

Asynchronous Simulation of Neuronal Activity

Thèse N° 7098

Présentée le 8 juillet 2019

à la Centres pour la recherche

BBP-CORE

Programme doctoral en neurosciences

pour l'obtention du grade de Docteur ès Sciences

par

Bruno Ricardo DA CUNHA MAGALHÃES

Acceptée sur proposition du jury

Prof. D. Ghezzi, président du jury

Prof. F. Schürmann, Prof. T. L. Sterling, directeurs de thèse

Prof. M. Diesmann, rapporteur

Prof. J. Labarta, rapporteur

Prof. S. DeParis, rapporteur

2019

*"I have no special talent.
I am only passionately curious."
— Albert Einstein*

Acknowledgements

I would like to express my most heartfelt gratitude to Prof. Felix Schürmann for the opportunity of performing the PhD thesis at the Blue Brain Project laboratory, and for his supervision and guidance throughout these four years. I sincerely appreciate the freedom he gave me in discovering and developing my research directions and to learn new side skills in many topics of personal interest.

I would like to extend my gratitude to Michael Hines at Yale University for the scientific discussions, and all my lab colleagues at EPFL for their companionship and technical support. A great thanks to Francesco Cremonesi for the very fruitful mathematical discussions. His positiveness and dedication to science were very inspiring.

Equally, I thank all the colleagues at the Center for Research in Extreme Scale Technologies at Indiana University, for the technical support and the rewarding moments spent in Bloomington. A special thanks to my thesis co-director, Professor Thomas Sterling, for the personal inspiration and motivation, and Tim Gilmanov and Sarah Carroll for their friendship and hospitality.

My warmest thanks to my Lausanne friends Ronny Hatteland, Henriette Gjaerde, Jessica Brühlmann, James King, Alexandre Chausson, Sheena Rupa, Luis Polo Carbayo, Ginal Schlatter and Dumas Galvez, for all the amazing moments and the strong friendship.

I express my sincere gratitude to my team mates at the Lausanne Natation waterpolo team, for all the moments of intense yet joyful physical training in the swimming pool, and all the moments of fun outside.

Last but not least, I would like to thank my parents, Alberto and Maria, and all my family for their unconditional support and love.

Lausanne, 1st July 2019

B. M.

Abstract

Simulations of the electrical activity of networks of morphologically-detailed neuron models allow for a better understanding of the brain. Short time to solution is critical in order to study long biological processes such as synaptic plasticity and learning.

State-of-the-art approaches follow the Bulk Synchronous Parallel execution model based on the Message Passing Interface runtime system. The efficient simulation of networks of simple neuron models is a well researched problem. However, the efficient large-scale simulation of detailed neuron models — including topological information and a wide range of complex biological mechanisms — remains an open problem, mostly due to the high complexity of the model, the high heterogeneity of specifications in modern compute architectures, and the limitations of existing runtime systems.

In this thesis, we explore novel methods for the asynchronous multi-scale simulation of detailed neuron models on distributed parallel compute networks.

In the first study, we introduce the concept of distributed asynchronous branch-parallelism of neurons. We present a method that extracts variable dependencies across numerical resolution of neurons topological trees and the underlying algebraic solver. We demonstrate a method for the decomposition of the neuron simulation problem into interconnected resolutions of neuron subtrees. Results demonstrate a significant reduction in runtime on heterogeneous distributed, multi-core, Single Instruction Multiple Data (SIMD) computing environments.

In the second study, we complement the previous effort with graph-parallelism retrieved from mathematical dependencies across the systems of Ordinary Differential Equations (ODEs) that model the activity of neurons. We describe a method for the automatic extraction of read-after-write variable dependencies, concurrent variable updates, and independent ODEs. The automation of the methods expose a computation graph of biological mechanisms inter-dependency, and an embarrassingly-parallel SIMD execution of mechanism instances, leading to an acceleration of the simulation.

The third part of this thesis pioneers the fully-asynchronous parallel execution model, and the *exhaustive yet not speculative* stepping protocol. We apply it to our use case, and demonstrate that by removing collective synchronization of neurons in time, by utilising a memory-linear neuron representation, and by advancing neurons based on their synaptic connectivity, a substantial runtime speed-up is achievable through cache efficiency.

Abstract

The fourth and last part advances the aforementioned fully-asynchronous execution model with variable-order variable-timestep interpolation methods. We demonstrate low dependency of runtime and step count on volatility of solution, and a high dependency on synaptic events. We introduce a variable-step execution model that allows for non-speculative asynchronous stepping of neurons on a distributed compute network. We demonstrate higher numerical accuracy, less interpolation steps, and shorter time to solution, compared to state-of-the-art implicit fixed-step resolution.

This thesis demonstrates that utilising asynchronous runtime systems is a requirement to succeed in efficiently simulating complex neuronal activity at large scale. Most contributions presented follow from first principles in numerical methods and computer science, thus providing insights for applications on a wide range of scientific domains.

Keywords

Asynchronous simulation, neural networks, NEURON simulator, branch-parallelism, graph-parallelism, cache-efficiency, variable timestep interpolation, asynchronous runtime systems, HPX, ParallelX

Résumé

Il est possible de mieux comprendre le cerveau en simulant l'activité électrique de réseaux de neurones morphologiquement détaillés. Le temps d'exécution de ces modèles doit être le plus court possible afin d'étudier de longs processus biologiques tels que la plasticité synaptique et l'apprentissage.

L'état de l'art actuel suit le modèle d'exécution Bulk Synchronous Parallel basé sur l'environnement d'exécution Message Passing Interface. La simulation efficace de réseaux de modèles de neurones simples est un problème bien recherché. Néanmoins, la simulation efficace à grande échelle de modèles neuronaux détaillés — comprenant des informations topologiques et une vaste collection de mécanismes biologiques complexes — reste un problème ouvert, principalement en raison de la grande complexité du modèle, de la grande hétérogénéité des spécifications dans les architectures d'ordinateurs modernes et des limites des environnements d'exécution existants.

Dans cette thèse, nous explorons de nouvelles méthodes pour la simulation asynchrone multi-échelle de modèles de neurones détaillés sur des réseaux parallèles et distribués de calcul.

Dans la première étude, nous introduisons le concept de parallélisme de branche asynchrone et distribué de neurones. Nous présentons une méthode qui extrait des dépendances variables à travers la résolution numérique des arbres topologiques de neurones et du résolveur algébrique sous-jacent. Nous démontrons une méthode pour la décomposition du problème de simulation de neurones en résolutions interconnectées de sous-arbres de neurones. Les résultats démontrent une réduction significative du temps d'exécution dans des environnements informatiques hétérogènes distribués, multicœurs, à instruction unique, données multiples (SIMD).

Dans la deuxième étude, nous poursuivons l'effort précédent avec le parallélisme de graphe extrait des dépendances mathématiques des systèmes d'équations différentielles ordinaires (ODE) modélisant l'activité électrique des neurones. Nous décrivons une méthode d'extraction automatique des dépendances de variable lecture-après-écriture, des mises à jour de variables simultanées et des ODE indépendants. L'automatisation des méthodes expose un graphe de calcul d'interdépendance des mécanismes biologiques et une exécution SIMD et parallèle d'instances de mécanismes, conduisant à une accélération de la simulation.

Resumé

La troisième partie de cette thèse inaugure le modèle d'exécution complètement asynchrone et le domaine de simulation *exhaustif mais non spéculatif*. Nous l'appliquons à notre cas d'utilisation et démontrons qu'en supprimant la synchronisation collective de l'interpolation temporelle des neurones, en utilisant une représentation linéaire de neurones en mémoire et en faisant avancer les neurones dans le temps en fonction de leur connectivité synaptique, une accélération substantielle de l'exécution est possible grâce à un meilleur accès de la mémoire cache.

La quatrième partie complète la méthode asynchrone susmentionnée avec une résolution numérique basée sur les méthodes implicites avec ordre et pas-de-temps variables. Nous analysons la sensibilité numérique de nos méthodes et démontrons une faible dépendance de l'exécution et du nombre d'étapes vis à vis de la volatilité de la solution, ainsi qu'une forte dépendance vis à vis des événements synaptiques. Nous introduisons un modèle d'exécution à pas-de-temps variable qui permet une interpolation asynchrone et non-spéculative de neurones sur un réseau de calcul distribué. Nous démontrons une plus grande précision numérique, moins de pas-de-temps d'interpolation et un temps de résolution plus court, par rapport à la résolution implicite de l'état de l'art.

Cette thèse démontre que l'utilisation de systèmes d'exécution asynchrones est une nécessité pour réussir à simuler efficacement activité neuronale complexe à grande échelle. La plupart des contributions présentées découlent des principes de base des méthodes numériques et de la science informatique, fournissant ainsi des pistes pour l'application dans un large éventail de domaines scientifiques.

Mots-clés

Simulation asynchrone, réseaux de neurones, simulateur NEURON, parallélisme de branche, parallélisme de graphe, efficacité d'appel de cache, interpolation à pas-de-temps variable, environnement d'exécution asynchrones, HPX, ParalleX

Contents

Acknowledgements	v
Abstract	vii
Resumé	ix
List of Figures	xiv
List of Tables	xvii
1 Introduction	1
1.1 State of the Art	2
1.1.1 Single Compute Node Executions	3
1.1.2 Branch-Parallelism	3
1.1.3 Spatial Decomposition	4
1.1.4 Variable Timestep Interpolation	4
1.1.5 Distributed Compute Environments	5
1.2 Limitations	7
1.2.1 Efficient Usage of Compute Resources	7
1.2.2 Handling of Complex Neuron Models	7
1.3 Motivation	8
1.4 Research Scope	9
1.5 Runtime Systems	9
1.5.1 MPI: Message Passing Interface	11
1.5.2 HPX: High Performance ParalleX	11
1.6 Reference Implementation	12
1.6.1 Computational Model	12
1.6.2 Spatial Discretization	14
1.6.3 Branching	14
1.6.4 Fixed Step Interpolation	15
1.6.5 Parallel Executions	16
1.6.6 Workflow	17
1.7 Thesis Structure	18
1.7.1 Asynchronous Branch-Parallelism	18

Contents

1.7.2	Asynchronous Graph-Parallelism	19
1.7.3	Fully-Asynchronous Fixed-Step Execution Model	19
1.7.4	Fully-Asynchronous Variable-Order Variable-Timestep Execution Model	20
2	Asynchronous Branch-Parallelism Extracted from Neuron Morphology	21
2.1	Abstract	21
2.2	Introduction	22
2.3	Methods	24
2.3.1	Dependency Parameters	24
2.3.2	Neuron Tree Decomposition and Parallel Execution	26
2.3.3	Vector-based Acceleration	28
2.3.4	Asynchronous Execution of State Updates	29
2.3.5	Distributed Load Balancing	29
2.4	Benchmark	31
2.4.1	Implementation	31
2.4.2	Use Case	32
2.4.3	Hardware Specifications	33
2.4.4	SIMD vs Multi-core Trade-off Optimization	33
2.4.5	Reference Branch-Parallel Implementation	33
2.4.6	Scaling of Single Neurons	34
2.4.7	Scaling of Networks of Neurons	35
2.5	Discussion	37
3	Asynchronous Graph-Parallelism Extracted from ODE Dependencies	39
3.1	Abstract	39
3.2	Introduction	40
3.3	Methods	41
3.3.1	Dependencies from Model Specification	41
3.3.2	Application to NEURON Modelling Language	43
3.3.3	Computation Graph from Individual Dependencies	44
3.3.4	Vector-Parallelism of Mechanism Instances	45
3.4	Benchmark	47
3.4.1	Implementation	47
3.4.2	Use Case	50
3.4.3	Hardware Specifications	50
3.4.4	Strong Scaling	50
3.4.5	Distributed Executions	52
3.5	Discussion	54
4	Fully-Asynchronous Cache-Efficient Simulation	55
4.1	Abstract	55
4.2	Introduction	56
4.3	Methods	57

4.3.1	Linear Data Structures	58
4.3.2	Time-Based Elements Synchronization and Stepping	60
4.3.3	Neuron Scheduler	61
4.3.4	Communication Reduce	62
4.4	Benchmark	63
4.4.1	Implementation	63
4.4.2	Use Case	64
4.4.3	Hardware Specifications	64
4.4.4	Linear Containers	64
4.4.5	Neuron Scheduler and Asynchronous Stepping	65
4.4.6	Communication Reduce	66
4.4.7	Single Compute Node Executions	67
4.4.8	Distributed Executions	67
4.5	Discussion	68
5	Fully-Async. Fully-Implicit Variable-Order Variable-Timestep Simulation	69
5.1	Abstract	69
5.2	Introduction	70
5.3	Methods	71
5.3.1	Resolution of Simple and Complex Neuron Models	71
5.3.2	Variable Step Implementation	72
5.3.3	Asynchronous Timestepping of Networks of Neurons	74
5.4	Results	75
5.4.1	Numerical Accuracy	75
5.4.2	Performance Dependency on Solution Stiffness	77
5.4.3	Performance Dependency on State Discontinuities	77
5.4.4	Simulation of a Laboratory Experiment	78
5.5	Benchmark	78
5.5.1	Implementation	78
5.5.2	Use Case	80
5.5.3	Fixed- vs Variable-Timestep Interpolators	82
5.5.4	Variable Step Event Grouping	84
5.5.5	Fully-Asynchronous vs Bulk-Synchronous Execution Models	85
5.5.6	Runtime Dependency on Input Size and Spike Activity	85
5.5.7	Overall Runtime Speedup Estimation	86
5.6	Discussion	87
6	General Discussion and Conclusions	89
6.1	Summary	89
6.1.1	Micro-Parallelism of Detailed Neuron Models	89
6.1.2	Fully-Asynchronous Execution Model	90
6.2	Contributions	91
6.3	Performance Outlook	95

Contents

6.4	Future Work	97
6.4.1	Combined Micro-Parallelism Methods	97
6.4.2	Automatic Tuning of Data Layout to Hardware Specifications	97
6.4.3	Variable Timestepping Without Discontinuities	98
6.4.4	Finer-Tuned CVODE Execution and Parameters	98
6.4.5	Asynchronous Speculative Execution	98
6.4.6	Distributed Load Balancing	99
6.4.7	Support for Graphics Processing Units	100
6.5	Closing Discussion	100
A	Generalization: from Gaussian Elimination to Hines Solver	103
B	Methods Availability and Reproducibility	105
C	Scientific Papers	107
	Bibliography	109
	References	109

List of Figures

1.1	The scope of the simulation scale and morphologically detailed neuron models covered in this thesis	2
1.2	Scope of the interpolation methods discussed	6
1.3	The topological structure of a sample neuron, and its representative sparse tridiagonal tree representation	15
1.4	Workflow of one compute step of the numerical resolution of our simulation model	17
2.1	Scope and key concepts of the branch-parallelism methods	23
2.2	Algorithms of the initial matrix values set-up, Gaussian Elimination and voltage update methods	25
2.3	A sample workflow of the algorithm that decomposes neuron morphologies into a tree of subtrees	26
2.4	The topological structure of a neuron and its representative matrix, before and after the subtree decomposition algorithm	27
2.5	The topological structure of a neuron after clustering into 5 subtrees, and memory layouts for the pre- and post- subtree clustering phases	28
2.6	Overview of the asynchronous producer-consumer model for dependency variables	30
2.7	Morphological structure of the Layer 5 neurons extracted from the digital reconstruction of the rodent neocortex used as input data	32
2.8	Strong scaling benchmark for the simulation of one second of electrical activity of one cell, comparing the multisplit implementation in NEURON and our methods, benchmarked on four compute architectures	34
2.9	Strong scaling benchmark for the simulation of one second of electrical activity of three different Layer 5 neuron models, benchmarked on four compute architectures	35
2.10	Benchmark of the simulation of one second of electrical activity of Layer 5 neurons, on a Cray XE6 cluster with 128 compute nodes	36
3.1	Diagram of the flow dependencies of the SK_E2 potassium ion channel	42
3.2	Diagram of the flow dependencies and concurrent outputs of the SK_E2 potassium ion channel	42

List of Figures

3.3	Workflow of the state-of-the-art approach for the state update of a neuron morphology with 23 mechanisms	44
3.4	Computation graph for the parallel execution of the state update function with flow dependencies and concurrent updates	45
3.5	Three alternative memory layouts for the processing of one mechanism with three state variables expressed in five instances	46
3.6	Mean and standard deviation of the number of instances per mechanism, extracted from 1000 neurons in the layer 5 of the Somatosensory cortex of the rodent brain	48
3.7	Average execution time for the timestep update of single instances of the mechanisms, extracted from 1000 neurons in the layer 5 of the Somatosensory cortex of the rodent brain	49
3.8	Benchmark for the simulation of 100ms of electrical activity of a single neuron from the somatosensory cortex of the young rodent, on four compute architectures	51
3.9	Benchmark for the simulation of 100ms of electrical activity of the neocortex, on a cluster of 128 Cray XE6 compute nodes	52
4.1	Distribution of synaptic delays in terms of count and percentage of all synapses in a network of 219.247 neurons	56
4.2	Memory representation of linear data structures	58
4.3	Representative schema of the time-dependency control method for the synchronization of stepping and spikes, applied to a network of seven neurons	60
4.4	A sample workflow of four iterations of the neuron scheduler applied to a network of seven neurons	61
4.5	Diagram of the communication required for the selective broadcast and all-reduce operations using regular versus locality-reduced communication	62
4.6	Benchmark of the synchronous versus Bulk Synchronous Parallel execution models of the simulation of 100ms of the electrical activity of different neural networks, on four different hardware specifications	66
5.1	Illustrative workflow of the speculative and the non-speculative scheduler-based variable timestep methods analysed	73
5.2	Voltage potential at the soma and interpolation steps of a layer 5 pyramidal cell, during a 6ms and 100ms simulation of a 1.3mA continuous current injection	75
5.3	Interpolation steps and runtime for the 1000ms simulation of the injection of a continuous constant current as a percentage of the threshold current (0.206mA), on a layer 5 pyramidal cell	76
5.4	Interpolation steps and runtime for the 1000ms simulation of the injection of short 1μs current pulses of different amplitudes I at different frequencies, on a layer 5 pyramidal cell	77
5.5	Number of discontinuity events and distribution of event time differences for three sample neurons throughout 7.5 seconds of simulation	79

5.6	Runtime for the simulation of one second of biological activity of five neuron networks described by different spiking rate dynamics	83
5.7	Distribution of neuron counts by spiking rates collected from the central minicolumn of the network utilized in the simulation of the laboratory experiment, in the interval 2-4 secs	85



List of Tables

4.1	Cache efficiency of linear and standard library containers, measured for the Asynchronous and Bulk Synchronous Parallel execution models	65
4.2	Performance of regular versus locality-reduced communication in terms of runtime, and number of point-to-point and reduce communications, on the Bulk Synchronous Parallel and asynchronous execution models	67

1 Introduction

This chapter is adapted from the preprint version of the following articles:

Magalhães B., Hines M., Sterling T., Schürmann F, "Exploiting Flow Graph of System of ODEs to Accelerate the Simulation of Biologically-Detailed Neural Networks", accepted at IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2019

Magalhães B., Hines M., Sterling T., Schürmann F, "Asynchronous Branch-Parallel Simulation of Detailed Neuron Models", submitted to Frontiers in Neuroinformatics 2019

Magalhães B., Hines M., Sterling T., Schürmann F, "Fully-Asynchronous Cache-Efficient Simulation of Detailed Neural Networks", accepted at International Conference on Computational Science (ICCS) 2019

Magalhaes B., Hines M., Sterling T., Schürmann F, "Fully-Asynchronous Fully-Implicit Variable-Order Variable-Timestep Simulation of Neural Networks", published on arXiv

Personal contributions: conceptualization, writing.

Over the past years, the simulation of the activity of large neural networks has received increasing interest (Kandel et al., 2013; Markram, 2012; Shepherd et al., 1998). Experimental advances such as high resolution recording of neurons *in vivo* and *in vitro* have supported quantitative modeling. Biologically inspired simulations of neural circuits present an enormous opportunity for understanding the behaviour of the brain. Simulations can be performed on different details of neurons activity (Brette et al., 2007), ranging from biochemical reactions to a conductance-based simulation, or a simpler model such as integrate-and-fire (Brunel & Hakim, 1999). Moreover, different scales can be simulated, from a point neuron representation to a more complete morphologically detailed neuron model.

Recent efforts (Markram et al., 2015) presented for the first time a simulation of a morphologically detailed model of the neocortical microcircuit, simulated in the NEURON scientific application (Hines & Carnevale, 1997; Carnevale & Hines, 2006). This effort was part of the main mission of the Blue Brain Project (Markram, 2006; Hill & Markram, 2008) to digitally reconstruct and simulate the brain. The simulation was based on the multi-compartment Hodgkin-Huxley (HH) formalism (Hodgkin & Huxley, 1952). The HH model computes an approximation of the current passing through a section of the neuron's membrane, as a capac-

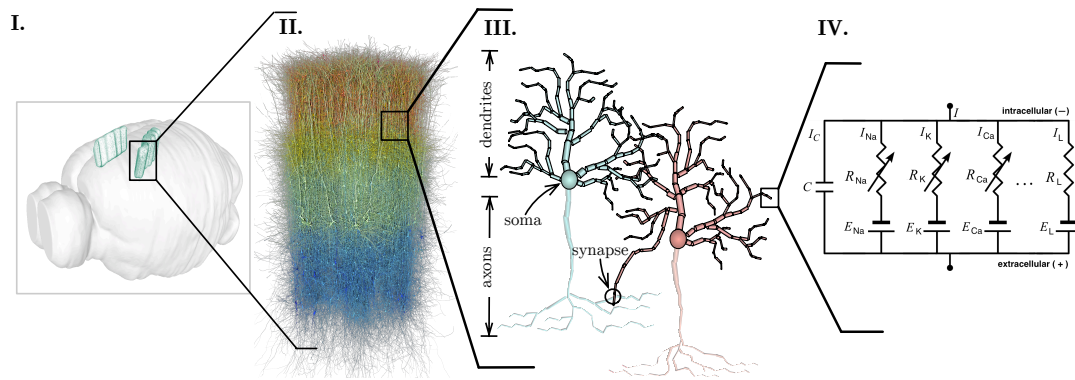


Figure 1.1 – The scope of the simulation scale and morphologically detailed neuron models covered in this thesis. **I:** The rodent brain with an exposed digital reconstruction of the lower limb of the primary somatosensory area; **II:** a cortical column of the previous brain region; **III:** model 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 dendrites (north of soma, in dark). A synapse is a connection between an axon and a dendrite of different neurons; **IV:** the resistor–capacitor circuit (RC circuit) representing the electrical activity on the membrane of a single capacitor with ionic current leaks, also referred to as compartment.

itor with ionic conductances. Each neuron is modelled by a system of Ordinary Differential Equations (ODEs), that includes the change in voltage at the capacitor and the change in the opening and closing of the ion channels that drive the fluxes of ionic currents at the capacitor’s membrane. Neurons are coupled with each other through synapses, that are electro-chemical transducers. For completeness, Figure 1.1 illustrates the scope and scale of the methods discussed.

Since it is not feasible to analytically solve complex HH equations, simulations typically employ numerical methods and standard time-discretized ODE solvers (Hines & Carnevale, 1994). The large number of equations involved in such systems leads to computationally costly simulations, that may be required to run for long periods of time (Helmstaedter & Mitra, 2012). The acceleration of such simulations is particularly relevant for the understanding of biological phenomena such as synaptic plasticity and learning. Such use cases may focus on the study of the dynamics between few neurons (Markram et al., 2012), small networks of neurons (Chindemi, 2018) or large populations of neurons (Morrison et al., 2007), with a biological time spanning from few minutes to several days to be expressed.

1.1 State of the Art

Similarly to simulations in several scientific fields, acceleration of the simulation of such models typically follows the Bulk Synchronous Parallel (BSP) execution model (Gerbesiotis &

Valiant, 1994), computing several neurons simultaneously via a synchronized multi-threaded distributed execution (Tikidji-Hamburyan et al., 2017). Execution time is divided in equidistant time intervals, equivalent to the time duration of the shortest synaptic delay across all neuron pairs in the network (Migliore et al., 2006). The synaptic delay for a given synapse between a **pre-** and a **post-synaptic** neuron is defined by the time for current propagation from the soma of a pre-synaptic neuron to the extremity of the axon branch, and the neurotransmitter release to the post-synaptic receptor. Stepping of neurons is performed independently within the boundaries of each synchronization interval.

The theoretical limit of acceleration on the BSP model is dictated by the most complex compute kernel in the network, whose state update takes the longest to compute. The extent of a kernel, representing the activity of a group of neurons, a single neuron, a subsection of the neuron topology, a compartment, or a biological mechanism such as ion channels, varies extensively and depends on the implementation and scale of parallelism exposed, as detailed next.

1.1.1 Single Compute Node Executions

Acceleration of the simulation of networks of point neuron models has been previously demonstrated with multi-core parallelism of individual neurons, combined with Single Instruction Multiple Data (SIMD or vectorized) computing of neuron state variables, in Brian (Goodman & Brette, 2008; Brette & Goodman, 2011), Auryn (Zenke & Gerstner, 2014) and NEST simulators (Gewaltig & Diesmann, 2007).

The acceleration of networks of branched morphologies utilising multi-core Single Instruction Single Data (SISD) has been covered extensively by the NEURON simulator (Hines & Carnevale, 1997), with added SIMD capabilities demonstrated by the CoreNeuron (Kumbhar et al., 2016, 2019) and Arbor (Klijn et al., 2017) scientific applications. The SIMD methods proposed take advantage of the similarity across ODEs of individual neurons, allowing them to be grouped together in a SIMD-friendly memory layout, enabling full usage of the processor's register file width.

Multi-core execution of neurons is typically performed with OpenMP (Dagum & Menon, 1998) and/or POSIX threads (Butenhof, 1997), with synchronization of neurons stepping performed with a threading barrier executed at an interval equivalent to the shortest synaptic delay. This guarantees no overstepping of solution interpolation in time, and no missed synaptic events in the stepping interval between the current and the next synchronization instants.

1.1.2 Branch-Parallelism

Further acceleration can be achieved on the strong scaling axis with finer-grained parallelism of individual neuron models, via the decomposition of simulation steps into smaller compute kernels, leading to the increase of compute units assigned to each neuron. Finer-grained parallelism efforts have been demonstrated based on branch sections parallelism, by extracting

variable dependencies across compartmental neuron trees, and performing a sub-sectioning of the topological structure of neurons. Related work has been presented on the NEURON scientific simulator (Hines et al., 2008), demonstrating parallelism of neuron subtrees on a network of single-core architectures with a Single-Instruction Single-Data (SISD) instruction stream.

1.1.3 Spatial Decomposition

Orthogonal volumetric decomposition and parallel-distributed processing of volumetric regions have been explored for branched neuron models (Kozloski & Wagner, 2011). This approach is most suitable for the simulation of spatially organized and computationally costly elements, as the computation must be large enough to overlap the high communication required — executed at every computation timestep — between neighbouring spatial regions in different compute nodes. Memory-wise, such implementations based on spatial decomposition have been shown to yield a high memory overhead and load imbalance due to the duplication (*ghosting*) of branch sections in high density regions in networks of detailed neural networks (Magalhães et al., 2016).

Alternative implementations based on the tessellation of compartmental space has been presented for lower scales of simulations, with the main efforts driven by the STEPS simulator (Hepburn et al., 2012) for the stochastic simulation of reaction-diffusion systems at the molecular level of neuron models and extra-cellular space.

1.1.4 Variable Timestep Interpolation

An acceleration based on improved numerical resolution is also possible by utilizing a method for variable-step interpolation of individual neurons, and has been presented and implemented in NEURON (Lytton & Hines, 2005). The method details an implementation of adaptive-step interpolation of individual neurons, by utilising the CVODE library (Cohen & Hindmarsh, 1996), a C implementation of the VODE algorithm (Brown et al., 1989), part of the SUNDIALS package (Hindmarsh et al., 2005). The VODE is a Backward Differentiation Formula (BDF) method of variable-order and variable-step for the resolution of Initial Value Problems (IVPs) for stiff ordinary differential equations (Cash, 1980). For a given function and time, it approximates the derivative of a function using information from previous steps (stored in the state history), thereby increasing the accuracy of the numerical resolution. The step size is tentatively computed in order to respect an user-provided absolute tolerance, thus adapting the step size to rapid variations of voltage trajectory. Current events that cause a discontinuity of solution force the integrator to start again with a new IVP. Interpolation methods presented allow for two distinct stepping models: (1) a globally synchronous step for all neurons, mostly suitable for models with short event delays and no discontinuities such as gap junction and simulations with detailed axon branching; or (2) a local variable step per neuron advancing neurons speculatively with reversal of state in the occurrence of

overstepping and missed events.

1.1.5 Distributed Compute Environments

Due to storage and compute requirements, large scale simulations require the use of super-computing infrastructures. Similarly to single node implementations, executions follow the BSP execution model, extended to a network of compute nodes. Stepping synchronization and exchange of synapses are performed at the node and at the network level (Migliore et al., 2006; Plesser et al., 2007; Morrison et al., 2005). This is typically performed via Message Passing Interface (MPI) communication (Lusk et al., 2009), alongside local neurons synchronization via OpenMP or other threading control library. Synaptic communication is performed via collective communication calls as part of the previous synchronization step, with a collective scatter-gather (MPI_Alltoall) or gather (MPI_AllGather) operation. A hardware-specific and more scalable solution has been presented on an IBM BlueGene/P using the Deep Computing Messaging Framework (DCMF) runtime (Hines et al., 2011; Kumar et al., 2010), based on immediate selective broadcasts of spikes and a synchronization barrier at the end of every communication step.

Efficient usage of resources is possible when the input dataset is *large enough* to allow enough flexibility to balance neurons across compute units, in such way that static load balancing can be performed accurately beforehand. This has been demonstrated by the Least Processing Time (LPT) algorithm (Korf, 1998), yielding quasi-balanced workload distribution by iteratively assigning neurons to the compute node with the least total computation time.

Simulations at very large scales, particularly those requiring petascale compute power, have been covered extensively by the NEST simulator (Plesser et al., 2007) for point neuron models. At such scale, collective communication yields a significant overhead in the overall runtime. Memory and communication imbalance due to heterogeneous synaptic connectivity are the major scaling obstacles (Morrison et al., 2005; Kunkel et al., 2012). Studies have been conducted to analyse the technical aspects of running such simulations on large supercomputers and on the study of brain-scale networks, identifying memory overhead of the connection framework (i.e. synaptic connectivity) as the scaling limit of present implementations (Helias et al., 2012). However, the problem of memory imbalance is a hard problem for large network sizes, due to the heterogeneity in connectivity across synapses, and the overhead of MPI buffers required for accumulation of synapses throughout computation steps (Ippen et al., 2017). Communication-level optimizations are possible by taking advantage of the nature of the problem: neurons are connected to a small subset of other neurons that are held across a small fraction of the compute network. This allows for an optimization in communication by grouping synaptic target lists by types and number of synapses of a given type (Kunkel et al., 2014). Moreover, due to the upper limit of connectivity of single neurons, the connectivity sparsity across very large networks of neurons can be exploited by having a two-tier connectivity table that allows selective scatter-gather of synapses instead of the commonly-used collective

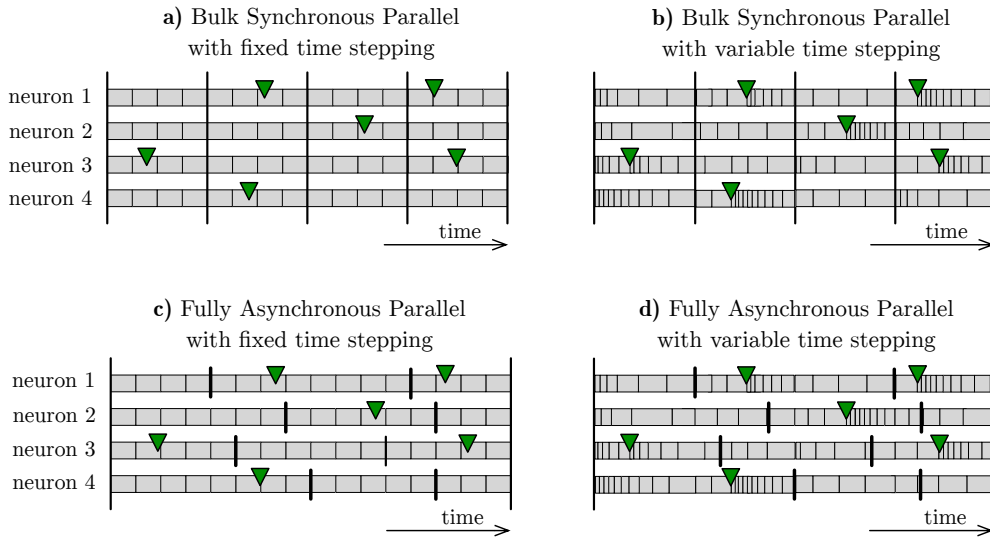


Figure 1.2 – Scope of the interpolation methods discussed, illustrated by four alternative approaches for the simulation of four neurons, on a left-to-right execution timeline. A gray cell represents a step of a neuron, with its width representing the timestep duration. Inverted green triangles represent delivery of synaptic events (solution discontinuities). Vertical bars across all neurons represent collective communication. Vertical bars across single neurons represent limit of stepping dictated by pre-synaptic neurons’ time instants. **a)** Bulk Synchronous Parallel (BSP) model with fixed timestepping, and collective synchronization; **b)** BSP model with variable time step implemented in NEURON (Lytton & Hines, 2005) and collective synchronization. Backstepping operations are omitted; **c)** Fully-Asynchronous Parallel (FAP) execution model applied to fixed step interpolation, introduced in Chapter 4, with individual synchronization of neurons based on their connecting pre-synaptic neurons; **d)** FAP variable-step method presented in Chapter 5.

gather operation in NEURON, reducing the size of the MPI receive buffers, eliminating buffers’ growth with the total number of processes, and yielding ideal scaling properties (Jordan et al., 2018).

An alternative provided by NEURON allows for a distributed execution with variable stepping, guaranteeing coherent time interpolation by enforcing neurons synchronization with a BSP-based communication barrier, therefore avoiding synaptic events being delivered in preceding instants in time (Migliore et al., 2006). Speculative stepping is allowed by a single step that traverses the synchronization instant, with posterior backstepping for missed events. The step size is limited to the instant of the nearest synaptic or discontinuity event.

For clarity, the BSP-based distributed implementations of fixed and variable timestep interpolations are illustrated in layouts a) and b) in Figure 1.2, respectively.

1.2 Limitations

The problem of efficiently computing large scale networks of highly heterogeneous neuron models is then twofold.

1.2.1 Efficient Usage of Compute Resources

At first, due to the large scale of data and computation involved in the problem specification, simulations require high performance computing techniques in order to compute the solution in a feasible time. Thus, acceleration efforts rely on the parallel and distributed computation of several neurons simultaneously.

The highly-heterogeneous specifications of modern compute architectures requires a computation model that adapts the problem to the host architecture. Due to the wide extent of scales and models required for different biological use cases, such a computation model should be flexible enough to fully utilize the available compute resources, independently of the complexity of the neuron and network models. However, state-of-the-art approaches are based on a wide collection of simulators that are often scale specific, and/or utilise only a reduced subset of the compute architecture capabilities.

Moreover, the commonly utilized MPI runtime provides a powerful communication and synchronization framework for collective operations on distributed compute nodes, with a minimal interface for point-to-point communication, making it most suitable for problems characterized by homogeneous data types and computation across compute nodes. Nevertheless, it does not provide core-level parallelism, remote procedure calls, futures, callback methods, pointer addressing in distributed memory, data balancing primitives, remote flow-control objects such as semaphores and mutual exclusion gates. In practice, it lacks several features that would be essential for the resolution of more complex problems. In addition, any extra feature that is not part of the library requires a strenuous development effort combining third-party libraries and a user-programmed workflow performing communication of intermediate data structures across nodes. Complementary, state-of-the-art efforts for *on node* acceleration such as OpenMP (Dagum & Menon, 1998) and the ones provided by the C++20 standard (Standard C++ Foundation, 2019) provide parallelism and asynchronicity, yet are limited to a single compute node. The problem is therefore not trivial.

1.2.2 Handling of Complex Neuron Models

As a second point, advancements in biological and computational neuroscience fields introduce new use cases that increase the complexity of the models currently being simulated, and that have not been properly accounted for by existing methods. To name a few:

- Higher resolution of morphological trees, combined with new ion channels and biologi-

cal mechanisms, that greatly increase the complexity of the model and the disparity in computational workload across neurons, and require micro-parallelism techniques that accelerate the simulation of individual neurons;

- Models of very large networks display highly-heterogeneous spatial and temporal activity across neurons and require an interpolation model that adapts the allocation of compute resources to the neuronal activity of significance during runtime;
- New models of synaptic plasticity require a dynamic reconfiguration of network connectivity throughout the simulation, in order to overcome the limitations of the static-connectivity graphs currently utilised;
- New models of neuronal branching growth, require dynamic fine-tuning of neuron topological trees to the hardware specifications throughout execution;
- Further complexity from biological phenomena such as astrocytes, gap junctions, and learning models that are characterized by highly-correlated and non-linear ODEs, whose computational implications have not been covered by previous research;
- Multi-scale simulations that combine several physical and time scales, introducing new data dependencies that go beyond the synaptic delay connectivity utilised in common simulations.

1.3 Motivation

These limitations motivate the search for a computation model that accelerates the simulation of networks of detailed neural models, independently of the network size, and on a wide range of compute architectures, fully utilising multi-core and vector-based capabilities on distributed networks of compute nodes.

Modern runtime systems such as the High Performance ParalleX 5 (HPX-5) (Sterling et al., 2014), Charm++ (Kale & Krishnan, 1996), Legion (Bauer et al., 2012), HPX-3 (Kaiser et al., 2014) and OmpSs (Duran et al., 2011) provide programming models and runtimes for asynchronous parallelism and heterogeneity, with the capability of handling heterogeneous tasks and control objects in distributed compute and memory architectures. Such tools increase the possibilities of new neural simulations use cases — until now restricted to the BSP paradigm and MPI communication model — and are of particular relevance in the simulation of morphologically detailed neuron models, due to their capabilities in handling connectivity, computation, and memory asynchronously. Moreover, better-parallelism of neurons should allow for better usage of computing resources in modern architectures, and consequently, for simulations to compute at a runtime closer to real time.

While there is an ongoing debate on the most appropriate execution model, with an open space of research being continuously pursued to improve current runtime systems, the functionalities provided by existing asynchronous execution models are worth exploring.

1.4 Research Scope

With that in mind, this thesis explores the field of asynchronous simulation of morphologically detailed neural networks. Our objective is to investigate the efficient simulation of large networks of highly-heterogeneous neuron models characterized by diverse activity dynamics, executed on a network of compute nodes with a wide range of hardware specifications.

We will show that asynchronous runtime systems with distributed memory addressing is not only very suitable, but *very likely* the only solution able to handle the high complexity in such resolution.

Our methods provide novel insights on the development of neuron simulators targeting efficient executions of complex neuron models on heterogeneous distributed compute architectures. The extent of the contributions presented covers from micro-parallelism methods focused on individual neuron models, to the dynamics of medium- and large-sized networks of neurons. The range of analysis in this thesis includes multi-core and SIMD accelerations, cache-efficiency, network communication, numerical accuracy, computation flows from numerical resolution, network activity, applications to biological use cases, and reduction of overall time to solution.

The capabilities of our methods are demonstrated on a prototype implementation developed on the core compute kernel of the NEURON scientific application, yielding a fully-asynchronous distributed and parallel simulation of neuron networks. Asynchronicity capabilities are provided by the HPX-5 runtime system (Sterling et al., 2014) for the ParalleX execution model (Kaiser et al., 2009) on a global address memory space with transactional memory capabilities (Kulkarni et al., 2016). We provide implementation details, drawbacks and a comparison to the BSP counterpart based on MPI and OpenMP. The flexibility in the distributed, multi-core and vector parallelism methods presented is shown to fully utilise all computing resources across a wide spectrum of host architectures, when enough computation is available. To provide substantial evidence of our results, comparisons are provided on four heterogeneous compute architectures: Intel Xeon 6140, Intel Knights Landing, Intel E5 and Cray XE6. Performance on distributed executions are demonstrated on a network of Cray XE6 nodes.

Our work addresses several limitations and advances state-of-the-art methods for large scale neuron simulations and asynchronous computation. Nevertheless, most of the methods introduced follow from first principles of numerical simulation and computer science, thus being applicable to a wide range of scientific problems.

1.5 Runtime Systems

A runtime system is a collection of components described as software (binaries, operative system, libraries), hardware, or both, that allows for an application to run on a system. The

runtime describes the set of instructions that run throughout the execution of the application. This includes user-introduced code (the application per se) and the instructions that were not written but are required for the proper execution of the program. Examples of runtime systems methods include communication, network bootstrapping, processor interfacing (threading control, atomic operations, etc.), hardware calls (IO and network), the hardware instructions set architecture (ISA), synchronization control methods (mutual exclusion objects, semaphores, atomic updates) and network operations (sending, receiving, querying and probing for messages), among others.

A wide collection of runtime systems is available nowadays with distinct features, with development of new components being actively pursued (Sterling et al., 2017). To name a few:

- GPU acceleration (Bueno et al., 2012), distributed asynchronous task-based parallelism (Bueno et al., 2011), hierarchical task-based programming (Planas et al., 2009) and self-adaptive tasks (Planas et al., 2013) in OmpSs (Duran et al., 2011);
- adaptive dynamic task scheduling to resources (Sterling & Zhang, 2018), message-drive computation (Brodowicz & Sterling, 2017), multi-level stack and hardware messaging (Autonomic Performance Environment on eXascale, APEX) (Huck et al., 2015), offload of computation to network cards (Anderson et al., 2017) in HPX-5 (Sterling et al., 2014);
- passive and active distributed global address space (Kulkarni & Lumsdaine, 2015) , and execution on embedded devices (van Wagensveld & Margull, 2017) in HPX-3 (Kaiser et al., 2014) and HPX-5; and
- automated mechanisms for data movement across compute nodes in Legion (Bauer et al., 2012).

In the context of our research, we will focus on the methods for asynchronism in Global Address Space (GAS). Asynchronism refers to the timing of compute, communication and synchronization operations across compute units not being synchronized, predetermined, or set at regular intervals. Instead, it is decided dynamically throughout the execution, typically when a previous operation is completed and the adequate hardware resource becomes available. A distributed memory space abstracts the addresses of physical memory (or memory page mapped by the Operative System) into an unique address representation across the whole network, managed by the runtime system. In practice, it allows unique data and functions addressing in the global address space across all physical compute nodes and distributed memory regions.

For brevity, we provide descriptions of two sample runtime systems, that will be utilized as the base of analysis in the following thesis: the synchronous MPI with no GAS, and the asynchronous HPX with GAS.

1.5.1 MPI: Message Passing Interface

The Message Passing Interface (MPI) (Gropp et al., 1999) is a standard of a protocol that defines operations for communication, network control, Input/Output (IO), management of compute nodes on the network, among others, running a similar parallel process distributed across all compute nodes on a network. MPI is the dominant execution model used in high performance computing in current times (Sur et al., 2006).

The most common MPI-based execution workflow performs a synchronous computation of similar compute operations in distinct datasets stored across compute nodes, and exchanges data across nodes at predetermined collective communication steps. MPI works on a distributed memory environment, however data structures addressing is performed at the level of memory of individual nodes. Point-to-point messaging allows for selective messaging across pairs of nodes in the network. Selective broadcasts are possible on communication windows to a subset of the network. Other methods include collective and individual operations for input and output, network topology management, and one-way communication with *get/put/accumulate* methods on remote memory windows.

The MPI programming challenge is to properly load balance a distinct subset of input data across the network nodes, serialize and synchronize inter-node data exchange, and program the control of the execution flow of compute nodes, in order to avoid missed messages, incoherent data states, deadlocks, etc.

The newest MPI v3 (Forum, 2012) release provides non-blocking collective operations, inexistent in the previous version 2. These, alongside the point-to-point non-blocking routines in MPI v2, are sometimes also referred to as asynchronous communication routines, to refer an operation that can be performed without halting the execution on the initiator compute node, and without waiting for the destination processor to receive the message.

1.5.2 HPX: High Performance ParalleX

The HPX-5 interface and runtime library (Sterling et al., 2014) is a realization of the ParalleX execution model for exascale execution (Cimini et al., 2011; Kulkarni et al., 2016; Kissel & Swany, 2016). It consists of lightweight threads and active message parcels, operating within the context of an active global address space, and synchronizing through lightweight local control objects (LCOs) such as futures or local reductions and distributed collectives that support dynamic and irregular participation.

The Global Address Space (GAS) is flat and byte addressable and supports block-based allocation through a *malloc/free* API. Blocks may be allocated locally or remotely, or as part of distributed arrays. Array distributions can be cyclic or user-defined. The mapping of blocks to physical compute nodes (henceforth also denominated as **localities**) can vary dynamically at runtime. Threads may access global data directly through local aliases, may access remote global data through an asynchronous *memget/memput* API, and/or may send parcels to

global addresses.

Parcels contain immediate data, a specification of the action to perform, and continuation information such as "return to me" or "forward the computed value as an input to an LCO." When a parcel arrives at the locality associated with its target global address the HPX-5 runtime will invoke it as a new lightweight thread. The parcel-thread isomorphism and global address space are key to writing programs that work portably across shared memory, distributed memory, and hybrid architectures.

HPX-5 applications are written to be adaptive and data-driven. Parcels move computation to data while LCOs provide runtime-visible data and control dependent execution. Combined with the ability to remap global data, this design can minimize network traffic and allows a dynamic scheduler to map irregular computation onto the available resources.

1.6 Reference Implementation

Our use case is the digital reconstruction of an *in vivo* laboratory experiment, applied to a previously published network of morphologically detailed neuron models (Markram et al., 2015).

The reference implementation follows the simulation workflow implemented in NEURON (Carnevale & Hines, 2006), categorized by the following properties: (1) neurons are branched representations of spatially-discretized capacitors with ionic current channels — hereinafter referred to as **compartments**; (2) neurons are represented by ODEs that define the current on the capacitor and the voltage-dependent opening of each ion channel or biological mechanism; and (3) ODEs are coupled with a time dependency, based on the synaptic connectivity between neurons. The mathematical formalism follows.

1.6.1 Computational Model

The topology of a neuron is described by a tree of resistors, with capacitors and nonlinear resistive current flows at each node (compartment) connected to ground. The RC circuit that models the electrical current passing through the membrane of a compartment n is modelled by:

$$C \frac{dV_n}{dt} = - \sum_i g_i x_i (V_n - E_i) + I(t) \quad (1.1)$$

where g_i and E_i describe the conductance and reversal potential of the ionic channels, respectively. Synaptic currents or injected current stimuli, if any, are included in $I(t)$. The term x_i models the opening probability of the transmembrane ion channel currents, typically described by a voltage-gated ODE. The RC circuit underlying the current passing through a compartment is illustrated in Figure 1.1 IV.

1.6. Reference Implementation

The original formulation of ionic activity was introduced by the Hodgkin-Huxley (HH) model, and includes the activity of the ion channel gating states that model the flux of sodium (index Na) and potassium (index K) currents. All remaining currents are included in the leak current (index L), leading to the formulation:

$$\sum_i g_i x_i (V_n - E_i) = g_{Na} m^3 h (V_n - E_{Na}) + g_K n^4 (V_n - E_K) + g_L (V_n - E_L). \quad (1.2)$$

For brevity, the equations of the voltage-dependent variables m , n and h describing the opening of the ion channels as first-order ODEs were omitted. Together with the current function in Equation 1.1, they describe the system of ODEs that describes the dynamics of a neuron defined by a single compartment, or a point neuron. A more detailed (extended) HH model includes a wider range of ion channels and currents such as Calcium. Refer to Channelpedia (Ranjan et al., 2011) for a collection of existing ion channel specifications.

The computational model does not describe capacitance, resistance and conductance as a total value but as a value per unit length instead, and adds the axial resistance of the neurites (between compartments) to the current expression in Equation 1.1, leading to an equation of the form:

$$C \Delta x \frac{dV}{dt} = - \sum_i g_i \Delta x (V - E_i) + I(t) \Delta x + \frac{V_{n+1} - V}{r_{n+1} \Delta x} - \frac{V - V_{n-1}}{r_{n-1} \Delta x} \quad (1.3)$$

where the subscripts $n-1$ and $n+1$ represent the indices of the previous and following compartments. The new contributions are provided by Ohm's Law and the neuronal cable theory (Niebur, 2008). The term r defines the axial resistance per unit length as a function of the diameter and the cytoplasmic resistivity. Dividing both sides of the equation by Δx leads to the final formulation:

$$C \frac{dV}{dt} = - \sum_i g_i (V - E_i) + I(t) + \frac{V_{n+1} - V}{r_{n+1} \Delta x^2} - \frac{V - V_{n-1}}{r_{n-1} \Delta x^2}. \quad (1.4)$$

We can simplify it using the definition of approximation ($\lim_{x \rightarrow 0}$) of the second derivative using finite difference methods:

$$\frac{V_{n+1} - V}{r \Delta x^2} - \frac{V - V_{n-1}}{r \Delta x^2} = \frac{1}{r} \frac{V_{n+1} - 2V + V_{n-1}}{\Delta x^2} \approx \frac{1}{r} \frac{d^2 V}{dx^2}. \quad (1.5)$$

We now replace Equation 1.4 with the value for the second derivative, leading to the final formulation:

$$C \frac{dV}{dt} = - \sum_i g_i (V - E_i) + I(t) + \frac{1}{r} \frac{d^2 V}{dx^2}. \quad (1.6)$$

1.6.2 Spatial Discretization

As we cannot find an analytical solution on arbitrary neuron geometries/morphologies, we resort to discretization and numerical approximation as a way to make the problem tractable. Therefore, we perform a spatial discretization of the neuronal morphology — from biologically inspired to HH-based compartmental representation. Moreover, the model assumes the spatial discretization to be small enough, so that the second order correctness implies that the value at any spatial location is sufficiently close to the average value of the compartment, represented by the nodal value of the solution to the discretised problem.

Subsequently, it assumes the state of the axonic branches to be constant throughout the execution, therefore simulating only soma and dendrite sections. Thus, upon an action potential (known as a spike) of a neuron, the synaptic propagation delay between two neurons includes the current propagation from the soma (or axon initial segment) to the bouton in the axon, plus the time required for the electro-chemical reaction and neurotransmitter release at the synapse.

1.6.3 Branching

A branched representation of a neuron allows for more details from the morphology, by adding the neighbouring compartments' contributions according to the neuronal cable theory for multiple compartments. Branched neuron trees include the terms defining the currents derived from the parent and children branches, extending Equation 1.3 to the final formulation:

$$C \frac{dV}{dt} = - \sum_i g_i (V - E_i) + I(t) + \frac{V_{p(n)} - V_n}{r_{p(n)} \Delta x^2} - \sum_{c: p(c)=n} \frac{V_n - V_c}{r_c \Delta x^2} \quad (1.7)$$

where $p(c) : \mathbb{N} \rightarrow \mathbb{N}$ returns the index of the parent compartment of a given compartment c and r_c is the resistance of the connectivity to neighbouring compartments, if any. A parent compartment is the compartment immediately above on a sequence or tree structure. A converse logic holds for the definition of children compartments. Due to having no analytic solution, numerical methods are employed with a problem-optimized solver used for the resolution of the system of equations.

NEURON's algebraic solver (Hines, 1984) describes each neuron as a sparse-tridiagonal matrix that represents the voltage in a compartment as a function of the main diagonal of the matrix, the contributions from parents and children on the upper and lower diagonals respectively, and the mechanism contributions to the voltage on the right hand side of the matrix-vector multiplication. Following the compartment numbering convention used by NEURON (Hines et al., 2008), the tree of compartments is numbered using a depth-first search scheme from root to leaves, ensuring that the index of a parent compartment is larger than all its children and smaller than its parent. The aforementioned ordering rule guarantees that the matrix is symmetric and in each row i there exists a single non-zero element with columns index j

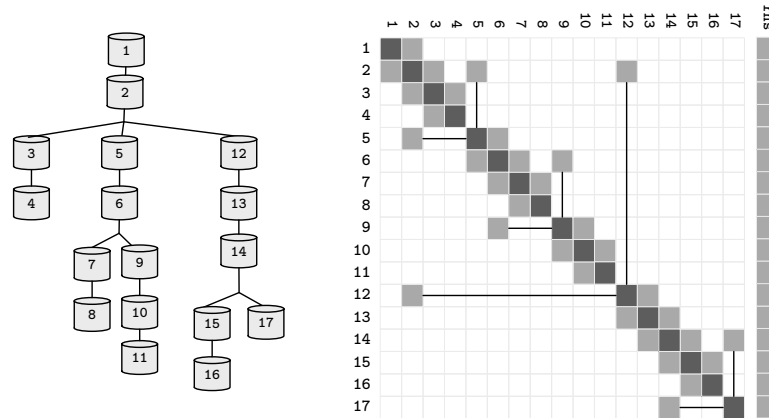


Figure 1.3 – The topological structure of a sample neuron, and its representative sparse tridiagonal dendritic tree representation. Lower and upper diagonals include parent and children contributions to the current function (a and b in Equation 1.8). Main diagonal (d) includes the changes to the compartment voltage dV/dt induced by the capacitance and mechanisms. Remaining terms are included in the right hand side (rhs) vector. Straight lines display connections between compartments with non-sequential indices (referenced by p).

such that $j < i$, i.e. a single parent compartment per branch. Upon the computation of all mechanism contributions, all compartments must solve:

$$b_{p(n)}V_{p(n)} + d_nV_n + \sum_{c: p(c)=n} a_cV_c = rhs_n \quad (1.8)$$

where $d, a, b \in \mathbb{R}$ are the coefficients of the voltage contribution of the compartment n (as lower, upper and main diagonal respectively), its children and parent compartment, on a neuron with N compartments. For completeness, refer to Figure 1.3 for a sample neuron and its sparse tridiagonal matrix representation.

Given a tree with the aforementioned terms updated, the final solution of the system can be computed by a problem-specific implementation of the Gaussian Elimination, that only modifies the d and rhs vectors, as a and b are constant. Further details are provided later in Chapter 2, where branching structure is covered in detail.

1.6.4 Fixed Step Interpolation

Fixed timestep interpolation is possible with either implicit Backward Euler or Crank-Nicholson methods. The gap between the analytic and numerically solved solution is particularly high when the system's voltage is changing rapidly, making the conditionally A-stable Forward Euler method particularly susceptible to oscillations and therefore not used (Carnevale & Hines, 2006). The Backward Euler method computes the solution of a set of nonlinear simultaneous equations at each step. To reduce the complexity and number of equations of the system at each timestep, NEURON uses staggered resolution, and offsets the calculation of gating

variables and compartment by half-step. Its numerical error is proportional to Δt , making it unable to deal *accurately* with discontinuities, in practice delivering events at the next timestep and yielding an error propagation with an average of $\Delta t/2$ (Casalegno et al., 2016).

An alternative variant of Crank-Nicholson method applies Strang splitting to calculate the values of the compartment voltage and Hodgkin-Huxley gating states on interleaved time intervals, providing second-order accuracy (Carnevale & Hines, 2006). In practice, a direct solution of voltage equation using a linearized membrane current $I(V, t) = g(V - E)$ at a timestep $t \rightarrow t + \Delta t$ is possible if the conductance g and reversal potential E have second-order accuracy at time $t + \Delta t/2$. Since the conduction of HH-type channels is given by a function of state variables $n (K^+)$, m and $h (Na^+)$, this second-order accuracy at $t + \Delta t/2$ is achieved by performing a calculation with a timestep offset of $\Delta t/2$ from the current voltage timestep. Nevertheless, NEURON adopts Backward Euler as the default method, as Crank-Nicholson is not L-stable: its second-order accuracy yields high oscillations around solutions, particularly for events with infinitesimal small duration such as ideal current pulses and gap junctions.

Each fixed step iteration is defined by a fixed step size of length Δt , enough to capture the resolution of the fastest mechanism — typically the fast Na^+ channels. The community-defined standard value for the fixed step size is $25\mu s$.

1.6.5 Parallel Executions

Parallel executions require the synchronization of stepping across neurons in the simulation. The maximum time distance that a neuron can distance itself from a given pre-synaptic neuron is determined by the synaptic delay of the fastest synapse of the pair. The rationale is that in the event of a spike from a pre-synaptic neuron at time t , the state of the receiving post-synaptic neuron cannot have been resolved for a later time than $t + t_{syn}$ — where t_{syn} is the synaptic delay between the pre- and the post-synaptic neuron — or the neuron will have *overstepped* and missed the interpolation instant. However, an alternative execution is possible with a *speculative* execution model. In practice, events delivered in past time instants are recovered by discarding the extra timesteps and *backstepping* the state of the neuron to the delivery time of the event. This topic will be covered in depth later in Chapter 5.

The event delay interval varies extensively across pairs of neurons. Therefore, the shortest propagation delay in a neural network that rounds down the multiple of the computation timestep is used as the collective synchronization and communication step size for the exchange of synapses to be delivered within the next interval. In our model, this is computed as 0.1 milliseconds, or equivalently four computation timesteps. An illustration of this synchronization protocol is illustrated in diagram a) in Figure 1.2.

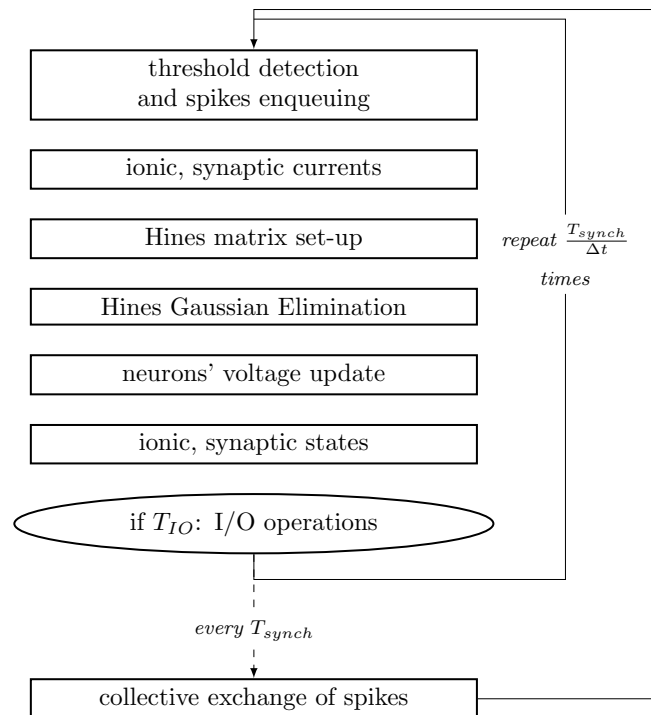


Figure 1.4 – Workflow of a compute step of the numerical resolution of a detailed neuron interpolation, based on interleaved voltage-states resolution. T_{synch} and T_{IO} are the communication and IO step intervals, respectively. Simulations on single compute nodes exclude the step for collective exchange of spikes.

1.6.6 Workflow

The workflow of a computation timestep of duration Δt , executed on a network of neurons with minimum synaptic delay T_{synch} , is presented in Figure 1.4 and is as follows:

1. Voltage at the Axon Initial Segment (AIS) compartment of the neuron is checked against the spike threshold. The threshold value is not constant in the HH model, thus voltage at AIS is checked against a value high enough to guarantee a spike;
2. Spikes delivered during the previous communication interval are put in a time-ordered event queue. Neurons check for events to be delivered in the following step, and deliver them, if any;
3. Mechanisms activation leads to a change in the conductance of the compartments, and to a new current contribution of each ion channel to the main current function;
4. The Gaussian Elimination method computes the algebraic resolution of the system of linear equations, providing the compartmental voltages at the new step;
5. The internal states of mechanisms are updated from the new voltage value;

6. For every step where Δt divides T_{synch} , a collective communication call delivers the synapses for the subsequent period. Received spikes are added to the time-ordered queue. A similar logic applies to T_{IO} for the output of state.

1.7 Thesis Structure

This thesis is organized as a compilation of published and submitted scientific articles, in line with the following EPFL Regulations: Doctoral Commission (CDoct) Decisions 109 (November 2015) and CDoct 110 (January 2016). The contributions are detailed in four chapters, referring to individual publications, and summarized in the next sections.

1.7.1 Asynchronous Branch-Parallelism

Chapter 2 advances the topic of distributed asynchronous branch parallelism, a method that explores dependencies on the topological structure of neurons. The contributions are organized in six components: (1) a formal description of the data dependencies underlying the branching structure of neurons; (2) a method that describes and extracts read-after-write dependencies from the resolution of the underlying solver; (3) a load balancing algorithm that decomposes neurons representation into an subtrees, enabling a balanced multi-core execution; (4) a method for the transformation of subtrees into a vector-friendly memory layout that accelerates execution in the SIMD axis; (5) an asynchronous parallel execution model for the resolution of the mathematical dependencies between equations of connecting branches, based on an active producer-consumer execution model supported by distributed placeholders; and (6) a dynamic finer-grained version of the LPT-based load balancing algorithm for distributed networks, that delegates neuron sections to different compute nodes in order to balance the computational workload, while minimizing network communication.

The computation model exposes tree parallelism on a distributed memory environment, with data representations dynamically adapted to maximise multi-core and SIMD processing across compute nodes in the network. Active point-to-point communication asynchronously updates three dependency values held on a local memory region or in remote nodes. Communication is minimized by reducing the number of subtree interconnections on different compute nodes, and by removing transitive connectivity across remote subtrees. Moreover, we take existing Gaussian parallelism concepts to the limit in a useful way that yields significant performance gains in distributed multi-core SIMD architectures.

Benchmark results demonstrate that multi-core and SIMD acceleration are competitive axes of acceleration, and that there is an optimal value for subtree size that maximizes the achievable speedup. Benchmarks demonstrate 2.2-10.5x speedup on four distinct compute architectures, and 1.9-7.4x on a distributed network of 128 Crazy XE6 compute nodes, compared to state-of-the-art implementations.

1.7.2 Asynchronous Graph-Parallelism

Chapter 3 describes an automated method for the acceleration of simulations on a wide range of scientific problems represented by large systems of ODEs. The application of the methods to our use case exposes a balanced SIMD-enabled computation graph from the system of equations. The following components are detailed: (1) a method for the extraction of the flow dependencies and concurrent output operations from the interdependencies across state variables on a system of ODEs; (2) an algorithm for the embarrassingly parallel processing of blocks of independent ODEs; (3) a method for SIMD acceleration of blocks of similar instances of ODEs; (4) a load balancing algorithm for balanced execution blocks based on the computation time of individual mechanism ODEs; and (5) an asynchronous execution model that allows for good leverage of computing resources by producing enough micro-parallelism to fully utilizing compute units on a wide range of architectures, based on a trades off between two competitive parallel resources — SIMD units versus multiple cores.

Benchmark demonstrates a speedup of 4-7x on the state-of-the-art serial SIMD implementation and 13-40x over its Single Instruction (SISD) counterpart in four distinct compute architectures. Benchmark of distributed executions on 128 Cray X6 compute nodes yields a speedup of 2-8x with almost ideal strong scaling for large datasets.

1.7.3 Fully-Asynchronous Fixed-Step Execution Model

Chapter 4 introduces the fully-asynchronous parallel execution model and the *exhaustive yet not speculative* stepping protocol that improve cache locality and provide cache-level acceleration. The model performs a fully-asynchronous simulation — with asynchronous computation, communication and synchronization — that advances neurons' ODEs timestep beyond synchronization barriers, and based on the time couplings between equations, as illustrated in layout c) in Figure 1.2.

The 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.

The methods introduced allow for an acceleration beyond the BSP theoretical limit, demonstrating a superlinear speedup of 25%-65% across four distinct compute architectures, and 15%-40% on distributed executions on 32 Cray compute nodes compared to the state-of-the-art BSP implementation.

1.7.4 Fully-Asynchronous Variable-Order Variable-Timestep Execution Model

Chapter 5 follows from the aforementioned asynchronous strategy and introduces a fully-asynchronous fully-implicit solver of variable-order variable-timestep interpolation of detailed neuron models, that benefits from cache-efficient barrier-free synchronization and performs timesteps larger than BSP communication intervals. The execution model is illustrated in layout d) in Figure 1.2.

The description of our methods start with the mathematical formalism underlying the simulation of our use case and its resolution with variable stepping interpolation. We show that by following an *earliest neuron steps next* scheduler, we allow for large time interpolation intervals, and maximise the efficiency of the variable step interpolator beyond what was believed to be possible in fixed-step executions. We perform an analysis of numerical accuracy, demonstrating higher precision in our methods compared to the fixed step counterpart, due to an exact delivery of events on a continuous time line, and better interpolation of the stiff dynamics that underlie the activity of neurons. An analysis of the sensitivity of our model follows, based on stiffness in solution and solution discontinuities from synaptic events. The results demonstrate low dependency of speedup and step count on solution trajectory, and a high dependency on synaptic activity on up to circa 1000 spikes per minute. The feasibility of our methods is studied, by performing a digital reconstruction of a laboratory experiment on 217 thousand neurons. The results demonstrate a substantial possibility for acceleration, due to a discontinuity rate below the 1000 Hz threshold, and the suitability of our methods for most (but not all) neurons in the network.

We analyse the efficiency of our variable step methods with a benchmark of the simulation of neural networks ranging from 1024 to 65536 neurons on 64 Cray XE6 compute nodes. We simulate the electrical activity of five spiking regimes that characterize distinct dynamics of the mammal brain. Benchmarks demonstrate a possible acceleration of 544-65x for a mean spiking rate frequency of 0.25Hz representing a majority of neurons in the brain during regular activity, down to 7.7-1.8x to a moderate regime of 6.5Hz, and 2x to no acceleration for a spiking regime of 38 Hz, a pattern of unlike occurrence or of short duration. We detail the dependency of runtime on input size and spiking activity, and show that the computational complexity of our method is dependent on the network size and synaptic activity. These results are measured in the context of the aforementioned laboratory experiment set up, demonstrating a possibility of a speedup of 224.5-11.9x, for the variable timestep methods with a precise delivery of events in a continuous time line, on the 1024-65536 dataset tested. Two higher-efficiency yet lower-accuracy implementations based on half- and full-step fixed-timestep grouping of events are presented, demonstrating a speedup of 225.1-17.1x and 228.5-24.6x, respectively. Finally, we demonstrate that preservation of the acceleration achieved is almost fully guaranteed for larger neuron networks, as over 95% of neurons are characterized by spiking spiking regimes that demonstrate the preservation of the speedup measured.

Chapter 6 draws the final discussion and conclusions.

2 Asynchronous Branch-Parallelism Extracted from Neuron Morphology

This chapter is adapted from the preprint version of the following article:

Magalhães B., Hines M., Sterling T., Schürmann F., "Asynchronous Branch-Parallel Simulation of Detailed Neuron Models", submitted to *Frontiers in Neuroinformatics* 2019

Personal contributions: conceptualization, formal analysis, investigation, methodology, software, validation and writing.

2.1 Abstract

Simulations of the electrical activity of networks of morphologically detailed neuron models allow for a better understanding of the brain. State-of-the-art simulations describe neurons by the dynamics of the branching topology, synaptic currents and biochemical processes. Acceleration of such simulations is possible in the weak scaling limit by modelling neurons as indivisible computation units and increasing the computing power. Strong scaling and simulations close to biological time are difficult, yet required, for the study of synaptic plasticity and other use cases requiring simulation of neurons for long periods of time. Current methods rely on parallel Gaussian Elimination, computing triangulation and substitution of many branches simultaneously. Existing limitations are: (a) high heterogeneity of compute time per neuron leads to high computational load imbalance; and (b) difficulty in providing a computation model that fully utilises the computing resources on distributed multi-core architectures with Single Instruction Multiple Data (SIMD) capabilities.

To address these issues, we present a strategy that extracts flow-dependencies between parameters of the ODEs modelling the current equation and the algebraic solver of individual neurons. Based on the map of dependencies exposed, we provide three techniques for memory, communication, and computation reorganization that yield a load-balanced distributed asynchronous execution. The new computation model distributes datasets and balances computational workload across a distributed memory space, exposing a tree-based parallelism

of the neuron topological structure, an embarrassingly parallel execution model of neuron subtrees, and a SIMD acceleration of subtree state updates.

The capabilities of our methods are demonstrated on a prototype implementation developed on the compute kernel of the NEURON scientific application (Kumbhar et al., 2019), built on the HPX runtime system for the ParalleX execution model. Our implementation yields an asynchronous distributed and parallel simulation that accelerates single neuron to medium-sized neural networks. Benchmark results display better strong scaling properties, finer-grained parallelism, and lower time to solution compared to the state of the art, on a wide range of distributed multi-core compute architectures.

2.2 Introduction

This chapter introduces a method for the acceleration of the simulation of individual neurons, by exploring micro-parallelism extracted from inter-compartmental connectivity in neuron topological trees.

Finer-grained parallelism of individual neuron models has been exploited in a limited way through the NEURON multisplit approach (Hines et al., 2008) — henceforth denominated as previous branch-parallelism efforts or simply **multisplit** — executed on a single-core, Single Instruction Single Data, distributed compute architecture. The multisplit method implements a parallel computation of neuron branches by converting a given topology into a tree-structure of connecting backbones (linear sequence of unbranched compartments) and leaf subtrees. The method performs a numerical reduction of each backbone’s tridiagonal matrix to a N-shaped matrix (diagonal and full first and last columns). All subtrees and backbones are triangularized simultaneously and the ends of the M backbones form a $M \times M$ reduced tree matrix that, after Gaussian Elimination, allows backsubstitution, again in parallel of all the subtrees and backbones. The matrix is later converted to a 2×2 matrix that aggregates the contributions to be exchanged to parent and leaf subtrees. Subtrees are distributed across compute nodes. Contributions of subtrees to the main tree are computed independently and reduced at every computation step. Tree-based parallelism requires information spawn/reduce throughout connecting subtrees, yielding a limit of parallelism dictated by the slowest subtree to perform a computation step. The method provides a substantial speedup on the architectures tested at the time, however it lacks support for load-balancing and processing based on SIMD-based acceleration available in modern compute architectures.

Complementary, acceleration of the linear solver describing the system of equations of individual neurons has been implemented in the NEURON simulator with *The Hines Solver* (Hines, 1984), allowing a computational complexity of $O(n)$ for a neuron tree with n compartments. The Hines solver is a specialization of the Gaussian Elimination to the sparse tridiagonal representation of neuron morphologies detailed — refer to Appendix A for the mathematical reduction. An acceleration of the Hines solver has also been proposed via Exact Domain Decomposition on GPUs of Hodgkin-Huxley (HH) based neuron models, where branching is

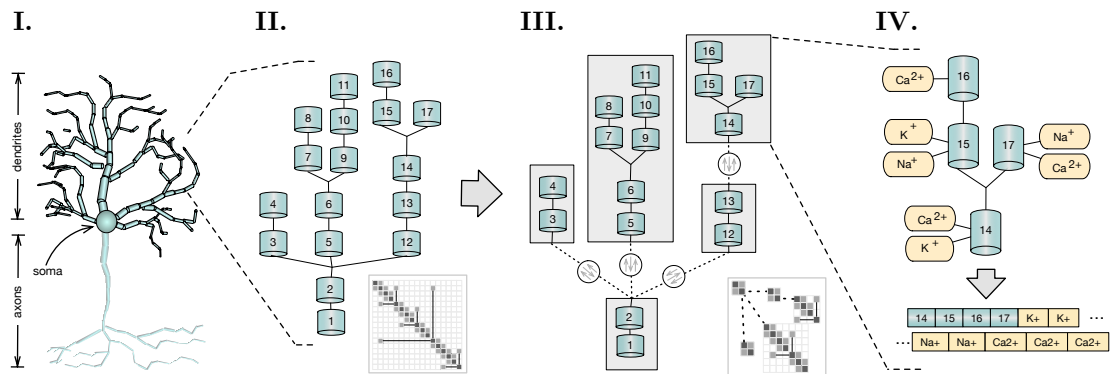


Figure 2.1 – Scope and key concepts of the methods presented. **I.** A spatial discretization of a neuron into two compartmental trees representing axons and dendrites branching; **II.** A sample of a dendritic tree and its representative branching matrix structure in thumbnail. Simulation dependency parameters across compartmental connectivity and respective matrix are extracted from these structures; **III.** A method for load balancing recursively tests for the optimal tree decomposition, based on the computational workload of each possible subtree. Initial compartmental tree is decomposed into a tree of subtrees and distributed across a distributed memory space. Matrix decomposition follows accordingly. Resolution of independent subtrees is computed asynchronously, with three variable dependencies on parent-children connectivity across subtrees (in dashed), synchronized throughout the execution; **IV.** Subtree memory layouts are reorganized to provide SIMD-acceleration of mechanism (ion channel) state updates and solver resolution.

performed via Exact Domain Decomposition by creating subdomains at the graph bifurcations with degree greater or equal than two (Vooturi et al., 2017). However, parallelism based on bifurcation points does not properly handle detailed topologies that yield a high disparity of branch lengths, and does not take into account workload imbalance for complex non-HH compartment models.

The work presented advances the state-of-the-art branch-parallelism method available in the NEURON simulator (Hines et al., 2008), extending it to an asynchronous execution model on distributed multi-core SIMD-enabled computing platforms. Our methods are based on the decomposition of the topological tree of neurons into a tree of subtrees which represent a subset of the initial problem. Its main object is to provide finer-grained representation of neuron models and lower time to solution, by performing parallel computation of balanced SIMD subtrees across a large network of compute nodes. The main point of relevance of this work is the acceleration of single neurons. However, as we will later demonstrate, the fine granularity, strong scaling capabilities, and load balancing methods introduced allow as well for a significant acceleration of medium-sized neural networks of up to few thousand neurons.

The contributions of this chapter are the following: (a) a formal description of the data dependencies underlying the branching structure of neurons; (b) a method that describes and ex-

tracts read-after-write dependencies from the underlying algebraic solver; (c) a load-balancing algorithm that decomposes neurons representation into a minimal number of subtrees that enables a balanced multi-core execution; (d) a method for the transformation of subtrees into a vector-friendly memory layout that accelerates execution in the SIMD axis; (e) an asynchronous parallel execution model for the resolution of the mathematical dependencies between equations of connecting branches, based on an active producer-consumer execution model of flow dependency variables; and (f) a dynamic finer-grained version of the LPT-based load balancing algorithm for distributed networks, that delegates neuron sections to different compute nodes in order to balance the workload, while minimizing network communication. An overview of the scope of the research and key conceptual advancements covered in this chapter are displayed in Figure 2.1.

In order to demonstrate the efficiency of the methods presented, our strategy was implemented on the compute kernel of the NEURON scientific application (Kumbhar et al., 2019), with asynchronicity capabilities provided by the HPX runtime system (Sterling et al., 2014) for the ParalleX execution model (Kaiser et al., 2009) on a global address memory space with transactional memory capabilities (Kulkarni et al., 2016). Applied to a network of morphologically detailed neuron models from (Markram et al., 2015), our strategy is shown to provide close to full usage of computing resources (when enough computation is available), finer grained parallelism, and higher multi-threading and SIMD capabilities, validated on three highly heterogeneous neuron models. The benchmarks of individual neurons demonstrate a 2.2x to 10.5x speedup compared to the reference NEURON multisplit implementation on four distinct compute architectures. Large scale executions yield a speedup of almost twofold on a network of 128 Cray XE6 compute nodes simulating 4096 neurons.

2.3 Methods

2.3.1 Dependency Parameters

In Section 1.6.3 we detailed the branching structure of a neuron, and mentioned that each compartment must resolve $b_{p(n)}V_{p(n)} + d_nV_n + \sum_{c: p(c)=n} a_cV_c = rhs_n$ at every iteration, where b , d and a are constants as a term of the parent, current, and children compartment contributions (Equation 1.8). With that in mind, the algorithms describing the implementation of the matrix set-up and solver resolution methods are presented in Figure 2.2. Parameters that are required to be communicated between connecting compartments are color-coded. The direction of the parameter flow dependencies is indicated by the direction of the arrow in the same color — top-down for parent-to-children, bottom-up for children-to-parent. The read-after-write dependencies are:

1. *Branching contributions to RHS*: the set-up of the Right Hand Side requires the contribution of the difference in potential between connecting compartments ($v_{p(n)} - v_n$) and their constant branching contributions a_n and b_n . An updated voltage from chil-

Algorithm 1 Matrix Initial Set-up	Algorithm 2 Hines Solver and Voltage Update
<pre> 1: Mechanism contributions to dV/dt and RHS: 2: for $n \in [1, N]$ do 3: $d_n \leftarrow d_n + \text{current}_d(n)$ 4: $rhs_n \leftarrow rhs_n + \text{current}_rhs(n)$ 5: end for 6: Branching contributions to RHS: 7: for $n \in [2, N]$ do 8: $\uparrow rhs_{p(n)} \leftarrow rhs_{p(n)} + a_n (v_{p(n)} - v_n)$ 9: $rhs_n \leftarrow rhs_n - b_n (v_{p(n)} - v_n)$ 10: end for 11: Voltage decay and branching contributions to D: 12: for $n \in [2, N]$ do 13: $d_n \leftarrow d_n + \text{voltage_decay}(n)$ 14: $d_n \leftarrow d_n - b_n$ 15: $d_{p(n)} \leftarrow d_{p(n)} - a_n$ 16: end for </pre>	<pre> 1: Backward triangulation 2: for $n \in [N, 2]$ do 3: $\uparrow d_{p(n)} \leftarrow d_{p(n)} - b_n a_n/d_n$ 4: $\uparrow rhs_{p(n)} \leftarrow rhs_{p(n)} - rhs_n a_n/d_n$ 5: end for 6: Forward substitution (solution to system of ODEs) 7: $rhs_0 \leftarrow rhs_0/d_0$ 8: for $n \in [2, N]$ do 9: $\downarrow rhs_n \leftarrow rhs_n - b_n rhs_{p(n)}$ 10: $rhs_n \leftarrow rhs_n/d_n$ 11: end for 12: Voltage update with dV/dt 13: for $n \in [1, N]$ do 14: $v_n \leftarrow v_n + rhs_n$ 15: end for </pre>

Figure 2.2 – Algorithms of the initial matrix values set-up (left) and Gaussian Elimination and voltage update methods (right) of a neuron discretized by N compartments. Data-dependency variables are emphasized in coloured text, with direction of arrow in same colour pointing up if children-to-parent flow dependency, or down otherwise. Variables notation follow Equation 1.8.

- dren to parent ($\uparrow v_n$) allows for the computation of the rhs vector set-up, yielding a children-to-parent flow dependency;
- 2. *Backward triangulation*: a children-to-parent flow dependency allows for the completion of the Backward Triangulation step by providing the children compartment contributions to their parents' diagonal and right hand side values ($\uparrow d_n, rhs_n$);
- 3. *Forward substitution*: rhs values are required to be modified in the root-to-leaves direction, in order to compute the final value of children's rhs , leading to a parent-to-children flow dependency ($\downarrow rhs_{p(n)}$). The outcome of the substitution step is the updated rhs for the current step, i.e. the voltage values;

For subtrees with a single connection point to a parent node, the back-triangulation is complete, i.e. triangulation at its parent subtree can be initiated immediately after. For subtrees with two connection points, triangulation stalls until the triangulation on both children branches is completed. Substitution follows a converse logic, being optimally computed for leaf (single connection point) subtrees, and requiring substitution at parent subtrees to be completed beforehand if not a leaf subtree.

Mechanism state update of compartments (*current* and *state* functions in the workflow presented in Figure 1.4) can be performed independently, as long as the previous dependency variables are resolved. Thus, the finest granularity of parallelism (or the smallest compute task) can now be modelled at the compartment level.

More importantly, granularity of tasks can be increased and decreased by clustering connecting

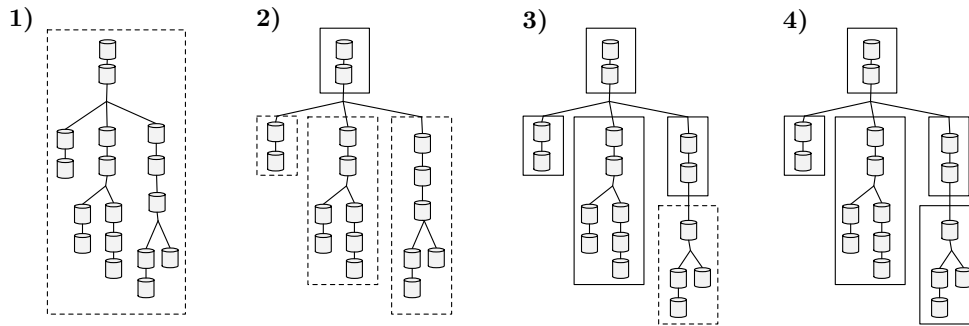


Figure 2.3 – A sample workflow of the algorithm that decomposes neuron morphologies into a tree of subtrees. Dashed contours display groups of compartments being tested against the maximum computational complexity threshold. Straight contours represent a set of compartments with a total computational complexity below the threshold, clustered into a subtree. **1)** The total runtime of the initial tree is computed and compared with the complexity threshold. **2)** The previous cluster exceeds the allowed computational complexity threshold. The following sequence of non-bifurcated compartments that yields smaller complexity (top two compartments) is clustered into a subtree. The connecting three branches are tested. **3)** Both left and center sections are benchmarked and their execution time is below the threshold, thus becoming two independent subtrees. The same threshold test for the right region fails. The first two compartments in the right region yield less runtime than the complexity threshold and are clustered into a subtree. **4)** The remaining compartments on the right branch of the tree comply with the complexity test and are clustered, leading to the final data representation.

compartments. With that in mind, the following section provides a clustering method that takes advantage of this property to perform load-balancing on multi-core execution units.

2.3.2 Neuron Tree Decomposition and Parallel Execution

The problem of scaling the model of a neuron efficiently across any number of compute units with vectorization (SIMD) capabilities is twofold: at first, there should be a *large enough* number of compute tasks to provide enough flexibility in the distribution of tasks. The rationale is that a high number of tasks allows for a better balancing of the total workload (sum of task runtimes) across processors. Secondly, the number of total tasks should ideally be as small as possible, so that vectorization can be maximised inside each task and threading (de)allocation overhead is minimized. This leads to a competitive trade-off between multi-core and vector acceleration.

To fine-tune the data representation to the host compute architecture, our strategy takes advantage of the flexibility in granularity presented in the previous section. A loading algorithm recursively traverses the tree and clusters connecting compartments that yield a computational workload as close as possible to a given threshold. The cluster is a subtree of compartments whose computational complexity is provided by the runtime of all com-

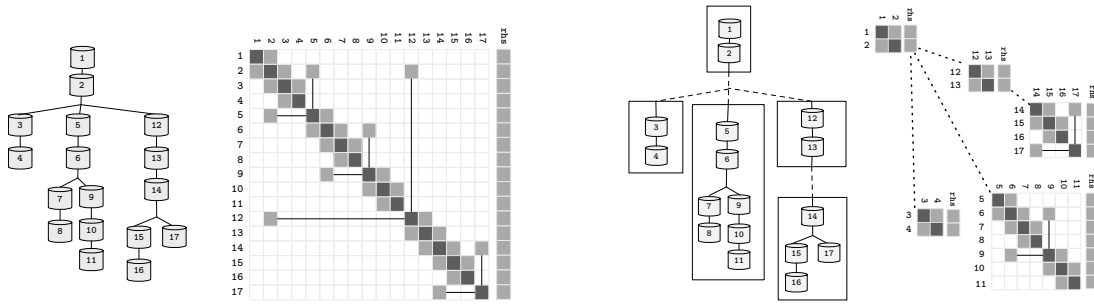


Figure 2.4 – **Left:** The topological structure of a sample neuron, and its representative sparse tridiagonal dendritic tree representation; **Right:** alternative representations of the neuron model, after the decomposition into subtrees pictured in Figure 2.3. Straight lines represent parent-children data-dependencies within the same subtree. Dashed lines represent data dependencies to different subtrees.

partments it contains. Remaining compartments, not included in the previous subtree, will recursively be clustered using the same maximum-threshold testing algorithm, until all compartments are traversed. Bifurcations require independent tests on each branch. Thus, if a branch connects to several children, then all branches in the lower level are either in the same subtree as the parent, or on independent subtrees. This rule avoids having subtrees connected to compartments that are neither root or leaf in other subtrees, so that synchronization of computation is not required except at terminal compartment connections. The complexity or maximum computation time assigned to a subtree of a given neuron n is scaled by a constant k and provided by:

$$\max work_{subtree}(n) = k \frac{runtime_n}{cores\ count} \quad (2.1)$$

an approach similar to NEURON's multisplit which can be interpreted as *each subtree must yield a max complexity to fit at most $1/k$ subtrees per compute core*. An analogous interpretation is that the initial neuron tree is decomposed in several subtrees, each guaranteed to have a runtime of at most $runtime_n/cores\ count$ scaled to a factor k — where the constant k provides flexibility in the number of subtrees generated, essential for load balance of computation across compute cores. An illustrative example of the application of the clustering algorithm to a single neuron is presented in Figure 2.3. Two cases may lead to a subtree with an assigned workload which does not approximate Equation 2.1: a linear sequence of compartments of small complexity that requires a bifurcation into multiple subtrees; or a leaf subtree of the topology that does not aggregate enough computational complexity.

Following the partitioning algorithm, parallelism of subtrees is possible, requiring synchronization of the connecting flow dependencies at every computation step. Due to the recursive tree-traversal nature of the algorithm, the final representation of the neuron is a tree of subtrees, where each subtree is a tree of compartments by itself. Analogously, the initial solver

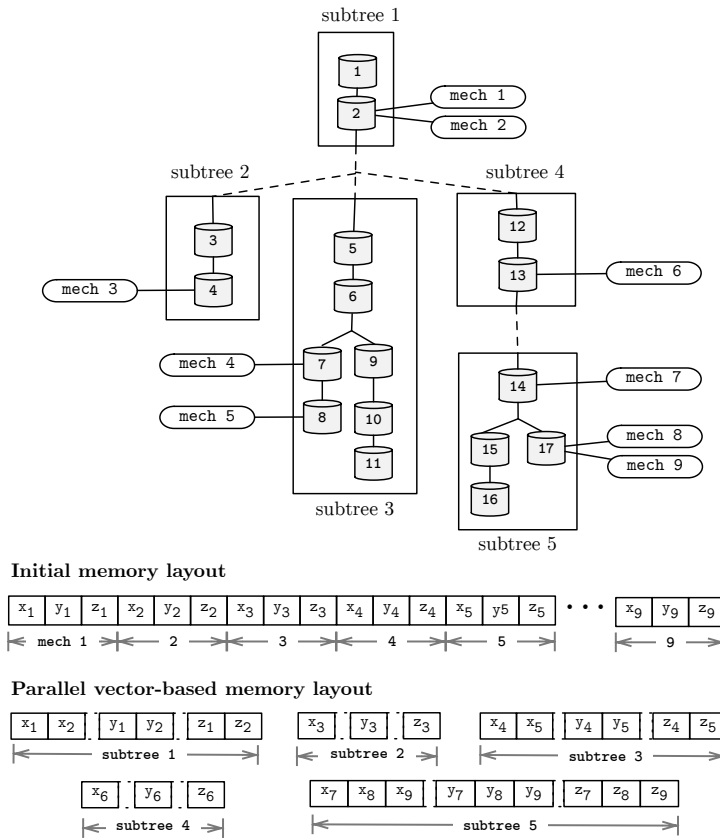


Figure 2.5 – Representation of a neuron topological structure after clustering into 5 subtrees (top); and memory layouts for the pre- and post- subtree clustering phases presented in Figure 2.3 (bottom). Morphology includes 17 compartments, and 9 instances (mech 1-9) of a mechanism type with state variables x , y and z . Distinct mechanism types and solver parameters a , b , p , d , v and rhs were omitted for simplicity.

problem is now decomposed into several smaller solver problems with a data dependency between only the root and leaf rows in different subtrees — refer to Figure 2.4 for an illustrative sample of a neuron and its linear solver structure after the clustering method. This property allows for a vector-based acceleration of individual subtrees, as will be covered in the following section.

2.3.3 Vector-based Acceleration

Similarity in the computation of the mechanisms and compartments update function allow for an acceleration in the SIMD-axis by performing vectorized computation of state variables. Vectorization can be enabled in two distinct core computations. At first, on the execution of the Gaussian Elimination method, by performing simultaneous (memory-aligned) read operations

of a , b and p , and update of the variables d , v and rhs . Secondly, vectorization can also be achieved by performing simultaneous computation of instances of the same mechanism types, placed in different compartments, holding different states, yet defined by similar state-update functions. To enable vectorization, the memory layout for individual subtrees is serialized and realigned on a SIMD-friendly layout, after the subtrees decomposition algorithm presented. For completeness, Figure 2.5 displays the post-vectorization memory layout for the morphological representation studied previously.

The computation workload allocated to each task (subtree) during load balancing is measured as the runtime of the subtree in the vectorised memory layout. In practice, a test for a subtree complexity requires the tentative set of compartments to be merged and SIMD memory-arranged. As high disparity in workload across subtrees may occur, ideal core workload balancing is not guaranteed. An asynchronous execution model mitigates this issue by dividing the subtree stepping workflow in smaller kernels and dynamically running available kernels as soon as dependency variables are resolved. This procedure is detailed next.

2.3.4 Asynchronous Execution of State Updates

To handle the disparity in computation times across subtrees and to fully utilize compute resources, an asynchronous producer-consumer execution model was implemented. Flow dependencies can be resolved by actively exchanging information between connecting subtrees, providing parent and bottom subtree contributions required for the completion of the Gaussian Elimination step. Execution is driven by shared placeholders across connecting subtrees. Compute threads wait for variable update(s) on each placeholder: when all values have been provided by the value-producing subtrees, the value-consuming subtree is allowed to continue. The implementation details will be detailed further in Section 2.4.1.

Execution follows an asynchronous threading model where lightweight threads (guiding connections between subtrees) go dormant and active when waiting for a placeholder or upon a placeholder value update. Figure 2.6 provides the diagram of the resolution of flow dependencies based on the three placeholders demonstrated previously, inline with the algorithm displayed in Figure 2.2. The combination of SIMD-enabled compute tasks, placed in and out of context of compute cores throughout execution, allows for an asynchronous execution model and the full utilization of computing resources on a single compute node.

2.3.5 Distributed Load Balancing

The extension of the methods presented to a network of compute nodes rely on a load balancing algorithm that aims to equalize the workload across the network, while minimizing inter-node communication. To that extent, the previously mentioned vector vs multi-core efficiency trade-off is extended with a communication optimisation on distributed localities: the framework must now deliver enough balanced SIMD-tasks that utilise all compute cores

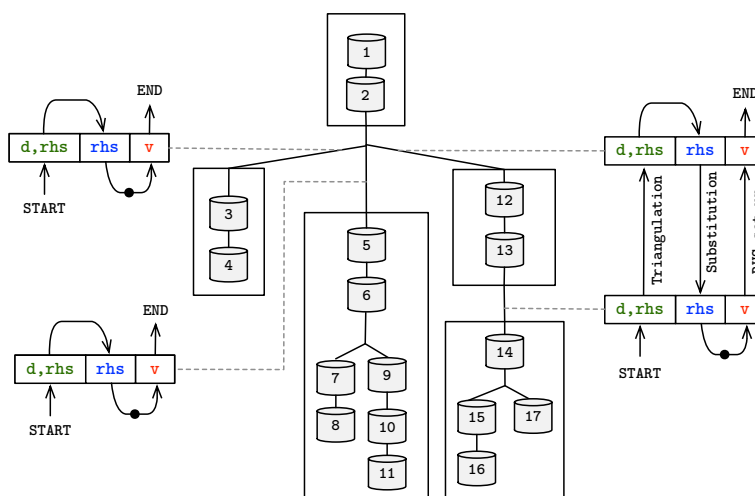


Figure 2.6 – Schematic overview of the asynchronous producer-consumer model, displaying the shared placeholders between subtrees — coloured and matching Figure 2.2. Arrow head (tail) on placeholders represents a set (get) operation by the producer (consumer) of the placeholder’s value. Dots in arrow line comprise the update of mechanism states, leading to an updated voltage value to be used in the following iteration.

across the network, while minimizing the inter-node communication from the placeholders connecting neuron sections held on different localities.

The load balancing algorithm implemented follows a dynamic distributed implementation of the Least Processing Time algorithm (Korf, 1998). To allow finer-granularity in the load balancing, the LPT is applied to groups of connected neuron subtrees instead of whole neurons. The method relies on the active update of a shared table holding the total computational time assigned to each locality so far. At the onset of the execution, neurons are loaded across the network, assigned a weight based on a computational workload (measured as the mean runtime of 10 sample 100 ms simulations of each subtree), serialized and finally communicated to the least busy compute node. As a reminder, the workload of individual subtrees remains quasi-constant throughout the execution, thus justifying (1) static load balancing, once and at the onset of execution, and (2) the usage of a computationally-heavier yet very accurate metric of computational workload (weight) based on simulated runtime and not on predictive methods. In most cases, only whole neurons are moved to other localities. This rationale allows connecting subtrees to be placed within the same memory region, and reducing inter-node communication — required once at every synaptic delay between connecting neurons, and three times per step for connecting subtrees. However, terminal (leaf) subtree groups of a neuron may be assigned to a different locality if the load imbalance caused by a whole-cell placement in a single memory region exceeds the locality runtime threshold. To avoid communication delays caused by more than two computed nodes involved in any resolution of a single neuron, only terminal sections of neurons are delegated to a remote compute

node. This rationale avoids transitive communications: in practice, dividing a single neuron across three localities 1..3 may yield a communication pattern of $1 \rightarrow 3$ and $2 \rightarrow 3$ for two unconnected arborizations 2 and 3, but not $1 \rightarrow 2 \rightarrow 3$. The maximum computation threshold assigned to each compute node is provided by the mean runtime of the dataset per locality — computed at the onset of the execution — with a tolerance value of about 10% that, when exceeded, signals the delegation of the remaining arborization of a neuron to a remote memory locality.

2.4 Benchmark

2.4.1 Implementation

The methods presented rely on the efficient implementation of HPX control objects across neuron subtrees (or subsets of the initial neuron topology), built on a distributed memory space. The most relevant implementation features are:

- Subtrees are allocated on the Global Memory Address Space (GAS), with a physical allocation on the compute node provided by the load-balancing method detailed in Section 2.3.5. Each GAS address is unique and can be called transparently from any compute node. The GAS addresses of connecting parent-children subtrees and synaptically-connected neurons are shared at the onset of the execution;
- Synaptic deliveries are performed with a remote procedure call to the address of the target (post-synaptic) subtree, with spiking time as argument;
- Synchronization of neurons time advancement is enforced with a communication barrier at equidistant time frames, equivalent to the smallest synaptic delay in the network;
- Computation flow on each neuron is guided by threads attached to the shared placeholders between connecting subtrees, that go dormant and active when waiting for a placeholder or upon a placeholder value's update. The HPX thread scheduler handles the scheduling of compute resources to the queued compute kernels;
- Shared placeholders for dependency variable among connecting subtrees are built on top of asynchronous calls to *set* and *get* operations, supported by a *future* for probing of state. A thread gate (or *and gate*) underlies the implementation of each placeholder, with an initial counter set to the number of contributions that must be set before execution is allowed to continue. There is one contribution to be set for parent-to-children dependencies, and a number of contributions equal to the number of children in the converse direction;
- The distributed execution on the GAS is based on remote procedure calls, that initiates a lightweight thread on the appropriate compute node, or places it on a queue of dormant

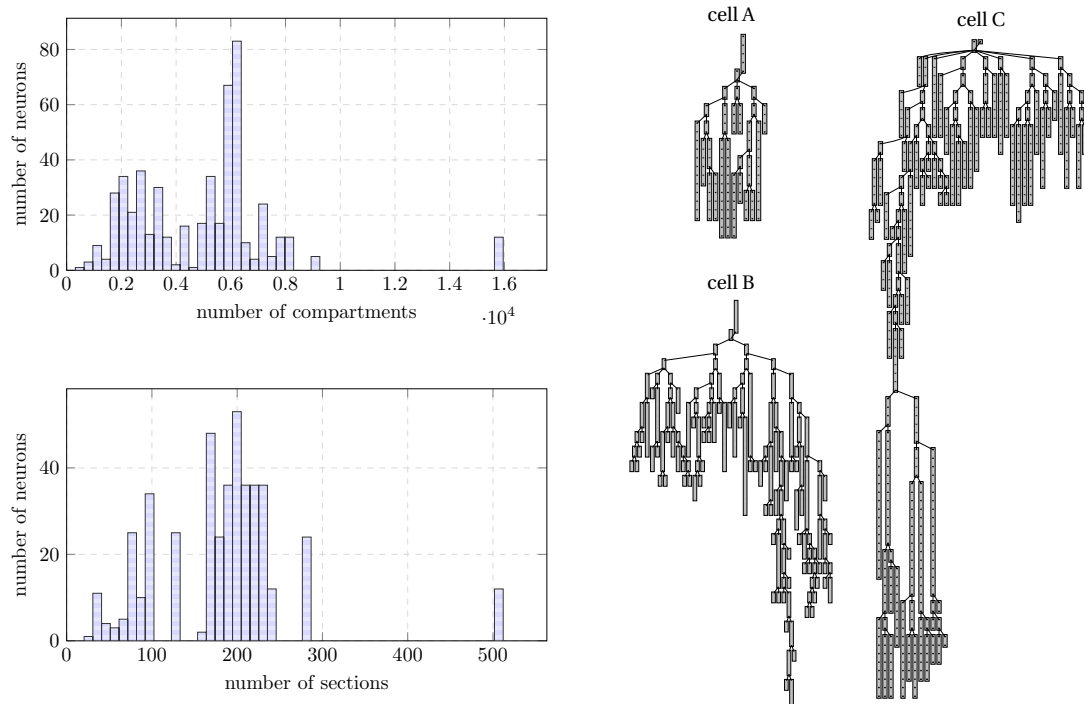


Figure 2.7 – **Left:** Morphological structure of the Layer 5 neurons extracted from the digital reconstruction of the rodent neocortex used as input data. Data provided as a histogram (50 bins) in terms of distribution of number compartments (top) and number of sections (unbranched cables of a neuron, bottom). **Right:** Dendritic compartmental trees of the three individual neurons used as input dataset on single node benchmarks.

threads to be dynamically allocated to idle compute resources throughout the execution. Upon the execution of the method at a remote location, an asynchronous call is sent back to the request initiator with the return value. The runtime system handles the communication, execution and callbacks for the remote procedure calls.

2.4.2 Use Case

Our benchmark simulates the biological activity of a digital reconstruction of a morphologically detailed neural network extracted from the model of Markram et al. (Markram et al., 2015). The reconstruction model underlies the research of synaptic plasticity and learning from Chindemi et al. (Chindemi, 2018), requiring a simulation of dozens of minutes to several days to be expressed. Input neurons were extracted from layer 5 of rodent brain neocortex. Neuron models are characterized by highly heterogeneous neuron topologies — see Figure 2.7 for the distribution of compartments and branches across the dataset. Each neuron requires 4-10MB of memory for the storage of the topological structure, linear solver, mechanisms states, and dynamic data containers for synaptic events.

For brevity, in the following benchmarks we will refer to our implementation as neurox, as

in NEURON on HPX. Moreover, the initial load balancing and memory layout realignment processes take approximately 1-2 seconds of execution time and are excluded from the analysis, as they are considered negligible in the overall execution.

2.4.3 Hardware Specifications

Execution times on a single compute node were measured on four distinct compute architectures: (1) Intel Sandy Bridge E5-2670 with 16 cores at 2.6 Ghz, with and AVX capabilities (256-bit floating point vector operations), and 128 GB of RAM; (2) Intel Knights Landing (KNL) with 64 cores at 1.3 Ghz, 96 GB of RAM, and AVX-512 (512-bits register file width); (3) Cray XE6 with 2x AMD Opteron 6380 with 16 cores at 2.5 Ghz each, 64 GB of RAM and 256-bit floating point units; and (4) Intel Xeon Gold 6140 with 18-core at 2.3Ghz with AVX-512, turbo-boost up to 3.7Ghz, and 98 GB RAM. State values are stored at double floating-point precision, leading to a maximum SIMD acceleration of 4 and 8 simultaneous operations for 256- and 512-bit register file width, respectively.

To allow for efficient point-to-point communication, selective broadcasts, and remote direct access memory, we use specialized Infiniband network hardware on the network of Cray XE6 compute nodes, interfaced via the photon API library (Kissel & Swamy, 2016).

2.4.4 SIMD vs Multi-core Trade-off Optimization

The optimal value of the constant k defining the maximum computational complexity per subtree was computed with a grid search between 0.1 and 2 with a step of 0.2, a method commonly used in problems of the same domain (Hines et al., 2008). The optimal value depends on the number of active cores available at runtime, and was measured at approximately $k = 0.8$ for 2 cores, $k = 1$ for 4 cores, $k = 1.5$ for 8 cores, $k = 1.8$ for 16 cores and $k = 2$ for more than 16 cores. A deviation of circa 20% over the aforementioned values is possible, as it depends on the input morphology and architecture.

2.4.5 Reference Branch-Parallel Implementation

We compared our methods against the reference branch-parallel implementation in the NEURON multisplit (Hines et al., 2008). Our testbench measures the runtime of the simulation of one second of the electrical activity of the cell A (illustrated in Figure 2.7), on the four compute architectures mentioned previously. The benchmark results are presented in Figure 2.10. The speedup achieved was of approximately: 10.5x for the single-core execution, down to 2.64x on the 64-core execution on the intel KNL; 2.2x-1.8x for single- to 32-core execution on the Cray XE6; 3.2x-1.8x for single- to 32-core on the Intel E5; and 3.5x-1.9x on the single- and 18-core implementation on the Intel Xeon 6140.

As expected, better acceleration is achieved by the Intel Xeon and KNL architectures, mostly

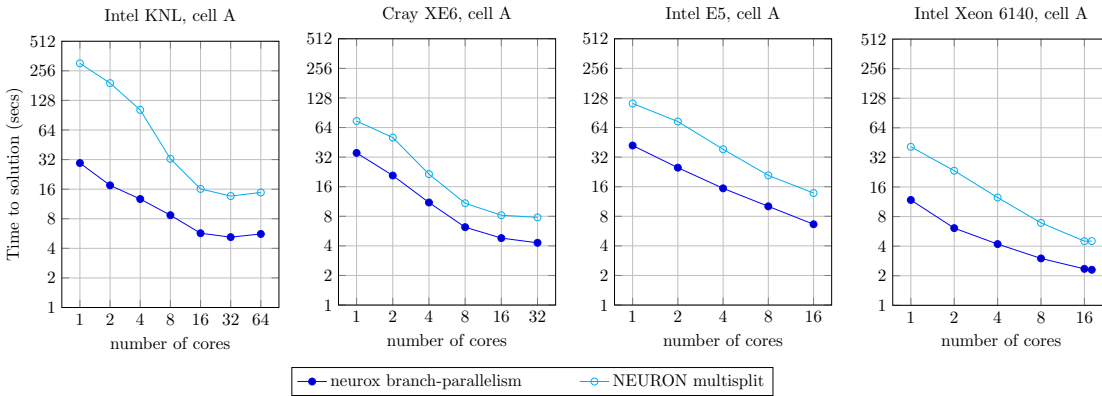


Figure 2.8 – Strong scaling plot for the simulation of one second of electrical activity of the neuron A, illustrated in Figure 2.7. Benchmark results presented for the NEURON multisplit approach, and our methods (neurox). Hardware specifications: Intel KNL with 64 cores at 1.3 Ghz and AVX-512; Cray XE6 compute node with 2 × AMD Opteron 6380 with 16 cores at 2.5 Ghz; Intel E5 with 16 cores at 2.6 Ghz and AVX2; and Intel Xeon Gold 6140 with 18-core at 2.3Ghz with AVX-512.

noticeable for a small number of cores, as the register file width is twice the amount of the XE6 and E5. As an important remark, some of the single-core benchmarks presented (particularly the KNL) exceed the theoretical limit of SIMD acceleration of 4x or 8x. This acceleration is due to the structure-of-arrays data layout in memory allowing better memory access compared to the non-SIMD array-of-structures, allowing an extra speedup on top of the SIMD instruction-level parallelism.

2.4.6 Scaling of Single Neurons

We analysed the strong scaling properties of our methods, simulating one second of electrical activity of three neuron morphologies characterized by different levels of width and depth of branching, illustrated in Figure 2.7. The benchmark results are presented in Figure 2.9. This analysis provides the theoretical limit of acceleration for single-neuron execution, as a basis for the study of the acceleration for the larger networks of neurons that will follow. The benchmark runtimes demonstrate a speedup of: 5.3x, 3.3x and 2.3x for cells A, B and C, respectively, on the intel KNL; 8.2x, 7.5x and 6.0x on the Cray XE6; 6.3x, 8.2x, and 6.3x on the Intel E5; and 5.1x, 4.2x and 3.2x on the Intel Xeon 6140.

Linear acceleration with good strong scaling on up to 8 threads on single-core compute units with Single Instruction Single Data processing has been previously demonstrated by the NEURON multisplit approach (Hines et al., 2008). Similarly, in our methods, ideal scaling is not visible and an acceleration beyond 16 threads per neuron is strenuous. This limitation has already been studied by Hines et al. (2008) on the multisplit method, and is unrelated to the implementation but due to the nature of the problem instead. Subtrees are quasi-balanced in

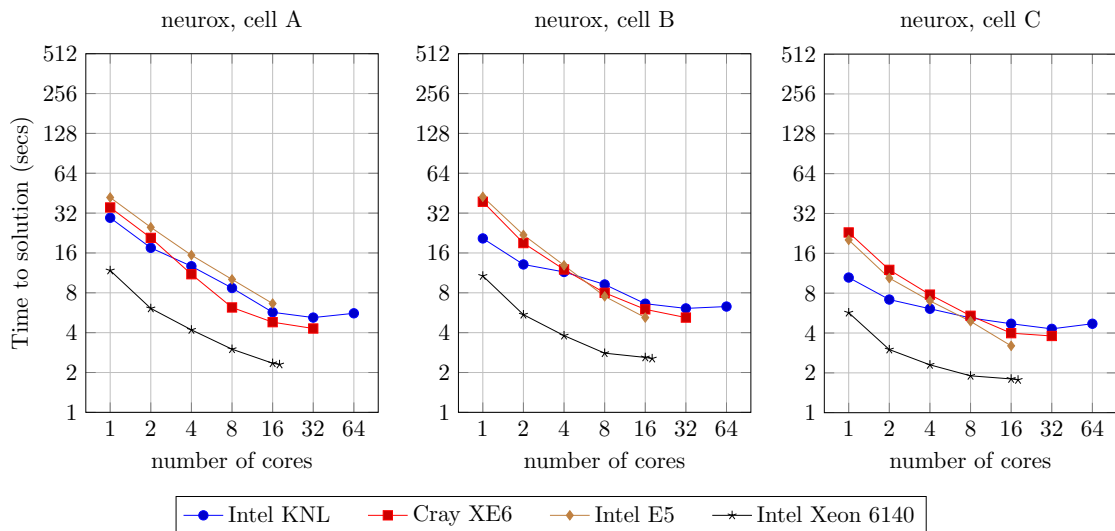


Figure 2.9 – Strong scaling benchmark for the simulation of one second of electrical activity of three different models of single neurons illustrated in Figure 2.7, benchmarked on four compute architectures: Intel KNL with 64 cores at 1.3 Ghz and AVX-512; Cray XE6 compute node with 2× AMD Opteron 6380 with 16 cores at 2.5 Ghz; Intel E5 with 16 cores at 2.6 Ghz and AVX2; and Intel Xeon Gold 6140 with 18-core at 2.3Ghz with AVX-512.

terms of workload, yet stochastic temporal events such as user-defined current injections and synaptic activity account for extra computation that can not be accounted for by the subtree clustering algorithm, leading to an unpredictable imbalance at certain time intervals. More importantly, the tree structure of the dataset limits the parallelism exposed when computation is concentrated at higher levels of the neuron tree, such as during Gaussian Elimination. This property is noticeable when comparing the benchmarks of the three cells A-C with increasing branching depth, where an increase of the cell depth leads to a reduction in the strong scaling capabilities of our method.

Finally, the deceleration of the speedup with the increase of compute cores is also related to the aforementioned trade-off between SIMD parallelism (maximized for single core execution) and vector-based parallelism: to fully utilize all available cores, smaller SIMD data structures are created, causing a loss on the exposed vector acceleration, i.e. register file width is not fully utilised. As a smaller factor of performance loss, an extra overhead is added by threading (de)allocation as we increase the number of active cores.

2.4.7 Scaling of Networks of Neurons

An analysis of scaling of our implementation on a larger dataset was executed on a network of 128 Cray XE6 compute nodes, performing a simulation of one second of electrical activity on the same neocortical model. The benchmark presents the scaling properties of our branch-

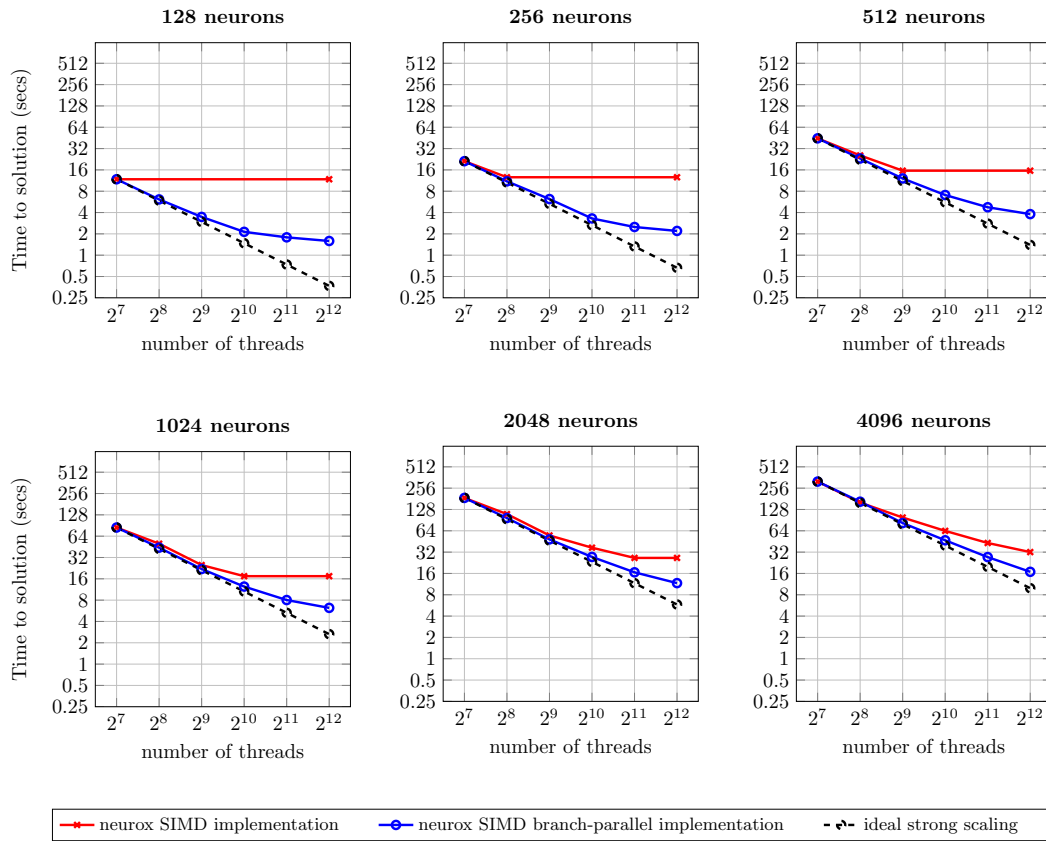


Figure 2.10 – Benchmark of the simulation of one second of electrical activity of Layer 5 neurons, on a Cray XE6 cluster with 128 compute nodes. Each node contains two AMD Opteron 6380 with 16 cores at 2.5 GHz each with 256-bit register file width. Ideal strong scaling assumes complete overlap of computation and communication.

parallel implementation, similar to the previous section, and compares it against the non branch-parallel counterpart, that simulates neurons as indivisible units.

The execution runtime for an increasing number of neurons is presented in Figure 2.10. The first benchmark performs a strong scaling analysis with the same ratio of threads-per-neuron as the previous single neuron benchmarks. The results demonstrate the preservation of strong scaling properties between the single and 128 compute nodes use cases. Similar benchmarks were performed for 16, 32 and 64 compute nodes — omitted for brevity — and provide identical results. A weak scaling analysis follows by increasing the workload from 128 to 4096 neurons (following the benchmarks from left to right), in the same hardware and number of compute units. Results suggest that an increase of the input dataset per compute node approximates the runtime to the ideal strong scaling limit. This is due to an overlap of computation and communication across different neurons subtrees, leading to a better usage of compute resources.

The maximum speedups achieved were of 7.4x for 128 neurons, 5.8x for 256 neurons, 4.2x for 512 neurons, 2.8x for 1024 neurons, 2.3x for 2048 neurons and 1.9x for 4096 neurons. For the 128-2048 neuron datasets, most of the speedups are due to the saturation of threads with compute tasks on the branch-parallel execution, but not on the non-branched execution (due to an insufficient number of neurons). However, in the scenarios where both implementations fully utilise their compute units, a significant speedup is noticeable. This property is more prominent on the largest benchmark tested, where the dataset yields one neuron per thread, thus providing enough tasks to fully occupy all computing resources on both implementations, and an acceleration of almost twofold is visible. This speedup is justified by the finer-grained parallelism and load balancing exposed by our methods, allowing better dynamic allocation of tasks to threads, thus leading to an overall reduction in runtime.

2.5 Discussion

This chapter presented an algorithm and an implementation that expose numerical dependencies at the topological level of branched neurons, and accelerate the simulation of morphologically detailed neuron models. We detailed the method for the numerical resolution of our problem, and showed that (1) the activity of the electrical current at the level of neuron topological trees depends on three parameters of the numerical solver that include current contributions between connecting tree sections; (2) the previous dependency allows for the grouping of connecting compartments into subtrees, where each subtree is a subset of the initial problem, and the set of subtrees holds the same data representation as the initial neuron; and (3) subtrees can be grouped into a tree of subtrees — holding the previous cover and distinct set properties — and allocated to any locality on the network in order to allow for accurate distributed load balancing.

Our analysis showed that a numerical resolution with full usage of compute resources on a distributed network of compute nodes is possible at three layers of parallelism: (1) at the level of compute nodes, a load balancing method delegates sections of neuron arborizations to localities at the onset of execution; (2) at the compute node level, load balancing follows analogously with the clustering of compartments into subtrees, allowing a multi-core acceleration by dynamically delegating subtrees to compute cores throughout the execution; and (3) at the core level, where SIMD-based acceleration of state variable updates and numerical solver acceleration is possible by realigning the memory layout of each subtree.

The methods were implemented on the compute kernel of the NEURON scientific application, yielding an asynchronous simulator on a global memory address space, with synchronization and threading supported by the HPX runtime system. The benchmark results compared our methods with the reference branch-parallelism method in the NEURON simulator, yielding a speedup of up to 10.5x on an Intel Knights Landing with 64 cores, 2.2x on a 32-core Cray XE6, 3.2x on an 16-core Intel E5, and 3.5x on a 18-core Intel Xeon 6140. A following benchmark studied the efficiency of our methods on three highly heterogeneous neuron models, display-

Chapter 2. Asynchronous Branch-Parallelism Extracted from Neuron Morphology

ing good strong scaling properties on up to 16 cores, and a dependency of the parallelism efficiency on the depth of the neuron tree.

We extended our methods to larger neuron networks, and assessed its scaling properties on a network of 128 Cray XE6 compute nodes simulating up to 4096 neurons. Our implementation was shown to deliver a speedup of 7.4x for small datasets, and 1.9x when full occupancy of compute resources was guaranteed. Moreover, it displayed a good preservation of its strong scaling properties, with almost ideal scaling on the largest dataset tested.

The preservation of the scaling properties — independently of the network size and compute cores per neuron — combined with the added capability of generating a varying number of compute tasks allocated in a balanced way across all localities, allows our strategy to be fine-tuned to a wide range of distributed architectures and inputs.

3 Asynchronous Graph-Parallelism

Extracted from ODE Dependencies

This chapter is adapted from the following article:

Magalhães B., Hines M., Sterling T., Schürmann E, "Exploiting Flow Graph of System of ODEs to Accelerate the Simulation of Biologically-Detailed Neural Networks", accepted at IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2019

Personal contributions: conceptualization, formal analysis, investigation, methodology, software, validation and writing.

3.1 Abstract

Exposing parallelism in scientific applications has become a core requirement for efficiently running on modern distributed multi-core SIMD compute architectures. The granularity of parallelism that can be attained is a key determinant for the achievable acceleration and time to solution.

Motivated by a scientific use case that requires the simulation of long spans of time — the study of plasticity and learning in detailed models of brain tissue — we present a strategy that exposes and exploits multi-core and SIMD micro-parallelism from unrolling flow dependencies and concurrent outputs in a large system of coupled ordinary differential equations (ODEs).

An implementation of a parallel simulator is presented, running on the HPX runtime system for the ParalleX execution model, providing dynamic task-scheduling and asynchronous execution. The implementation was tested on different architectures using a previously published brain tissue model. Benchmark of single neurons on a single compute node present a speedup of circa 4-7x when compared with the state-of-the-art Single Instruction Multiple Data (SIMD) implementation and 13-40x over its Single Instruction Single Data (SISD) counterpart. Large scale benchmarks suggest almost ideal strong scaling and a speedup of 2-8x on a distributed architecture of 128 Cray X6 compute nodes.

3.2 Introduction

This chapter describes a strategy that allows for a speedup on the simulation of a wide range of scientific problems represented by large systems of ODEs, by exposing a balanced SIMD-enabled computation graph that describes data-flow dependencies and concurrent outputs across variables on a system of differential equations. The work presented introduces a novel method for micro-parallelism that allows for further acceleration of individual neurons and groups of neurons described by similar model descriptions.

Data-flow dependencies via data-flow programming execution models have been previously explored by the OmpSs (Edwards et al., 2014) and OpenMP (Dagum & Menon, 1998) programming models. Such methods require the manual specification of input, output and input-output data dependencies in order to create Direct Acyclic Graphs (DAGs) of computation. To put our work into perspective, the methods presented start at the level of mathematical abstraction and rely on the specification of the system of ODEs, in order to automatically extract not only the data-flow dependencies (DAG) but also concurrent output/update operations, without the need of user-provided input/output dependencies. This feature allows for the automatic extraction of dependencies and DAGs from very large system of ODEs whose dependencies would be infeasible to extract manually.

The contributions presented are the following: (a) a method for the extraction of flow dependencies and concurrent output operations from the interdependencies across state variables on a system of ODEs; (b) an algorithm for the embarrassingly parallel processing of blocks of independent ODEs; (c) a method for SIMD acceleration of blocks of similar instances of ODEs; (d) a load balancing algorithm for balanced execution blocks based on the computation time of individual equations; and (e) an asynchronous execution model that allows for good leverage of computing resources by producing enough micro-parallelism to fully utilize available SIMD and multi-core compute units on a wide range of architectures, based on a novel method for strong scaling that trades off between two types of parallel resources — SIMD units vs multiple cores.

Our methods were applied to the simulation of the electrical activity of a network of biologically inspired neuron morphologies. Benchmark results are shown to expose a finer-grained level of parallelism and a large speedup compared to the reference solution on a single compute node and on distributed architectures. Moreover, the flexibility in the multi-core parallelism and vectorization methods presented allows the full usage of computing resources across a wide spectrum of host architectures (when enough computation is available), leading to a significant acceleration in the strong scale limit.

3.3 Methods

3.3.1 Dependencies from Model Specification

We present a method for the extraction of the flow (read-after-write) dependencies and concurrent write operations between mechanisms and compartments state variables, based on the mathematical specifications of the ODEs. In our model, mechanism specifications are provided by NEURON's domain specific language (NMODL), introduced by Hines and Carnevale (2000). Our methods parse the textual representation of the mathematical specification of ODEs defining the change in opening variables and the current contribution of each mechanism, and extract the flow dependencies and concurrent output operations across the set of mechanisms and the compartments they're placed on.

The flow dependencies of a given ODE are provided by the list of ODEs whose states updates must precede it. Conversely, concurrent outputs are defined by the set of parameters whose updated value must be summed, in order to compute the final value of another parameter. As an example, take the calcium-activated potassium channels from mammalian brain from Kohler et al. (1996). The potassium current in this mechanism is formulated by:

$$I_K = g_K \cdot z \cdot (V - E_K). \quad (3.1)$$

The equation provides the current contribution to the ionic current term in Equation 1.7, thus following the same symbol notation. The equation follows a z activation kinetics described by a first-order ODE as:

$$\tau_z \frac{dz}{dt} = \frac{1}{1 + \left(\frac{0.00043}{[Ca]_{in}}\right)^{4.8}} - z. \quad (3.2)$$

where $[Ca]_{in}$ is the internal calcium concentration and τ_z is the time constant of the channel. The equilibrium potential for a given ion (in this example K) is given by the Nernst equation:

$$E_K = \frac{RT}{zF} \ln\left(\frac{[K]_{out}}{[K]_{in}}\right) \quad (3.3)$$

where R is the universal gas constant ($8.314JK^{-1}mol^{-1}$), T is the temperature in Kelvin ($K = \check{C} + 273.15$), z is the valence of the ionic species (+1 for Na^+ , +1 for K^+ , +2 for Ca^{2+} , -1 for Cl^-), F is the Faraday's constant ($96485Cmol^{-1}$) and $[K]$ is the concentration of the ion K in the intracellular (in) and extracellular (out) neuron fluid.

In brief, the update of the channel state depends on the resting potential of potassium currents in the system (E_K in Eq. 3.3), and the updated value of its gating variable z that depends on the state of the calcium ions ($[Ca]_{in}$ in Eq. 3.2). On the other hand, the potassium and calcium currents are voltage-driven (similarly to I_K in Eq. 3.1), requiring an updated voltage value at the compartment. For clarity, the corresponding diagram of flow dependencies is displayed in Figure 3.1.

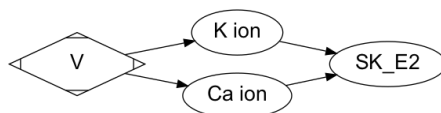


Figure 3.1 – Diagram of the flow dependencies of the SK_E2 potassium ion channel. V represents the start of the computation with an updated compartment state (voltage), Ca and K ions are the mechanisms that represent the states of Calcium and Potassium ion currents in the system. Straight arrow tails illustrate read-after-write dependencies (the DAG) and flow dependencies between mechanism equations.

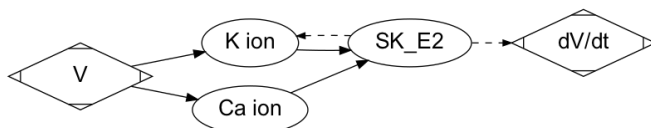


Figure 3.2 – Diagram of the flow dependencies (straight lines) and concurrent outputs (dashed lines) of the SK_E2 potassium ion channel. Start and end of computation graph are represented by V and dV/dt respectively. Ca and K ions are the mechanisms that represent the states of Calcium and Potassium ion currents in the system. Straight arrow tails illustrate read-after-write dependencies (the DAG) and flow dependencies between mechanism equations. Dashed arrow tails illustrate concurrent variable updates.

Conversely, the computation of the voltage value dV/dt in Equation 1.7 requires a sum of the updated values of mechanism currents — i.e. the value of I_K in Eq. 3.1 and all remaining currents — leading to a concurrent update of the value holding the sum of all current contributions. A similar sum operation is required to update the state of the mechanisms representing ion currents: in practice, the current contribution of individual ion channels needs to be summed to the mechanism holding that ion type, leading also to a concurrent update. These two operations can be inserted into the previous diagram, leading to the dependency diagram displayed in Figure 3.2.

As a side note, mechanisms and compartment ODE updates interact in a feedback loop and yield a circular dependency that needs to be resolved: in practice, an update of the state of the compartment alters the state of the voltage-dependent ODEs, and vice-versa. Although these interdependencies could be resolved with general numerical method algorithms, our implementation relies on a problem-specific optimization, the Hines solver (Hines, 1984) implemented in NEURON. In brief, interleaved timestepping handles the mechanism-compartment interdependency, resolving voltages at every full step and mechanism updates at every half of a timestep, allowing for a *quasi*-implicit numerical resolution and without iterations. Once mechanism states are updated for a new timestep, a Gaussian Elimination is applied to the branched structure of the neuron, solving Equation 1.7 for connecting compartments in terms of the voltage of a compartment, its children’s and its parent’s voltage, providing the updated voltage value V at the new step from the voltage change dV/dt . This procedure refers to the steps Hines matrix set-up, Hines Gaussian Elimination and neuron’s voltage update on the workflow displayed in Figure 1.4, and has been covered extensively in Chapter 2.

To summarize, given the description of a set of ODEs describing the state updates of a given mechanism A , the dependencies collection proceeds in accordance with the following rules: (1) if an equation in A depends on a variable of an external mechanism B , then B must be computed beforehand and the flow dependency $B \rightarrow A$ must hold; and (2) when a equation update in A is given by a sum of elements from other mechanisms C_1, C_2, \dots, C_n executed in parallel, then the updates $C_1 \rightarrow A, C_2 \rightarrow A, \dots, C_n \rightarrow A$ lead to a concurrent output. Note that a concurrent output implies a flow dependency whose individual updates must be handled with an atomic, sequential or mutually exclusive operation. The application of these methods to a collection of mechanisms allows the creation of an inter-mechanism flow dependency and concurrent output, which will be covered in detail in the following section.

3.3.2 Application to NEURON Modelling Language

The mechanism specification provided by NEURON's domain specific language NMODL (Hines & Carnevale, 2000) complies with the mathematical description of individual mechanisms. As an example, the function that calculates the conductance of the current ion channel and individual contribution to the potassium currents of the previous SK_E2 mechanism is described by:

```

1   gSK_E2 = gSK_E2bar * z
2   ik = gSK_E2 * (v - ek)

```

a representation equivalent to Equation 3.1, where gSK_E2bar is the potassium conductance in the system and ik is the individual contribution to the potassium currents from this mechanism. The update of the internal state of the ion channel is computed based on the derivative of z , represented by z' and described similarly to Equation 3.2 as:

```

1   zInf = 1 / (1 + (0.00043 / cai) ^ 4.8)
2   z' = (zInf - z) / zTau

```

where $zTau$ is a time constant and cai is the internal calcium concentration in the system. It follows that the total potassium current ik across this channel depends on the resting potential of the potassium current in the system ek , the voltage v at the compartment, and the conductance of the ion channel (gSK_E2 , proportional to the gating state z). On the other hand, z depends on the internal concentration of calcium cai . In brief, it follows that SK_E2 depends on the compartment voltage, and on the voltage-driven calcium and potassium currents. Moreover, the total potassium current in the system and the state of the compartment requires an updated value of all related ionic currents, including the SK_E2 ion channel's (ik), leading to two concurrent updates. These results follow inline with the dependencies extracted in the previous section and pictured in Figure 3.2.

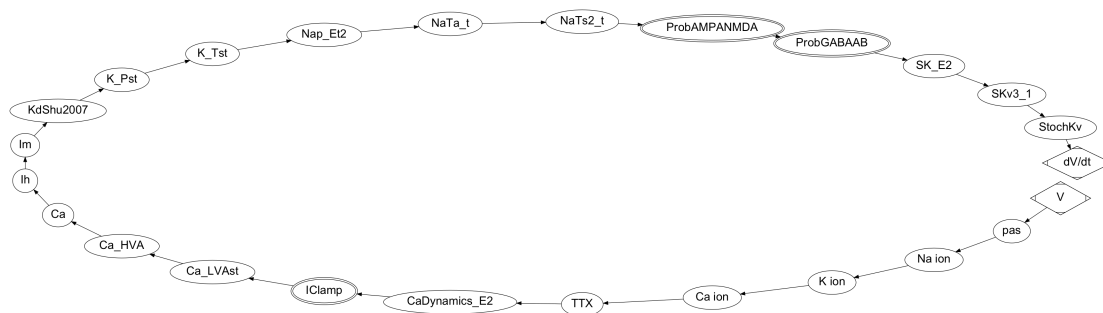


Figure 3.3 – Workflow of the state-of-the-art (sequential) state update function of a neuron morphology with 23 mechanisms. Execution order displayed as clock-wise, starting on the compartment state update (V node), followed by the membrane passive currents (pas), ionic currents, remaining mechanisms, and finishing on the update of the compartment current ODE (dV/dt node). Nodes with a double contour account for mechanisms that are event-driven, e.g. current injections or synaptic currents.

3.3.3 Computation Graph from Individual Dependencies

The methods presented were automated and applied to a biologically inspired neuron network of the somatosensory cortex of the young rodent, with a total of 23 mechanisms and 44 ODEs, extracted from the work of Markram et al. (Markram et al., 2015). The state of each mechanism is modelled by a maximum of four distinct ODEs. Mechanism state variable are replicated for every mechanism instance across all compartments. Thus, a computation graph of distinct mechanisms is created for the whole network of neurons (or a group of neurons), and the complexity of a graph is not dependent on the number of neurons, but on the number of unique ODEs in the system.

The graph extraction procedure follows similarly to the method described in the previous section: (1) a textual parser builds the syntax tree of all ODEs that describes the mechanism (model) specifications, in accordance with the NMODL domain specific language; (2) variables on the left and right hand side of the ODEs are flagged as dependee and dependency, respectively; (3) the direction of variable dependencies — most specifically *which variable waits for what value* — across mechanism ODEs is stored in a dependee-to-dependency matrix, representing row-to-column variable flow dependencies; and (4) sum operations over a set of elements, or equivalently multiple columns-per-row dependencies, lead to a many-to-one flow dependency and are tagged as requiring a concurrent update, and subject to a memory-protected value update.

For comparison, the serial workflow of mechanism updates adopted in existing state-of-the-art implementations is displayed in Figure 3.3. The execution order is as follows: (1) update of passive current in the compartment (pas); (2) sequential update of the states all ionic currents (3) update of ion channel gating states and remaining mechanisms.

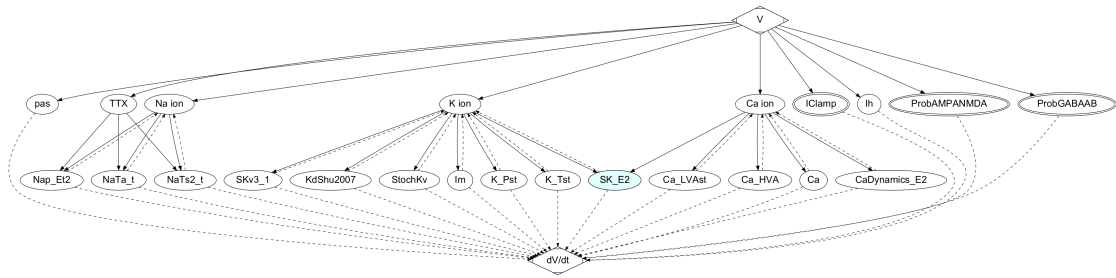


Figure 3.4 – Computation graph for the parallel execution of the state update function with flow dependencies and concurrent update. The workflow displays the execution pictured in Figure 3.3, after application of the dependencies extraction method. Execution flow follows from node V to dV/dt . Single-line arrows represent execution dependencies. Dashed-line arrows represent concurrent updates to mechanisms' (upward) or capacitor's state (downward). Nodes with a double contour display mechanisms that are event-driven (e.g. current injections or synaptic currents). The SK_E2 mechanism used for detailing the graph creation processed is coloured cyan.

The graph of flow dependencies and concurrent updates was extracted from the same model, using the methods mentioned previously. Individual dependencies were combined to build the computation graph with the flow dependencies and concurrent update phases displayed in Figure 3.4. The SK_E2 mechanism exemplified in the previous section is emphasized in blue colour, with a flow dependency from potassium and calcium ions (K ion and Ca ion, respectively), and a concurrent output to the current of potassium ions. The resulting computation graph yields a maximum of 19 parallel kernels, where each node of the graph represents the set of all instances of that mechanism type. The computation of a timestep terminates when all updates triggered by the concurrent processes at the bottom level are finalized. As an important remark, despite the high parallelism exposed, the maximum acceleration is dictated by the slowest mechanism in the graph. This limitation is a major bottleneck for use-cases of very high number of instances of the same mechanism, or mechanisms with high computational runtime per instance. In the following section we will address this issue, by presenting a method that exposes SIMD-computation, block-parallelism and accurate computational load balancing by exploring the similarities between instances of the same mechanisms.

3.3.4 Vector-Parallelism of Mechanism Instances

The graph-based computation detailed in the previous sections exposes the parallelism of different mechanisms running simultaneously. Notwithstanding, instances of mechanisms of the same type are independent and defined by similar operations. Therefore, a significant speedup can be achieved by computing mechanism instances simultaneously, using vector instructions. In practice, vectorization on the computation of instances of the same mechanism can be efficiently performed if data is represented in a SIMD-friendly memory layout. To

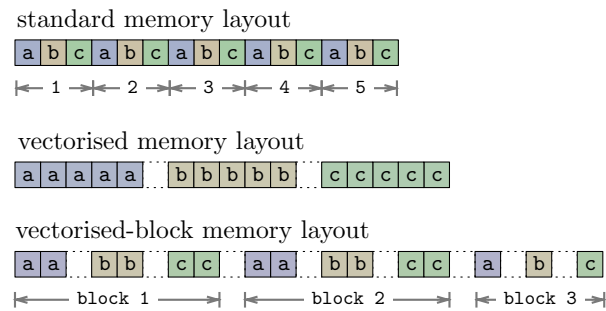


Figure 3.5 – Three alternative memory layouts for the processing of one mechanism with three state variables (a–c) expressed in five instances (1–5). Standard memory layout represents data as an array of structures. Vectorised memory layout displays data on a SIMD-friendly alignment, with added memory padding between variables for vector-instruction processor alignment. Vectorised-block memory layout follows the SIMD approach, dividing instances across several blocks of fixed maximum size (2 in the example above).

overcome the limitation of concurrent writing, shadow vectors are populated simultaneously with individual contributions of instances. A post-mechanism reduce operation sums all contributions and writes the final update value.

Nevertheless, computation disparity at every node of the graph – describing all instances of a mechanism type – leads to a computational load imbalance. The theoretical speedup limit is now dictated by the execution time of the slowest mechanism. This bottleneck is most prominent when a few mechanisms require a very large runtime and/or are exhibited by a high number of instances when compared to the remaining mechanisms. In the example above, this is the case of the ProbAMPANMDA and ProbGABAAB mechanisms, referring to synaptic connections between neurons, as will be shown later in the benchmark results.

A second limitation occurs on neuron models with a reduced number of mechanisms, or in compute architectures with a high number of cores. In such scenarios, the parallelism exposed by the graph may not be enough to fully utilize all compute cores available in the architecture.

These two issues can be circumvented by exploring the parallelism of mechanism instances. Instances of the same mechanisms can be divided in several blocks (subsets), in accordance with the SIMD-friendly memory layout of the initial block. This approach solves both the computational imbalance and lack of exposed parallelism issues mentioned previously: by creating smaller computation tasks, it allows the dynamic scheduler to better balance the execution, while creating an embarrassingly parallel execution model for blocks of instances, that exposes further parallelism. For completeness, Figure 3.5 presents the memory layout of a sample mechanism in the standard, vectorised, and vectorised-block memory format. The vectorised-block memory layout approach provides vector-based computation of independent blocks of instances.

The main question now lies on the decision of instances block size per mechanism. To account

for the disparity of runtime across single instances of different mechanisms, our method sets a cap on the total computation assigned to each block. The algorithm is the following: (1) single instances of every mechanism are executed and benchmarked independently; (2) the block size for a given mechanism is determined by the number of instances whose total execution time falls below a maximum computation threshold; and (3) memory layout for every mechanism instances is reorganized into the vectorised-block layout, based on the block size assigned to that mechanism. For optimal usage of SIMD compute units, block sizes are set to a multiple of the width of the processors register file.

The value of the maximum complexity threshold per block plays a critical role in the acceleration provided: the rationale is that this constant needs to reflect the optimal trade-off between block-size (exposed vector-based acceleration, limited by vector word size) and number of blocks (exposed thread-base parallelism, limited by number of cores per CPU). In practice, high thread-parallelism forces a small block-size per thread, leading to high overhead on thread (de)allocation and possibly insufficient vector-parallelism per thread. On the other hand, small thread-parallelism will fully utilize SIMD units but may not yield enough tasks to occupy or correctly load balance all compute cores. Therefore, the acceleration provided by these two resources becomes competitive for small datasets or highly-parallel architectures, and the balance of both is defined by the block size constant, by itself dependent on the runtime of the update functions of each mechanism, which depend on the compute architecture they are computed on.

To conclude, the detection of the optimal value for the block size is performed following a grid search approach, commonly used in existing efforts in the same problem domain (Hines et al., 2008). Although having a direct method for the computation of block size would be ideal, little work is available on analytical formulations or regression models for the discovery of optimal execution hyper-parameters (in this case ideal block size for update functions) based on the input data (number of neurons, distribution of mechanisms and mechanism instance runtime) and relevant hardware specifications of the host architecture (cache size, vector-word size, number of cores, number of NUMA sockets, number of simultaneous SIMD operations, processor clock speed, memory bandwidth, to name a few).

3.4 Benchmark

3.4.1 Implementation

The reference implementation was extracted from the multi-threaded SIMD-optimized compute kernel of the NEURON scientific application (Kumbhar et al., 2019), available as open source (Blue Brain Project, 2015a), implementing a serial execution of mechanisms workflow — as illustrated in Figure 3.3. Executions enabled with graph-parallelism exhibit a maximum numerical discrepancy of 10^{-5} due to a different order of execution of mechanisms yielded by the computation graph, considered negligible as the smallest range in values of interest

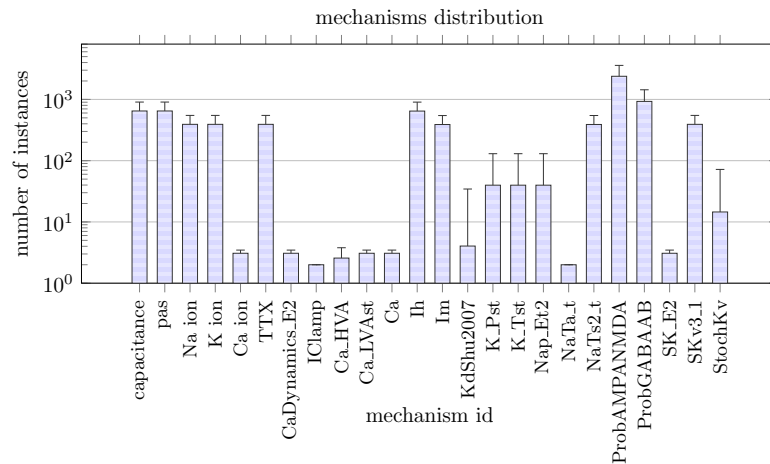


Figure 3.6 – Mean and standard deviation of the number of instances per mechanism, extracted from 1000 neurons in the layer 5 of the Somatosensory cortex of the rodent brain.

represents the *opening* of ionic channels, a percentual value between 0 and 1. Data is stored as 64-bit double-precision floating point values. Moreover, the runtime of the initial data, graph, and block set-up methods are excluded from the following measurements as they run on a minimal runtime compared to the overall simulation.

To allow for an asynchronous execution model and to enable graph- and instance- parallelism in the reference solution, the communication, threading and memory handling methods were replaced by HPX routines:

- Graph read-after-write dependencies were implemented via HPX *and-gates* with an initial value set to the number of dependencies on the graph node. Upon completion of the update function, each dependency decrements the and-gate value of its dependees. When the gate value reaches zero, the update function on the dependee node is executed, and its gate reset to the initial value. Subsequently, its dependees' gate values are decremented at the end of the assigned compute kernel;
- Output dependencies are handled by temporary arrays that store the contribution of each ODE. After the last update from the dependencies, the final value is computed as the sum of all contributions. Alternative approaches for concurrent writing — such as serial writing and *mutexes* — were tested and were deemed less efficient;
- Inter-node communication (synaptic delivery) is performed via asynchronous one-off events (remote procedure calls), with the event information being stored by a *future* that allows for probing and querying of event state;
- Memory allocations were replaced by HPX methods allowing datasets to be transparently distributed and accessed across localities. The structure of the parallelism-graph is local

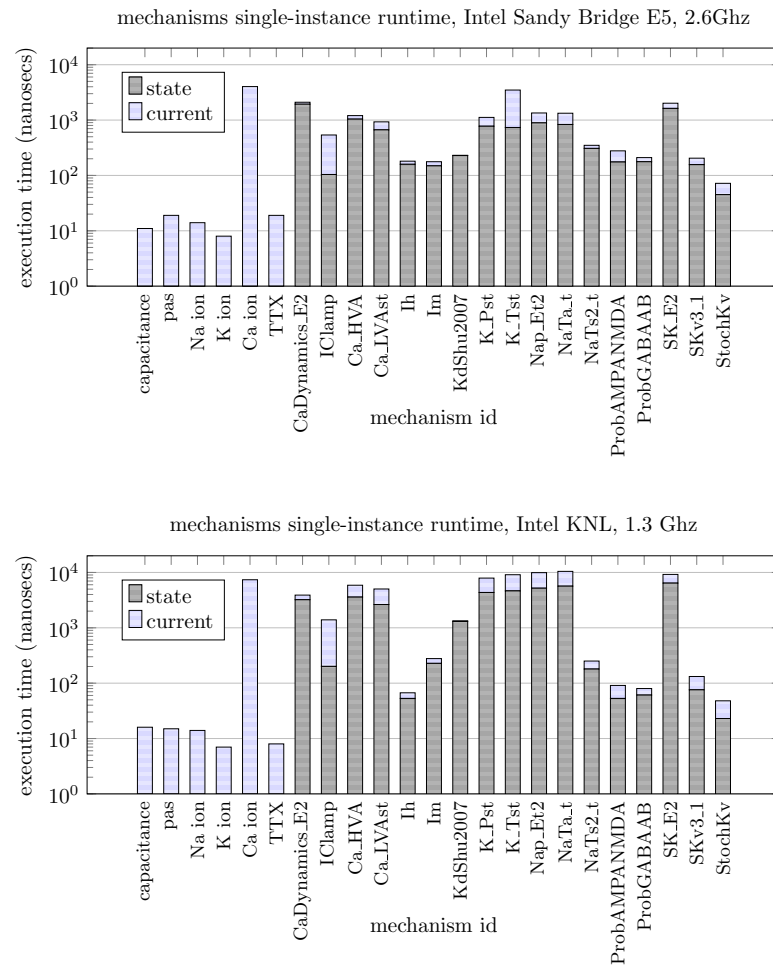


Figure 3.7 – Average execution time for the timestep update (the current and state functions) of single instances of the mechanisms, extracted from 1000 neurons in the layer 5 of the Somatosensory cortex of the rodent brain. Execution times presented for the Intel Sandy Bridge E5 (top) and Knights Landing architectures (bottom).

to every compute node, for efficiency purposes. Neurons are stored as a distributed array in the global memory address space;

- The asynchronous execution follows a dynamic thread allocation model. Micro-tasks such as mechanism function updates or synaptic deliveries are executed by (lightweight) threads and stay *in context* until task termination or held by a synchronization object. If the thread initiator is a timed event (such as incoming synapse), thread is terminated and cleared. If it is a repetitive event (e.g. a timestep's update function at every step), its initiator (and-gate) is reset and thread goes into dormant state, waiting for next wake-up signal (and-gate counter);

Due to the usage of a global memory address space, synchronization objects may wait, probe

and initiate processes in remote localities. Moreover, threading, events and procedure calls at local or remote localities are indistinguishable in terms of implementation.

3.4.2 Use Case

We benchmarked our methods on a morphologically detailed neural network retrieved from the model of a digital network reconstruction of Markram et al. (Markram et al., 2015). The benchmark performs a simulation of 100ms or 4000 computation timesteps of biological activity of the layer 5 of the somatosensory cortex of the young rodent brain. The test case represents a fraction of the time window of the model for synaptic plasticity from Chindemi et al. (Chindemi, 2018), which requires from few minutes to several days of simulation to be expressed. Each neuron is comprised of up to 23 different biological mechanism types, of which it may instantiate up to several thousands. The total number is varying from neuron to neuron; how the mechanisms are distributed across multiple neurons is shown in Figure 3.6. Each neuron requires 4 to 10 MB of memory, comprising topological structure, mechanism instances metadata, and temporary data structures for communication and events handling.

3.4.3 Hardware Specifications

Single compute node implementations were benchmarked on the following four architectures: (1) Intel Sandy Bridge E5-2670 with 16 cores at 2.6 Ghz with AVX (256-bit register file width, allowing vector-processing of four parallel 64-bit floating point units) and 128 GB of RAM; (2) Intel Knights Landing (KNL) Xeon Phi with 64 cores at 1.3 Ghz with AVX-512 (512-bits register file width), and 96 GB of RAM; (3) Intel Xeon Gold 6140 with 18-core at 2.3Ghz with a turbo-boost frequency of up to 3.7Ghz and AVX-512, and 98 GB RAM; and (4) Cray XE6 with 2× AMD Opteron Abu Dhabi 6380 with 16 cores at 2.5 Ghz each, 256-bit floating point units, and 64 GB of RAM. In practice, an optimal SIMD usage yields 8 simultaneous double-precision floating point operations on the Xeon 6140 and KNL, and 4 operations on the Cray XE6 and Intel E5.

3.4.4 Strong Scaling

The average execution time per mechanism instance — measured at the onset of the execution and utilised to compute the instance block size of each function and mechanism — is presented in Figure 3.7 for the E5 and KNL architectures. Combined with the heterogeneous distribution of mechanisms across neurons demonstrated in Figure 3.6, the high disparity of runtime presented across mechanisms strengthens the need for the micro-parallelism of individual neurons and for the load balancing methods, provided by graph and instances-block parallelism, respectively.

The benchmark results for the multi-core execution on a single compute node on the strong scaling axis are presented in Figure 3.8, displaying the runtime of a single neuron against an

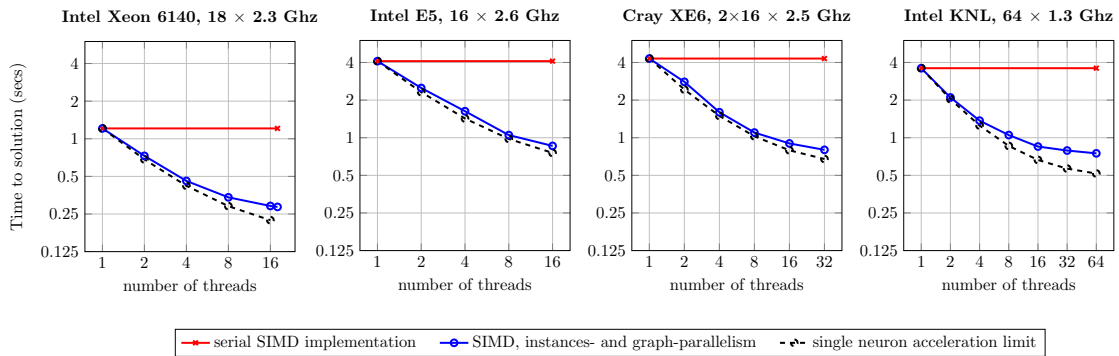


Figure 3.8 – Benchmark for the simulation of 100ms of electrical activity of a single neuron from the somatosensory cortex of the young rodent, on an Intel E5 with 16 cores at 2.6 Ghz and AVX2, an Cray XE6 compute node with 2× AMD Opteron 6380 with with 16 cores at 2.5 Ghz each and 256-bit register file width, an Intel KNL with 64 cores at 1.3 Ghz and AVX-512, and an Intel Xeon Gold 6140 with 18-core at 2.3Ghz with AVX-512. Single neuron acceleration limit represents the theoretical minimum runtime with multi-threaded execution of the parallel processes, computed as $time_{serial\ proc.} + time_{parallel\ proc.}/threads$, with serial and parallel processes taking 13% and 87% of overall time respectively.

increasing range of compute cores. For brevity, the runtime for SIRD implementations are omitted, and only the runtimes for SIMD based-implementations are presented. The ideal speedup limit on a single neuron is based on the multi-threaded execution of the parallel processes, and takes into account the serial operations — such as the delivery of events to synapses and the resolution of the linear solver detailed in Chapter 2 — that are processed beforehand and hold remaining compute cores as idle during their execution, as dictated by Amdahl’s law (Hill & Marty, 2008).

The results provide an indication of the acceleration of the simulation by the methods proposed over the single-core SIMD-accelerated reference implementation. Single neuron simulation yields a speedup of 7.0x on the KNL, 5.4x on the Cray XE6, 4.5x on the Intel E5 and 4.2x on the Xeon 6140, compared to its non-parallel SIMD counterpart. When benchmarked against the Single Instruction Single Data implementation found in the NEURON simulator, the speedup achieved was of 40.7x on the KNL, 9.3x on the XE6, 13.1x on the E5 and 14.7x on the Xeon. Moreover, the methods presented allow for an almost ideal acceleration on the Intel E5 and Cray XE6. In such scenarios, three efficiency properties are achieved: (1) high graph-parallelism; (2) high parallelism of instances block; and (3) block sizes large enough to provide full usage of the processor’s vector compute unit. Ideal scaling is strenuous due to the overhead of the low-level synchronization and threading calls by the HPX runtime to the architecture — when handling tasks in the order of nanoseconds — which becomes visible as we increase the number of compute cores. Good yet not ideal scaling is visible on the Intel KNL and Xeon architectures. A trade-off between multi-core and vector-based acceleration plays a major role in accelerating single neuron models in these two architectures, both with 512-bit register file width. As more processing units are added, the data available for SIMD instruction

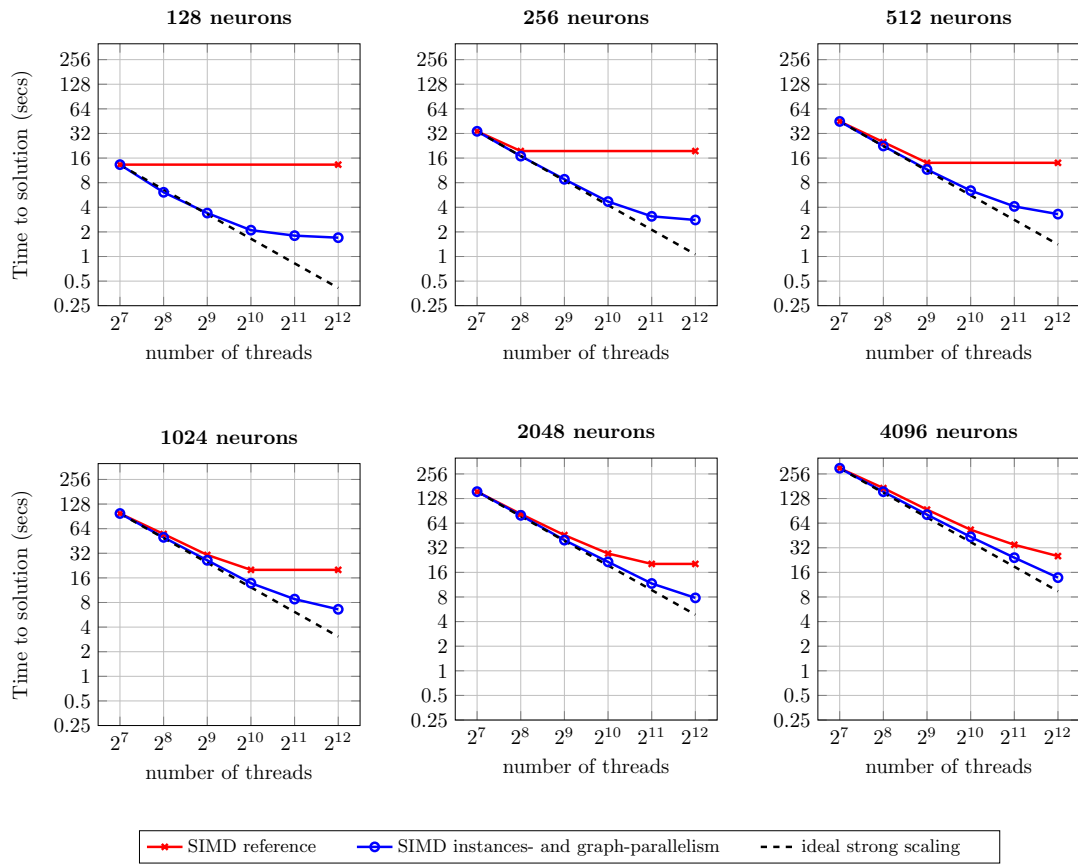


Figure 3.9 – Benchmark for the simulation of 100ms of electrical activity of the neocortex, on a cluster of 128 Cray XE6 compute node, each with a CPU containing 2x AMD Opteron 6380 with 16 cores at 2.5 GHz. Leftmost top plot displays the single neuron per locality use case, similar to Figure 3.8; following plots iteratively double the neurons/locality ratio. Ideal strong scaling assumes full overlap of serial and parallel processes, and is computed as a factor of the overall runtime as $(time_{serial\ proc.} + time_{parallel\ proc.}) / threads$.

stream is reduced so that enough work is provided for all cores. Thus, the speedup derived from vectorization is reduced as the number of instances per mechanism block decreases. The converse holds: large mechanism blocks allow for high SIMD acceleration but yields load imbalance at the core level that reduces multi-core acceleration capabilities.

3.4.5 Distributed Executions

Our performance analysis was extended to a benchmark of a larger neural circuit executed on a compute cluster of 128 Cray XE6 compute nodes, presented in Figure 3.9. The test case simulates 100ms of the electrical activity of biologically inspired model of a cortical column of the rodent neocortex, extracted with the same reconstruction method as the aforementioned somatosensory cortex. The acceleration provided at such scale becomes relevant to the simulation of large datasets that require long biological time to express its phenomena, such

as synaptic plasticity. The benchmark set-up is the following: a strong scaling analysis with a single neuron per compute unit (in line with the single node use case) is presented at the leftmost top plot. An analysis of larger datasets follows by presenting the runtime plots for an increasing number of neurons (128 to 4096 neuron plots, left to right, top to bottom), on the same hardware configuration. At the largest scaling scenario, both the reference and the proposed implementations yield enough tasks to utilize all available compute units.

In such distributed executions, acceleration is limited by the slowest compute node in the network, which is related to the sum of the computational complexity of the neurons it holds. As a side note, it is relevant to mention that runtime of individual neurons can vary up to fourfold and neurons are distributed in a round-robin fashion. However, this distribution does not alter the speedup achieved as both implementations follow the same input data distribution.

The results suggest that the strong scaling properties for single neuron simulation are preserved between the single and the 128 compute nodes use cases. In practice, the speedup achieved holds for similar ratios of neurons per compute cores. The guaranteed preservation of scaling properties allows for an accurate performance modelling on a very wide spectrum of number of neurons per total compute cores across the network, making our model adjustable to a wide range of distributed architectures.

As the input size increases, an increase of the number of neurons leads to a better overlap of computation and communication, decreasing the overall runtime in both implementations. The speedup achieved when utilising all available threads was of 7.9x for 128 neurons, 6.9x for 256 neurons, 4.2x for 512 neurons, 3.1x for 1024 neurons and 2.6x for 2048 neurons, due to coarse-grained task parallelism on the reference solution leading to higher compute load imbalance. At the largest scale of 4096 neurons (rightmost bottom plot), the methods presented are shown to be executed with a speedup of 2.1x over the reference solution at a runtime that approximates the optimal speedup given by the ideal limit. In such scenarios, higher parallelism efficiency is achieved by processing a *large enough* number of neurons that allows serial processes to be overlapped with the computation of other neurons, keeping all compute cores busy. For datasets equal or greater than 4096 neurons, both the proposed and reference implementations yield enough tasks to allow full usage of computing resources, as at least one neuron per core is readily available for processing. Nevertheless, an acceleration of more than twofold of the proposed graph- and instances-parallel version over the reference implementation is noticeable. Similarly to the branch-parallelism method covered in the previous chapter, this is due to the finer-grained parallelism exposed, allowing better dynamic task distribution across compute cores, thus better adapting to the disparity of execution runtime across neurons.

As closing remarks, the distributed implementation of our methods requires communication between compute nodes at every synaptic-exchange time interval, contrarily to existing branch-parallelism efforts (Hines et al., 2008) that require branch-section communication at

every computation timestep. The low overhead of communication and the overlap of communication and computation justify the aforementioned preservation of scaling properties.

3.5 Discussion

We developed a strategy for exploiting micro-parallelism of systems of ODEs, based on five methods: (1) a computation graph extracted from variable dependencies, exposing read-after-write and concurrent output dependencies; (2) an embarrassingly parallel computation of blocks of independent ODEs; (3) a vector-based computation of blocks of instances of similar ODEs; (4) a load balancing scheme based on individual ODEs' execution time; and (5) an asynchronous execution model that efficiently delegates tasks to computing resources on a distributed compute environment.

We applied our strategy to the compute kernel of the NEURON scientific application (Kumbhar et al., 2019), with asynchronous methods for global memory addressing space, remote procedure call, threading, synchronization and communication methods provided by the HPX runtime system. Benchmark results showed that the techniques presented allow for good strong scaling properties; a single neuron speedup of 7.0x on an Intel Knights Landing compute node, 5.4x on the Cray XE6, 4.5x on the Intel E5 and 4.2x on the Xeon 6140, compared to the state-of-the-art SIMD implementation; and over threefold this efficiency when compared to the NEURON simulator with non-vectorized computation.

Applied to a distributed network of 128 Cray X6 compute nodes and a large network of neurons, our implementation delivered a speedup of 2.1x for a network with a distribution of one neuron per thread — and full occupancy of compute resources — with the speedup increasing up to 7.9x for smaller datasets. Moreover, benchmarks of distributed executions showed that single-node strong scaling properties are preserved, and suggested an approximation to ideal scaling following an increase of the input size.

Due to the flexibility in the level of fine-grained parallelism exposed, our methods are shown to allow for an almost ideal usage of computing resources on large datasets on a wide range of compute architectures.

4 Fully-Asynchronous Cache-Efficient Simulation

This chapter is adapted from the preprint version of the following article:

Magalhaes B., Hines M., Sterling T., Schürmann E, "Fully-Asynchronous Cache-Efficient Simulation of Detailed Neural Networks", accepted at International Conference on Computational Science (ICCS) 2019

Personal contributions: conceptualization, formal analysis, investigation, methodology, software, validation and writing.

4.1 Abstract

Modern asynchronous runtime systems allow the rethinking 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 model allows for an increase of processor's prefetching capabilities, increased cache locality, and an 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 (i.e. synaptic transmission), and takes advantage of the pairwise time dependency between equations, leading to a fully-asynchronous parallel execution model, supported by an *exhaustive yet not speculative* stepping protocol. 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 at each stepping iteration.

Our methods were implemented on the compute 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 speedup that leads to a reduced runtime compared to the bulk synchronous execution, yielding a speedup between 25% to 65% across four distinct compute architectures, and 15% to 40% on a distributed execution on 32 Cray XE6 compute nodes.

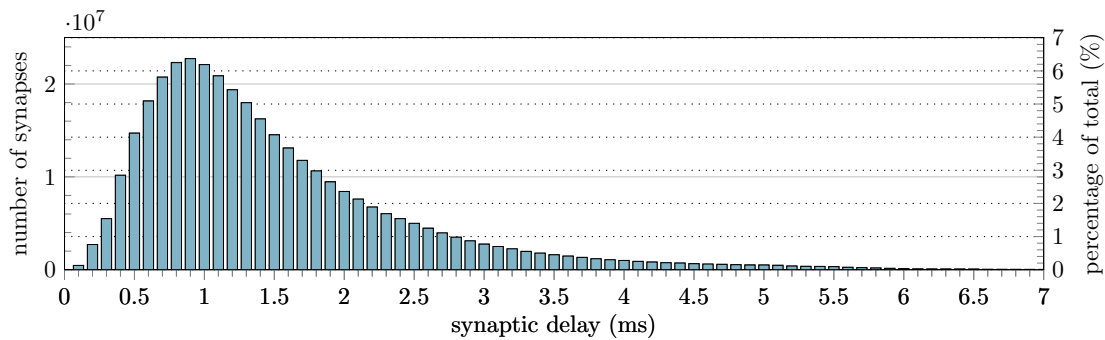


Figure 4.1 – Distribution of synaptic delays in terms of count (left y-axis) and percentage (right y-axis) of all synapses in a network of 219,247 neurons, extracted from a biologically inspired digital reconstructed model of the rodent neocortex from Markram et al. (2015). Histogram contains one bin per interval of 0.1 ms . The leftmost bar ($x = 0.1\text{ ms}$) represents the communication step size of state-of-the-art implementations based on the Bulk Synchronous Parallel execution model.

4.2 Introduction

Similarly to state-of-the-art approaches for large scale simulations on a wide range of scientific domains, the simulation of the electrical activity of neural networks follows the Bulk Synchronous Parallel (BSP) execution model. 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. Neurons are interpolated independently, and their synaptic activity and stepping is synchronized at the end of each interval.

Synaptic communication is typically performed with Message Passing Interface (MPI). However, 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 (Ovcharenko et al., 2015). 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 (Zenke & Gerstner, 2014). Similar scalability issues derived from communication and synchronization are also prone to occur and have already been discussed in other high-impact large scale simulations of time- and space-bounded elements, such as N-body simulations (Gordon Bell Prize (GBP) winners for the years 2009, 2010 and 2012 (Ishiyama et al., 2012)), cardiac model simulations (GBP 2015 finalist (Randles et al., 2015)), fluid dynamics (GBP 2013 winner (Rossinelli et al., 2013)), materials crystalization (GBP 2011 winner (Shimokawabe et al., 2011)), weather forecasting (Rodrigues et al., 2010) and direct volume rendering (Kutluca et al., 2000).

The stepping constraint implicit in the short interval that determines the communication step size follows from the assumption of the worst case scenario derived from synaptic activity of interconnected neurons. The minimum synaptic delay between a pre- and a post-synaptic neuron defines the maximum difference in stepping allowed, with the guarantee that no spike

is missed if the pre-synaptic neuron spiked in the previous synaptic delay interval. To ensure that such guarantee holds for a network of neurons, the minimum synaptic delay across all pairs of neurons delimits the length of the global synchronization/communication step. However, the minimum synaptic delay that dictates the communication step size, accounts for a very small portion of the overall synaptic connectivity. In practice, such value is computed as $0.1ms$ or equivalently four compute steps and accounts for circa 0.13% of all the synaptic delays. For completeness, the histogram of distribution of synaptic delays on a network of 219 thousand neurons is provided in Figure 4.1.

Motivated by the search of new execution models that overcome the aforementioned issues, and accelerate the simulations of detailed neuron models, this chapter presents the **Fully-Asynchronous Parallel** (FAP) execution model that improves cache locality and provides cache-level acceleration by removing synchronous communication steps. The model presented relies on an **exhaustive yet not speculative** stepping protocol that advances ODEs timestepping beyond synchronization barriers, based on the time coupling 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 (locality).

We implemented our methods on the compute kernel of the NEURON scientific application (Kumbhar et al., 2019), available as open source (Blue Brain Project, 2015a), with communication, synchronization, and threading enabled by the HPX-5 runtime library (Sterling et al., 2014). A benchmark was performed on four different compute architectures, and demonstrate a decrease of 31% to 65% in runtime compared to the reference solution on Intel E5, Intel Knights Landing and Intel Xeon architectures, and a decrease of 25% to 31% on an AMD Opteron. A distributed execution on a Cray XE6 compute cluster with 32 compute nodes demonstrates a speedup of 15%-40%.

4.3 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

Chapter 4. Fully-Asynchronous Cache-Efficient Simulation

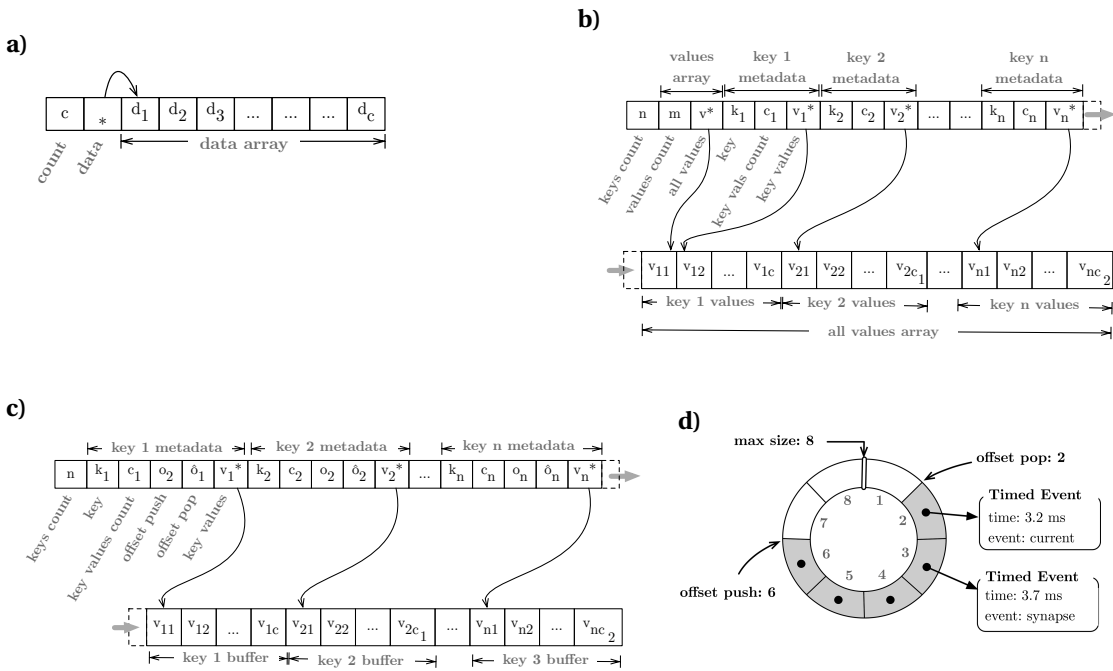


Figure 4.2 – Memory representation of linear data structures. Gray arrows represent connections between contiguous memory regions. **a)** linear vector; **b)** linear map; **c)** linear priority queue; **d)** a circular array representing a sample entry in the priority queue.

maximum extent once it has been mapped to the cache. Following this reasoning, the next section details the implementation of our cache efficiency methods.

4.3.1 Linear Data Structures

To avoid fragmentation of data layouts in memory due to dynamic memory allocations and to 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. A single memory buffer allocation follows, and an in-place instantiation of state variables and containers yields a single contiguous memory representation of the whole neuron. The description of the containers follows in the following paragraphs.

Linear Vector: implemented as a serialization of the `std::vector` class found in the C++ standard library (Josuttis, 2012), with meta data, pointers, and data elements placed on a sequential memory space. An illustration of the linear vector data structure is displayed in Figure 4.2 (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 the metadata containing all (ordered) keys, thus yielding similar computational complexity as the `std::map` implementation with a red-black tree, at $O(\log n)$. The metadata related to each entry in the map contains the linear vector holding the elements referring to the respective key. Because the size of each vector in a worst case scenario can be computed beforehand (e.g. maximum number of incoming synapses at any given time), a continuous memory region containing all entry vectors — and consequently a linearization of the map — is possible. Moreover, the linear data representation of the map values allows for operations such as minimum value, maximal value and value search can be performed with the same efficiency as a single vector by traversing the continuous memory of all data arrays. For clarity, the memory layout of the linear map is presented in Figure 4.2 (b).

Linear Priority Queue: storing time-driven events such 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 neuron index (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 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 4.2 (c). At the level of each key, given a pre-synaptic neuron index, 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 completeness, Figure 4.2 (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 (Arge et al., 2002) improve memory access pattern yet do not guarantee a 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 all the 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).

As a closing remark, linear data structures allow for better cache locality and introduce a speedup in the overall execution time by reutilizing previously cached memory locations.

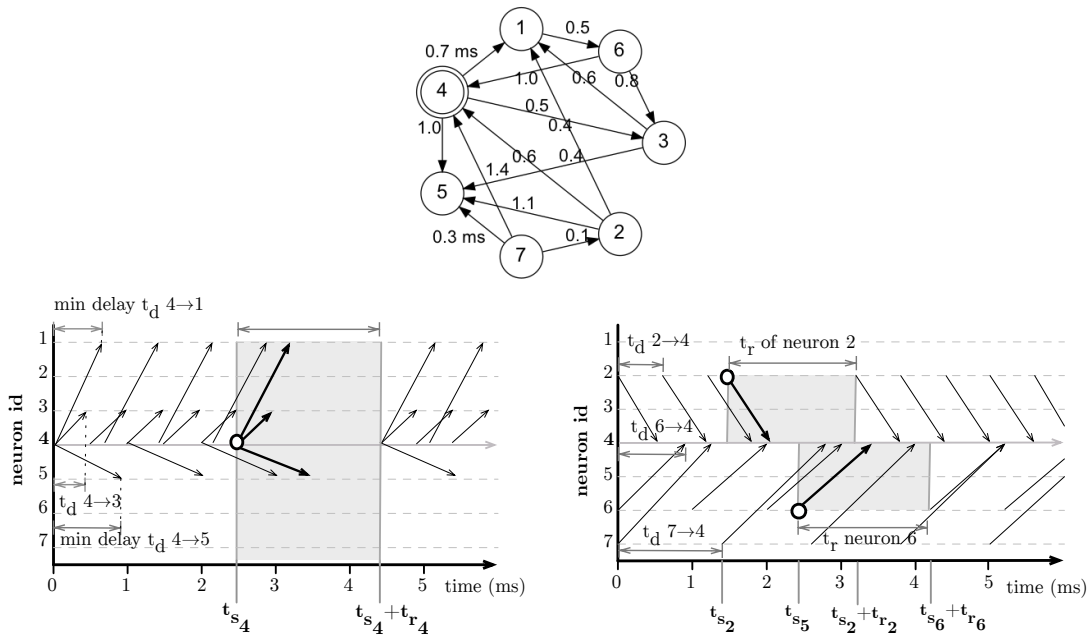


Figure 4.3 – Representative schema of the time-dependency control method for the synchronization of stepping and spikes, applied to a network of seven neurons. **Top:** sample network of neurons (vertices 1-7). Arrow heads (tails) connect to post- (pre-) synaptic neurons. Labels on edges describe the minimum synaptic delay across all synapses from a pre- to a post-synaptic neuron connection. **Bottom left:** 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 synaptic neuron, given by his present time plus the minimum transmission delay between itself (the pre-) and the post-synaptic neuron — represented as t_d *pre* → *post* and conforming to the graph on the left. Spike notifications (t_s , circles) also provide progress updates and allow post-synaptic neuron to freely proceed to a time distance equivalent time of spike plus the refractory period (t_r) of the pre-synaptic neuron. **Bottom 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.

The main goal is then to keep these linear data structures in cache for as long as possible, to maximize their reutilization, or equivalently, the number of steps taken on each stepping iteration. This topic is detailed in the following section.

4.3.2 Time-Based Elements Synchronization and Stepping

To allow for a flexible progress of neurons in time, that detach stepping from the constraints of the minimum synaptic delay across all pairs of neurons in the system, our method creates a distributed graph holding the time dependencies between neurons, as illustrated in Figure 4.3 (top). The main rationale is to allow for a given post-synaptic neuron to advance in time based on the progress of its pre-synaptic dependencies. The result is an exhaustive stepping mechanism, that maximises the number of steps per neuron, and an increased reutilisation

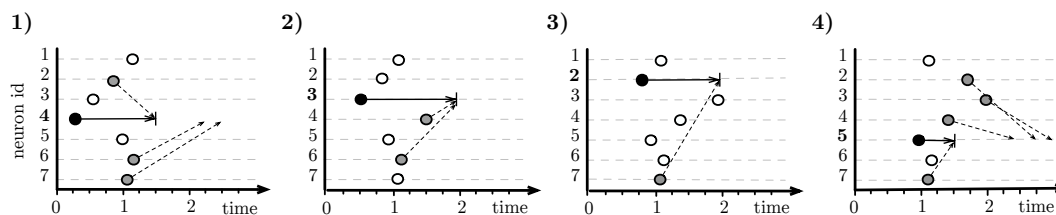


Figure 4.4 – A sample workflow of four iterations of the neuron scheduler applied to the network of seven neurons displayed in Figure 4.3 (top). On the iteration (frame 1), neuron 4 is the earliest in time (coloured black) and is allowed to proceed to time $1.5ms$, dictated by the transmission delay 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), 3) and 4), respectively.

of 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 every computation step, a neuron notifies its post-synaptic neurons of its current step (if necessary), and stores in a queue the next time instant at which a new notification will be 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. the time 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 4.3 (bottom). This fully-asynchronous execution yields three main advantages compared to its BSP counterpart: (1) larger stepping intervals by completely removing collective synchronization barriers; (2) less often communication as the pairwise communication delays are generally two orders of magnitude longer than the global minimum transmission delay; and (3) full overlap of computation and communication. To maximise the number of steps taken on any stepping interval, a neuron scheduler allows for an optimal decision of the next neuron to step, by keeping track of the progress of neuron throughout simulation. This topic is covered next.

4.3.3 Neuron Scheduler

In order to increase cache efficiency even further, 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. At the onset of stepping, a neuron queries its dependencies map for the time allowed by its pre-synaptic dependencies, and performs the

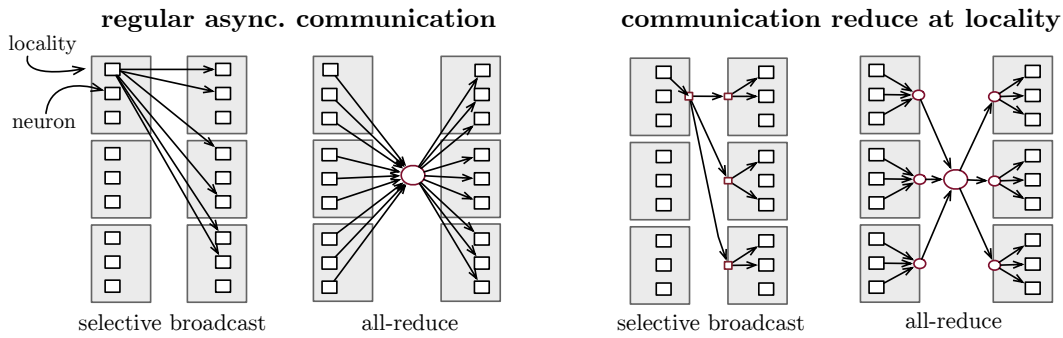


Figure 4.5 – A sample diagram of the communication required for the selective broadcast and all-reduce operations using regular (left) versus locality-reduced (right) communication.

maximal stepping allowed. For clarity, Figure 4.4 provides a schematic example of 4 iterations based on the scheduled stepping.

4.3.4 Communication Reduce

Global Memory Address Space (GAS) — as defined by the ParalleX standard — allows for remote procedure executions on 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 a physical address in local memory. However, on a distributed execution, each call is an instantiation of a procedure to be executed in an object held possibly in a different locality. Therefore, large amount of object-to-object communications may become a bottleneck by saturating the network bandwidth. This issue is trivial to overcome on MPI-based implementations, as the sender locality is responsible for buffering, packing and initiating the communication, while the converse operations must follow from the receiver. On the ParalleX runtime system, its resolution is not as simple, as data representation in GAS arrays removes the locality-awareness of each object on 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 the communication phase, as single packets to remote localities. At the recipient locality, a map of pre-synaptic neuron indices to the list of local GAS addresses allows for a message to be unfolded and locally spawn to the recipient GAS addresses in the locality. In practice, for a selective broadcast operation — typical of a spike exchange — this method replaces n remote communications to neurons, by l communications to localities followed by a lightweight threads spawn at each locality, where $n \gg l$. For completeness, Figure 4.5 provides an illustration of the communication reduce methods.

4.4 Benchmark

4.4.1 Implementation

Our strategy was implemented on the compute kernel of the NEURON scientific application (Kumbhar et al., 2019), available as open source (Blue Brain Project, 2015a). Communication, synchronization and memory allocations performed with MPI, OpenMP and malloc, were replaced by the equivalent HPX counterparts. Both the fully-asynchronous and the reference implementations follow the same numerical resolution.

The following features were implemented with HPX primitives:

- Memory allocations of individual neurons are performed at the onset of execution. Neurons are placed on the Global Address space based on an offset given by the pre-computed (linear buffer) size of each neuron. Neurons metadata and linear containers are initialized on the previous memory space;
- Point-to-point communication guiding neuron stepping notifications is implemented with asynchronous remote procedure calls. The delivery of a message updates the respective pre-synaptic neuron's entry in the map of interpolation times instant per pre-synaptic neuron (held on the memory space of the post-synaptic neuron);
- Communication reduce is implemented with a protocol that performs communication to remote locality addresses (instead of neuron GAS addresses), with posterior lightweight threads performing the delivery of data to the local neurons;
- The scheduler at every compute node is built with a memory-protected priority queue holding the time instant of the local neurons, and the set of neurons being processed. Upon the termination of each stepping iteration, a neuron updates its new time instant in the scheduler's priority queue, and removes itself from the set of executing neurons. The next neuron to be picked by the scheduler is the first in the queue which is not part of the set of executing neurons.

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. The waking of dormant neuron threads is supported by an and-gate per neuron (initiated as 1).

Remaining details on the HPX implementation of other features — such as synaptic delivery — have been covered in previous chapters and will be omitted for brevity.

4.4.2 Use Case

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. (2015), with the distribution of synaptic connectivity previously presented in Figure 4.1. Each representation of a neuron requires a total memory of 4 to 12 MB, including the neuron structure and the supporting data structures for the full-asynchronous execution.

4.4.3 Hardware Specifications

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: (1) Intel Sandy Bridge E5-2670 with 16 cores at 2.6 Ghz; (2) Cray XE6 compute node with an AMD Opteron 6380 with 16 cores at 2.5 Ghz each; (3) Intel Knights Landing (KNL) Xeon Phi with 64 cores at 1.3 Ghz; and (4) 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.

Distributed benchmarks were executed on 32 compute nodes of Cray XE6 compute nodes, with specialized Infiniband network hardware for efficient point-to-point communication.

4.4.4 Linear Containers

Cache efficiency of linear containers was measured with the *likwid* suite for performance monitoring and benchmarking (Treibig et al., 2010) on the Xeon 6140 processor. The performance counters were set to measure the containers' performance only, in order to isolate its performance analysis from other features. The benchmark testbench compares cache efficiency of linear versus standard library's containers. The estimated amount of read/write operations are:

- a spike or event notification at approximately every 15 ms, requiring a loop through the map of post-synaptic neurons' information;
- 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 the maximum 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 — with a push to the priority queue — at almost every time step.

The cache efficiency results on the BSP-based stepping protocol, with four continuous steps

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, steps distribution in Figure 4.6)

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

Table 4.1 – Cache efficiency of linear and standard library (std) containers, for the BSP execution model (4 steps per neuron, top) and the Fully-Asynchronous Parallel execution model (bottom, with the steps distribution presented in Figure 4.6).

per neuron, and a communication interval at every 0.1ms, is provided in Table 4.1 (top). Results demonstrate a lower time to solution of circa 4x 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.

4.4.5 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 was measured and is presented in Figure 4.6 (c). The step sizes vary depending on the circuit size due to an increased inter-neuron connectivity for larger circuits. In practice, an increase in the number of neurons leads to a possibly increased amount of pre-synaptic connectivity, and to a higher probability of having a smaller minimum synaptic delay for a given pair of neurons, leading to smaller stepping intervals.

With this in mind, we performed a similar cache efficiency benchmark for the asynchronous

Chapter 4. Fully-Asynchronous Cache-Efficient Simulation

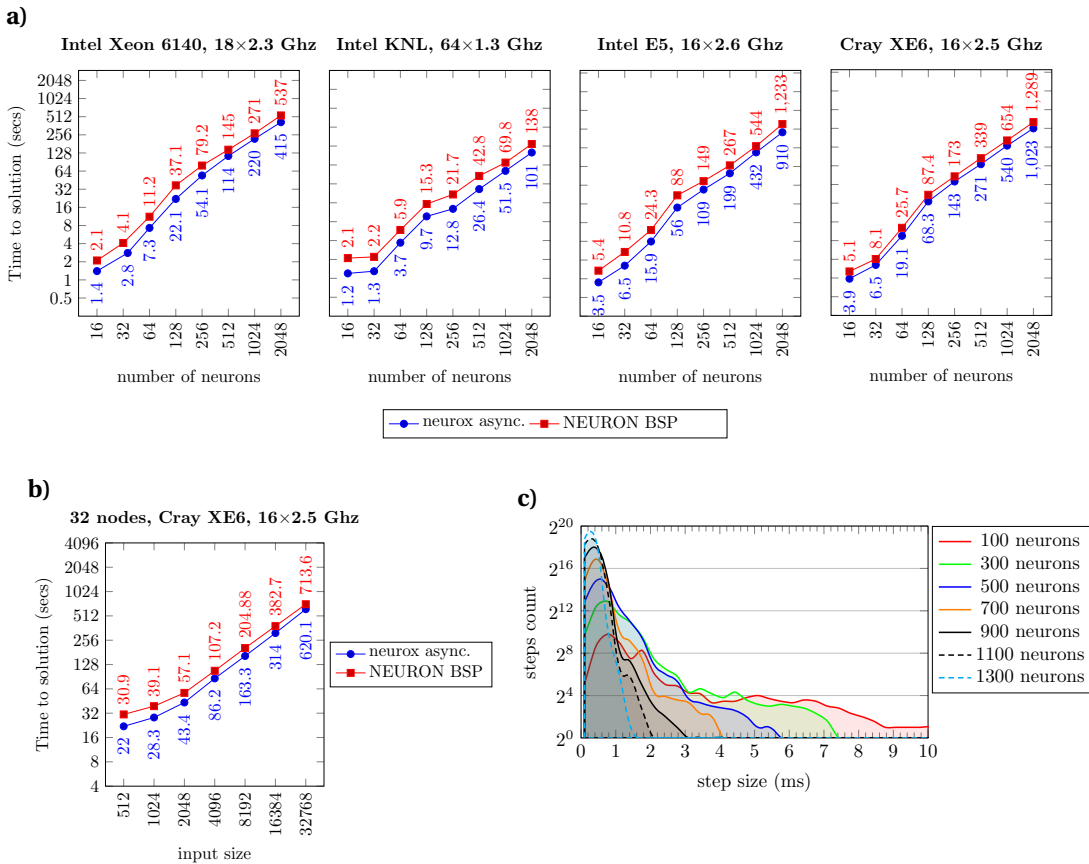


Figure 4.6 – **a)** Time to solution of the methods presented (neurox async.) and the Bulk Synchronous Parallel equivalent (NEURON BSP) on the simulation of 100ms of the electrical activity of differently sized neural networks, on four different hardware specifications. **b)** Benchmark results for the 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 4.1.

execution model, and the details are provided in Table 4.1 (bottom). Results of linear versus standard library’s (std) implementations follow in line with the BSP use case, displaying better memory access and lower time to solution when comparing both implementations. Asynchronous scheduled stepping yielded a lower runtime in the order of 5 – 10%, and a more efficient memory access compared with the previous BSP benchmark, on both linear and std implementations.

4.4.6 Communication Reduce

The efficiency of the communication reduce was benchmarked on a network of 32 Cray XE6 compute nodes, measuring the runtime, number of point-to-point (p2p) and number of reduce operations per locality, on the previous testbench. A benchmark compares the reduced

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	2168	2327	7543	8855	24.3K	33.4K	70.1K	124K	188K	480K
reduce comm.	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 neurons	
	red.	simple	red.	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	623K	665K	2.34M	2.72M	8.25M	11.09M	44.77M	25.79M	71.75M	181.46M

Table 4.2 – Performance of regular versus locality-reduced communication in terms of runtime, and number of point-to-point and reduce communications, on the BSP (top) and asynchronous (bottom) execution models.

versus non-reduced (simple) communication implementations. Results are presented for the BSP execution model — with a point-to-point communication for transmission of synapses and a reduce operation for the synchronization of neurons stepping — and the FAP model, where point-to-point communication guides synaptic activity and neurons stepping notifications. The results are provided in Table 4.2 and demonstrate a reduction of communication workload and runtime, on both the BSP and the FAP models. The gap in communication workload between reduced and non-reduced implementations increases with the circuit size, as more neurons incur more synaptic activity and communication. An acceleration of circa 5% – 10% is visible when moving from the BSP to the asynchronous execution model.

4.4.7 Single Compute Node Executions

The benchmark for a single compute node of the four aforementioned compute architectures is displayed in Figure 4.6 (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 speedup 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 4.6 (c). On the Intel Xeon 6140, the methods yield a speedup between 31% — for the largest network of 2048 neurons — and 51% for the network of 16 neurons. The speedups for the remaining architectures are 36%-65% for the KNL, 35%-54% on the Intel E5, and 26%-31% on the Cray XE6.

4.4.8 Distributed Executions

In order to understand whether the efficiency of the fully-asynchronous execution model in single compute nodes holds in a distributed compute environment, we extended our benchmark to a network of 32 Cray X6 compute nodes. 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 4.6 (b), and

display a speedup of 16% for the largest dataset of 32768 neurons, up to 40% for 256 neurons i.e. one neuron per core per locality.

4.5 Discussion

In this chapter, we explored the capabilities of new runtime systems for the numerical simulation of large systems of ODEs. We presented the fully-asynchronous parallel execution model, 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 detailed 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 were introduced and detailed: (1) a linear data representation of a vector, map and priority queue containers that allow for the sequential instantiation of whole neuron data structures in memory; (2) an exhaustive yet not speculative stepping of individual equations based on their time dependencies, supported by (3) a point-to-point communication protocol that actively notifies neuron time dependencies of the time advancement 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 compute kernel of the NEURON scientific application and tested on a biologically inspired branched neural network. We analysed and demonstrated the efficiency of the features introduced in terms of communication, cache efficiency, patterns of data loading, and time to solution. Benchmark results yielded a significant speedup in runtime in the order of 25% to 65% across different compute architectures and up to 40% on distributed executions.

5 Fully-Asynchronous Fully-Implicit Variable-Order Variable-Timestep Simulation

This chapter is adapted from the preprint version of the following article:

Magalhaes B., Hines M., Sterling T., Schürmann F., "Fully-Asynchronous Fully-Implicit Variable-Order Variable-Timestep Simulation of Neural Networks", published on arXiv

Personal contributions: conceptualization, formal analysis, investigation, methodology, software, validation and writings.

5.1 Abstract

State-of-the-art simulations of detailed neural models follow the Bulk Synchronous Parallel (BSP) execution model. Execution is divided in equidistant communication intervals, equivalent to the shortest synaptic delay in the network. Neurons stepping is performed independently, with collective communication guiding synchronization and exchange of synaptic events.

Commonly to most biological simulations, the interpolation step size is fixed and chosen based on some prior knowledge of the fastest possible dynamics in the system. However, simulations driven by a stiff dynamics or a wide range of time scales — such as multiscale simulations of neural networks — struggle with fixed step interpolation methods, yielding excessive computation of intervals of quasi-constant activity, inaccurate interpolation of periods of high volatility in solution, and being incapable of handling unknown or distinct time constants. A common alternative is the usage of adaptive stepping methods, however they have been deemed inefficient in parallel executions due to computational load imbalance at the synchronization barriers that characterize the BSP execution model.

We introduce a distributed fully-asynchronous execution model that removes global commu-

nication and synchronization, allowing for long variable timestep interpolation of neurons. Asynchronicity is provided by active point-to-point communication notifying neurons' time advancement to synaptic connectivities. Time stepping is driven by scheduled neurons time advancement based on synaptic delays across neurons, yielding an *exhaustive yet not speculative* adaptive-step execution.

Execution benchmarks on 64 Cray XE6 compute nodes demonstrate a reduced number of interpolation steps, higher numerical accuracy and a lower time to solution, compared to state-of-the-art methods. Efficiency is shown to be activity-dependent, with scaling of the algorithm demonstrated on a simulation of a laboratory experiment.

5.2 Introduction

In the previous Chapter 4, we presented a method for acceleration at the cache level, where we pioneered the Fully-Asynchronous Parallel (FAP) execution model applied to fixed timestep interpolations, illustrated in diagram c) in Figure 1.2. The underlying logic is that, because the BSP communication interval refers to the shortest event delay across the system, the removal of collective synchronization barriers and stepping neurons based on their pairwise connectivities, allows for stepping intervals longer than the BSP communication timeframe. This was shown to promote better processor prefetching, lower data volume across memory layers, data representations being kept longer in faster cache levels, and ultimately, a reduced time to solution.

An orthogonal axis of acceleration relies on improved numerical resolution by utilizing a variable step interpolation of individual neurons, with synchronization at each BSP communication interval. Lytton et al. (Lytton & Hines, 2005) presented an implementation of adaptive step interpolation in NEURON, by utilising the CVODE library (Cohen & Hindmarsh, 1996). For a given function and time, the CVODE approximates the derivative of a function using information from previous steps (stored in the state history), thereby increasing the accuracy of the numerical resolution. The step size is tentatively computed in order to respect a user-provided tolerance (**atol**), thus adapting the step size to rapid variations of voltage trajectory. Current events that cause a discontinuity of solution forces the integrator to start again with a new state. The method guarantees coherent variable stepping on distributed compute nodes by exchanging events at BSP-based communication barriers, therefore avoiding synapses being delivered in preceding instants in time. However, it limits the steps length to the BSP communication barrier, or the instant of the nearest synaptic or discontinuity event. For clarity, an illustration of this method is displayed in layout b) in Figure 1.2.

Following from the two previous efforts, this chapter introduces a method for the distributed fully-asynchronous variable-order variable-timestep interpolation of detailed neuron models, that benefits from cache-efficient barrier-free synchronization and performs variable timesteps on the FAP execution model. We show that by following an *earliest neuron steps next* scheduler, we allow for large time interpolation intervals, and maximise the efficiency of the

variable step interpolator beyond what was believed to be possible in BSP-based executions. The scope of our methods are presented in Figure 1.2, layout d).

The contributions of this chapter are as follows. We present the mathematical formalism underlying the simulation of our use case and its resolution with variable step interpolation. We perform a study of numerical precision on the electrical activity of a single neuron and demonstrate higher numerical accuracy than its fixed step counterpart found in NEURON. We analyse and discuss the benefits and bottlenecks of two distinct distributed variable-step execution models — a speculative model with backstepping, and a scheduled non-speculative model — and provide insights on their feasibility on distributed simulations of large networks of neurons. We demonstrate a low sensitivity of our model to stiffness of solution (spiking rate), and high susceptibility to discontinuity events (synaptic activity) with a guaranteed speedup for a discontinuity rate below approx. 1000 Hz. To measure the relevance of our findings on a real use case, we simulate a laboratory experiment based on the spontaneous activity of 219 thousand neurons, demonstrating a mean discontinuity frequency of 94 Hz, and large periods of absence of discontinuities, demonstrating the suitability of our methods to the problem domain.

An implementation of our methods on the compute kernel of the NEURON scientific application is detailed, and a benchmark is performed on 64 Cray XE6 compute nodes. Distributed asynchrony and multi-core executions on a global memory address space is provided by the HPX runtime system (Sterling et al., 2014). We simulate five spiking regimes that characterize several dynamics of the mammal brain, on an input of 1024 to 65536 neurons, and compare our methods against five state-of-the-art numerical solvers. Results demonstrate a speedup of 544-65x for a quiet spiking regime of 0.25Hz representing a majority of neurons in regular brain activity, down to 7.7-1.8x to a moderate regime of 6.5Hz, and 2x to no acceleration for 38 Hz, a pattern of unlike occurrence or short duration. An analysis of the overall performance achievable on the previous laboratory experiment demonstrates a speedup of 224.5-11.9x for an execution with precise delivery of events, increasing to 225.1-17.1x and 228.5-24.6x for two optimized alternatives that group events delivery in the next half and full timestep (a numerical precision similar to the state-of-the-art fixed-step methods). To finalize, we show that over 95% of neurons fall in three spiking regimes that guarantee the preservation of the speedup in larger networks, demonstrating good scaling properties of our methods to very large networks of neurons.

5.3 Methods

5.3.1 Resolution of Simple and Complex Neuron Models

In Section 1.6 we detailed a problem specific optimization that allows for a substantial speedup on the resolution of **simple neuron models** such as the Hodgkin-Huxley (Equation 1.2), where state variables m , n and h are described by linear ODEs and depend only on the voltage V ,

and vice-versa. In such scenarios, an implicit resolution based on interleaved timestepping of voltage and states, by solving voltages at a given time t and states at time $t + \Delta t/2$, allows for the resolution of the system of ODEs as a system of linear equations.

Resolution of **complex models**, including non-linear and/or correlated state equations cannot be resolved with the aforementioned method, and require a fully-implicit resolution. A use case of high importance is the model of synaptic plasticity and learning, such as the one presented by Graupner et al. (Graupner & Brunel, 2012), with cubic ODEs and correlated calcium and synaptic efficacy values, or Chindemi et al. (Chindemi, 2018) with higher complexity introduced by ODEs describing synaptic weight and calcium activity models. For such scenarios, numerically reliable resolutions rely on fixed-step iterative implicit methods such as the Backward Euler. Alternatively, an implicit variable timestep method with variable order is possible, and its application to our use case is detailed in the following paragraphs.

5.3.2 Variable Step Implementation

The CVODE — C Variable-step solver for ODEs, (Cohen & Hindmarsh, 1996) — is an implementation of the Backward Differentiation Formula (BDF) for the variable-step multistep implicit method solving the Initial Value Problem (IVP, $\dot{y} = f(t, y)$, $y(t_0) = y_0$ where $y \in \mathbb{R}^N$) for ODEs as:

$$\sum_i^q \alpha_{n,i} y_{n-i} + \Delta t_n \beta_n \dot{y} = 0 \quad (5.1)$$

where $y = [..., V_{k-1}, V_k, V_{k+1}, ..., x_{i-1}, x_i, x_{i+1}, ...]$ is a vector representing the state variables of a neuron or a set of neurons, following the variable notation in Equation 1.1. q is the order of the current iteration, α and β are the BDF-method coefficients which are q -dependent and unfolded as:

$$\begin{aligned} \text{BDF-1: } & y_n - y_{n-1} = \Delta t_n \dot{y}; \\ \text{BDF-2: } & 3y_n - 4y_{n-1} + y_{n-2} = 2\Delta t_n \dot{y}; \\ \text{BDF-3: } & 11y_n - 8y_{n-1} + 9y_{n-2} - 2y_{n-3} = 6\Delta t_n \dot{y}; \\ \text{BDF-4: } & 25y_n - 48y_{n-1} + 36y_{n-2} - 13y_{n-3} + 3y_{n-4} = 12\Delta t_n \dot{y}; \\ \text{BDF-5: } & 137y_n - 300y_{n-1} + 300y_{n-2} - 200y_{n-3} + 75y_{n-4} - 12y_{n-5} = 60\Delta t_n \dot{y}; \end{aligned} \quad (5.2)$$

Note that the interpolation of order 1 refers to the Backward Euler. In brief, CVODE returns the Δt_n and y_n that solve BDF- q for an user-provided tolerance. The computation is performed iteratively, with a suggested step size for each iteration based on the solution gradient and order q . For a new step n , iteration m : (1) CVODE suggests $y_{n(m)}$ and Δt_n based on the user provided jacobian \dot{y}_n . The initial guess $y_{n(0)}$ is computed explicitly from history; (2) CVODE provides the Right Hand Side (RHS) computed from the BDF- q formula, and expects an user provided $y_{n(m+1)}$; (3) the implicit resolution relies on Newton iterations, with a stop condition

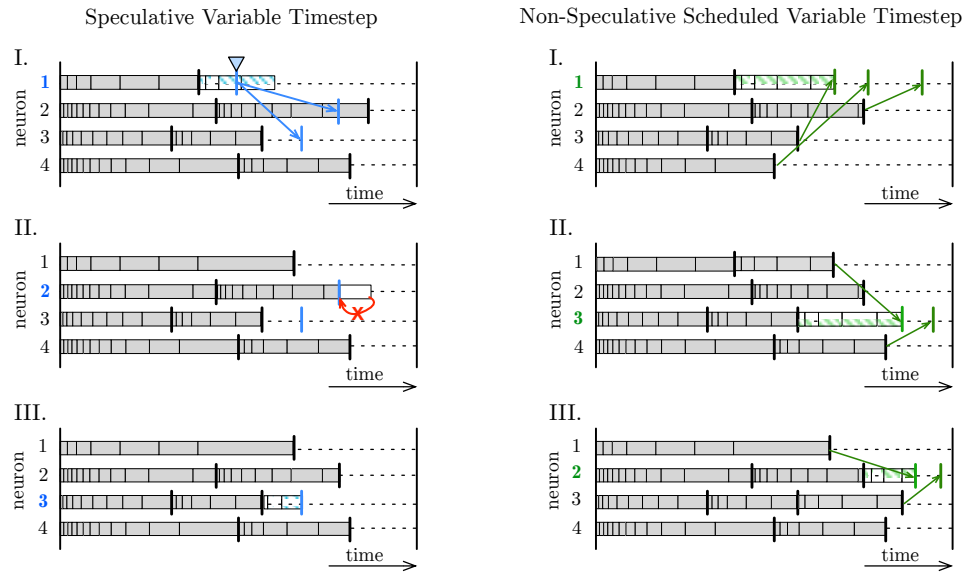


Figure 5.1 – Illustrative workflow of the distributed variable timestep methods analysed. **Left:** Speculative interpolation advances neurons iteratively; **I.** Neuron 1 performs a step (blue area) and spikes during the step interval, with spike instant marked with an inverted triangle. Spike delivery times to post-synaptic neurons 2 and 3 are marked with blue arrow heads; **II.** Neuron 2 steps next. Its current interpolation time exceeds the spike delivery time, requiring backstepping to the missed event delivery time (red arrow); **III.** Neuron 3 interpolates until the next (spike) event time. **Right:** Non-Speculative scheduled time stepping advances the earliest neuron in time, and steps until the next event delivery time or maximum time allowed by pre-synaptic connectivity (green area), to guarantee no backstepping; **I.** Neuron 1 advances to the earliest time instant allowed (red arrow heads), given by the time instants of pre-synaptic neurons 2, 3 and 4 and synaptic delays 2→1, 3→1, and 4→1, respectively; **II.** Neuron 3 is now the earliest neuron in time, and follows analogously based on the synaptic delays of neurons 1 and 4; **III.** Neuron 2 advances based on the synaptic connectivity of neurons 1 and 3.

based on an test of $\|y_{n(m)} - y_{n(0)}\| \leq \epsilon$. If error is greater than threshold, a reiteration follows with a smaller $\Delta t_{n(m+1)}$; if error is smaller, proceeds to step $n+1$ with larger $\Delta t_{(n+1)(0)}$. The user provides the function that computes the ODE right-hand side for a given value of time t and state vector y ; and the function that computes the Jacobian $j = \partial f / \partial y$ or an approximation to it.

We utilise the Jacobian function with the CVODE-based preconditioning function that solves $Px = b$, where b is the input and P approximates $M = I - \gamma J$. This is the default Jacobian in NEURON. Two alternative Jacobian implementations were tested and deemed infeasible: (1) a diagonal linear approximation to the Jacobian, given by $Jy = \dot{y}$, displayed faster computation yet highly inaccurate results; and (2) a dense matrix approximation $J_{ij} = (f_i(y + h_j, t) - f_i(y, t)) / h_j$ for a parameter variation h_j was shown to be accurate yet infeasible due to the high time to solution and high memory requirements for the dense matrix of states.

5.3.3 Asynchronous Timestepping of Networks of Neurons

Control of neurons time advancement on synchronized distributed executions is a solved problem, by enforcing a BSP-like synchronization barrier (Hines & Carnevale, 1997), as in layouts a) and b) in Figure 1.2. Here we discuss alternatives following an asynchronous execution model. We implemented and analysed the feasibility of two distinct fully-asynchronous barrier-free execution models, displayed in Figure 5.1 and detailed next.

The initial approach is inspired on the speculative interpolator previously designed for NEURON for a single compute node (Lytton & Hines, 2005). Neurons are described by individual interpolators, and advance in time under the best assumption that no **discontinuity** of solution (synaptic current) will arrive with a delivery time earlier than the neuron current time. Discontinuities lead to a **reset of the IVP** problem and interpolator state history, and consequently to small steps in the following iterations. When a discontinuity is required to be delivered in a past instant in time, a **backstepping** operation must precede, in order to reset the recent step and interpolate neuron state back to a time instant of confidence (the time of the discontinuity). Simulations of small neuron networks on single compute nodes are possible and have previously shown a substantial runtime acceleration utilising this model (Lytton & Hines, 2005). The state of neurons is local to the compute node, thus backstepping of neuron states and synaptic activity are not computationally heavy and do not require communication. However, in our implementation this approach demonstrated two main drawbacks: (1) large spiking networks lead to a high number of IVP resets, and large amount of time spent on speculative stepping with posterior backstepping. This is mainly due to, in practice, one being unable to tell which neuron should be stepped at a time, such that the risk of backstepping would be minimized and the step length maximized. Moreover, (2) on distributed executions, a main problem arises when synaptic activity (exchanged across different compute nodes) needs to be reverted, requiring further communication. On large networks of neurons, the reversal may lead to an extremely complex cascade chain of reversal notifications and backstepping across neurons on different compute nodes, leading to a high communication and computation workload, deeming this methodology infeasible.

An alternative approach was implemented, based on the non-speculative asynchronous stepping methodology detailed in the previous Chapter 4. Neurons hold a map storing the time instant of their pre-synaptic connectivities. The map is updated by stepping notifications received actively at a certain frequency, throughout the stepping of its pre-synaptic dependencies. The frequency value is set at a neuron pair level, to a minimum value that guarantees no deadlocking. More importantly, this value is at least the BSP communication interval of 0.1ms , and can reach up to several milliseconds (Figure 4.1), thus minimising overall communication. Neurons step to the maximum time allowed by their synaptic connectivities. This guarantees synapses to be delivered in future time instants, thus removing backstepping and reversion of sent synaptic spikes. The method is improved with an *earliest neuron steps first* scheduler at each compute node that keeps track of neurons advancement, and picks the earliest neuron in time as the next to interpolate. This guarantees the maximisation of the step length and

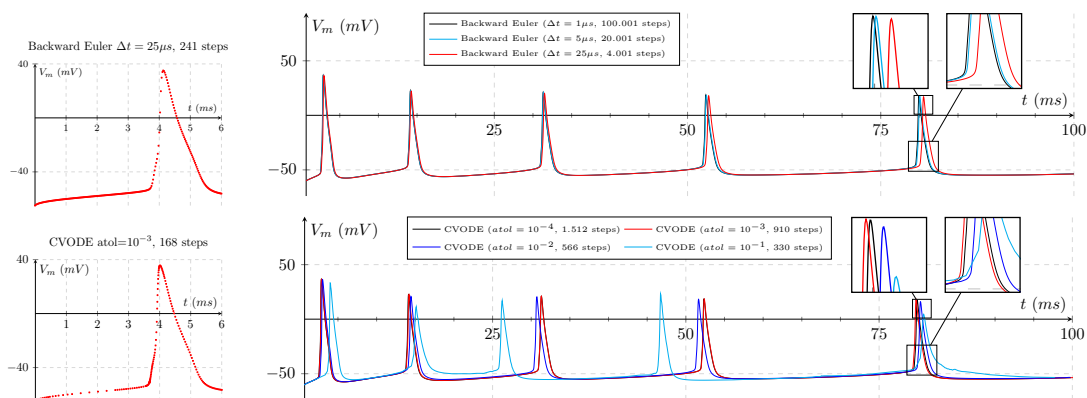


Figure 5.2 – Voltage potential at the soma and interpolation steps for a layer 5 pyramidal cell (L5_TTPC2_cADpyr232_1 (Human Brain Project, 2014)) during a 6ms (left) and 100ms simulation (right) of a 1.3mA continuous current injection, interpolated with Backward Euler (top) and CVODE (bottom) methods, respectively. The reference implementations are the Backward Euler method with $\Delta t = 1\mu s$ and CVODE with absolute tolerance 10^{-1} , presented in black and considered indistinguishable. The standard NEURON step size and tolerance values are $\Delta t = 25\mu s$ and 10^{-3} and are presented in red.

provides a larger variable step interval. Due to the reduced communication and computation, the removal of solution resets and backstepping, and larger stepping intervals, this method will be the default used in the asynchronous variable step simulations of networks of neurons.

5.4 Results

5.4.1 Numerical Accuracy

Backward Euler is an A-stable and L-stable method of order 1. The numerical accuracy of CVODE depends on the order of the Backward Differential Formula iteration and the user-provided tolerance. The underlying BDF is A-stable at order 1 or 2. At orders 3 to 5, it is not A-stable but *stiffly-stable*: the region of instability grows as the order increases from 3 to 5, and stability depends on the step size (Cohen & Hindmarsh, 1996).

We compare the numerical accuracy of both models by measuring the time difference of the main unit of interest in the activity of spiking neuron networks — the spiking time instants. Figure 5.2 presents the voltage trajectory and number of steps of a 1.3mA current clamp experiment for a single (6ms) and several (100ms) spikes of a layer 5 pyramidal cell. The default settings utilised in NEURON are plotted in red, with $\Delta t = 25\mu s$ and $atol = 10^{-3}$ for Euler and CVODE methods, respectively. The reference solutions are plotted in black, with $\Delta t = 1\mu s$ and $atol = 10^{-4}$, and their numerical difference is considered negligible.

Results on the single spike voltage trajectory (6ms) display a reduced step count and better adaptation to trajectory change when comparing CVODE to Euler method. The rationale

Chapter 5. Fully-Async. Fully-Implicit Variable-Order Variable-Timestep Simulation

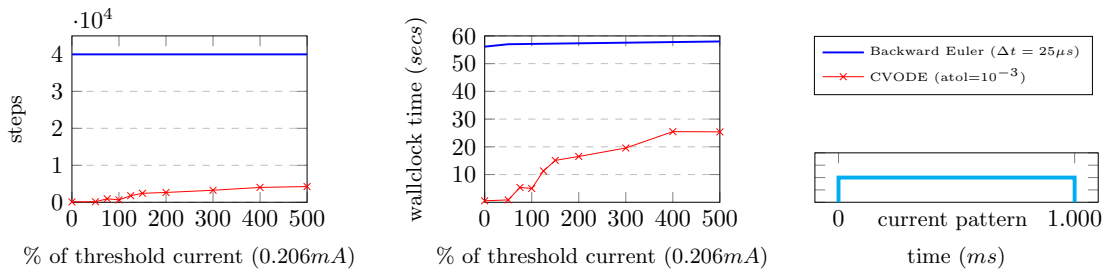


Figure 5.3 – Interpolation steps and runtime for the $1000ms$ simulation of the injection of a continuous constant current as a percentage of the threshold current ($0.206mA$), on a layer 5 pyramidal cell (L5_TTPC2_cADpyr232_1 (Human Brain Project, 2014)). Results presented for the Backward Euler with $\Delta t = 25\mu s$ and CVODE with $atol=10^{-3}$. Hardware specifications: Intel core i5 with 2×1.6 Ghz.

behind the better performance of adaptive stepping is that it is gradient sensitive and thus it follows the natural behaviour of a neuron: a cell spike requires small timesteps for higher precision, followed by interspike intervals or resting periods with little synaptic input therefore allowing large step sizes. CVODE displays less steps during long periods of low gradient (e.g. $1 - 2.5ms$), and greater number of steps for steep trajectories (the uprising trajectory of the spike) and sudden changes in gradient (the trajectory proximal to the peak voltage).

The $100ms$ simulation studies the impact and numerical accuracy of both interpolators to longer executions. Results display a phase shift in solution (measured as the time difference between peak voltage values of the reference and benchmark curves) that increases with the increase of the step size on the Euler methods. In practice, the timestep determines the fastest reaction time of the system. Thus, a large timestep will inevitably cause the system dynamics to be *slow*. The analysis of the performance of the CVODE tolerance values show that tolerance of 10^{-2} approximates the resolution of the default Euler step size of $25\mu s$, with a significant reduction of $7 \times$ in step count. At longer runs, the variable step demonstrated to be more precise, due to no accumulation of phase shift, with the maximum trajectory shift measured at approximately $1.1ms$. On the other hand, a tolerance value of 10^{-3} approximates closely the optimal solution with 40% less steps, and with a margin of error similar to its $5\mu s$ Euler counterpart for the period of $100ms$, while yielding $22 \times$ less interpolations.

On a longer execution, the measured phase shifting of the trajectory for a period of $1000ms$ of simulation (omitted for brevity) was of approximately $2ms$ for $\Delta t = 5\mu s$, and $6.5ms$ for $\Delta t = 25\mu s$, considered a large value in the timescale of neurons activity. The $100ms$ execution displayed demonstrated a phase shifting of approximately a tenth of the whole second simulation.

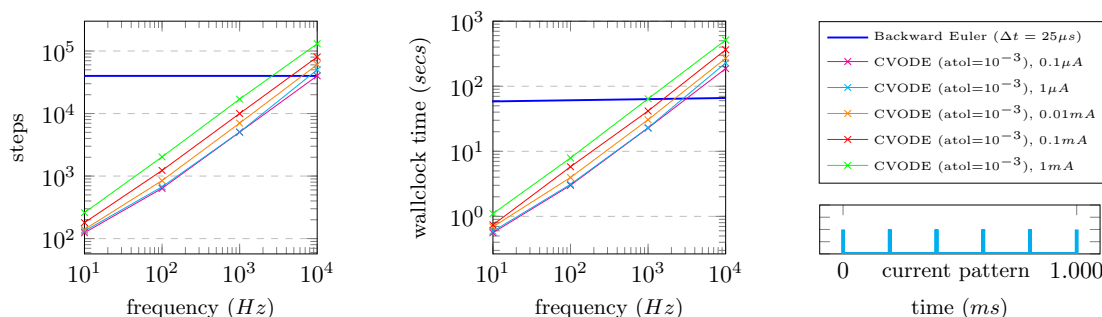


Figure 5.4 – Interpolation steps and runtime for the 1000ms simulation of injection of short $1\mu s$ current pulses of different amplitudes I at different frequencies, on a layer 5 pyramidal cell (L5_TTPC2_cADpyr232_1 (Human Brain Project, 2014)). Results presented for the Backward Euler with $\Delta t = 25\mu s$ and CVODE with $atol=10^{-3}$. Hardware specifications: Intel core i5 with 2×1.6 Ghz.

5.4.2 Performance Dependency on Solution Stiffness

We measured the response of both stepping methods with the NEURON default parameters to different levels of changes in spiking frequency. Changes in trajectory were enforced by injecting a continuous current of a given amplitude on a neuron during 1000ms. Current intensity is measured as a percentage of the **threshold current**, the minimum continuous current value that needs to be injected to force the neuron to spike. Performance was measured in terms of steps count and time to solution on an Intel i5 at 1.6 GHz. Results are presented in Figure 5.3, and demonstrate that high dynamics of the solution degrade the CVODE performance. This is justified by CVODE requiring smaller steps on high trajectory variations in order to respect the absolute tolerance value. For the range of tested scenarios, CVODE runs on less steps and shorter time to solution when compared to Backward Euler. For a neuron without any spikes, or equivalently for a current injection below the spiking threshold, it was measured a reduction of: (1) $434\times$ in step count and $98\times$ in runtime for injected currents below 50%; (2) $62\times$ step count and $11.6\times$ runtime for 100% of the threshold current; and (3) $9.4\times$ step count and $2.5\times$ runtime for 500% of the threshold current, a worst case scenario of little probability of occurrence.

5.4.3 Performance Dependency on State Discontinuities

We measured the effect of discontinuities on both methods by injecting several current pulses at a fixed frequency on a neuron soma, in the same Intel i5 compute architecture, mimicking synaptic events. The experiment results are displayed in Figure 5.4 and suggest that the performance depends on the trajectory change from each discontinuity, i.e. the amplitude of the current injected. This is due to the increase of the voltage trajectory being dependent on the amount of current injected. The larger the voltage increase, the larger the change in trajectory gradient, thus the more interpolation steps are required. Furthermore, and as

Chapter 5. Fully-Async. Fully-Implicit Variable-Order Variable-Timestep Simulation

expected, results demonstrate that the number of discontinuities plays a major role on the CVODE performance. CVODE is shown to deliver a reduction of steps in the order of $153-322\times$ for a frequency of 10 discontinuities per second. The equilibrium between Euler and CVODE method lies in the interval between $10^{3.1} - 10^{3.2} Hz$ for the current values of $1 mA$ to $0.1 \mu A$.

The runtime demonstrates a similar dependency on the injected current, yielding a speedup of $51\times$ for $10 Hz$ decreasing linearly up to the speedup equilibrium value at $1000 Hz$ for the strongest current. For the lightest current injection, a speedup of $100\times$ is visible for the $10 Hz$ discontinuity rate, decreasing to an Euler matching value at circa 1600 events per second ($10^{3.2} Hz$). Although this exercise does not represent the stochastic pattern of spikes arrival on real neurons, typically described by a Poisson distribution with a long tail, it provides an estimation of the CVODE speedup allowance. With that in mind, the following section measures the discontinuity rates on a simulation of a laboratory experiment.

5.4.4 Simulation of a Laboratory Experiment

We tested the suitability of variable step methods to our problem by measuring the spiking activity of a simulation of 7.5 secs of electrical activity mimicking a laboratory experiment. The experimental set-up performs a fixed step simulation of the spontaneous activity of 219.247 neurons during tonic depolarization. The network exhibits spontaneous slow oscillatory population bursts, initiated in layer 5 (L5), spreading down to L6, and then up to L4 and L2/3 with secondary bursts spreading back to L6. For further details, refer to section *Simulating Spontaneous Activity* in (Markram et al., 2015).

A representative distribution of discontinuity events for three groups of neurons — organized by highest 1%, median 1%, and lowest 1% number of discontinuities — is displayed in Figure 5.5. The simulation incurred a total of circa 155 million events, with the following distribution: (a) top 1% of neurons, between 3040 and 6146 events in 7.5 secs, or 405-820 Hz; (b) median 1%, from 541 to 558 events (72-74.4 Hz); and (c) bottom 1%: less than 100 events ($\leq 10 Hz$). The average number of events was of 707 events for the 7.5 secs of simulation, or equivalently, 94 Hz, significantly below the $1000 Hz$ threshold discussed in the previous section. Moreover, the results on the distributions of time interval between discontinuities, plotted in red on the right, display large periods of *silence* between events arrival in the median and bottom use cases, but not on the top, suggesting the suitability of adaptive stepping to most (but not all) neurons in the population.

5.5 Benchmark

5.5.1 Implementation

Our methods were implemented on the compute kernel of the NEURON scientific application (Kumbhar et al., 2019). The mathematical specifications of biological mechanisms were

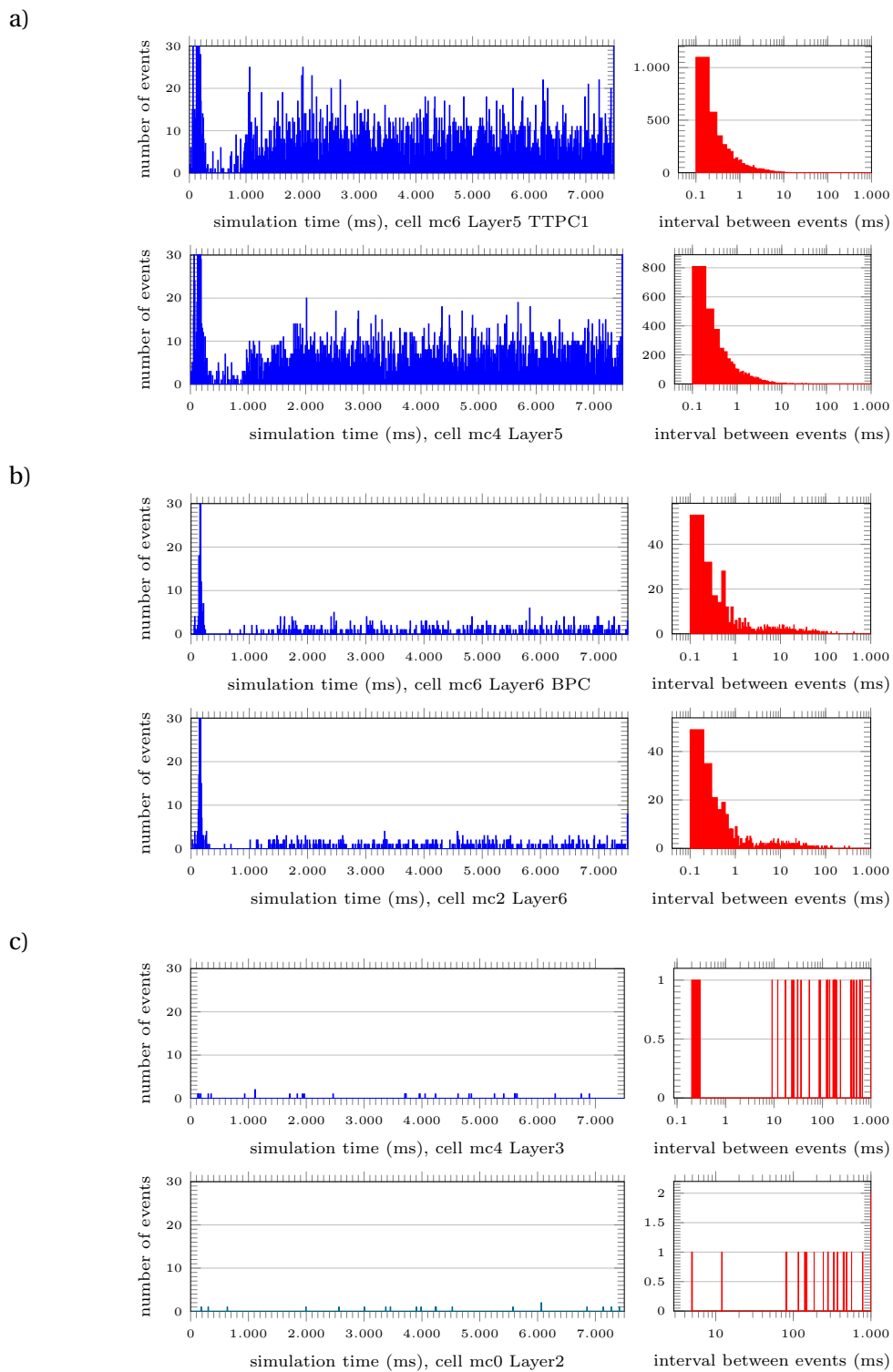


Figure 5.5 – Number of discontinuity events (incoming spike currents, bin size 0.25ms, left) and distribution of events time difference (bin size 0.1ms, right) throughout 7.5 secs of simulation, for a sample neuron collected from the **a)** top 1% neurons, receiving between 3040 and 6146 events; **b)** median 1%, from 541 to 558 events; and **c)** bottom 1% of neurons receiving less than 100 events. Total events count: circa 155 Million; mean per neuron: 707.

Chapter 5. Fully-Async. Fully-Implicit Variable-Order Variable-Timestep Simulation

provided by the NEURON modelling language (NMODL). Code changes to support SIMD capabilities and methods for variable step interpolation were added to the MODL-to-C code generator (Hines & Carnevale, 2000).

Single Instruction Multiple Data (SIMD) capabilities were implemented on all fixed and variable timestep methods discussed hereafter. Communication, synchronisation control objects, memory allocation, threading, distributed execution and parallelism were implemented with HPX. Implementation details have been covered in Chapter 4, and will be omitted for brevity.

Resolution of variable timestep interpolation was implemented via the CVODE library (Cohen & Hindmarsh, 1996), a C implementation of the VODE algorithm (Brown et al., 1989), part of SUNDIALS (Suite of nonlinear and differential/algebraic equation solvers (Hindmarsh et al., 2005)).

5.5.2 Use Case

We simulate one second of biological activity of a digital reconstruction of a morphologically detailed neural network extracted from the model of Markram et al. (Markram et al., 2015). Neuron models include 23 distinct biological mechanism types modelled by 44 ODEs, and highly heterogeneous neuron morphologies. Each neuron requires a storage of 4-10 MB for its state, times q for a q -order interpolation, plus circa 4 MB for intermediate storage of data structures supporting the fully-asynchronous execution model. Our CVODE solver is configured to utilise the default maximum BDF order value of 5.

On the set-up of the simulation test bench, it is relevant to mention that neuronal activity is highly dependent on the mammal specie, brain region and momentary activity, among other factors. Moreover, the network behaviour simulated must approximate a real use case, as spiking activity affect heavily the performance of variable step methods, as detailed in Section 5.4.3. An analysis of a single simulation combining several brain dynamics would be of little interest in the context of efficiency analysis, due to all free variables that affect performance. Therefore, our test bench simulates and studies the efficiency of five different brain dynamics described in the literature:

1. a model of *quiet dynamics* with a mean spiking rate of 0.25 Hz per neuron, representing neurons almost at rest and/or with little activity. This model is of high relevance, as it provides an upper bound of the runtime of circa 90% of neurons in the brain during regular activity — see Table 1 in Shoham et al. (Shoham et al., 2006) for evidence of silence and highly sparse activity among neurons; supported by Kerr et al. (Kerr et al., 2005) for estimates of rat's neocortex spiking rate at circa 0.1 Hz due to sparsity of activity; and the estimation of 0.16 Hz by Lennie et al. (Lennie, 2003) based on brain energy levels;
2. a model of *slow dynamics* at 1.5Hz, representing the lower bound of active neurons, described next;

3. a model of *moderate dynamics* with a spiking rate of 6.5 Hz, an approximation of the irregular regime of slow oscillations displayed by the Brunel network (Brunel, 2000); also an upper limit to the 0.005-5Hz spiking rate of the rate frontal cortex (Watson et al., 2016); and a loose approximation of the to visual cortex of the cats in the 3-4 Hz interval (Baddeley et al., 1997).
4. a model of *fast dynamics* of 38 Hz, in the range of 30-40 Hz characterizing cortical and thalamic neuronal activity during periods of high vigilance (Steriade et al., 1998); and the regular regime of the theoretical model of inhibition-dominated model of the Brunel Network (Brunel, 2000); and
5. a model of *burst dynamics* at 55.8 Hz, typically a by-product of depolarizing current injections, similar to the first instants of simulation in Figure 5.5; and representative of the fast spiking regime of the inhibition-dominated irregular dynamics of the Brunel Network (Brunel, 2000).

The following benchmarks measure the scaling of our methods by simulating quasi-homogeneous activity across neurons on densely connected neural networks. This set-up is mostly favourable to fixed step and BSP-based methods, therefore the results presented next are a lower bound of possible acceleration. Neurons activity is triggered by a constant current injection in all neurons throughout the whole duration of the simulation, strong enough to approximate the spiking rate of the network to the regimes described.

For complete coverage of the topic, we include the following state-of-the-art solvers for simple neuron models (labelled 1a to 1c, and restrained to linear ODEs with uncorrelated states) and complex models (2a-2c), both detailed previously in Section 5.3.1:

- 1a)** the *cnexp* fixed step solver in NEURON, with added SIMD, providing an intercalated resolution of current and states as linear equations, with an analytical resolution of the first order ODE describing state variables;
- 1b)** the *Euler* solver in NEURON, with added SIMD, resolving the current-states dependency with an Euler method with staggered voltage-states timestepping, and as a linear equations; a model computationally less expensive than the previous due to no exponential and division operator;
- 1c)** the same *Euler* method on a fully-asynchronous parallel execution model, presented in Chapter 4, and illustrated in diagram c in Figure 1.2;
- 2a)** the BSP fixed step *derivimplicit* solver available in NEURON, with added SIMD, with interleaved-timestep resolution of current as a linear equation, and implicit resolution of individual mechanism state ODEs;
- 2b)** the BSP variable step method in NEURON with added SIMD and a collective communication barrier (diagram b in Figure 1.2); and

Chapter 5. Fully-Async. Fully-Implicit Variable-Order Variable-Timestep Simulation

2c) the SIMD-enabled fully-asynchronous protocol with variable timestepping introduced in this chapter (diagram d in Figure 1.2).

The following biological constraints were taken into account to guarantee that variable-step runtimes represent a biologically plausible use case: we verified that there were no continuous periods of silence in the network, and no collective synchrony of spiking or voltage trajectory across neurons, that would promote additional efficiency in variable timestep methods. Our input data is retrieved from neurons in layers 4 and 5 of the brain, typically represented by the longest dendritic trees, therefore representing a worst case scenario in terms of number of pre-synaptic (dependency) neurons. Finally, the number of synapses as AMPA and GABA receptors was measured, and are characterized by a mean of 2289.7 and 4418.8 and a std. dev. of 1284.9 and 3200.4 per neuron, below the counts described in the literature, but inline with the reference digital reconstruction of the rodent brain detailed in Section 5.4.4.

Execution times were collected on 64 Cray XE6 compute nodes, powered by an AMD Opteron 6380 with 16 cores at 2.5 GHz, 64 GB of RAM and 256-bit floating point units. Efficient point-to-point communication and remote direct access memory is provided with specialized Infiniband network hardware, interfaced via the photon API library (Kissel & Swamy, 2016). To study the dependency of the algorithm on the input size and synaptic connectivity, we tested our methods in neural networks ranging from 1024 to 65536 neurons, a scale that approximates two columns in the rodent neocortex, and the maximum allowed due to memory requirements of the BDF solver of order 5 we used. Neurons were equally distributed in a round robin fashion across compute nodes. Due to the long simulation time required, runtimes for moderate, fast and burst dynamics were extrapolated from an execution of 250, 100 and 100ms, respectively.

The benchmark results are presented in Figure 5.6. The FAP variable step method (2c•) is presented alongside two variants — labelled 2c• and 2c• — that group and deliver instantly the discontinuity events within an interval equivalent to the timestep $\Delta t/2$ of the interleaved fixed timestep method, and the Δt of the regular fixed step methods, respectively. This approach yields a level of reduced precision in the delivery of events — similar to fixed step methods — however maintaining the same high variable-order variable-step accuracy during periods of activity without discontinuities, with the main advantage of reducing significantly the number in IVP resets. CVODE-based executions are displayed for an absolute tolerance (atol) of 10^{-3} , a value equal to the default value in the NEURON simulator. Executions with an absolute tolerance of 10^{-2} yielded a reduction in runtime of 5%-8%, and were omitted for brevity. The performance analysis for fixed-step simple model solvers (1a-1c) was covered in depth in the previous Chapter 4, and is omitted for brevity.

5.5.3 Fixed- vs Variable-Timestep Interpolators

Fixed step methods do not yield significantly-different execution times across different spiking regimes. This is due to the homogeneous computation of neuron state updates throughout time, and the light computation attached to synaptic events and collective communication

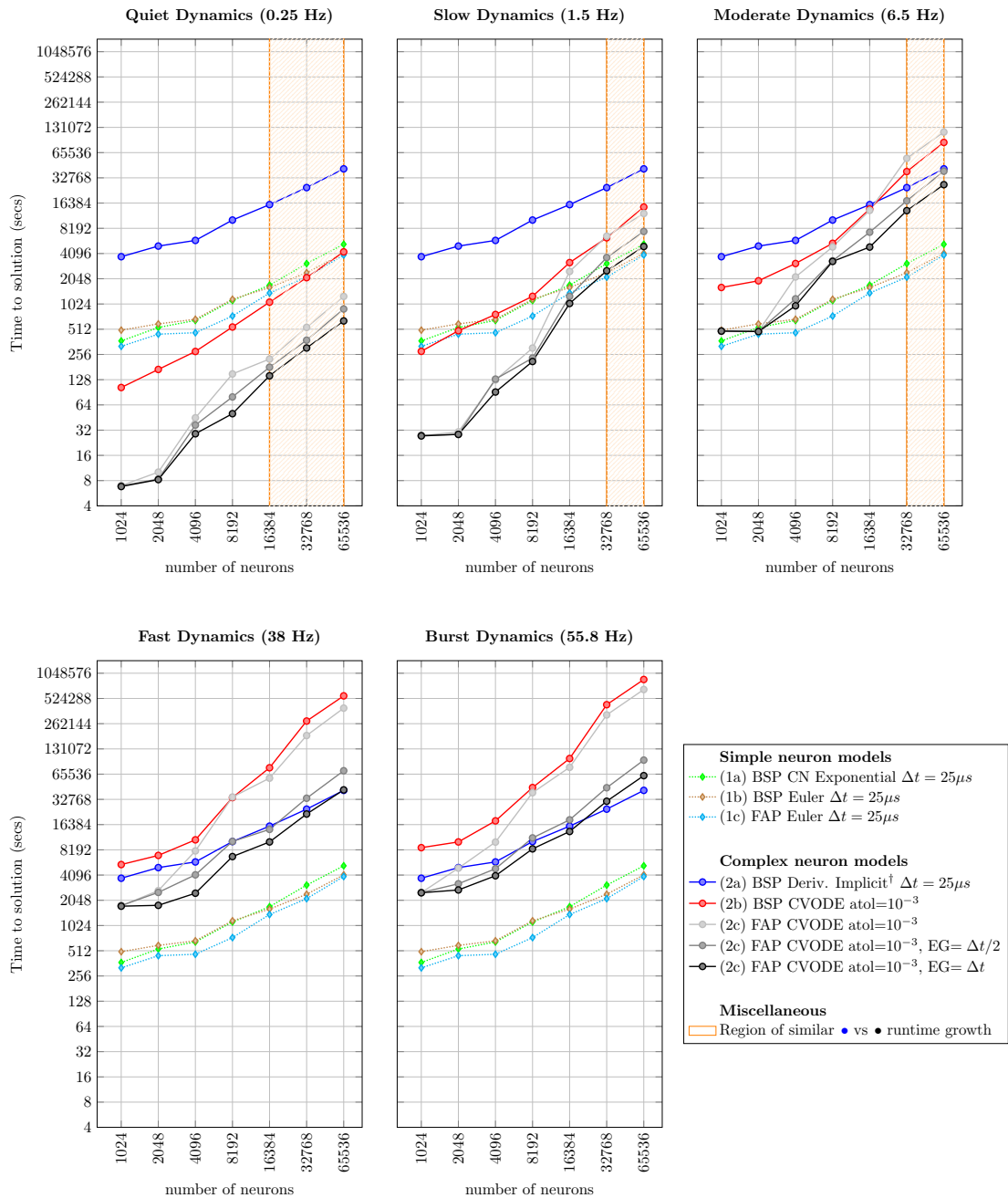


Figure 5.6 – Runtime for the simulation of one second of biological activity of five neuron networks described by distinct spiking rate dynamics. Results measured for increasing input sizes on a network of 64 Cray XE6 nodes, with an AMD Opteron 6380 with 16 cores at 2.5 Ghz. **Key:** BSP: Bulk Synchronous Parallel; FAP: Fully-Asynchronous Parallel; atol: absolute tolerance; EG: event grouping interval; [†]: able to solve non-linear ODEs implicitly, and unable to solve correlated mechanism states implicitly.

not yielding a substantial increase of runtime. The difference in execution times measured across the five regimes was of about 2%, which we consider negligible. On the other hand, as expected, variable step executions are penalized on regimes with high discontinuity rates. It is noticeable that the runtimes of fixed- and variable-step solvers approximate as we increase the spiking rate, i.e. the increase of runtimes with the input size is steeper for variable timestep (2b• and 2c•••) compared to fixed timestep methods (2a•). This is due to discontinuities in variable-step being delivered throughout a continuous time line, compared to the discrete delivery instants of the fixed-step methods — therefore increase the number of interpolation steps; and the iterative model of the variable timestep reinitializing the state computation with small step sizes on each IVP reset, compared to the constant-sized step of fixed step methods. A remarkable performance is visible on the quiet dynamics use case, where our fully-implicit ODE solver of complex models (with Newton iterations), still runs faster than the simple solver resolving only a system of linear equations. The underlying rationale is that — despite the inherent computation cost of Newton iterations in the variable step methods — the low level of discontinuities allow for very long steps, that surpass the simulation throughput of simple solvers running on fixed step methods. The measured speedup of our reference method (2c•) compared to the reference fixed step method (2a•) was of 544-65× across input sizes for the quiet dynamics, down to 7.7-1.8× to the moderate dynamics. The fast dynamics presented a speedup of twofold for the dataset of 1024 neurons, and a similar runtime for the 66K neurons. The burst dynamics, although of very unlikely probability of occurrence, demonstrated an acceleration of 1.5× for 1024 neurons and a deceleration of 1.5× for 66K neurons.

5.5.4 Variable Step Event Grouping

On the analysis of the performance of the CVODE with grouping of events within half fixed timestep (2c•), when applied to the largest dataset tested, the previous acceleration was reduced to 47× for quiet, 4.4× for slow, and 1.2× for moderate dynamics, with an inferior performance on the remaining regimes. A further reduction of speedup to 33× for quiet and 1.9× for slow dynamics was noticeable on the CVODE implementation without events grouping (2c•), with lower performance for the remaining spike regimes. Although being more precise and solving correlated states implicitly, this method runs slower than the reference implicit fixed step method 2a• in the use cases characterized by a high number of neurons and/or strong network activity. This goes in line with the conclusions in Section 5.4.4, confirming that performance is activity dependent, and the achievable speedup depends on the network connectivity.

The speedup introduced across FAP CVODE variants (2c•••) increases with the amount of discontinuities in the system — correlated to high network activity or size — as the efficiency of the event grouping method is related to the amount of events in the same grouping interval that are delivered at once. No difference in runtime was measured for the smallest dataset tested due to the high sparsity of synaptic events in small networks.

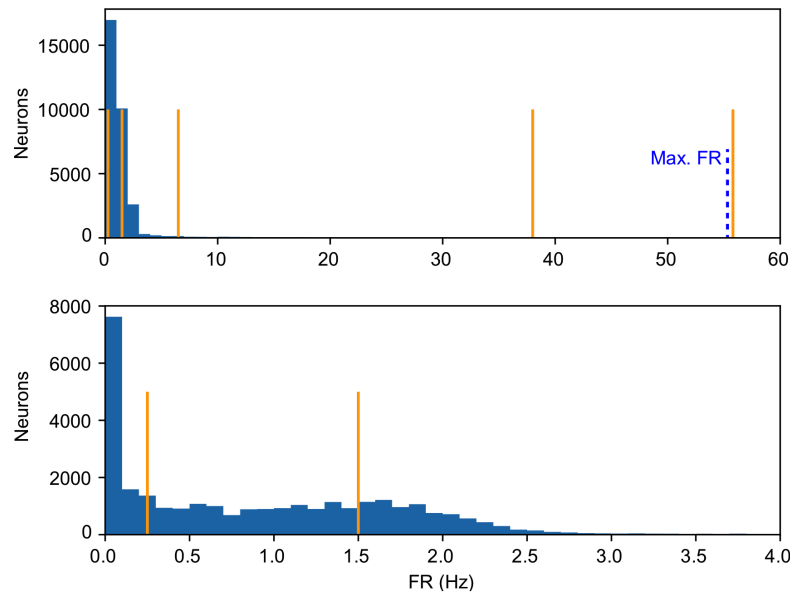



Figure 5.7 – Distribution of neuron counts by spiking rates collected from the central minicolumn (31346 neurons) of the network utilized in the simulation of the laboratory experiment described in Section 5.4.4, in the interval 2-4 secs. Orange bars represent the frequencies 0.25 Hz, 1.5 Hz, 6.5 Hz, 38 Hz and 55 Hz that describe the boundaries of the 5 spiking regimes analysed. Top: a bin accounts for 1Hz. *Max. FR* is the maximum firing rate measured; Bottom: zoomed dataset with a cut off at 0-40 Hz. A bin accounts for 0.1 Hz.

5.5.5 Fully-Asynchronous vs Bulk-Synchronous Execution Models

Our next analysis focuses on the performance difference between the BSP and FAP execution models. Results show that the runtimes of both implementations approximate with the increase of the input. This is visible by comparing the fixed step trajectories 1b♦ and 1c♦, and the variable step trajectories 2b• and 2c•. For small network sizes, the difference in runtime is noticeably few orders of magnitude higher than for larger network sizes. On large models, the runtimes are similar. This property was demonstrated in Chapter 4. In brief, an increase of network leads to a higher number of network connectivity, therefore reducing the maximum allowed stepping interval per neuron, and approximating it to the minimum communication delay utilised in the BSP methods. On fixed step methods, it is noticeable a similar runtime on large (66K) networks of neurons, as timesteps are computationally homogeneous. On variable step methods, similar runtimes are only noticeable when significant network activity is present (moderate, fast and burst dynamics), as little network activity leads to few discontinuities, allowing stepping intervals to be modelled in a reduced number of variable steps.

5.5.6 Runtime Dependency on Input Size and Spike Activity

It is known that, on simulations of small networks, variable-timestep methods yield a significant acceleration in time to solution compared to fixed timestep methods (Lytton & Hines,

2005). The rationale is that the small number of neurons in the network leads to a reduced number of synapses per neuron, thus less discontinuities. However, the connectivity in larger networks reaches up to 10 thousand synapses per neuron, with the number of discontinuities being related also to the overall network activity. The question lies now on which conditions are required for similar computation complexity in both interpolators. To that extent, we measured the regions of similar runtime growth for the reference fixed step (2b•) and our variable step methods (2c•). The region is labelled as  in Figure 5.6.

As expected, fixed step methods yield a quasi-linear runtime growth with the increase of the input size, and are independent of the spiking regime, due to the almost ideal scaling of the algorithm in the BSP model. On the other hand, the runtime of variable timestep methods — dependent on the number of discontinuities — demonstrates a rapidly increasing growth with the input size outside the region of similar growth, and almost linearly inside. Moreover, as it depends on the network activity, the lower limit of the region increases with the spiking rate, and is delimited at 16.4K, 32.8K and 32.8K neurons or more for the quiet, slow and moderate dynamics, while not visible in the fast and burst dynamics. In the three spiking regimes where such region exists, the similar growth in both approaches provides a confidence of the scaling capabilities of our methods in larger network models. This is of high importance as it provides an estimation of runtime upper bound in simulations combining neurons with heterogeneous spiking rates, as discussed next.

5.5.7 Overall Runtime Speedup Estimation

To conclude our analysis, we computed an estimation of the performance acceleration on a simulation combining several spiking regimes. For that purpose, we measured the the distribution of neuron spike rates and neurons per spiking regime, following the laboratory experiment simulation described in Section 5.4.4. Estimations were collected from the central minicolumn (31346 neurons) of the 219K neurons network, to avoid boundary-effects that would improve results due to neurons placed on the edges, with reduced connectivity. The counts relate to the 2-4s simulation time interval, to exclude the initial artificial synaptic burst due to the current injection. The results are presented in Figure 5.7. The measured percentage of neurons on each regime is 31.43%, 38.44%, 27.02%, 3.10% and 0.01%, relating to 68.9K, 84.3K, 59.2K, 6.8K and 22 neurons. Following the runtimes described in Section 5.5.3, the speedup range for the interval of 1024-66K neurons when comparing our methods with the state-of-the-art solver for complex models (2a•) are estimated as: 224.5-11.9x for the variable step method with precise event delivery (2c•); 225.1-17.1x for the similar implementation with delivery of events within the next half timestep (2c•); and 228.5-24.6x for the use case with full-timestep event group delivery (2c•). As a side note, the percentage of neurons in the quiet dynamics regime does not agree with the 90% described in the literature. We believe this is due to the reduced size of the network, and that simulations of larger networks and complete brain models include a larger portion of neurons in this regime. Therefore, we assume these results to be a lower bound estimate of possible acceleration.

Moreover, since the quiet, slow and moderate dynamics regimes weight over 95% in the runtime calculation, and as for datasets above 32.8K the reference vs benchmark runtimes have a similar runtime growth in those regimes, we believe the overall runtime for larger circuits do not yield a significant reduction in the speedup values presented, and that the scaling properties are almost fully-preserved on larger networks.

5.6 Discussion

This chapter presented a strategy for the fully-asynchronous distributed simulation of detailed neuron models with variable-order variable-timestep interpolation. We detailed state-of-the-art approaches based on the Bulk Synchronous Parallel execution model (BSP), their limitations on the numerical resolution of complex neuron models, and computation load imbalance at synchronization barriers in variable-step simulations. We discussed the problem of synaptic exchange and synchronization of networks of neurons in speculative variable-step executions, and the inherent issue of a cascade chain of states reset and reversal of false synaptic events. To overcome this issue, we proposed an alternative non-speculative scheduled execution model on a distributed network of compute nodes. The approach follows a novel Fully-Asynchronous Parallel (FAP) execution model (yielding asynchronous computation, communication and synchronisation), that relies on the individual stepping of neurons based on the time instant of their synaptic connectivities, avoiding backstepping and allowing for stepping intervals beyond the BSP-based synchronization interval. Step lengths are maximised by a scheduler that tracks neurons' time advancement and dynamically allocates compute resources to the earliest neurons in time.

We performed an analysis of numerical accuracy and demonstrated better precision and less interpolation steps of the methods presented compared to the reference fixed-step implicit solver in the NEURON scientific application. An analysis of variable-step performance based on step count and time to solution was performed on three experimental set-ups. (1) A continuous current injection causing fast gradient changes in state on a single neuron demonstrated a reduced step count and runtime, and low dependency of performance on the stiffness of solution. (2) An injection of a sequence of current pulses simulating network activity demonstrated a high dependency of step count and runtime on incoming synaptic activity, which cause a solution discontinuity and a reset of solution state. We demonstrated a reduced step count for an incoming spike current rates between 1000 and 1600 Hz, depending on the current value. (3) A digital reconstruction of a laboratory experiment on a network of 219 thousand neurons measured the relevance of our result to a real use case. Results displayed a highly heterogeneous distribution of discontinuity rates across neurons, demonstrating the suitability of our methods to the problem domain.

A proof of concept simulator was implemented on the compute kernel of the NEURON scientific application. Distributed asynchrony, global memory addressing space, remote procedure call, threading, synchronization and communication methods were replaced and

Chapter 5. Fully-Async. Fully-Implicit Variable-Order Variable-Timestep Simulation

implemented by the HPX runtime system (Sterling et al., 2014). We analysed and benchmarked five spiking regimes that describe distinct patterns of brain activity in mammals, and six state-of-the-art solvers for simple and complex neuron models. Benchmark results on a network of 1024-65536 neurons demonstrate an overall reduction in runtime in the order of 224.5-11.9x for the most accurate method with precise delivery of events in a continuous timeline, and 225.1-17.1x and 228.5-24.6x for two optimized variants with half-step and full-step grouped delivery of events. We demonstrated that performance is activity-dependent and that almost ideal scaling is possible, as over 95% of neurons in a biologically inspired neuron network — from a simulated laboratory experiment — fall in spiking regimes with guaranteed preservation of the speedup achieved.

6 General Discussion and Conclusions

6.1 Summary

The general object of this thesis was to introduce new computational models that underlie an efficient numerical resolution of the simulation of highly-heterogeneous neuron models via asynchronous runtime systems on parallel-distributed compute resources.

Chapter 1 introduced the scope of our research, the problem description, our motivation, a literature review, a review of state-of-the-art approaches, and an analysis of runtime systems. We described our mathematical formulation, based on an extended Hodgkin-Huxley model (Hodgkin & Huxley, 1952), and our use case, extracted from a previously published digital reconstruction of the young rodent brain (Markram et al., 2015). We analysed our dataset and demonstrated high heterogeneity in neuron topological structure, ionic current activity, synaptic delays, and thus highly-heterogeneous computational complexity across neurons in the dataset. We discussed new asynchronous runtime systems and methodologies that introduce new possibilities in the simulation of such detailed neuron models.

6.1.1 Micro-Parallelism of Detailed Neuron Models

Chapter 2 developed the field of distributed asynchronous micro-parallelism exposed from the decomposition of neuron morphological trees into interconnected subtrees. The work detailed numerical dependencies across compartments, and presented a data decomposition method yielding load-balanced multi-core execution of neuron subtrees, with Single Instruction Multiple Data (SIMD) processing biological mechanisms on the compartments of each subtree. Two major scaling limitations were identified and discussed: (1) the dependency of parallelism on the depth of the neuron topology, and (2) the competitive speedup provided by vector and multi-core acceleration on single compute nodes. Benchmarks of the simulation of three distinct neurons on four compute architectures (Intel KNL, Intel i5, Intel Xeon and AMD Opteron) demonstrated a significant speedup compared to the branched Single Instruction Single Data (SISD) implementation in NEURON, with the speedup decreasing

with the increase of compute cores. The application of the methods to a distributed network of 128 Cray XE6 compute nodes simulating medium-sized networks of up to 4000 neurons, demonstrated a time to solution that approximates the theoretical strong scaling limit, with a twofold speedup compared to its non-branched SIMD counterpart due to finer-grained parallelism and improved task scheduling.

In **Chapter 3**, we introduced a novel approach for the micro-parallelism of neuron state updates, by extracting flow dependencies, concurrent variable updates, and independent variables from ODE specifications. The methods presented accelerate the simulation of individual neurons or sets of neurons by exposing a computation graph of interdependent Ordinary Differential Equations (ODEs). We detailed a method that automates the parsing of ODEs specified by NEURON's domain specific language. We applied the parsing method to a neuron model of 23 biological mechanisms modelled by 44 ODEs, demonstrating up to 19 parallel mechanism kernels. Further kernel parallelism was provided by a method that extracts a load-balanced embarrassingly parallel execution model of similar and independent ODE instances, providing multi-core- and SIMD-parallel acceleration of individual mechanisms. A single neuron benchmark was performed and demonstrated a speedup very proximal to the theoretical limit of the state update function, on the aforementioned four compute architectures. A distributed execution on 128 Cray XE6 compute nodes, simulating networks ranging from 128-4096 neurons yielded an acceleration that approximated ideal strong scaling limit with the increase of the input size. A speedup of over twofold was measured when comparing graph- versus non-graph parallel implementations, due to dynamic workload balancing of finer-grained compute tasks.

6.1.2 Fully-Asynchronous Execution Model

Chapter 4 introduced the fully-asynchronous parallel execution model, and the *exhaustive yet not speculative* stepping protocol, that replaced collective synchronization of neurons with active point-to-point messaging that provides time stepping notifications to synaptic connectivities. We detailed a method for fully linearizing neurons memory representation in memory, yielding a reduction of memory read operations and improved cache efficiency. A method for neuron scheduling was provided, that further improves performance by dynamically allocating available compute resources to the earliest neuron in time, leading to better cache locality due to a higher number of steps performed in cache. The method delivered up to a 100× increase in the number of steps on a single stepping iteration, decreasing with the number of neurons in the network, due to the implicit shorter synaptic delays of larger networks. Complementarily, we implemented a communication reduce from neuron-to-neuron to neuron-to-locality, and demonstrated its efficiency in terms of communication count and time to solution compared to the non-reduced counterpart. A benchmark measuring the overall speedup followed, and displayed a reduction of 25%-65% in time to solution across the four architectures tested for a network of 16 to 2048 neurons, and a reduction of circa 15%-40% on 32 Cray XE6 nodes simulating 512 to 32768 neurons.

The fully-asynchronous execution model was further developed in **Chapter 5**, where we studied the application of variable-order variable-timestep interpolation in the resolution of complex neuron models that define use cases such as synaptic plasticity and learning. We tested and demonstrated higher numerical accuracy of adaptive-step methods compared to fixed-step methods. We measured the efficiency in terms of number of interpolation steps and runtime, and we showed that the performance of variable-step methods is highly-dependent on synaptic currents (discontinuities), and little dependent on the solution trajectory or stiffness. These insights were verified against a simulated reconstruction of a laboratory experiment and demonstrated high heterogeneity in discontinuities across neurons, and the general suitability of variable-step methods to the problem domain. On the context of distributed asynchronous variable-step executions, we identified speculative stepping with synaptic spiking as the main issue in distributed variable-timestep interpolation methods for network of neurons. To overcome the limitation, we introduced a new method that — contrarily to existing methods — yields no solution discontinuities, no backstepping of solution for missed synaptic events, and no BSP-based collective synchronization.

We described and benchmarked five networks with distinct spike frequencies that characterize different regimes of brain activity in mammals, and compared our methods with six state-of-the-art solvers, on a network of 64 Cray XE6 compute nodes, and increasing input ranging from 1024 to 65536 neurons. Results demonstrated a speedup of 544-65 \times against the state-of-the-art fixed-step counterpart, for the quiet spiking regime representing the majority of neurons in regular brain activity. The speedup was shown to decrease with an increase of network activity due to the higher number of events implicit, and a similar runtime was measured at a spike frequency close to a moderate spike regime of 6.8 Hz. A speedup estimation applied to the same laboratory experiment, presented a possibility of an acceleration in the order of 224.5-11.9 \times for the most precise variable timestep method described, and 225.1-17.1 \times and 228.5-24.6 \times for two detailed optimized variants that group and deliver discontinuities within half- and full-step fixed-timestep intervals, with a small loss in precision. Finally, we showed that over 95% of the input dataset is defined by a quiet to a moderate spiking regime that yields a linear speedup dependency on the input size, demonstrating good scaling properties of the asynchronous variable-step methods to biologically inspired use cases.

6.2 Contributions

This thesis advanced the state of the art in simulation and distributed parallel executions, and provided insights on the future directions of neural simulations. We provided methods for the acceleration of simulations via modern asynchronous runtime systems on the use case of simulation of morphologically detailed neural networks, and improved state-of-the-art methods in scheduling and compute heterogeneity in a distributed memory space.

In the study of small- and medium-sized networks, ranging from a single to few thousand neurons, our scientific advancements pushed micro-parallelism by exploring increased strong

Chapter 6. General Discussion and Conclusions

scaling parallelism via graph-parallelism from variable dependencies across systems of ODEs, and via branch-parallelism from dependencies across topological trees of individual neurons. We showed that:

- The Bulk Synchronous Parallel runtime commonly used on the resolution of such complex neuron morphologies, is inadequate in the resolution of small datasets due to computational load imbalance;
- State-of-the-art simulation methods provide insufficient compute granularity. An increase of granularity is possible with micro-parallelism;
- Micro-parallelism techniques allow for the decomposition of a problem of *any* large size, down to a workflow of interconnected smaller kernels. The sequence of branch, compartment, mechanism, and mechanism instance decomposition allows for an extremely large number of micro-kernels that *easily* saturate all parallel units in modern compute architectures;
- Distributed micro-parallelism requires a coherent decomposition and cross-linkage of neuron kernels across distributed memory regions. This is a task of direct implementation with the global memory space underlying modern asynchronous runtime systems, yet a strenuous task for BSP-based execution models;
- Distributed threading, remote procedure calls, placeholders and execution gates with support to a global memory address space allow for an efficient asynchronous kernel synchronization and message-passing execution. This feature removes BSP-like synchronization across compute units, yielding a major performance increase;
- On-the-fly decomposition and aggregation of underlying kernels — at neuron, subtree, mechanism, and mechanism instance levels — allows for the fitting of the complexity of the problem to the hardware specifications. However, the computational complexity of each kernel is guided by a constant value whose fine-tuning is not trivial;
- The limit in acceleration provided by micro-parallelism techniques are limited by (1) Amdal's law due to serial processes in graph-parallelism methods; and (2) depth of neuronal topology in branch-parallel workflows;
- Ideal strong scaling of branch-parallelism of individual neurons is not possible as multi-core and vector processing units provide competitive acceleration efforts, due to a trade-off between SIMD acceleration from kernel size (related to the deficient usage of the full register file width), and multi-core acceleration from the number of kernels (due to the overhead of managing a very high number of small compute kernels), with such limitation being most prominent on multi-core executions beyond 16 cores;
- Contrarily to the common belief that micro-parallelism is mostly necessary for single-neuron simulations, the techniques are also relevant to achieve ideal strong scaling of

medium-sized networks on distributed executions with 4 or more neurons per compute node. This is due to the overlap of computation and communication, and better load balancing due to finer-grained parallelism.

The data decomposition of the dataset in distributed load-balanced kernels allowed for the detachment of the problem representation from the hardware specifications, and introduced — to our knowledge — the first implementation of a distributed, multi-core, SIMD-enabled neuron simulator that fine-tunes the data memory layout to the host architecture on a distributed network of compute nodes. The overlap of asynchronous computation and communication maximises the usage of compute units, when computation is available. These properties demonstrate that **asynchronous runtime systems are suitable for the efficient simulation of highly-heterogeneous data representations on highly-heterogeneous compute architectures**, and due to the quasi-infinitesimal number of kernels that can be generated by the decomposition methods, **the computational complexity of detailed neuron models can be fine-tuned to *any* host architecture**.

On the study of the activity of medium and large networks of neurons, our work demonstrated that the synaptic time-dependency across neurons enforces a synchronization barrier in the simulation that delimits the performance of BSP-based executions. To that extent, we introduced the fully-asynchronous execution model, characterized by three axes of asynchrony:

1. Asynchronous computation, defined by the independent stepping of neuron ODEs in a distributed multi-core compute environment;
2. Asynchronous communication, via non-blocking point-to-point communication and remote procedure calls between neurons;
3. Asynchronous synchronization, enabled by the removal of synchronization barriers and collective communication operations, replaced by:
 - (a) Non-speculative stepping of ODEs based on pre-synaptic connectivity delays; and
 - (b) Active stepping notification to post-synaptic connections of neurons.

Our work showed that:

- A cache-level acceleration of individual neurons is possible with an asynchronous execution. Unlike micro-parallelism efforts that accelerate solution in the strong scaling axis by increasing the number of compute units for a fixed-sized problem size, the fully-asynchronous methods allow for a super-linear speedup due to cache efficiency and an acceleration beyond the BSP theoretical limit, without an increase of computing power;
- Neuron data structures can be fully linearized, including dynamic containers for map and priority queue implementations, due to a reliable estimation of worst-case scenario memory layout; and

Chapter 6. General Discussion and Conclusions

- Cache-efficiency is correlated to the number of steps in cache, and performance can be increased with a scheduler; however it decreases with an increase of the network size due to implicit shorter stepping intervals from synaptic delays.

As a relevant remark, the exploration of the cache-efficiency axis of acceleration and **the fully-asynchronous execution model requires mechanisms for remote execution flow control (thread gates, mutual exclusion, conditional gates and futures), available only in asynchronous runtime systems with global memory addressing capabilities.**

Our final contribution focuses on the activity of individual neurons and network dynamics of biologically inspired use cases with complex neuron models. We showed that:

- The activity of neurons is defined by a stiff solution, with long periods of low activity and brief periods of high volatility;
- The dynamics of a network during the activity of a biologically inspired use case is characterized by neurons that spike at highly heterogeneous spiking rates;
- Gradient-sensitive variable-step methods allow for a better interpolation of solution during action potential and a reduced number of steps during periods of refraction and low activity, yielding an overall reduction in interpolation time steps and a shorter time to solution;
- Most neurons that characterize regular brain activity are defined by low volatility in solution and low rate of discontinuities, and thus can be simulated with a significant increase of accuracy and speedup with variable-step methods;
- Variable-step interpolation of individual neurons is little sensitive to variation of solution (in terms of runtime and step count), yet highly sensitive to synaptic activity that causes discontinuity of solutions;
- The BSP execution model is not suitable for distributed variable-step methods due to the computational load imbalance at synchronization barriers;
- Regular speculative stepping models are not suitable for asynchronous distributed executions due to discontinuity events requiring a reset of solution state. This issue represents a performance bottleneck, due to distributed synapses communication requiring to be reversed as part of the backstepping mechanism;
- Asynchronous non-speculative interpolations (beyond communication barriers of the BSP model) are possible as long as neurons do not surpass the stepping limit dictated by the interval of confidence computed from pre-synaptic connectivity, as it guarantees no backstepping and no reversal of states;
- Asynchronous variable timestep executions allow for large stepping intervals that go beyond the BSP-based collective synchronization barriers, promoting a significant speedup due better cache-efficiency and larger individual steps;

- Variable timestepping allows for precise interpolation of solution trajectories and delivery of events on a continuous time frame, to the cost of a runtime increase in neurons with a high number of discontinuities. This limitation is overcome by grouping discontinuities in a similar fashion to fixed timestep, increasing the performance and providing a similar numerical accuracy to its fixed step counterpart;

In brief, we showed that brain activity is characterized by very distinct behavioural patterns across space and time, and that the **BSP-based fixed-step methods are not optimal to simulate the activity of large networks of complex neuron models due to: (1) an unnecessary large amount of interpolation steps during periods of low volatility in solution trajectory; and (2) insufficient interpolation steps during periods of high volatility, and as a consequence low numerical accuracy.** Moreover, brain activity is driven by time-bounded ODEs with highly-heterogeneous solution trajectories. Thus, **asynchronous variable stepping provides an efficient simulation of biologically inspired use cases, and accounts correctly for the combination of multiple scales and time constants that characterizes detailed brain models.**

6.3 Performance Outlook

We provide an overview of expected performance of the techniques presented for models covering larger parts of the cortex, and possible future bottlenecks derived from input and compute architectures heterogeneity.

Branch-Parallelism of Neuron Topologies. The efficiency of the branch-parallelism techniques in future applications will depend on the directions of hardware development and the network size. Single neurons may be accelerated extensively on multi-node multi-core architectures, yet performance increase will be limited for 16 or more compute cores. Medium-sized networks of neurons (up to a few thousand neurons) can benefit from this method due to improved computational load balancing at synchronization barriers from finer-grained parallelism. However, very large networks of neurons may be computed efficiently without branch-parallelism (Ovcharenko et al., 2015), when it is guaranteed that (1) an accurate strategy for computational load balancing of neurons across compute units — such as the Least Processing Time (Korf, 1998) applied in NEURON — provides a similar total workload per compute unit, and (2) the dataset is *large enough* to allow for enough flexibility in the distribution of neurons across compute units, in order for the aforementioned algorithm to be able to balance the dataset.

Compute architectures with large vector units such as GPUs may not be fully leveraged by this method, as only linear solver variables — but not mechanism state variables — in most models tested are expressed in enough instances to fully occupy the large register file width of common graphic processors.

Chapter 6. General Discussion and Conclusions

As a final remark, the efficiency and applicability of the branch-level parallelism method is expected to increase with neuron models characterized by more detailed topologies — particularly models including axonal branching, long-range connectivity and myelinated brain areas — as they add further complexity in the topological structure of neurons, and therefore increased opportunities for parallelism.

Graph-Parallelism of Neuron State Updates. Graph-parallelism was demonstrated for the dependencies across ODEs that describe the state of individual neurons. In this technique, SIMD acceleration was provided by memory-aligning similar ODEs across compartments on each neuron. While we demonstrated its efficiency on multi-core CPU architectures and up to a few thousand neurons, this method is also highly advantageous on very large networks of neurons and on compute architectures with large vector units such as GPUs. At first, the grouping of similar ODEs can be performed across groups of neurons, yielding a SIMD-friendly data structure that yields several thousands of memory-aligned similar ODEs. Secondly, it reduces the number of compute kernels — that would otherwise reach the order of millions in large executions — reducing the computational overhead of thread handling.

Moreover, we demonstrated that the efficiency of this method in single neuron executions is limited by the acceleration of the state update kernel in the single-step workflow. A combination of branch-parallelism with graph-parallelism overcomes this limitation, providing promising results in achieving linear strong scaling of individual neurons. On individual neurons characterized by a small system of ODEs — i.e. low graph-parallelism — or very large number of compute cores, the need for the combination of graph- and branch-parallelism becomes more prominent. However, on neuron networks composed by a few neurons we demonstrated that ideal strong scaling is possible across a wide range of multi-core and SIMD-defined compute architectures, using only graph-parallelism.

Fully-Asynchronous Cache-Efficient Fixed-Step Executions. The Fully-Asynchronous Parallel (FAP) execution model demonstrated a significant speedup of small datasets, with a decrease of acceleration following an increase of the network size. For large network models, it is expected for the maximum number of steps per stepping iteration — and consequently the time to solution — to approximate the Bulk Synchronous Parallel (BSP) execution model. On the other hand, cache-efficiency techniques such as neuron memory linearisation provide a general speedup to networks of any size and neuron model complexity. Furthermore, the communication reduce methods presented provide a significant speedup for large network models, and we expect this efficiency to increase with the network size, due to the increased number of point-to-point communication calls involved.

As an important remark, benchmark results of the FAP execution model in regular compute networks (omitted for brevity) demonstrated that an efficient implementation of the FAP — on both fixed- and variable-step methods — requires specialized point-to-point network hard-

ware such as the Infiniband utilised in our test bench. This limitation is possibly not as heavy on executions of neuron models with a higher complexity (or higher attached computational workload), as the communication overhead may be partially overlapped with computation.

Fully-Asynchronous Variable-Step Executions. The speedup presented for the FAP variable-step implementation is preserved for large neuron networks on single compute node executions, where compute cores are guaranteed to be fully-occupied with computation. However, an efficient execution of distributed variable-step simulations is a major challenge, whose efficiency relies heavily on the data distribution of neurons across compute nodes. In practice, the computational workload of individual neurons depends on their momentary activity, which cannot be predicted beforehand. We believe this is the major challenge in scaling fully-asynchronous variable-step interpolation techniques to large networks of neurons, and further research needs to be pursued in this direction. To that extent, two alternative static and dynamic load balancing strategies are detailed in the following Future Work section.

Finally, we emphasize that the acceleration results of the 65536 neuron network tested are a lower-bound of the capabilities of our methods, and that simulations of larger neuron networks are expected to yield an increase of performance beyond the results presented, due to a higher ratio of neurons in low spiking-activity brain regions.

6.4 Future Work

Due to the extent of the research fields covered by this thesis, several methods for improved numerical resolution and asynchronous computation are yet to be assessed, and are listed next.

6.4.1 Combined Micro-Parallelism Methods

Our work developed the field of micro-parallelism of neuron networks in several independent approaches. However, both micro-parallelism efforts displayed a hard limit in strong scaling. As future work, we believe that the combination of the branch-parallelism, graph-parallelism and cache-efficiency methods presented allows for improved strong scaling on the simulation of individual neurons, and to a simulation with a runtime faster than biological time.

6.4.2 Automatic Tuning of Data Layout to Hardware Specifications

An important area of research in the work presented focused on the fine tuning of the data layout to the hardware specifications. Similarly to other implementations found in literature, this parametrization is performed beforehand with a grid search approach. An approach based on an optimization method would be ideal, yet the complexity of such optimization is tremendous, due to the high dimensionality of the variables in the problem, hardware

specifications and neuron topology definitions. This problem is of general concern in the field of computer science, and we expect to cover a problem-specific implementation in future work.

6.4.3 Variable Timestepping Without Discontinuities

The variable timestepping interpolation can be improved for models that incur no discontinuities. Adding the axonal branch information to the simulation would allow for a continuous interpolation of the synaptic state throughout the branch and at axon terminal, so far limited to a model of discontinuity events caused by a timed synaptic activation function in the shape of a current pulse (*Dirac delta function*). For such use cases, the network of neurons could be interpolated by a single solver that would include the state of all the neurons in the network. A similar approach has been implemented by the module *global vardt* in NEURON (Lytton & Hines, 2005), lacking SIMD speedup capabilities.

6.4.4 Finer-Tuned CVODE Execution and Parameters

The variable timestep results could be improved by exploring the flexibility of the methods in the CVODE (C Variable-step solver for ODEs) library. To that extent, we propose an evaluation of a more accurate Jacobian, a lower-order resolution of the BDF for lower memory consumption, finely-tuned tolerance values (relative tolerance in CVODE, and forward and adjoint sensitivity analysis capabilities in CVODES (Serban & Hindmarsh, 2005), a superset of CVODE), and a method for the *fallback* to a fixed step implicit resolution — e.g. by discarding user-provided tolerance values and enforcing a minimum step size on the BDF order 1 — for neurons with very large number of discontinuities.

6.4.5 Asynchronous Speculative Execution

The *exhaustive yet not speculative* asynchronous execution model demonstrated in Chapter 4 and Chapter 5 took advantage of the maximum time instant allowed by the pre-synaptic connectivities to compute the stepping interval of confidence. This allows for a non-speculative execution and guarantees that no events are missed. This is of particular importance as a missed event would cause a reset of state at a previous time instant of confidence. Nevertheless, and in particular in the case of variable time stepping where the computational complexity of neurons is highly heterogeneous and activity dependent, this may lead to periods of idleness at the compute nodes.

An alternative implementation based on a *carefully speculative* execution model would improve the performance of the system. This method would allow neurons to step speculatively (beyond the synaptic delay -based instant of confidence), while being careful enough to avoid situations that would initiate a cascade of spiking and solution resets throughout neurons. In practice, this could possibly be implemented by enforcing an interpolation limit at instants of

spike initiation in speculative steps.

6.4.6 Distributed Load Balancing

Our simulation model provided an efficient execution of a static data structure of a neuronal ensemble whose internal state changed over time. Distributed load balancing and rearrangement of data memory layouts are precomputed and performed once at the onset of execution. However, complex use cases require *mutable* data structures or dynamic redistribution of neurons across compute nodes. To name a few:

Synaptic plasticity. A major use case is the structural plasticity covered in our model, driven by the recruitment of new glutamatergic receptors in existing synaptic spines. Such change is simply modelled by a change in the total capacitance of the ion channel mechanism. Ongoing research aims at modelling structural plasticity from the growth of new synapses. This would require new instantiations of mechanisms being created on the compartments during runtime and new connections between neurons.

Growth of neural morphologies. A second example of dynamic data structures relates to the growth of neuronal arborization, important during embryonal and adult neurogenesis, and essential for the regeneration of neuronal connections and the increase of neuronal connectivity. The study of a theoretical model for topological development of neurons has already been presented (Kanari, 2018), yet it lacks an implementation that accounts for the growth of data structures.

Load balancing on the simulation of distributed variable-timestep methods. We showed in Chapter 5 that the simulation time of individual neurons in variable timestep executions is highly dependent on the discontinuity events triggered by the synaptic activity in the network. However, while in small neuron networks the spiking activity is quasi-homogeneous across neurons, we demonstrated that it is highly heterogeneous in large networks. A distribution of neurons across compute nodes based on activity would provide a balanced compute workload across compute resources, leading to a significant reduction in overall time to solution. However, the computational workload depends on the momentary activity, which cannot be predicted beforehand and changes throughout the simulation.

One possible solution is the static load balancing at the onset of the execution, with a round-robin allocation of neurons to compute architectures, based on their physical location in the neural network. The rationale is that neighbouring neurons are highly connected and likely to produce similar activity throughout time. Thus, this would potentially yield a distribution of sets of neurons with similar activity across compute nodes, delivering a similar amount of workload.

A second option is the dynamic allocation of neurons to compute resources throughout the simulation. A possible implementation could rely on a hypergraph partitioning method that continuously allocates neurons to compute nodes at a predetermined interval, based on their most recent activity. As a relevant remark, the FAP execution model combined with a dynamic load balancing schema may solve the long-standing problem of computational load imbalance at synchronization barriers, common to BSP-based variable-step simulations.

Multi-scale simulation. A final example covers the modelling of neuronal activity at multiple levels, defined by high heterogeneity of computation across the combined data structures that represent population, point, compartmental, and molecular information of neurons. At such level, the load balancing task is already a complex task, with an increased complexity in the event of dynamic data structures. Additionally, computations utilising cross-referencing (*pointers*) of data structures becomes an extremely convoluted process as data dependencies may lie anywhere in the compute network. This is particularly a hard task in regular memory structures with local (RAM level) memory addressing, as internal pointers and host rank should be updated if data structures are required to be moved to other compute nodes.

To overcome these issues, dynamic load balancing on a global address space would be a viable solution. The active GAS capabilities in HPX provide support for automatic dynamic *on-the-fly* data movement across compute nodes, based on user-provided (de)serialization functions. Similarly to the passive GAS implementation implemented in this thesis, cross-referencing would be automatically solved with global (distributed) address pointers managed by the runtime system.

6.4.7 Support for Graphics Processing Units

Graphics Processing Units (GPUs) perform limited operations very quickly and in parallel, due to a reduced instruction set that performs specialised operations in thousands of threads simultaneously. Benchmarks of synchronous executions of our simulation model using GPUs are part of ongoing research and have been described in the literature (Kumbhar et al., 2019). Results demonstrated a high computational throughput of similar kernels derived from several instances of similar mechanisms, mostly visible in datasets with large networks of neurons due to a high number of similar ODEs. GPU support is not provided by HPX but has been introduced in different asynchronous runtime systems such as OmpSs (Bueno et al., 2012) and Legion (Bauer et al., 2012). Therefore, the feasibility of GPU-like compute units in asynchronous execution models should be investigated in the future.

6.5 Closing Discussion

Modern advancements in digital imaging techniques allow for new levels of detail in neuronal topologies that were not possible until recent times. Such advancements allow for the

discovery of new computational models of biological phenomena that unveil a level of computational complexity that goes way beyond the original Hodgkin-Huxley and Integrate-and-Fire formulations of point neuron models.

On the other hand, the spectrum of available computing resources continues expanding in several directions, with state-of-the-art compute architectures being defined by highly-distinct hardware specifications. To this extent, massively-parallel multi-core CPUs, neuromorphic chips, embedded devices, and highly-vectorized compute units such as GPUs are being developed concurrently. The efficient usage of such wide range of hardware specifications on the simulation of networks of detailed neuron models is a hard problem. In addition, simulations of large networks require the usage of distributed networks of compute nodes, increasing even further the problem difficulty due to the added distributed memory and communication axes that needs to be taken into account.

Runtime systems play a major role in the handling of such convolution of problem and hardware complexity. However, common runtime systems — particularly the Message Passing Interface (MPI) — are mostly suitable for problems defined by homogeneous computation across neurons, with synchronized activity at node and network level. Nonetheless, the introduction of asynchronous runtime systems with distributed memory addressing capabilities, allows for the rethinking of current strategies for the resolution of simulations characterized.

The general object of this thesis was to explore novel computational methods from the usage of asynchronous runtime system on the simulation of neural networks. Our research demonstrated that asynchronous computation is a requirement for succeeding in efficiently utilising the compute capabilities of modern compute architectures. This is of particular relevance for the study of networks of detailed Hodgkin–Huxley neuron models, characterized by a large collection of biological mechanisms and an extensive arborization. At such level of complexity, we demonstrated the efficiency of asynchronous runtime systems with global memory addressing space, and the limitations of the current synchronous runtime systems based on the BSP execution model.

The application of our research methods to a biologically inspired neural network yielded: (1) improved load balancing and lower time to solution from distributed micro-parallelism of asynchronous tasks; (2) efficient asynchronous execution of neuron kernels across different memory localities; (3) acceleration of solution at the SIMD, cache, multi-core and network axes; (4) a fully-asynchronous execution model providing asynchronous communication, computation, and synchronization; and (5) an adaptive-step method for the efficient and accurate interpolation of the periods of stiffness and low volatility in solution that characterizes regular neurons activity. The scientific advancements introduced were shown to be possible only via asynchronous runtime systems, due to the unavailability of distributed memory management and asynchronous control flow objects in synchronous runtime systems. Moreover, the behaviour of neurons in large networks was shown to be very distinct across physical locations and throughout simulation time, therefore making unrealistic the usage of

Chapter 6. General Discussion and Conclusions

heuristics such as predetermined load balancing techniques that would promote an efficient BSP-based execution, characteristic of synchronous runtime systems.

Although our use case introduced already a high intricacy of distinct dynamics in the simulation, the complexity of future simulations tends to increase even further. A major use case is the multi-scale modelling of neuronal activity, part of the mission of the Blue Brain Project. Future simulations shall combine models of large populations of neurons (Omurtag et al., 2000), point neuron information (NEST simulator (Gewaltig & Diesmann, 2007)), molecular level information (STEPS simulator (Hepburn et al., 2012)), and the Hodgkin-Huxley neuron models covered in this thesis (Hodgkin & Huxley, 1952). The complexity increases further with models for the growth of branches and synapses in the system, and other biological entities such as glial cells, extra-cellular space and gap junctions, are also part of ongoing research. For such use cases, a computational model such as the one presented, with dynamic allocation of compute tasks to distributed compute resources and adaptive-step interpolation of the dynamics of individual ODEs, may very likely be the only viable solution to efficiently manage the complexity derived from the highly-heterogeneous time scales and temporal activity across different biological mechanisms.

As a final remark, this thesis provided insights for the design of future simulators across a wide range of scientific domains, driven by large heterogeneous datasets and compute architectures. Although being applied to a network of neuron models, several methods presented are problem-independent and do not require intrinsic knowledge of the problem domain from the user, therefore opening the prospectus for the acceleration of a wide domain of scientific problems modelled by systems of ODEs.

A Generalization: from Gaussian Elimination to Hines Solver

We have mentioned in Chapter 2 that the Hines Solver utilised on the NEURON multisplit is a specialisation of a reverted Gaussian Elimination applied to a tree branching structure. We will present the proof for the Backward Triangulation step. The proof for the Forward Substitution step is analogous.

Take the regular Gaussian elimination formulation:

1. Forward-triangulation:

$$cell_n \leftarrow cell_n - \frac{b_n}{d_{n-1}} cell_{n-1}$$

2. Backward-substitution:

$$cell_n \leftarrow (rhs_n - a_n * rhs_{n+1}) / d_i$$

where $cell_n$ is an iterator over all elements in row n , i.e. a_n , d_n , b_n and rhs_n ; and $cell_{n-1}$ is the element immediately above $cell_n$ in the matrix. We present the proof of the Gaussian Elimination generalization for Eq. 1. The proof for Eq. 2 follows analogously. Starting with a tridiagonal matrix representing a linear sequence of compartments, changing function variable in Eq. 1 from n to $n + 1$ leads to:

$$cell_{n+1} \leftarrow cell_{n+1} - \frac{b_{n+1}}{d_n} cell_n.$$

Converting the loop from a regular (top-down) to a reversed (bottom-up) representation:

$$cell_{n-1} \leftarrow cell_{n-1} - \frac{a_{n-1}}{d_n} cell_n.$$

Top compartment (cell soma) has no parent compartment and bottom (last index) compartment has no children. To fit the representation in a matrix with the same dimension as compartments count, Hines solver shifts down all elements in the bottom diagonal (a) by one position. Voltage contributions to (from) parent compartment are now represented on the

Appendix A. Generalization: from Gaussian Elimination to Hines Solver

upper (left) cell, respectively:

$$cell_{n-1} \leftarrow cell_{n-1} - \frac{a_n}{d_n} cell_n.$$

By replacing parent compartment index on the single linear cable ($n - 1$) by the index on a tree-based representation ($p(n)$) we get:

$$cell_{p(n)} \leftarrow cell_{p(n)} - \frac{a_n}{d_n} cell_n.$$

Finally, a and b are terms in order of the inter-compartmental resistances that are assumed to be constant. Also, the term b_n lies above d_{n+1} is in the Hines solver matrix. Thus, by unfolding the iterator $cell_n$ in the non-constant d_n and rhs_n elements, one must only solve at every step:

$$d_{p(n)} \leftarrow d_{p(n)} - \frac{a_n}{d_n} b_n, \text{ and}$$
$$rhs_{p(n)} \leftarrow rhs_{p(n)} - \frac{a_n}{d_n} rhs_n.$$

leading to the formulation of Backward Triangulation step presented by the Hines solver in Figure 2.2.

B Methods Availability and Reproducibility

The source code of the application utilised in the benchmarks detailed in this thesis is available in the Blue Brain Project's neurox repository (Bruno Magalhaes, 2017a). The code is provided *as is*, and is believed to be stable and tested thoroughly with the dependencies and datasets detailed hereafter.

Dependencies The fixed step data structures and interface with mechanism files are provided by Coreneuron (Blue Brain Project, 2015a). The NMODL to C code generation extends the original MOD2C application (Blue Brain Project, 2015b) with variable step and micro-parallelism capabilities, and is available in a specialized branch of MOD2C (Bruno Magalhaes, 2017b). Additionally, the following external dependencies are also required: cmake 2.8.12+, HPX-5 4+, google tclap 1.2.1+, and Sundials CVODES 3.1.0+.

Input datasets The datasets for this study were generated with the Blue Brain Project's Neurodamus toolkit dated 4th July 2018, and will be made available by the authors, without undue reservation, to any qualified researcher.

Compilation Compilation instructions are detailed in the main README.md file of the neurox source code.

Execution Execution parameters and modules are defined by command line flags. Existing options include the configuration of constant and minimum step size for fixed and variable step interpolators, tolerance values, Jacobian methods and parameters, BSP and FAP execution model, mechanisms graph-parallelism, branch parallelism, load balancing complexity constants, communication reduce, and output of distribution of mechanisms and topological informations. Detailed execution flags information is provided by running `neurox_exec --help`.

Appendix B. Methods Availability and Reproducibility

Testing and Validation Unit and numerical testing relies on the Coreneuron library test suite and can be executed with `make test`. Validation of numerical results is performed on-the-fly throughout the execution with bitwise comparison of state variables against Coreneuron, and is activated by compiling neurox with debugging flags.

Coding Standard Coding format and layout follows the Google C++ Style Guide standards.

API documentaion Documentation follows the doxygen notation and can be exported after compilation with `make doc`.

C Scientific Papers

List of scientific papers prepared or published during the doctoral studies:

1. Magalhaes B., Hines M., Sterling T., Schürmann F, Asynchronous SIMD-Enabled Branch-Parallelism of Morphologically-Detailed Neuron Models, submitted to *Frontiers in NeuroInformatics*;
2. Magalhaes B., Hines M., Sterling T., Schürmann F, Exploiting Implicit Flow Graph of System of ODEs to Accelerate the Simulation of Neural Networks, published at *Proc. International Parallel & Distributed Processing Symposium (IPDPS 2019)*, Rio de Janeiro, Brazil;
3. Magalhaes B., Hines M., Sterling T., Schürmann F, Fully-Asynchronous Cache-Efficient Simulation of Detailed Neural Networks, published at *Proc. International Conference on Computational Science (ICCS 2019)*, Faro, Portugal;
4. Magalhaes B., Hines M., Sterling T., Schürmann F, Fully Implicit, Fully-Asynchronous, Variable Order, Variable Timestep Simulation of Detailed Neural Networks, published on *arXiv*;
5. Magalhães B., Tauheed F, Heinis T, Ailamaki A., Schürmann F, An efficient parallel load-balancing strategy for orthogonal decomposition of geometrical data, published at *Proc. International Super Computing (ISC 2016)*, Frankfurt, Germany;
6. Markram H., Muller E., Ramaswamy S., Reimann M.W., et al., Reconstruction and Simulation of Neocortical Microcircuitry, published at *Cell* 163 vol. 163, 2015;

Bibliography

References

- Anderson, M., Brodowicz, M., Swamy, M., & Sterling, T. (2017). Accelerating the 3-d fft using a heterogeneous fpga architecture. In *European conference on parallel processing* (pp. 653–663).
- Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., & Munro, J. I. (2002). Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the thirty-fourth annual acm symposium on theory of computing* (pp. 268–276).
- Baddeley, R., Abbott, L. F., Booth, M. C., Sengpiel, F., Freeman, T., Wakeman, E. A., & Rolls, E. T. (1997). Responses of neurons in primary and inferior temporal visual cortices to natural scenes. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 264(1389), 1775–1783.
- Bauer, M., Treichler, S., Slaughter, E., & Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *High performance computing, networking, storage and analysis (sc), 2012 international conference for* (pp. 1–11).
- Blue Brain Project. (2015a). *Coreneuron - simulator optimized for large scale neural network simulations*. <https://github.com/bluebrain/CoreNeuron>. GitHub.
- Blue Brain Project. (2015b). *Mod2c - nmodl to c converter for coreneuron*. <https://github.com/BlueBrain/mod2c>. GitHub.
- Brette, R., & Goodman, D. F. (2011). Vectorized algorithms for spiking neural network simulation. *Neural computation*, 23(6), 1503–1535.
- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., ... others (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23(3), 349–398.
- Brodowicz, M., & Sterling, T. (2017). Simultac fonton: A fine-grain architecture for extreme performance beyond moore's law. *Supercomputing Frontiers and Innovations*, 4(2), 27–37.
- Brown, P. N., Byrne, G. D., & Hindmarsh, A. C. (1989). Vode: A variable-coefficient ode solver. *SIAM journal on scientific and statistical computing*, 10(5), 1038–1051.
- Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of computational neuroscience*, 8(3), 183–208.
- Brunel, N., & Hakim, V. (1999). Fast global oscillations in networks of integrate-and-fire

Appendix C. Scientific Papers

- neurons with low firing rates. *Neural computation*, 11(7), 1621–1671.
- Bruno Magalhaes. (2017a). *Neurox: A parallel and distributed asynchronous simulator of extended hodgkin-huxley neuron models*. <https://github.com/bluebrain/neurox>. GitHub.
- Bruno Magalhaes. (2017b). *The 'nmodl to c converter for coreneuron' with extension ofr variable time step and micro-parallelism operations*. https://github.com/BlueBrain/mod2c/tree/brunos_use_case. GitHub.
- Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R. M., ... Labarta, J. (2011). Productive cluster programming with ompss. In *European conference on parallel processing* (pp. 555–566).
- Bueno, J., Planas, J., Duran, A., Badia, R. M., Martorell, X., Ayguade, E., & Labarta, J. (2012). Productive programming of gpu clusters with ompss. In *Parallel & distributed processing symposium (ipdps), 2012 ieee 26th international* (pp. 557–568).
- Butenhof, D. R. (1997). *Programming with posix threads*. Addison-Wesley Professional.
- Carnevale, N. T., & Hines, M. L. (2006). *The neuron book*. Cambridge University Press.
- Casalegno, F., Cremonesi, F., Yates, S., & Hines, M. L. (2016). Error analysis and quantification in neuron simulations. *ECCOMAS Congress 2016*.
- Cash, J. (1980). On the integration of stiff systems of odes using extended backward differentiation formulae. *Numerische Mathematik*, 34(3), 235–246.
- Chindemi, G. (2018). Towards a unified understanding of synaptic plasticity: parsimonious modeling and simulation of the glutamatergic synapse life-cycle. *EPFL Infoscience scientific publications*.
- Cimini, M., Siek, J. G., & Sterling, T. (2011). The semantics of parallel, v1. 0. *Indiana University Computer Science Technical Reports*.
- Cohen, S. D., & Hindmarsh, A. C. (1996). Cvode, a stiff/nonstiff ode solver in c. *Computers in physics*, 10(2), 138–143.
- Dagum, L., & Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1), 46–55.
- Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., & Planas, J. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02), 173–193.
- Edwards, H. C., Trott, C. R., & Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12), 3202–3216.
- Forum, M. (2012). *Mpi: A message-passing interface standard, version 3.0*. USA: University of Tennessee Knoxville, Tennessee.
- Gerbessiotis, A. V., & Valiant, L. G. (1994). Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2), 251–267.
- Gewaltig, M.-O., & Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, 2(4), 1430.
- Goodman, D. F., & Brette, R. (2008). The brian simulator. *Frontiers in neuroscience*, 3, 26.
- Graupner, M., & Brunel, N. (2012). Calcium-based plasticity model explains sensitivity of

- synaptic changes to spike pattern, rate, and dendritic location. *Proceedings of the National Academy of Sciences*, 201109359.
- Gropp, W. D., Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using mpi: portable parallel programming with the message-passing interface* (Vol. 1). MIT press.
- Helias, M., Kunkel, S., Masumoto, G., Igarashi, J., Eppler, J. M., Ishii, S., . . . Diesmann, M. (2012). Supercomputers ready for use as discovery machines for neuroscience. *Frontiers in neuroinformatics*, 6, 26.
- Helmstaedter, M., & Mitra, P. P. (2012). Computational methods and challenges for large-scale circuit mapping. *Current opinion in neurobiology*, 22(1), 162–169.
- Hepburn, I., Chen, W., Wils, S., & De Schutter, E. (2012). Steps: efficient simulation of stochastic reaction–diffusion models in realistic morphologies. *BMC systems biology*, 6(1), 36.
- Hill, M. D., & Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7), 33–38.
- Hill, S., & Markram, H. (2008). The blue brain project. In *2008 30th annual international conference of the ieee engineering in medicine and biology society* (pp. clviii–clviii).
- Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., & Woodward, C. S. (2005). Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3), 363–396.
- Hines, M. (1984). Efficient computation of branched nerve equations. *International journal of bio-medical computing*, 15(1), 69–76.
- Hines, M., & Carnevale, N. T. (1994). Computer simulation methods for neurons. *The handbook of brain theory and neural networks*, 1–18.
- Hines, M., Kumar, S., & Schürmann, F. (2011). Comparison of neuronal spike exchange methods on a blue gene/p supercomputer. *Frontiers in Computational Neuroscience*, 5, 49. Retrieved from <http://journal.frontiersin.org/article/10.3389/fncom.2011.00049> doi: 10.3389/fncom.2011.00049
- Hines, M. L., & Carnevale, N. T. (1997). The neuron simulation environment. *Neural computation*, 9(6), 1179–1209.
- Hines, M. L., & Carnevale, N. T. (2000). Expanding neuron's repertoire of mechanisms with nmodl. *Neural Computation*, 12(5), 995–1007.
- Hines, M. L., Markram, H., & Schürmann, F. (2008). Fully implicit parallel simulation of single neurons. *Journal of computational neuroscience*, 25(3), 439–448.
- Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4), 500–544.
- Huck, K. A., Porterfield, A., Chaimov, N., Kaiser, H., Malony, A. D., Sterling, T., & Fowler, R. (2015). An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3), 49–66.
- Human Brain Project. (2014). *Digital reconstruction of neocortical microcircuitry*. <https://bbp.epfl.ch/nmc-portal/welcome>. (Accessed: 2016-01-31)
- Ippen, T., Eppler, J. M., Plesser, H. E., & Diesmann, M. (2017). Constructing neuronal network models in massively parallel environments. *Frontiers in neuroinformatics*, 11, 30.
- Ishiyama, T., Nitadori, K., & Makino, J. (2012). 4.45 pflops astrophysical n-body simulation on

- k computer: The gravitational trillion-body problem. In *Proceedings of the international conference on high performance computing, networking, storage and analysis* (pp. 5:1–5:10). Los Alamitos, CA, USA: IEEE Computer Society Press. Retrieved from <http://dl.acm.org/citation.cfm?id=2388996.2389003>
- Jordan, J., Ippen, T., Helias, M., Kitayama, I., Sato, M., Igarashi, J., ... Kunkel, S. (2018). Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. *Frontiers in neuroinformatics*, 12, 2.
- Josuttis, N. M. (2012). *The c++ standard library: a tutorial and reference*. Addison-Wesley.
- Kaiser, H., Brodowicz, M., & Sterling, T. (2009). Paralex an advanced parallel execution model for scaling-impaired applications. In *Parallel processing workshops, 2009. icppw'09. international conference on* (pp. 394–401).
- Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., & Fey, D. (2014). Hpx: A task based programming model in a global address space. In *Proceedings of the 8th international conference on partitioned global address space programming models* (p. 6).
- Kale, L. V., & Krishnan, S. (1996). Charm++: Parallel programming with message-driven objects. *Parallel Programming using C+*, 175–213.
- Kanari, L. (2018). Neuronal morphologies: the shapes of thoughts. *EPFL Infoscience scientific publications*.
- Kandel, E. R., Markram, H., Matthews, P. M., Yuste, R., & Koch, C. (2013). Neuroscience thinks big (and collaboratively). *Nature Reviews Neuroscience*, 14(9), 659–664.
- Kerr, J. N., Greenberg, D., & Helmchen, F. (2005). Imaging input and output of neocortical networks in vivo. *Proceedings of the National Academy of Sciences*, 102(39), 14063–14068.
- Kissel, E., & Swamy, M. (2016). Photon: Remote memory access middleware for high-performance runtime systems. In *Parallel and distributed processing symposium workshops, 2016 ieee international* (pp. 1736–1743).
- Klijjn, W., Cumming, B., Yates, S., Karakasis, V., & Peyser, A. (2017, Jul). Arbor: A morphologically detailed neural network simulator for modern high performance computer architectures.. Retrieved from <http://juser.fz-juelich.de/record/836542>
- Kohler, M., Hirschberg, B., Bond, C., Kinzie, J. M., et al. (1996). Small-conductance, calcium-activated potassium channels from mammalian brain. *Science*, 273(5282), 1709.
- Korf, R. E. (1998). A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2), 181–203.
- Kozloski, J., & Wagner, J. (2011). An ultrascale solution to large-scale neural tissue simulation. *Front. Neuroinform*, 5(15), 10–3389.
- Kulkarni, A., Dalessandro, L., Kissel, E., Lumsdaine, A., Sterling, T., & Swamy, M. (2016). Network-managed virtual global address space for message-driven runtimes. In *Proceedings of the 25th acm international symposium on high-performance parallel and distributed computing* (pp. 15–18).
- Kulkarni, A., & Lumsdaine, A. (2015). Active global address space (agas): Global virtual memory for dynamic asynchronous many-tasking (amt) runtimes. *Proceedings of the 2015 ACM/IEEE Conference on Supercomputing*.
- Kumar, S., Heidelberger, P., Chen, D., & Hines, M. (2010). Optimization of applications with

- non-blocking neighborhood collectives via multisends on the blue gene/p supercomputer. In *Ipdps.../international parallel and distributed processing symposium. ipdps (conference)* (Vol. 2010, p. 1).
- Kumbhar, P., Hines, M., Fouriaux, J., Ovcharenko, A., King, J., Delalondre, F., & Schürmann, F. (2019). *Coreneuron : An optimized compute engine for the neuron simulator*.
- Kumbhar, P., Hines, M., Ovcharenko, A., Mallon, D. A., King, J., Sainz, F., . . . Delalondre, F. (2016). Leveraging a cluster-booster architecture for brain-scale simulations. In *International conference on high performance computing* (pp. 363–380).
- Kunkel, S., Helias, M., Potjans, T. C., Eppler, J. M., Plesser, H. E., Diesmann, M., & Morrison, A. (2012). Memory consumption of neuronal network simulators at the brain scale. In *Nic symposium 2012 proceedings* (Vol. 45, pp. 81–88).
- Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., . . . others (2014). Spiking network simulation code for petascale computers. *Frontiers in neuroinformatics*, 8, 78.
- Kutluca, H., Aykanat, C., et al. (2000). Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids. *The Journal of Supercomputing*, 15(1), 51–93.
- Lennie, P. (2003). The cost of cortical computation. *Current biology*, 13(6), 493–497.
- Lusk, E., Huss, S., Saphir, B., & Snir, M. (2009). *Mpi: A message-passing interface standard*.
- Lytton, W. W., & Hines, M. L. (2005). Independent variable time-step integration of individual neurons for network simulations. *Neural computation*, 17(4), 903–921.
- Magalhães, B. R., Tauheed, F., Heinis, T., Ailamaki, A., & Schürmann, F. (2016). An efficient parallel load-balancing framework for orthogonal decomposition of geometrical data. In *International conference on high performance computing* (pp. 81–97).
- Markram, H. (2006). The blue brain project. *Nature Reviews Neuroscience*, 7(2), 153–160.
- Markram, H. (2012). The human brain project. *Scientific American*, 306(6), 50–55.
- Markram, H., Gerstner, W., & Sjöström, P. J. (2012). Spike-timing-dependent plasticity: a comprehensive overview. *Frontiers in synaptic neuroscience*, 4, 2.
- Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., . . . others (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell*, 163(2), 456–492.
- Migliore, M., Cannia, C., Lytton, W. W., Markram, H., & Hines, M. L. (2006). Parallel network simulations with neuron. *Journal of computational neuroscience*, 21(2), 119.
- Morrison, A., Aertsen, A., & Diesmann, M. (2007). Spike-timing-dependent plasticity in balanced random networks. *Neural computation*, 19(6), 1437–1467.
- Morrison, A., Mehring, C., Geisel, T., Aertsen, A., & Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural computation*, 17(8), 1776–1801.
- Niebur, E. (2008). Neuronal cable theory. *Scholarpedia*, 3(5), 2674. (revision 121893)
- Omurtag, A., Knight, B. W., & Sirovich, L. (2000). On the simulation of large populations of neurons. *Journal of computational neuroscience*, 8(1), 51–63.

Appendix C. Scientific Papers

- Ovcharenko, A., Kumbhar, P., Hines, M., Cremonesi, F., Ewart, T., Yates, S., . . . Delalondre, F. (2015). Simulating morphologically detailed neuronal networks at extreme scale.
- Planas, J., Badia, R. M., Ayguadé, E., & Labarta, J. (2009). Hierarchical task-based programming with starss. *The International Journal of High Performance Computing Applications*, 23(3), 284–299.
- Planas, J., Badia, R. M., Ayguade, E., & Labarta, J. (2013). Self-adaptive ompss tasks in heterogeneous environments. In *2013 ieee 27th international symposium on parallel and distributed processing* (pp. 138–149).
- Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., & Gewaltig, M.-O. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In *European conference on parallel processing* (pp. 672–681).
- Randles, A., Draeger, E. W., Ooppelstrup, T., Krauss, L., & Gunnels, J. A. (2015). Massively parallel models of the human circulatory system. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (p. 1).
- Ranjan, R., Khazen, G., Gambazzi, L., Ramaswamy, S., Hill, S. L., Schürmann, F., & Markram, H. (2011). Channelpedia: an integrative and interactive database for ion channels. *Frontiers in neuroinformatics*, 5, 36.
- Rodrigues, E. R., Navaux, P. O. A., Panetta, J., Fazenda, A., Mendes, C. L., & Kale, L. V. (2010). A comparative analysis of load balancing algorithms applied to a weather forecast model. In *Computer architecture and high performance computing (sbac-pad), 2010 22nd international symposium on* (pp. 71–78).
- Rossinelli, D., Hejazialhosseini, B., Hadjidoukas, P., Bekas, C., Curioni, A., Bertsch, A., . . . Koumoutsakos, P. (2013). 11 pflop/s simulations of cloud cavitation collapse. In *Proceedings of the international conference on high performance computing, networking, storage and analysis* (pp. 3:1–3:13). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2503210.2504565> doi: 10.1145/2503210.2504565
- Serban, R., & Hindmarsh, A. C. (2005). Cvodes: the sensitivity-enabled ode solver in sundials. In *Asme 2005 international design engineering technical conferences and computers and information in engineering conference* (pp. 257–269).
- Shepherd, G. M., Mirsky, J. S., Healy, M. D., Singer, M. S., Skoufos, E., Hines, M. S., . . . Miller, P. L. (1998). The human brain project: neuroinformatics tools for integrating, searching and modeling multidisciplinary neuroscience data. *Trends in neurosciences*, 21(11), 460–468.
- Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., . . . Matsuoka, S. (2011). Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis* (pp. 3:1–3:11). New York, NY, USA: ACM. Retrieved from <http://doi.acm.org/10.1145/2063384.2063388> doi: 10.1145/2063384.2063388
- Shoham, S., O'Connor, D. H., & Segev, R. (2006). How silent is the brain: is there a “dark matter” problem in neuroscience? *Journal of Comparative Physiology A*, 192(8), 777–784.
- Standard C++ Foundation. (2019). *C++ standard draft sources*. <https://github.com/>

- cplusplus/draft. GitHub.
- Steriade, M., Timofeev, I., Durmuller, N., & Grenier, F. (1998). Dynamic properties of corticothalamic neurons and local cortical interneurons generating fast rhythmic (30–40 Hz) spike bursts. *Journal of Neurophysiology*, 79(1), 483–490.
- Sterling, T., Anderson, M., Bohan, P. K., Brodowicz, M., Kulkarni, A., & Zhang, B. (2014, Apr). Towards exascale co-design in a runtime system. In *Exascale applications and software conference*. Stockholm, Sweden.
- Sterling, T., Anderson, M., & Brodowicz, M. (2017). A survey: Runtime software systems for high performance computing. *Supercomputing Frontiers and Innovations*, 4(1), 48–68.
- Sterling, T., & Zhang, B. (2018). Runtime system architecture for dynamic adaptive execution. *Big Data and HPC: Ecosystem and Convergence*, 33, 3.
- Sur, S., Koop, M. J., & Panda, D. K. (2006). High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In *Proceedings of the 2006 acm/ieee conference on supercomputing* (p. 105).
- Tikidji-Hamburyan, R. A., Narayana, V., Bozkus, Z., & El-Ghazawi, T. A. (2017). Software for brain network simulations: a comparative study. *Frontiers in neuroinformatics*, 11, 46.
- Treibig, J., Hager, G., & Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel processing workshops (icppw), 2010 39th international conference on* (pp. 207–216).
- van Wagensveld, R., & Margull, U. (2017). Experiences with hpx on embedded real-time systems. In *2017 international conference on applied electronics (ae)* (pp. 1–6).
- Vooturi, D. T., Kothapalli, K., & Bhalla, U. S. (2017). Parallelizing hines matrix solver in neuron simulations on gpu. In *High performance computing (hipc), 2017 IEEE 24th international conference on* (pp. 388–397).
- Watson, B. O., Levenstein, D., Greene, J. P., Gelinas, J. N., & Buzsáki, G. (2016). Network homeostasis and state dynamics of neocortical sleep. *Neuron*, 90(4), 839–852.
- Zenke, F., & Gerstner, W. (2014). Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Frontiers in Neuroinformatics*, 8(76). Retrieved from <http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2014.00076/abstract> doi: 10.3389/fninf.2014.00076

Bruno Magalhaes

PhD Neuroscience candidate with Computer Science background

@ brunomaga@gmail.com <https://brunomaga.github.io>  brunomaga  github.com/brunomaga
 Lausanne, Switzerland  Native in Portuguese, fluent in English and French, fair in Spanish and Slovenian



Education

- | | |
|----------------------|---|
| ongoing
Mar 2015 | PhD Neuroscience, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland <ul style="list-style-type: none">> Title : Large-scale Asynchronous Simulation of Neuronal Activity> Teaching Assistant (400 hours) for Unsupervised and reinforcement learning in neural networks, Projects in neuro-informatics and <i>In silico</i> neuroscience> Visiting scholar at Center for Research in Extreme Scale Tech., Indiana University (US), Summers 2015-17 <div style="display: flex; gap: 5px;">C C++ Python HPX-5 MPI LaTeX tensorflow google test TCLAP Sundials CVODEs API IBM BlueGene/Q</div> |
| Sep 2009
Oct 2008 | MSc Advanced Computing, Imperial College London, UK <ul style="list-style-type: none">> Final thesis on multi-core CPU, GPU and parallel computation of large Markov models in heterogeneous networks, awarded distinction and published at NSMC'10. Finished degree with Merit. <div style="display: flex; gap: 5px;">C NVIDIA CUDA Message Passing Interface (MPI) Posix threads Java</div> |
| Jul 2007
Oct 2002 | BEng (5 year programme) Systems Engineering and Computer Science, University of Minho, Portugal <ul style="list-style-type: none">> Exchange student at the University of Maribor, Slovenia, 2005/2006. Finished degree with final grade A. |

Work Experience

- | | |
|----------------------|--|
| Feb 2015
Mar 2011 | Scientific Assistant and HPC Engineer, The Blue Brain Project, EPFL, Lausanne, Switzerland <ul style="list-style-type: none">> Parallel algorithms for spatial decomposition of neural networks> Parallel algorithms for distributed task-stealing programming models on neural networks> Parallel algorithms for synaptic map reconstruction via efficient distributed sparse matrix transposition> Algorithms for the distributed spatial indexing of detailed neuron morphologies <div style="display: flex; gap: 5px;">C C++ Message Passing Interface (MPI) OpenMP CMake IBM BlueGene/P and /Q parallel IO (MPI, HDF5)</div> |
| Feb 2011
Sep 2009 | Junior Architect for IT infra-structures, Noble Group, Worldwide <ul style="list-style-type: none">> Network design of a contingency data centre for all EU Power & Gas trading infrastructure, London, UK> Network and infrastructure design of a port and warehouse for coffee and soy beans, Santos, Brazil> Implementation of a web-based software for metals and coffee trading, New York, USA <div style="display: flex; gap: 5px;">Cisco and 3Com network devices ASP.NET</div> |
| Oct 2008
Mar 2007 | Analyst programmer, MSCI (former IPD - Investment Property Databank), London, UK <ul style="list-style-type: none">> Development of a web-based geographical system for real estate data search and analytics> Development of software for data query and warehousing <div style="display: flex; gap: 5px;">C# Visual Basic F# ASP.NET MS SQL Server SSIS google maps API javascript</div> |
| Sep 2005
Jan 2005 | Software developer (part-time), Department of Physics, University of Minho, Portugal <ul style="list-style-type: none">> Development of parallel algorithms for analysis of collisions of particles, in collaboration with CERN <div style="display: flex; gap: 5px;">Fortran Message Passing Interface (MPI) C</div> |

Publications peer reviewed; first author unless mentioned otherwise

- | | |
|----------------|--|
| in preparation | An Efficient Algorithm for The Distributed Transpose Of Large-Scale Graphs And Sparse Matrices With High-Cardinality Cell Structures |
| in preparation | Distributed Asynchronous Execution Model Speeds and Scales Up Over Hundredfold The Detection Of Contacts Between Detailed Neuron Morphologies |
| submitted | Fully Implicit, Fully Asynchronous, Variable Order, Variable Timestep Simulation of Detailed Neural Networks |
| submitted | Asynchronous SIMD-Enabled Branch-Parallelism of Morphologically-Detailed Neuron Models, <i>Frontiers in Neuroinformatics</i> |
| 2019 | Fully-Asynchronous Cache-Efficient Simulation of Detailed Neural Networks, <i>Proc. International Conference on Computational Science (ICCS 2019)</i> , Faro, Portugal |
| 2019 | Exploiting Implicit Flow Graph of System of ODEs to Accelerate the Simulation of Neural Networks, <i>Proc. International Parallel & Distributed Processing Symposium (IPDPS 2019)</i> , Rio de Janeiro, Brazil |
| 2016 | An efficient parallel load-balancing strategy for orthogonal decomposition of geometrical data, <i>Proc. International Super Computing (ISC 2016)</i> , Frankfurt, Germany |
| 2015 | <i>(co-author)</i> Reconstruction and Simulation of Neocortical Microcircuitry, <i>Cell</i> 163, 456–492. |
| 2010 | <i>(MSc final project)</i> GPU-enabled steady-state solution of large Markov models, <i>Proc. 6th International Workshop on the Numerical Solution of Markov Chains (NSMC 2010)</i> , Williamsburg, Virginia |

