65% across different compute architectures, and in the order of 15% to 40% for distributed executions.

1 Introduction

Asynchronous runtime systems built on a global memory address space (GAS) opens up new possibilities for numerical resolutions without synchronization barriers at the core and compute node level, and allow for a substantial reduction of runtime by better utilizing the CPU's prefetching capabilities and cache-level acceleration. Our use case is the simulation of morphologically detailed neural networks, categorized with the following properties: (1) neurons are branched representations of spatially discretized capacitors with ionic current channels; (2) neurons are represented by Ordinary Differential Equations (ODEs) that

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22744-9_33

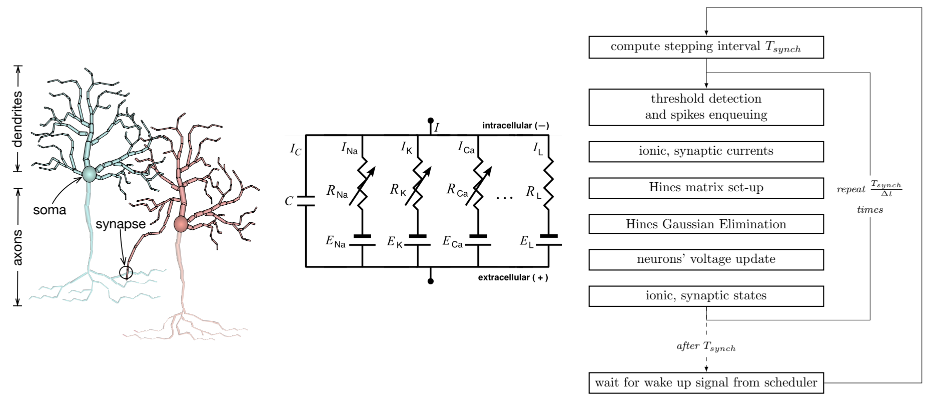


Fig. 1: **Left:** Model representation of two neurons and a synapse. Each neuron includes an axonic branch (south of soma, pictured in light) and a spatially discretized representation of a tree of dendrite compartments (in dark). A synapse is a connection between an axon and a dendrite of different neurons. **Middle:** the RC circuit representing the electrical activity on the membrane of a single compartment, between the intra and extracellular spaces. **Right:** The workflow of the algorithm. A neuron computes the stepping interval T_{synch} from the synaptic dependencies time instants, and performs $T_{synch}/\Delta t$ steps of length Δt.

define the current on the capacitor and the voltage-dependent opening of each ion channel; and (3) ODEs are coupled with a time dependency based on the synaptic connectivity between neurons. For clarity, refer to Figure 1 (left) for a schematic representation of the underlying model.

Due to the high complexity of the data representation — including topological structure, biological mechanisms, synaptic connectivity and external currents — simulations are computationally very costly. State of the art approaches for the acceleration of large neural simulations rely on common parallel and distributed computing techniques. Multi-core and multi- compute node acceleration can be found in NEURON [1]. Complementary efforts rely on Single Instruction Multiple Data (SIMD or vectorization) optimization of state variables replicated across ODEs [2]. Acceleration of small datasets of detailed neuron models have been explored with branch-parallelism [3] (single-core, Single Instruction Single Data, multiple compute nodes), and improved by Magalhaes et al. [4] (with added multi-core, SIMD, and distributed computation). Volumetric decomposition and tessellation with parallel processing of spatial regions has been presented by Kozloski et al. [5].

Similar to most large-scale scientific simulation approaches, synchronization of neurons in existing methods follows the Bulk Synchronous Parallel (BSP) model of computation: execution is split in time grids of equidistant intervals, a period of time with duration equivalent to the minimum synaptic delay across all pairs of neurons in the system. Synaptic communication is typically performed

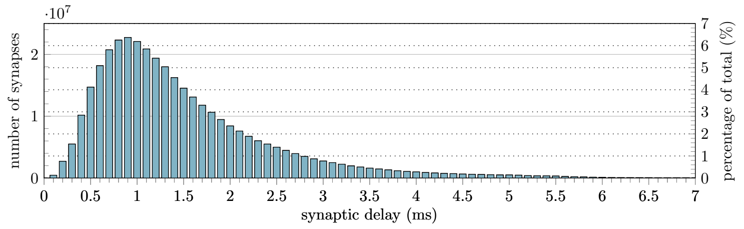


Fig. 2: Distribution of synaptic delays in terms of count (left y-axis) and percentage (right y-axis) of all synapses on a network of 219.247 neurons, extracted from a biologically inspired digital reconstructed model of the rodent neocortex from Markram et al. [6]. Histogram contains one bin per interval of $0.1ms$. The leftmost bar ($x = 0.1ms$) represents the communication step size of state of the art implementations following the Bulk Synchronous Parallel model.

with Message Passing Interface (MPI). It has been shown that, for extremely large networks of compute nodes, the synchronous collective communication can account for over 10% of the overall runtime [2]. This limitation is difficult to overcome in current approaches, as acceleration of the computation of complex models above one-tenth of real time is difficult, due to latency of inter-process communication [7].

In that line of thought, this work presents an **exhaustive yet not speculative** execution model that improves cache locality and provides cache-level acceleration by removing synchronous communication steps, and introducing a fully-asynchronous execution model that advances ODEs timestepping beyond synchronization barriers, based on the time couplings between equations. Our strategy includes five components. At first, (1) a fully-asynchronous stepping protocol that allows elements to perform several timesteps without collective synchronisation. Cache locality is improved by (2) a fully linear memory representation of the data structure, including vector, map and priority queue containers, and is further increased by (3) a computation scheduler that tracks the time progress of ODEs in time and advances the earliest element to its furthest instant in time. Network communication on distributed executions is minimized by (4) a point-to-point fully-asynchronous protocol that signals elements' time advancement to its dependees laid out in a Global Memory Address Space, and by (5) a local communication reduce operation at every compute node — herewith also referred to as locality.

We implemented our methods on the core computation of the NEURON simulator, available as open source [8], with communication, synchronization, and threading enabled by the HPX-5 runtime library [9], demonstrating a shorter time to solution on a wide range of architectures.

1.1 Mathematical Formulation

The main function that describes the currents passing through the membrane of a capacitor n (also referred to as compartment) is described by:

$$C_n \frac{dV_n}{dt} = -\sum_i g_i x_i (V_n - E_i) - \sum_{c:p(c)=n} \frac{V_c - V_n}{r_c} - \frac{V_n - V_{p(n)}}{r_{p(n)}} + I_n(t) \tag{1}$$

where V_n is the difference in potential across the membrane of the compartment, and r the resistance between connecting compartments, when available. The activity of different ions are represented by conductance g_i, opening probability x_i, and reversal potential E_i. The function $p(c) : \mathbb{N} \rightarrow \mathbb{N}$ returns the id of the parent compartment of a given compartment c. Refer to Figure 1 (middle) for the electrical model of the mathematical equation. The first right-hand side term refers to the ionic currents passing through the membrane, described by the Hodgkin-Huxley (HH) model [10]. The voltage-dependent variables x_i describe the opening of the ion channels as a voltage-gated first-order ODE and for brevity were omitted. The fixed step size of the numerical resolution is defined as the time interval *small enough* to capture the dynamics of the biological mechanism with the fastest kinetics — typically the fast Potassium channels — and is set in our model to 0.025 milliseconds. The second term extends the representation of a neuron to a branched morphology, by adding the neighbouring compartments' contributions according to the neuronal cable theory for multiple compartments [11]. To allow the removal of the spatial interpolation of state along each compartment, long compartments are divided into a sequence of smaller ones, and — as a result of their small length — assume that the average state of a compartment along its length is accurately represented by the state of a compartment at its center, and needs only interpolation at consecutive discrete time intervals. The final right hand side term $I(t)$ refers to external currents from time driven events such as injected current stimuli and synaptic activity. The synaptic delay for a given synapse connecting a pre- to a post-synaptic neuron is determined by the time required for the information following an Action Potential (spike) from the pre-synaptic neuron axon to reach its target post-synaptic neuron dendrite.

We apply the simplification that the spike propagation along the axon is stereotypic and that it can be approximated by converting the path from the soma to the synapse to a delay interval after which a simple event is delivered to the synapse. In the model of Markram et al. [6], the minimum synaptic delay in our model is set to $0.1ms$ or equivalently 4 compute steps — refer to Figure 2 for details. — and accounts for circa 0.13% of all the synaptic delays. The communication of spikes at the end of every minimum synaptic delay time frame, allows the update of neuron states in the subsequent period without loss of information.

2 Methods

Significant cache acceleration is difficult to achieve for scientific problems defined by complex data representations. Typically, the main principles to improve cache-efficiency are based on the following rules: using smaller data types and organizing the data so that memory alignment holes are reduced; avoiding the use of algorithms and data structures that exhibit irregular memory access patterns; using linear data structures, i.e. serial memory representations that improve access patterns; and improving spatial locality, by using each cache line to the maximum extent once it has been mapped to the cache. Following this reasoning, the next section details the implementation of our cache-efficiency methods. For completion, the workflow of the scheduled stepping and the kernels of individual compute steps discussed hereafter are presented in Figure 1 (right).

2.1 Linear Data Structures

To avoid fragmentation of data layouts in memory due to dynamic allocations and optimize cache memory reutilization, we implemented a fully linear neuron representation, including class variables and containers. Because the number of elements in the containers are either fixed or defined by a predictable worst case scenario, the size of the container data structures can be computed beforehand. The description of the containers follows in the following paragraphs.

Linear Vector: implemented as a serialization of the `std::vector` class with the meta data, address of array, and elements of the array placed on a sequential memory space. An illustration of the linear vector data structure is displayed in Figure 3 (a).

Linear Map: an unordered map structure storing the mapping of a key to a value or to an array of values. A search for a given key is performed with a binary search across all (ordered) keys, thus yielding similar computational complexity as the `std::map` implementation with a red-black tree, at $O(log\ n)$. The index of a key refers to the count and the pointer to the elements for that key. The memory layout is presented in Figure 3 (b). Moreover, the linear data representation of the map values allows for operations such as minimum value, maximal value and value query to be performed with the same efficiency as a vector.

Linear Priority Queue storing time-driven events as pairs of delivery time and destination. Capable of handling dynamic insertion and removal of events throughout the simulation on a queue of time ordered events. Our implementation relies on a map of circular arrays of ordered time events per pre-synaptic id (the key field). Circular arrays are dimensioned by a pre-computed maximum size, defined by the maximum number of events that can occur during the time window that two given neurons can be set apart at any time throughout the execution. As an example, for a given synaptic connectivity $A \rightarrow B$ with minimum synaptic delay of $1ms$ and the converse $B \rightarrow A$ of $5ms$, the maximum stepping time window

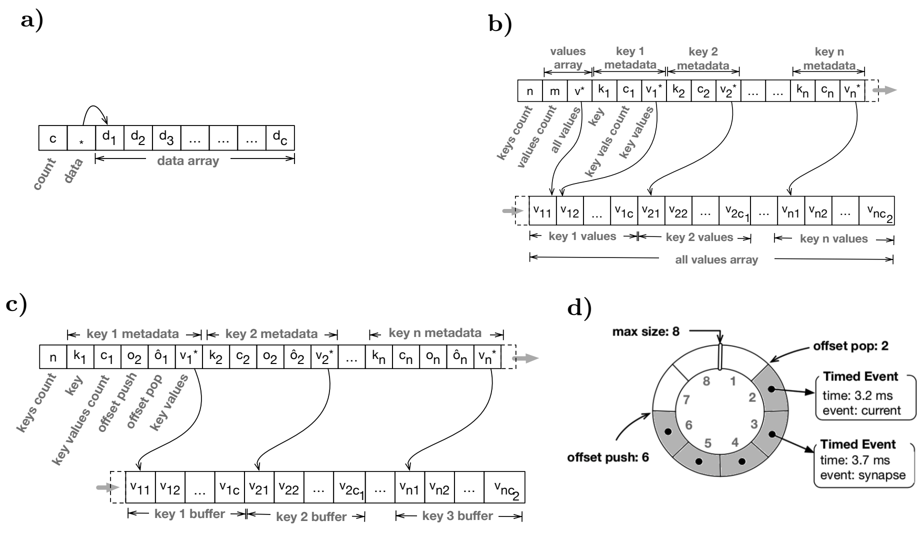


Fig. 3: Memory representation of linear data structures. Gray arrows represent connections between contiguous memory regions. **a)** linear vector; **b)** linear map; **c)** linear priority queue; **d)** a circular array representing a sample entry in the priority queue.

between both is $6ms$ long. To retrieve all subsequent events to be delivered in the following step, the algorithm loops through all keys, collects all events in the interval, and returns the time-sorted list of events. This replaces the iterative peak/top and pop operations underlying regular queue implementations. The memory layout is presented in Figure 3 (c). At the level of each key, given a pre-synaptic neuron id, the list of future events is retrieved in the pop-push interval of elements in the respective circular array. Push (pop) operations will increment the push (pop) offset variable and insert (retrieve) the element in that position. For completion, Figure 3 (d) displays an example of the circular array memory structure for a given key.

As a side note, cache-optimized implementations of priority queues such as funnel heap, calendar queue or other cache-oblivious queues [12] improve memory access pattern yet do not guarantee fully-linear memory allocation. For the sake of comparison, the computational complexity of both ours and the standard library `std::priority_queue` implementations are similar, requiring the retrieval of all events within the next timestep ($O(k)$ for a loop through the all k queues and extraction of the first element on the circular arrays), plus a sorting operation (with worst-case scenario $O(n\ log\ n)$) for a solution of size n, compared to the standard library implementation requiring a complexity in the order of $O(n\ log\ n)$ for n retrievals.

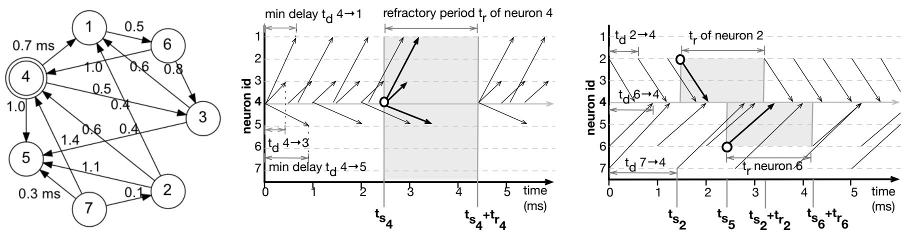


Fig. 4: A representative schema of the algorithm for dependency based synchronization of stepping. **Left:** a sample network of neurons (vertices 1-7). Arrow heads (tails) connect to post- (pre-) synaptic neurons. Labels on edges describe the minimum synaptic delay from a pre- to a post-synaptic neuron. **Center:** outgoing communication for neuron 4. Arrow tail (head) represents a message to the source (destination) neuron. A neuron transmits the time step allowed by the post-syn. neuron, given by his present time plus the minimum transmission delay the a post-synaptic neuron — represented by t_d *pre* $\rightarrow$ *post* and conforming to the graph on the left. Spike notifications (t_s, circles) allow post-synaptic neuron to freely proceed to a time equivalent to the spike time plus the refractory period (t_r) of the pre-synaptic neuron. **Right:** incoming communication for neuron 4. A post-synaptic neuron actively receives progress notifications and keeps track of the maximum step allowed based on pre-synaptic neuron status.

2.2 Time-Based Elements Synchronization and Stepping

To allow for a flexible progress of neurons in time that detach from the constraints of the minimum synaptic delay across all pairs of neurons in the system ($0.1ms$ or 0.13% of total delays, shown previously in Figure 2), we introduce a graph of time dependencies between neurons that allows for a given post-synaptic neuron to advance in time based on their pre-synaptic dependencies' progress. The result is an exhaustive stepping mechanism, that maximises the number of steps per neuron and the simulation time held on CPU cache. The pre- to post-synaptic neuron time updates are provided by an active asynchronous pairwise neuron notification messaging framework. Stepping notifications from a pre- to a post-synaptic neuron are sent at a period defined by their minimum synaptic delay. At the onset of every computation step, a neuron notifies its post-synaptic neuron ids of its stepping if necessary, and stores in a queue the next stepping when notification is required. To reduce communication, the transmission of a spike is also handled as a stepping notification by the post-synaptic size. As a problem-specific optimization, communication is further reduced by taking into account the refractory period, i.e. an interval after a spike during which a neuron is unable to spike again. A schematic workflow of the time-dependency algorithm is presented in Figure 4. The fully-asynchronous stepping yields a more flexible threading by completely removing collective synchronization barriers, less often communication as the pairwise communication delays are generally two orders of magnitude longer than the global minimum transmission delay and a full overlap

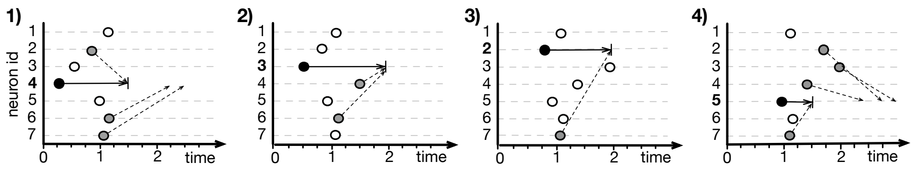


Fig. 5: A sample workflow of 4 iterations of the neuron scheduler applied to the 7 neuron network displayed in Figure 4. On the top-left (frame 1), neuron 4 is the earliest in time (coloured black) and is allowed to proceed to time 1.5ms, dictated by the transmission delay of the pre-synaptic neurons 2, 6 and 7 (coloured gray). The same logic follows in the following iterations, with neurons 3, 2 and 5 being the next ones to advance, as pictured in frame 2) and 3), respectively.

of computation and communication. To maximise the number of steps taken on any run, a neuron scheduler allows for an optimal decision of the next neuron to step, by keeping track of the progress of neurons. This topic is covered next.

2.3 Neuron Scheduler

To maximise cache efficiency, a scheduler was implemented to control and trigger the advancement of neurons in time based on their simulation time. At every iteration, the scheduler (one per locality) actively picks the earliest neuron in time and triggers its stepping. On multi-core architectures, a multi-threaded version of the scheduler allows for several neurons to be launched in parallel. A mutual exclusion control object (mutex) initiated with a counter equal to the number of threads serves as progress control gate. When all threads have been assigned a neuron, the scheduler waits on the mutex. Upon the end of the stepping from a neuron, its thread goes dormant and atomically decrements the mutex counter, waking up the scheduler, and updating its progress in the scheduler's progress map. At the onset of stepping, a neuron queries the time allowed by its pre-synaptic dependencies and performs all necessary steps. An example of scheduled stepping is illustrated in Figure 5.

2.4 Communication Reduce

Global memory address space (GAS) on the Parallax execution model allows for remote thread execution across multiple objects (neurons) distributed across several localities. On a single locality, each message incurs the overhead of a lightweight thread, as GAS addresses are an abstraction to local memory. However, on a distributed execution, each call is an instantiation of a procedure in an object held possibly in a different locality. Therefore, large amount of object-to-object communication may become a bottleneck by saturating the network bandwidth. This issue is trivial to overcome on MPI-based implementations, as the sender is responsible for buffering, packing and initiating the communication, while the converse operations must follow from the receiver. On the Parallax

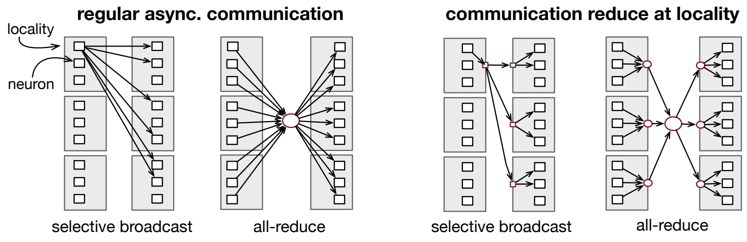


Fig. 6: A sample diagram of the communication required for a selective broadcast and an all-reduce operation with regular (left) versus locality-reduced (right) communication.

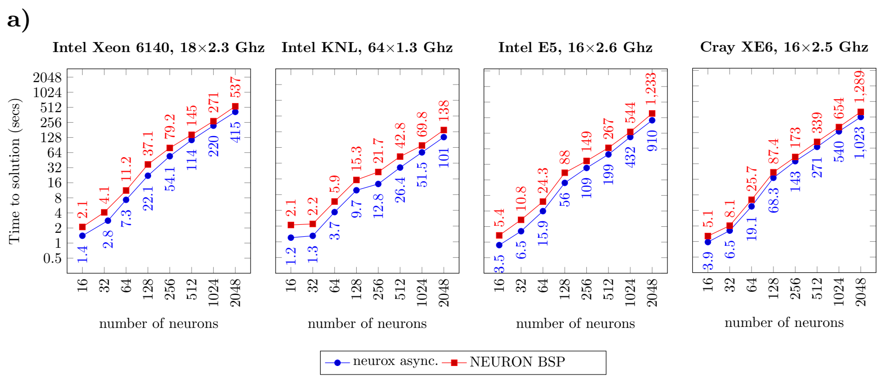
runtime system, its resolution is not as simple, as data representation in GAS arrays remove the locality-awareness of each object in a distributed array. To reduce the overhead of the high amount of point-to-point (inter-neuron) messaging, an extra layer of communication was introduced. Notifications of stepping and spikes for several post-synaptic neurons are packed at the onset of communication as single packets to remote localities. At the recipient side, a mapping of pre-synaptic id to the list of local GAS addresses, allows message to be unfolded and locally spawn to the recipient GAS addresses in the locality. This method replaces n remote communications by a single remote communication with n local lightweight threads spawn. For completion, Figure 6 provides an illustration of the communication reduce methods.

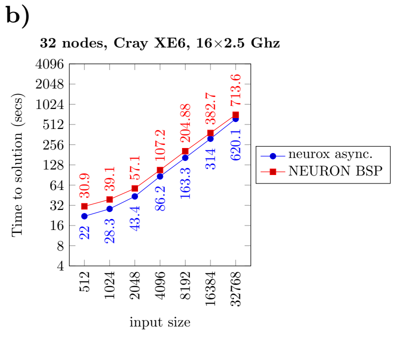
3 Results

Our strategy was implemented in the core computation of the NEURON scientific application, available as open source [8]. Communication, synchronization and memory allocations performed with MPI, OpenMP and malloc, were replaced by the equivalent HPX counterparts. Both our and reference implementations follow the same numerical resolution. The benchmark use case is the simulation of $100ms$ of electrical activity of a morphologically detailed neural network of layer 4 and 5 cells of the rodent brain, extracted from the model of Markram et al. [6], with the distribution of synaptic connectivity previously presented in Figure 2. To demonstrate general applicability of our methods to a wide range of compute architectures, we utilised four different compute architectures with high variability in processor architecture, CPU frequency, memory bandwidth and cache: an Intel Sandy Bridge E5-2670 with 16 cores at 2.6 GHz, a Cray XE6 compute node with an AMD Opteron 6380 with 16 cores at 2.5 GHz each, an Intel Knights Landing (KNL) Xeon Phi with 64 cores at 1.3 GHz, and an Intel Xeon Gold 6140 with 18 cores at 2.3GHz. The L1, L2 and L3 cache sizes for the architectures are: 448KB, 3.5MB and 35MB for the Intel E5; 768KB, 16MB and 16MB for the Opteron; 16KB, 1MB and 32 MB for the Intel KNL; and 576KB, 18MB and 24.75MB for the Xeon 6140. Each representation of a neuron requires a total memory of 4 to 12 MB. Distributed execution were executed on 32 compute nodes of Cray XE6 compute nodes, with specialized Infiniband network hardware for efficient point-to-point communication. We benchmarked the efficiency of each feature individually. The performance analysis of individual components follows in the following paragraphs.

Linear Containers: Cache efficiency of linear containers was measured with the *likwid* suite for performance monitoring and benchmarking [13] on the Xeon 6140 processor. The performance counters account for the containers performance only, isolating linear structures performance from other features. The benchmark test bench compares cache efficiency of linear versus standard library's containers. The estimated amounts of read/write workload are: a spike or event notification (loop through a map of post-synaptic neuron information) at approx. every 15 ms; a delivery of an event — spike information, external currents, time notification — at circa every 0.05 ms, requiring a query to the priority queue; a computation of max time time step allowed by querying the map of time instant per pre-synaptic neuron at every timestep (0.025ms); and an insertion of future events to be delivered at almost every time step (a push of a time event yo the priority queue). The results of cache efficiency on the BSP-based stepping protocol, with 4 continuous steps per neuron, and a communication interval at every 0.1ms, is provided in Table 1 (top). Results demonstrate lower time to solution of circa 4× on the linear implementations versus standard library's, caused by: (1) less instructions, suggesting a more efficient implementation; (2) less data volume across different cache levels and system, suggesting higher reutilisation of data structures across all memory layers; and (3) less memory data volume, suggesting a more compact representation of data leading to more information loaded per cache line. As a relevant remark, Layer 3 cache in the Xeon 6140 architecture is a *victim cache*, or a refill path of CPU cache. Thus, the L2/L3 data volume is higher in our implementation due to demotions of L2 data to L3 instead of main RAM, representing an advantageous behaviour compared to the reference implementation.

Neuron Scheduler and Asynchronous Stepping: Our analysis was extended with asynchronous stepping. Neuron step scheduling for *earliest neuron steps first* was enabled and the distribution of steps size for different input datasets is presented in Figure 7 (c). The step sizes vary depending on the circuit size due to increased inter-neuron connectivity for larger circuits. In practice, increased number of neurons leads to a possibly increased amount of pre-synaptic connectivity, and a higher probability of having a smaller minimum synaptic delay for a given pair of neurons, leading to smaller stepping intervals. We performed a similar cache efficiency benchmark for the asynchronous execution model, and the details are provided in the bottom of Table 1. Results of linear vs std implementations follow in line with the BSP use case, displaying better memory access and lower time to solution when comparing linear vs std container implementations. Asynchronous scheduled stepping yields circa $5-10\%$ lower runtime and a much more efficient memory access compared with the previous BSP benchmark, on both linear and std implementations.





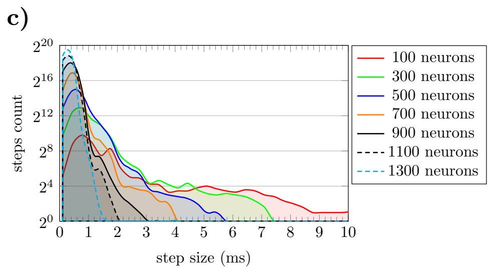


Fig. 7: **a)** Time to solution of the methods presented (neurox async.) and the Bulk Synchronous Parallel equivalent (NEURON BSP) on the simulation of 100ms of the electrical activity of differently sized neural networks, on four different hardware specifications. **b)** Benchmark results for the simulation of 100ms of electrical activity of an increasing number of neurons extracted, on a network of 32 Cray XE6 compute nodes. **c)** Distribution of maximum step size allowed when following the earliest neuron steps first scheduler in the network with synaptic delays represented in Figure 2.

Communication Reduce: The reduce of communications at locality level was measured in terms or runtime and number of point-to-point (p2p) and reduce operations on a similar test bench, and executed on 32 nodes of the Cray XE6 architecture. A benchmark compares the reduced vs non-reduced (simple) communication implementations, measured on the BSP execution model — with a point-to-point communication of synapses and a reduce operation for control gate of neurons time advancement — and the asynchronous model presented, where p2p communication guides synaptic activity and neurons stepping notifications. The results are provided in Table 2, and suggest a significant reduction of communication workload and runtime, on both the BSP and asynchronous execution models. The communication workload gap between reduced and non-reduced implementations increases with the circuit size, as more neurons incur

Table 1: Cache efficiency of linear and standard library (std) containers, for the BSP execution model (4 steps per neuron, top) and the Asynchronous execution model (with steps distribution presented in Figure 7).

Bulk Synchronous Parallel execution model (4 steps per iteration)

Metric	**128 neurons**		**256 neurons**		**512 neurons**		**1024 neur.**		**2048 neur.**	
	linear	std	linear	std	linear	std	linear	std	linear	std
Runtime (secs)	2.13	14.42	12.5	64.3	63.7	278	294	1206	1298	5182
Iterations count ($\times 10^3$)	12.9K	12.9K	25.8K	25.8K	51.7K	51.7K	103K	103K	206K	206K
Instructions count ($\times 10^9$)	12.2	50.9	53.2	221	231.5	953.4	1003.2	4089	4327	17.5K
Clock cycles Per Instr.	0.54	0.85	0.71	0.90	0.82	0.87	0.87	0.88	0.90	0.89
L1/L2 data volume (GB)	1.16	1.52	5.47	9.53	32.1	90.6	266	902	2138	5065
L2/L3 data volume (GB)	1.23	1.23	4.64	4.08	20.3	14.7	80.9	56.8	330	233
L3/system data vol. (GB)	0.77	1.81	3.49	6.94	15.8	27.3	63.8	95.4	254	346
Memory data volume (GB)	0.90	1.39	2.87	4.50	11.5	16.1	46.0	58.0	163	222

Scheduler-driven execution (4+ steps per iteration, following Figure 7)

Metric	**128 neurons**		**256 neurons**		**512 neurons**		**1024 neur.**		**2048 neur.**	
	linear	std	linear	std	linear	std	linear	std	linear	std
Runtime (secs)	2.03	13.6	11.9	60.9	60.4	263.6	277	1143	1222	4913
Iterations count ($\times 10^3$)	4.34	4.34	8.69	8.69	17.39	17.39	34.76	34.76	69.45	69.45
Instructions count ($\times 10^9$)	11.4	47.9	49.9	209	218.3	901.5	948.3	3868	4096	16.4K
Clock cycles Per Instr.	0.54	0.85	0.72	0.87	0.83	0.87	0.87	0.88	0.891	0.888
L1/L2 data volume (GB)	0.68	0.96	4.29	8.34	29.2	78.2	252.9	818.5	2036	4655
L2/L3 data volume (GB)	0.63	0.48	2.60	1.67	13.9	6.10	59.3	24.8	249.3	109.6
L3/system data vol. (GB)	0.43	0.96	2.10	3.95	10.6	13.7	43.03	42.06	172.3	148.1
Memory data volume (GB)	0.42	0.77	1.54	2.42	7.33	9.32	32.48	35.18	123.2	121.2

more synaptic activity and communication. An acceleration of circa 5% – 10% is visible when moving from a BSP to an asynchronous execution model.

Acceleration on Single Compute Nodes: The benchmark for a single compute node of the four aforementioned architectures is displayed in Figure 7 (top) and compares our methods (neurox async.) with the reference solution (NEURON BSP), for an increasing number of interconnected neurons. The results demonstrate that the speed-up achieved decreases as we increase the number of neurons in the dataset. This property is due to the reduction of maximal step allowed by the neuron scheduler as we increase the number of neurons, as presented in Figure 7 (c). On the Intel Xeon 6140, the methods yield a speed-up between 31% — for the largest network of 2048 neurons — and 51% for the network of 16 neurons. The speed-ups for the remaining architectures are 36%-65% for the KNL, 35%-54% on the Intel E5, and 26%-31% on the Cray XE6.

Acceleration on Distributed Executions: In order to understand whether the single node advantages of the asynchronous execution hold in a distributed setting with multiple nodes, we extended our benchmark to a network of 32 nodes of the Cray XE6 architectures. Similarly to the single compute node use case, the test bench provides the runtime for an increasing number of neurons, in this case for a fixed network of 32 compute nodes. The results are presented in Figure 7 (b), and display a speed-up of 16% for the largest dataset of 32768 neurons, up to 40% for 256 neurons i.e. one neuron per core per locality.

Table 2: Performance of regular versus locality-reduced communication in terms of runtime and point-to-point and reduce communications, on the BSP (top) and asynchronous (bottom) execution models.

BSP execution; 32 compute nodes; p2p comm. for spiking, reduce at every 0.1ms

Metric	512 neurons		1024 neurons		2048 neurons		4096 neurons		8192 neurons	
	reduce	simple	reduce	simple	reduce	simple	reduce	simple	reduce	simple
Runtime (secs)	3.90	4.07	4.93	5.51	7.48	8.70	12.96	15.66	28.38	31.61
point-to-point count	2168	2327	7543	8855	24.3K	33.4K	70.1K	124K	188K	480K
reduce comm. count	100	1600	100	3200	100	6400	100	12.8K	100	25.6K

Asynchronous Execution; 32 compute nodes; p2p for spiking and stepping notification

Metric	512 neur.		1024 neur.		2048 neurons		4096 neurons		8192 neurns	
	reduce	simple	reduce	simple	reduce	simple	reduce	simple	reduce	simple
Runtime (secs)	3.60	3.80	4.07	4.42	6.66	6.53	12.14	13.27	26.75	28.31
point-to-point count	623K	665K	2.34M	2.72M	8.25M	11.09M	44.77M	25.79M	71.75M	181.46M

4 Conclusions

In this article, we explore the capabilities of new runtime systems for the numerical simulation of large systems of ODEs. We present an asynchronous model of execution with the capability of removal of global synchronization barriers, leading to better cache-efficiency and lower time to solution, due to long timestepping of individual equations based on their time coupling information. We detail the implementation of a fully-asynchronous, cache-accelerated, parallel and distributed simulation strategy supported by the HPX runtime system for the ParalleX execution model, providing a Global Address Memory space, remote procedure calls and asynchrony capabilities. Five components are introduced and detailed: (1) a linear data representation of a vector, map and priority queue containers that allow fully sequential instantiation of data structures in memory; (2) an exhaustive yet not speculative stepping of individual equations based on its time dependencies, supported by (3) a point-to-point communication protocol that actively notifies time dependencies of time advancements of their dependees and allows for the full overlap of computation and communication; (4) an object scheduler that further improves cache locality by maximising the number of steps per run by tracking equations progress throughout the execution; and (5) a local communication reduce operation that translates point-to-point to point-to-locality communication in a global address memory space.

Our methods were implemented on the core computation of the NEURON scientific application and tested on a biologically-inspired branched neural network. We analyse and demonstrate the efficiency of the features introduced in terms of communication, cache efficiency, patterns of data loading, and time to solution. Benchmark results demonstrate a significant speed-up in runtime in the order of 25% to 65% across different compute architectures and up to 40% on distributed executions. To finalize, most techniques presented follow from first principles in Computer Science, and can therefore be applied to a wide range of scientific problem domains.

Acknowledgements

The work was supported by funding from the ETH Domain for the Blue Brain Project (BBP). The super-computing infrastructures were provided by the Blue Brain Project at EPFL and Indiana University. A portion of Michael Hines efforts was supported by NINDS grant R01NS11613.

References

1. M. L. Hines and N. T. Carnevale, “The neuron simulation environment,” *Neural computation*, vol. 9, no. 6, pp. 1179–1209, 1997.
2. A. Ovcharenko, P. Kumbhar, M. Hines, F. Cremonesi, T. Ewart, S. Yates, F. Schuermann, and F. Delalondre, “Simulating morphologically detailed neuronal networks at extreme scale.” Advances in Parallel Computing, 2015.
3. M. L. Hines, H. Markram, and F. Schürmann, “Fully implicit parallel simulation of single neurons,” *Journal of computational neuroscience*, vol. 25, no. 3, pp. 439–448, 2008.
4. B. Magalhaes, M. Hines, T. Sterling, and F. Schuermann, “Asynchronous simd-enabled branch-parallelism of morphologically-detailed neuron models,” 2019, unpublished.
5. J. Kozloski and J. Wagner, “An ultrascalable solution to large-scale neural tissue simulation,” *Front. Neuroinform*, vol. 5, no. 15, pp. 10–3389, 2011.
6. H. Markram, E. Muller, S. Ramaswamy, M. W. Reimann, M. Abdellah, C. A. Sanchez, A. Ailamaki, L. Alonso-Nanclares, N. Antille, S. Arsever *et al.*, “Reconstruction and simulation of neocortical microcircuitry,” *Cell*, vol. 163, no. 2, pp. 456–492, 2015.
7. F. Zenke and W. Gerstner, “Limits to high-speed simulations of spiking neural networks using general-purpose computers●,” *Frontiers in Neuroinformatics*, vol. 8, no. 76, 2014. [Online]. Available: http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2014.00076/abstract
8. Blue Brain Project, “Coreneuron - simulator optimized for large scale neural network simulations,” https://github.com/bluebrain/CoreNeuron.
9. T. Sterling, M. Anderson, P. K. Bohan, M. Brodowicz, A. Kulkarni, and B. Zhang, “Towards exascale co-design in a runtime system,” in *Exascale Applications and Software Conference*, Stockholm, Sweden, Apr 2014.
10. A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and its application to conduction and excitation in nerve,” *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.
11. E. Niebur, “Neuronal cable theory,” vol. 3, no. 5, p. 2674, 2008, revision 121893.
12. L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro, “Cache-oblivious priority queue and graph algorithm applications,” in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing.* ACM, 2002, pp. 268–276.
13. J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on.* IEEE, 2010, pp. 207–216.