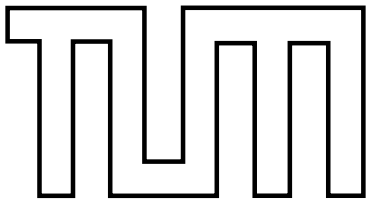


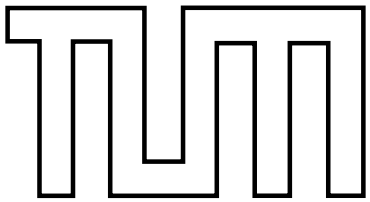
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

Parallelization of Biophysically Realistic Neural Simulations for Multi-Core Architectures

Hubert Eichner



FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

Parallelization of Biophysically Realistic Neural Simulations for Multi-Core Architectures

Parallelisierung Biophysikalisch Realistischer Neuronaler Simulationen für Multi-Core-Achitekturen

Hubert Eichner

Supervisor:	Prof. Dr. Arndt Bode
Advisor:	Dipl.-Inf. Tobias Klug
Submission Date:	17.12.2007

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this diploma thesis only supported by declared resources.

Contents

1 Introduction **5**

2 Introduction to Multi-Core Architectures **7**
2.1 History . . . 7
2.2 Challenges . . . 8
2.3 Homogeneous Multi-Core Architectures . . . 8
2.4 Heterogeneous Multi-Core Architectures . . . 10
2.5 Outlook . . . 10

3 Basics of Neurons and Neural Modeling **11**
3.1 Biophysical Foundations . . . 12
3.2 Electric Equivalent Circuits . . . 13
3.3 The Hodgkin-Huxley Model . . . 14

4 Neural Simulations **17**
4.1 NEURON . . . 17
4.2 Numerics underlying Neural Simulations . . . 18
4.2.1 General Remarks and Compartmental Modeling . . . 18
4.2.2 Spatial Discretization . . . 20
4.2.3 Temporal Discretization . . . 21
4.3 Defining the linear system of equations . . . 24
4.4 Solving the LSE . . . 27
4.4.1 Structure of the LSE . . . 28
4.4.2 Iterative Methods . . . 28
4.4.3 Gaussian Elimination and Back-Substitution . . . 28

5 Parallelizing Neural Simulations **35**
5.1 Threaded Execution Model and Terminology . . . 35
5.1.1 The Modified Fork&Join Model . . . 35
5.1.2 Mutex . . . 36
5.1.3 Mechanism Type . . . 36
5.1.4 Mechanism Instance . . . 36
5.2 Setting up the LSE . . . 37

5.2.1 Introduction . . . 37
5.2.2 Dependency Analysis for Axial and Capacitive Contributions . . . 38
5.2.3 Dependency Analysis for Mechanism Contributions . . . 38
5.2.4 Parallelization on the Compartment Level . . . 39
5.2.5 Parallelization on the Mechanism Type Level . . . 41
5.3 Solving in Parallel . . . 41
5.3.1 Networks of Neurons . . . 41
5.3.2 Single Neurons - Challenges . . . 42
5.3.3 Single Neurons - Cell Splitting . . . 43
5.3.4 Combining Whole Cell Balancing and Cell Splitting . . . 45
5.4 Comparison to Other Approaches . . . 47
5.4.1 Classification . . . 47
5.4.2 Other Approaches . . . 48

6 Results **51**
6.1 Practical Considerations . . . 51
6.1.1 Problems with Using a Stand-Alone Implementation . . . 51
6.1.2 Parallelizing Mechanism Code . . . 52
6.2 Technical Details . . . 54
6.2.1 POSIX Threads and OpenMP . . . 54
6.2.2 Thread Creation and Control . . . 54
6.2.3 Implementing Thread Waiting and Thread Notification . . . 55
6.2.4 The MC_NRN Environment Variable . . . 57
6.2.5 Measurement Problems . . . 58
6.2.6 Hardware and Software Environment . . . 59
6.3 Influence of Cache Architecture, Cache Size and Model Size . . . 60
6.4 Small Models . . . 64
6.5 Medium Size Models . . . 67
6.5.1 V1 . . . 68
6.5.2 VS Network . . . 73
6.6 Large Models . . . 75

7 Conclusions **79**

8 Acknowledgments **81**

A Mechanism Computation **83**
A.1 Using only Explicit Methods . . . 83
A.2 Using only Implicit Methods . . . 84
A.3 Mechanisms as Multi-Dimensional Functions . . . 84

B Strong Variations in Measurement Results **87**

Chapter 1

Introduction

With neurobiology and biochemistry advancing steadily, biophysically realistic modeling has become an indispensable tool for understanding neural mechanisms such as signal propagation and information processing. Increasing knowledge about anatomy and electrophysiological properties as well as increasing amounts of measurement data, however, lead to a higher computational complexity of neural simulations. Realistic neural simulations nowadays range from single cells modeled as a bunch of connected cables to whole cortical columns consisting of tens of thousands of neurons with highly accurate models of synapses and ion channels.

At the same time, a rather radical change in personal computer technology emerges with the upcoming of chip-level multiprocessors (CMP, multi-cores), high-density, explicitly parallel processor architectures for both high performance as well as standard desktop computers.

While this change in paradigms of processor development is responsible for a remarkable increase in processing power, it also challenges algorithm designers and application developers with the requirement of porting existing algorithms to this new architecture in order to exploit its high computational potential.

This work introduces strategies for the parallelization of biophysically realistic neural simulations based on the compartmental modeling technique and results of such an implementation, with a strong focus on multi-core and SMP (Symmetric Multi Processing) architectures. It first gives a rather short introduction to multi-cores, followed by an overview of neurons, neural mechanisms and neural modeling. Then, the numerics and algorithms underlying compartmental model simulations are introduced and analyzed, allowing the following sections to show up parallelization techniques for the computationally intensive parts of neural simulations. The techniques introduced are all based on the assumption that multi-cores are used; they will be compared to existing parallelization approaches, which are all based on message-passing architectures. Finally, the efficiency of the presented methods is evaluated by showing and discussing simulation results of different classes of neural models. The popular simulation software NEURON [30], [12] was used as the basis of this thesis and the implementation programmed therefore. An appendix is included that contains further information about certain topics that were found to be too extensive to be included directly in the text.

Chapter 2

Introduction to Multi-Core Architectures

2.1 History

Since the first microprocessors, Intel's 4004 [39] and Texas Instruments' TMS1000 [7], evolution in microprocessor technology was dominated by increasing the sequential performance of microprocessors. The main tactics were and/or still are increasing clock frequency, designing faster transistors, creating smaller circuits and transistors, and increase size and performance of caches. Significant effort is also put into effectively exploiting *Instruction Level Parallelism (ILP)* by a combination of pipelining (overlapping execution of consecutive instructions), executing instructions out of order, superscalar execution (the concurrent execution of different instructions) and even replicating functional units. The amount of ILP in a program is limited, however.

While manufacturing technologies regarding the size, power consumption and heat emission of a chip's underlying circuitry still improve, it has been clear for a long time that physical limitations impose upper limits on the distance an electrical signal can accomplish in a circuit within a given time interval like a clock cycle.

With both exploitation of ILP and sequential performance having been pushed to the limits by microarchitectures like Netburst [37] or Core [36], but potential left in designing chips with a higher transistor-density, the logical consequence was to integrate multiple, user-visible (as opposed to superscalar execution or pipelining techniques) processors on a single chip that appear to the user as independent processing units. This lead to the development of a new processor architecture, so called chip-level multiprocessors or multi-cores.

The concept of using multiple central processing units in one computer system has a long history, mainly in high performance computing and mainframes in the form of either *Symmetric Multiprocessing (SMP)*, where several processors (mostly 2 to 4, but usually never more than 16) are connected to main memory via a shared data bus, or in the form of *Non Uniform Memory Access (NUMA)* architectures like SGI's Altix family [62]. However, it has not been until now that this form of parallel architectures is widely used in personal computers as well.

In computer-scientific (as opposed to electrical or physical) terms, a multi-core processor does not differ significantly from an SMP-computer with the same number of processors. Therefore, many techniques and results from parallel computing research can be reused when creating

algorithms for or running programs on multi-core processors.

2.2 Challenges

Still, the evolution and spread of multi-core processors poses several challenges for developers and scientists alike because multi-cores differ in some important points from previous architectures:

1. Shared cache architectures, i. e. two or more cores on a chip using a common cache, might be preferred over currently more common separate cache architectures in the future. On one side, many parallel programs benefit from shared caches because data accessed by multiple threads is only stored in one cache, leading to a better utilization of cache memory; more importantly, however, communication between cores is much faster because it induces no main memory accesses as required when cores communicate using the MESI [40] cache coherency protocol. On the other side, shared caches result in problems due to mutual eviction of cache lines for certain applications.

2. It will become increasingly difficult to satisfy the main memory bandwidth and latency requirements of chips with many cores, enormously increasing the significance of cache efficiency of both parallel and non-parallel programs.

3. The ubiquity of multi-core processors changes the requirements for an application's parallel performance and usability. Formerly, most applications were either not available in parallelized versions at all or these parallel versions were specifically designed for special environments, requiring e. g. message passing libraries and shared file systems. In addition, parallelism was mostly not exploited automatically but required a high amount of user interaction.

The maximum number of fully functional CPUs per chip, as of the writing of this work, is 8 (prototypes with more cores but restricted instruction sets have been produced as well [38]). This number is very likely to further increase in the future. Therefore, one of the most important challenges to software developers, especially in the field of scientific computing, is to program software whose runtime scales well on multi-core architectures, by paying special attention to the above-mentioned multi-core-specific characteristics.

2.3 Homogeneous Multi-Core Architectures

Existing multi-cores may be divided into two groups, homogeneous and heterogeneous multi-cores. In homogeneous multi-cores, the single processing units are all of the same type. Currently, this is the most common multi-core architecture used, and the thesis will therefore concentrate on this kind of processors. Section 2.4 will give a short overview of heterogeneous multi-cores as well.

Figure 2.1 illustrates the architecture of a modern multi-core processor, Intel's Quad-Core Xeon 5300 series codenamed *Clovertown* [41]. The objective of this figure is to illustrate the current generation of multi-core processors used in standard personal computers and servers. Each core has its own *Level 1* (*L1*) caches. On the other hand, several cores share their *Level 2* (*L2*) cache, while other sets of cores have separate L2 caches. AMD's Quad-Core K10 series Opteron, for instance, features a *Level 3* (*L3*) cache shared between all cores [6].

Every core is a full-blown copy of a single-core processor, i. e. it features its own memory management units (MMU), floating point units including vector units, local interrupt controller (APIC) and even virtualization extensions (VT).

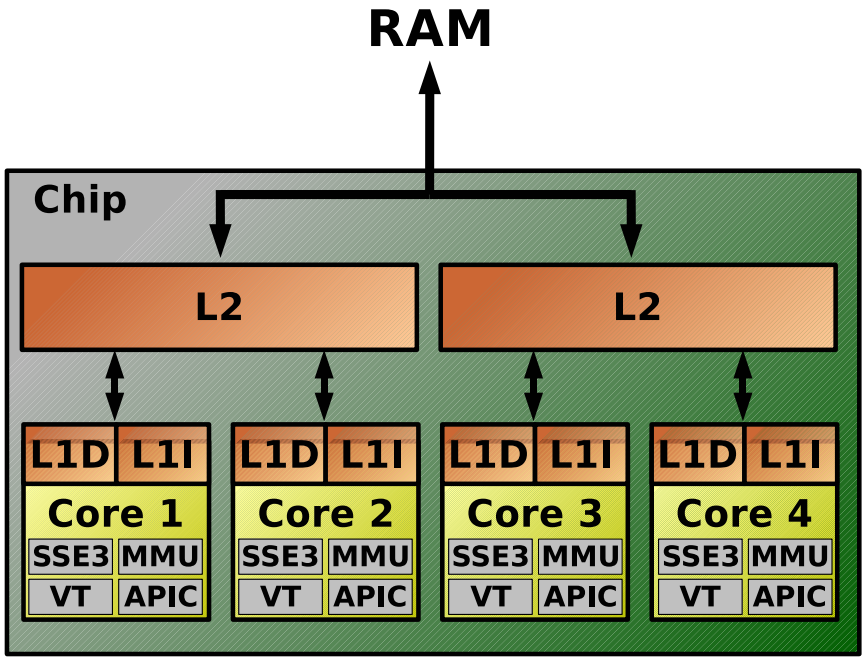


Figure 2.1: Intel's Quad-Core Xeon 5300 series processor

Shared caches might be the most important new characteristic of multi-cores. Communication bandwidth and latency are one of the main reasons why a lot of parallel programs on message passing or SMP architectures do not exhibit linear speedups for larger numbers of processors. Most parallel programs require a significant amount of inter-processor communication, for synchronizing processes or explicitly exchanging data needed by another processor. On message passing architectures, this is performed by explicitly sending messages between processors via a network connection. Multithreaded programs on multi-cores or SMP systems, on the other hand, do not exchange data explicitly because they have access to common main memory; thread synchronization is usually performed by atomically reading and/or changing specific variables. In both cases, data is exchanged between participating processors, or, to be precise, between their caches.

In either case, the more synchronizations and data exchanges are required by the implementation and/or the underlying algorithm, the higher is the impact of inter-core communication on

parallel performance. Multi-cores with shared caches, however, offer an extremely low communication latency, allowing for more fine-grained parallelization of algorithms.

On the other hand, care must be taken when processors are used that do not share a cache. Both synchronization of or exchanging data between these processors may lead to significant performance problems for programs that, at the same time, exhibit satisfying parallel performance for the same number of processors that share their caches.

This problem will be addressed and taken into account in this thesis during the design of parallel algorithms; it will also play an important role in section 6 where results are presented and discussed.

2.4 Heterogeneous Multi-Core Architectures

More heterogeneous and specialized multi-core architectures also exist. A very popular example is IBM's *Cell Broadband Engine Architecture (CBEA)* [42], which combines two general-purpose PowerPC cores with eight specialized processing units, so-called *Synergistic Processing Elements (SPE)*, that have a rather limited instruction set mainly intended for floating point vector and memory operations. This work will focus on homogeneous multi-core architectures, however; it is unclear if the CBEA or other architectures with specialized cores will become widely accepted in the future, whereas homogeneous multi-core architectures already established themselves as a standard in modern desktop computers. The main problem is that such new architectures either require re-implementing algorithms or full programs and/or their success depends on special compilers to exploit the power of the underlying architecture.

In contrast to these specialized architectures, homogeneous multi-core architectures do not require existing programs to be changed; instead, most threaded programs like servers or applications from the area of scientific computing can exploit the power of such processors without many or any changes at all.

2.5 Outlook

It is rather hard to predict what future developments will look like. While it is very likely that the number of cores per chip will further increase with shrinking transistor sizes, experiences with current multi-core architectures will have a great influence on these developments. cache efficiency will probably constitute one of the main areas of future research, but multi-core architectures might also lead to a revival of automatic parallelization techniques on the compiler-level or even on the language level. This might also advance the evolution of new programming languages as opposed to currently used languages like C that make it difficult for program analysis algorithms to automatically identify regions of potential parallelism, high memory usage etc.

Chapter 3

Basics of Neurons and Neural Modeling

This chapter gives a short introduction to single neurons, neural circuits and synapses. The area of neural sciences is extremely broad, so in this chapter, only those foundations will be dealt with that are necessary for understanding what neural simulations must accomplish.

Neurons, or nerve cells, are cells specialized for transmission of electrical signals. The human brain is thought to contain about 100 billion neurons; in contrast, the number of neurons in the fruitfly *Drosophila melanogaster* is estimated to be around 300.000, while the nervous system of the roundworm *Caenorhabditis elegans* consists of only 302 neurons. Neurons are connected with each other by synapses, either chemical or electrical (so called *gap junctions*). Neurons cover a vast range of different morphologies, but they typically consist of dendrites, the soma and the axon. The soma contains the cell nucleus and is often referred to as *cell body*. Connected to the soma are the dendrites, a tree-shaped, cellular extension that receives input via synapses from other neurons and forwards these signals to the soma and the axon. The axon is, in general, a long cable-like structure connected to the soma and forwards signals to other neurons. Axons often end in arborizations. An important aspect of neurons is that they are tree-shaped structures, i. e. there are no loops in neurons.

Many exceptions to these rules exist; for instance, the axon of an MSO cell in a gerbil's superior olivary nucleus connects to one root of the two dendritic trees instead of the soma; also, the type of ion channels and synapses often differs between different kinds of neurons. While most neurons in vertebrates encode information using so-called *action potentials*, short but high-amplitude voltage pulses, many neurons in the fly's lobula plate encode information using graded potentials, instead.

Figure 3.1 shows the general structure of a neuron in detail. In addition to the above mentioned characteristics of a nerve cell, the figure also illustrates chemical synapses in greater detail as well as axon myelination by Schwann cells, a kind of non-conducting sheath that allows for faster propagation of electric potentials by reducing the axon's capacity.

This chapter will first give a short introduction to the biophysics underlying neural information processing. Then, the basic concept of neural modeling and most simulations, using electric equivalent circuits, is presented. Finally, an example aiming to illustrate how neural modeling works is introduced, the Hodgkin-Huxley model describing action potential generation.

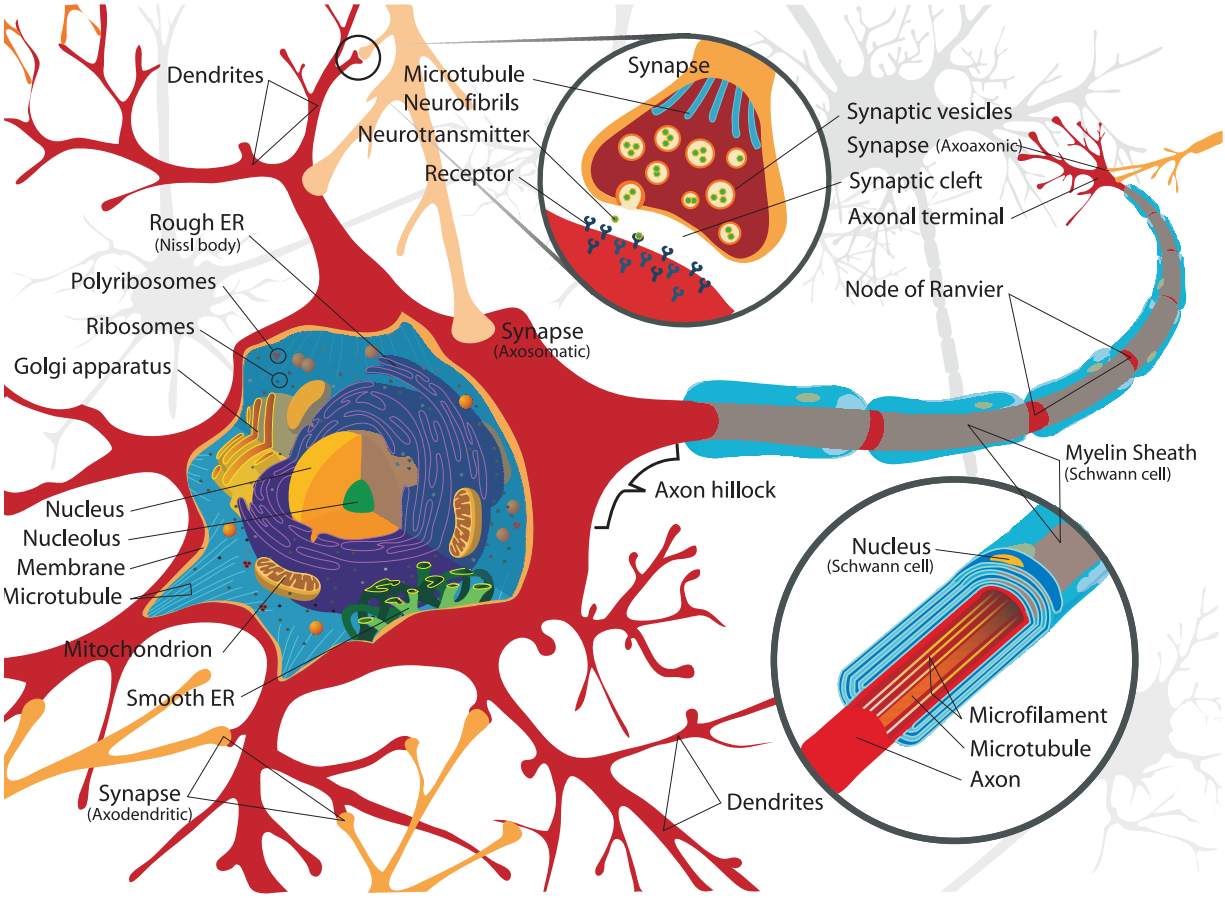


Figure 3.1: Detailed illustration of a neuron's structure. Picture is public domain and was taken from Wikipedia (http://en.wikipedia.org/wiki/Image:Complete_neuron_cell_diagram.svg).

3.1 Biophysical Foundations

Neurons allow for axial propagation of electrical signals along their membrane by locally changing the voltage over the membrane.

An electric field over the membrane exists due to differences in ion concentrations on the inside and outside of the membrane. Several ion species, most notably potassium (K^+), sodium (Na^+) and chloride (Cl^-), are transported over the membrane bidirectionally by both ion pumps and ion-selective channels. Thereby, two opposing forces for each ion species are built up, an electric gradient and a concentration gradient. The combination of these gradients results in an equilibrium state, a resting potential across a cell's membrane that is highly neuron and organism specific. It typically lies between -80mV and -40mV[43].

The actual transmission of information through electrical signals is performed by axial propagation of changes in a neuron's membrane potential. Focusing on a patch of a neuron's membrane, a rapid change in potential is achieved by ions flowing across the membrane through so-called *ion channels*. The resulting change in ion concentration of one or more ion species

leads to a change of the electric field over the membrane, influencing neighboring patches of the membrane, and thus resulting in an axial propagation of the signal. The deviation from a cell's ion concentration at its resting potential is subsequently (on a much slower timescale) brought back to zero by so-called *ion pumps*, transmembrane proteins that move specific kinds of ions across a cell's membrane.

Ion channels appear in many different forms, from so-called *passive* or *leaky* ion channels that allow passing ions all the time to voltage-dependent ion channels whose probability of being in a permissive (open) state depends on a non-linear function of time and voltage, concentration-dependent ion-channels or combinations thereof. Accordingly, the electric signals appear in a variety of forms, from graded changes to so-called action potentials or spikes.

3.2 Electric Equivalent Circuits

The key idea to most realistic neural simulations is the assumption that signal processing in neurons can be simulated by representing the realistic biophysical foundations by so-called *electric equivalent circuits* consisting of capacities, batteries and varying or constant conductances. Such equivalent circuit models exist for single ion channel types, patches of membrane and for single synapses; whole neurons are modeled by a combination of these small models by connecting patches of membrane with constant axial resistances. The general approach is to model the membrane as a parallel circuit separating the inside and outside of the cell, consisting of

- a capacitance modeling membrane capacitance,
- a constant conductance and battery, modeling the combination of passive ion channels and ion pumps,
- several variable conductances with batteries, modeling the combination of e. g. voltage-dependent ion channels and ion pumps and
- variable conductances with batteries, modeling the postsynaptic behavior of a synapse.

This technique of electric equivalent circuits allows to reproduce complex biophysical phenomena such as membrane rectification, generation of action potentials or kinetics underlying synaptic conductances. Mathematically describing the above mentioned circuit gives an ordinary, nonlinear differential equation in time. Taking into account axial currents as well would result in a partial, nonlinear differential equation in both time and space.

The most common usage of electric equivalent circuits in simulating whole neurons or networks of neurons is called *compartmental modeling*, a technique where neurons are modeled as a set of cylinders, so called *compartments*, featuring a subset or all of the mechanisms listed above; axial signal propagation is simulated by connecting these cylinders with constant axial conductances. Compartments may therefore be seen as a means of spatially discretizing the partial differential equation describing a neuron, resulting in a system of coupled ordinary differential equations in time.

This chapter, however, will first introduce an example for neural modeling of biophysical phenomena with electric equivalent circuits, the Hodgkin-Huxley mechanism for action potential generation. Compartmental modeling will be discussed in more detail in section 4.2.1.

3.3 The Hodgkin-Huxley Model

One of the most popular models in computational neuroscience is the so called Hodgkin-Huxley-model, first described in 1952 [32]. It models the electrical properties of a patch of membrane, including capacitive currents and the generation of action potentials by three different kinds of ion channels. Its popularity is mostly due to its exact prediction of action potentials; it has not been refuted by measurements in the last 55 years. In addition, it may be used to describe complete neurons by either modeling a neuron as a concatenation of several such membrane patches or representing the whole neuron with only one electric equivalent circuit. Also, due to its popularity and comprehensiveness, it will be used as an example of a complex and computationally demanding neural mechanism throughout this thesis. Figure 3.2 shows the electric equivalent circuit of the Hodgkin-Huxley model.

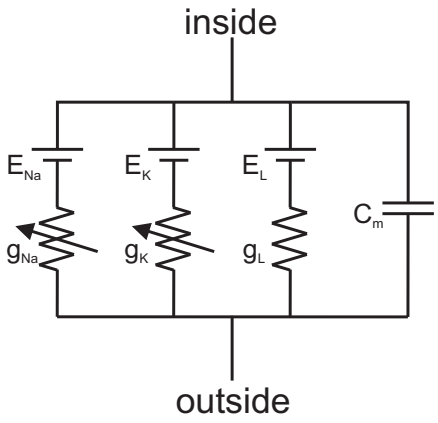


Figure 3.2: The Hodgkin-Huxley model's electric equivalent circuit

The Hodgkin-Huxley model may be seen as describing a patch of unspecified size of the membrane of a neuron. It defines the total current flowing over this patch as the sum of four types of outward currents:

- the capacitive current $I_C = C_m \frac{\partial V}{\partial t}$
- the leak current $I_L = g_L(V - E_L)$; this is the sum of currents (sodium, potassium, chloride, calcium) flowing over voltage-independent, passive channels in outward direction, with E_L being the leak equilibrium potential.
- the voltage dependent potassium current $I_k = g_K(V - E_K)$, where g_K is the voltage dependent potassium conductance and E_K the active sodium reversal potential, the membrane

potential where no sodium ions flow across voltage dependent sodium channels. Sodium may still flow across passive channels, however.

- the voltage dependent sodium current $I_{Na} = g_{Na}(V - E_{Na})$, where g_{Na} is the voltage dependent sodium conductance.

Using Kirchhoff's law, $I_{out} = I_{in}$, gives

$$C_m \frac{\partial V}{\partial t} + g_{Na}(V - E_{Na}) + g_K(V - E_K) + g_L(V - E_L) = 0 \tag{3.1}$$

Current injections may be simulated by simply adding the respective term, $I_{inj}(t)$, to the right hand side of the equation.

Key to the model are time- and voltage dependent conductances,

$$g_{Na} = \overline{g_{Na}} m^3 h; \;\; g_K = \overline{g_K} n^4$$

with $\overline{g_{Na}}$ and $\overline{g_K}$ being the (constant) maximal conductances for sodium and potassium, respectively. m, h and n may be seen as activation or inactivation particles that control the fraction of channels being in an open state. These parameters are time- and voltage dependent, dimensionless variables that are modeled using first-order kinetics:

$$m \leftrightharpoons 1 - m; \;\; h \leftrightharpoons 1 - h; \;\; n \leftrightharpoons 1 - n$$

with voltage dependent forward rates $\beta_m(V)$, $\beta_h(V)$ and $\beta_n(V)$ and voltage dependent backward rates $\alpha_m(V)$, $\alpha_h(V)$ and $\alpha_n(V)$ (in units of 1/sec) that specify how many transitions from the open to the closed state and from the closed to the open state, respectively, occur per second. The voltage dependent forward- and backward rates were defined by Hodgkin and Huxley to be

$$\alpha_m(V) = \frac{25 - V}{10(e^{(25-V)/10} - 1)}; \;\; \beta_m(V) = 4e^{-V/18}$$

$$\alpha_h(V) = 0.07e^{-V/20}; \;\; \beta_h(V) = \frac{1}{e^{(30-V)/10} + 1}$$

$$\alpha_n(V) = \frac{10 - V}{100(e^{(10-V)/10} - 1)}; \;\; \beta_n(V) = 0.125e^{-V/80}$$

The first-order kinetics for m, h and n are modeled using the following ordinary differential equations:

$$\frac{\partial m}{\partial t} = \alpha_m(V)(1 - m) - \beta_m(V)m$$

$$\frac{\partial h}{\partial t} = \alpha_h(V)(1 - h) - \beta_h(V)h$$

$$\frac{\partial n}{\partial t} = \alpha_n(V)(1 - n) - \beta_n(V)n$$

In summary, for every time step, m, h and n must be computed based on their previous values and the new, voltage-dependent forward and backward rates β and α. This is performed by a numerical integration method called *explicit Euler* which will be introduced in section 4.2.3. Then, the potassium and sodium current may be computed. The first-order differential equation 3.1 can then be solved for V using e. g. the explicit Euler method again.

This section's main intention is to show that detailed neural models of ion channels are not only often complex but also computationally demanding.

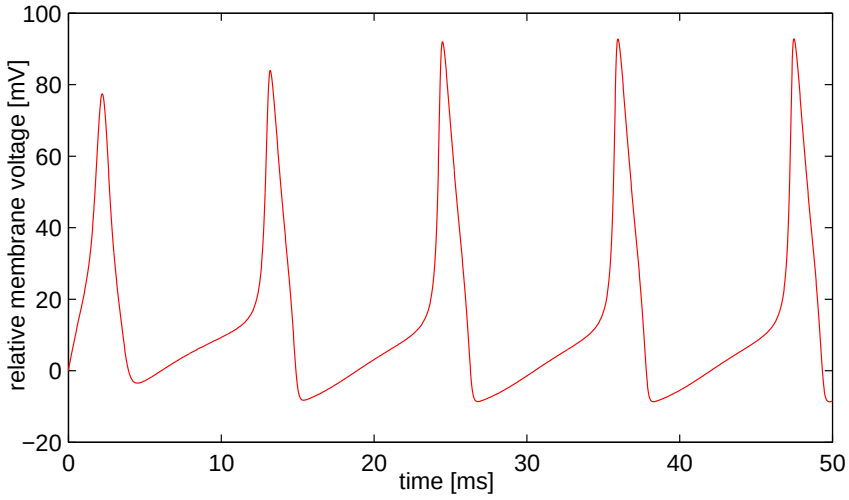


Figure 3.3: Simulation of the Hodgkin-Huxley model for a current injection of 20nA

Numerical simulation of the model for a positive current injection of $20nA$ yields a pulse of action potentials as illustrated in figure 3.3. Parameters used for the simulation were taken from [48] and are as follows: $\overline{g_K} = 36\frac{S}{cm^2}, \overline{g_{Na}} = 120\frac{S}{cm^2}, E_K = -12mV, E_{Na} = 115mV, g_L = 0.3\frac{S}{cm^2}, E_L = 10.613mV$.

Chapter 4

Neural Simulations

The last chapter gave a short overview of neural biophysics and the concept of using electric equivalent circuits to simulate patches of a neuron's membrane. This chapter will introduce the default technique used for biophysically realistic simulations of whole neurons and neural networks, *compartmental modeling*, where neurons are modeled as a set of electrically coupled cylinders called *compartments*, resulting in a big electric circuit representing the neuron and networks of neurons.

First, a popular software for neural simulations, *NEURON*, will be introduced, and it will be discussed why this software was used as the basis of the work performed in this thesis. Then, the concept of compartmental modeling and its properties compared to other approaches to neural simulation methods will be introduced. The main purpose of this chapter, however, is to establish the mathematical foundations of compartmental modeling, including a short overview of numerical methods for solving differential equations and their properties. Finally, a kind of Gaussian elimination optimized for neurons is introduced.

4.1 NEURON

This thesis is primarily based on techniques used by NEURON [30],[12]. NEURON is a software package for biophysically realistic neural simulations developed mainly by Michael Hines at Yale University and is, together with GENESIS [10],[3], the most popular software in that area. It is used by researchers all over the world, including huge projects such as BlueBrain [51]. For an extensive overview of neural simulators available, see [60].

One reason for using NEURON's techniques as the basis of this thesis is its popularity in the neuroscience community. This ensures that optimizations presented in this work will, in most cases, be applicable to NEURON, making the results of this work of practical use to neuroscientists. Another reason is that NEURON has evolved over more than twenty years and comprises algorithms and numerical methods that, in the course of time, have proven to be very robust in their field of application, while being extremely efficient due to their specialization to neural simulations.

Using and programming for NEURON instead of programming a stand-alone application

was also necessary for simulation of realistic models because of problems with importing these models, especially neural mechanisms specifically programmed for these models; section 6.1.1 further discusses this issue.

Still, most findings in this thesis are applicable to all kinds of neural simulators. The two main NEURON-specific restrictions are as follows.

- Only implicit integration methods for ordinary differential equations are considered. Explicit methods are not considered because of instabilities. While NEURON does not support explicit methods at all, GENESIS e. g. supports (in addition to the implicit methods *backward Euler* and *Crank-Nicholson*) *explicit Euler*, *exponential Euler* and *Adams-Bashforth* methods as well.

  Considering implicit methods only is not a real restriction. The main difference between implicit and explicit methods is that explicit methods require the computation of a matrix-vector product at each time step, while implicit methods usually require solving a linear system of equations. Matrix-vector multiplications are easily executed in parallel because they consist of a set of independent vector-vector products, one for every row/compartment, that may be computed simultaneously.

  Solving a linear system of equations in parallel, however, is a much more difficult task and will be handled in section 5.3; therefore this thesis will focus on implicit methods.

- All currents besides capacitive and axial currents are modeled using so called *mechanisms*. This includes gap junctions. These currents are approximated using an explicit integration method and restricting the temporal derivative of these currents to a function of membrane voltage only. This method is introduced in section 5.2.3; its implications on numerical stability and alternatives are discussed in appendix A.

4.2 Numerics underlying Neural Simulations

4.2.1 General Remarks and Compartmental Modeling

In section 3.2, the concept of modeling patches of a neuron's membrane with electrical equivalent circuits was introduced. The most popular and simplest way to simulate whole neurons or networks of neurons in a biophysically realistic way is to first perform a spatial discretization technique known as *compartmental modeling*. This technique simplifies the three-dimensional structure of a neuron into a set of uniform cylinders with arbitrary length and diameter, each; the resulting model is referred to as the compartmental model of a neuron.

Representing a neuron with cylinders is exemplarily demonstrated in figure 4.1. This figure shows a compartmental model of a *VS1* cell from the blowfly's visual system that was reconstructed from photographs of dye-filled cells. The inlet illustrates a magnification of a part of the dendritic tree to visualize how cylinders are used to model the original cell.

Regarding the simulation of current flow across a neurons membrane, every cylinder is modeled by representing its surface with one electric equivalent circuit. This electric equivalent

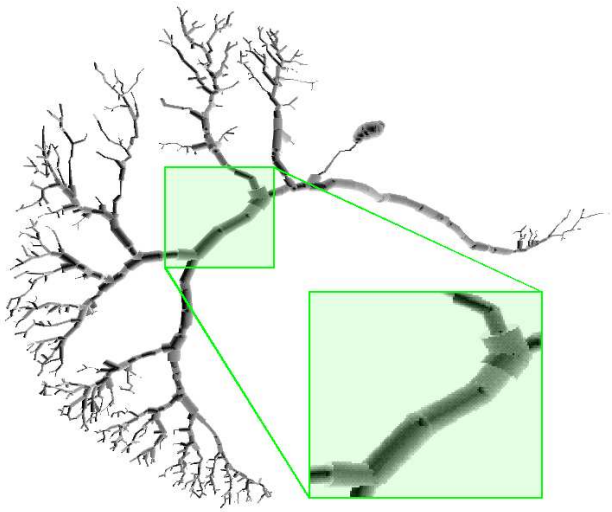


Figure 4.1: Compartmental model of a VS1 cell from the blowfly's visual system obtained by two-photon microscopy [14]. The inlet shows how cylinders/compartments are used to model the cell's morphology. The picture was generated by the Max-Planck-Institute of Neurobiology, Department of Systems and Computational Neurobiology.

circuit may be as simple as the one illustrated in figure 4.2 (this circuit models membrane capacitance, passive ion channels with a resistance of R_L and the equilibrium potential with a battery, E_L). However, electric equivalent circuits for modeling a compartment are usually more complex and consist of a combination of multiple circuits that model membrane capacitance as well as single ion channel types and ion equilibrium potentials.

The cylinders themselves are electrically coupled with neighboring cylinders with a constant conductance, the so-called *axial conductance*. This is exemplary shown in the lower part of figure 4.2.

Other approaches to biophysically realistic simulations of neurons exist as well. One possibility is to not use electric equivalent circuits as the basis of neural simulations but physical laws describing ion flow like e. g. Fick's law for diffusion and Ohm's law for drift [44], and applying these laws to a one-, two- or three-dimensional spatial discretization of a neuron. Current flow and distribution of charge may then be derived from the ion concentrations and changes thereof.

A second alternative is to model two- or three-dimensional representations with electric equivalent circuits making use of Maxwell's equations [66].

However, while compartmental modeling is not only the most popular but also a very efficient kind of neural simulation, it is also sufficiently realistic for reproducing all kinds of electrophysiological measurements. As for all kinds of simulations, the level of detail in simulation depends on the requirements of the user. While there are certain, although few, phenomena that are difficult to simulate with compartmental modeling, such as the radial (orthogonal to the membrane) diffusion of calcium, compartmental modeling has evolved as the de-facto standard in neural simulations because the majority of research done in neurobiology focuses on information processing in larger parts of cells, whole cells or networks of cells.

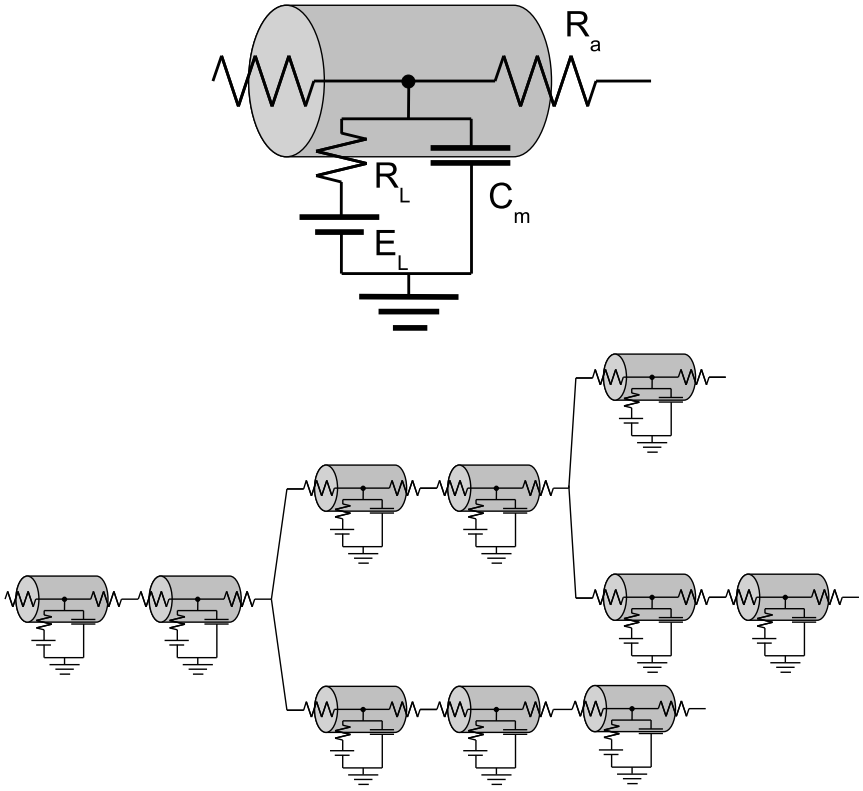


Figure 4.2: Upper part: Compartment with membrane capacitance, axial resistance, leak battery and leak resistance. Lower part: Network of compartments comprising a compartmentalized neuron

An alternative to compartmental modeling in neural simulations that is mentioned here for the sake of completeness only is to model neurons and neural nets not with biophysical realism but mathematical simplicity in mind. This is a common approach in theoretical neurobiology and computer science; it allows for mathematical analyses of neurons and neural nets by dynamic systems theory, for example, or implementing pattern matching using artificial neural nets. Therefore, strong simplifications like representing the neuron with only one compartment are often used, generalizing details of current flow like action potentials to a much simpler notion of *activity*. Synapses may be modeled by a non-linear or even linear I/O-function, not taking the time domain into account at all. This thesis, however, focuses on biophysically realistic simulations.

4.2.2 Spatial Discretization

Many (non-neural) simulation environments perform spatial discretization automatically, i. e. grids are generated (and sometimes refined and coarsened during simulation) automatically, and a certain spatial discretization method like finite differences, finite volumes or finite elements is used to simplify a partial differential equation to a system of coupled ordinary differential equa-

tions. Compartmental modeling is different from these methods in the sense that the simulation software is given an already discretized model, consisting of electrically coupled cylinders (although the generation of the compartmental model from e. g. photographs of dye-filled cells may have been performed in an automatic or semi-automatic manner).

The most intuitive way to explain how compartmental models are simulated is to set up an ordinary differential equation for each compartment that takes into account axial current contributions from neighboring compartments. This is the approach taken by the following sections.

For the sake of completeness, the term *cable equation* must be mentioned. In the context of neural simulations, documents sometimes describe simulations as a method for spatially and temporally discretizing an underlying differential equation, the cable equation [45]. This equation describes current flow in a geometrically uniform cylinder with axial, leak and capacitive currents only; current injections are usually modeled by specifying boundary- and initial conditions, and voltage-dependent ion channels or branches of the cable are not modeled at all.

Although the cable equation neither describes non-uniform diameters nor branched cable structures nor active membrane properties, it may be used to estimate the accuracy of compartmental modeling techniques by comparing an analytic solution of the cable equation to the solution of a simulation where a uniform cable with passive properties only (i. e. no voltage- or concentration dependent ion channels) is simulated. Then, one can view the method of compartmental modeling as a finite-differences discretization of the cable equation (see [12]). These considerations are out of the scope of this work, however.

4.2.3 Temporal Discretization

This section will take a look at two methods for the temporal discretization of the ordinary differential equation (*ODE*) underlying a compartment. The general tactic is to compute the voltage values for all compartments at every time step based on previously computed voltage values.

To simplify the following explanations, Kirchhoff's law will be used to set up the current balance equation for a simple compartment i like the one shown in the upper part of figure 4.2. Assuming a membrane capacitance C_m, leak resistance R_L and battery E_L, and two neighboring compartments $i-1$ and $i+1$ connected with an axial resistance of R_a, the equation looks as follows:

$$C_m \frac{\partial V_i}{\partial t} = \frac{1}{R_a}(V_{i-1} - V_i) + \frac{1}{R_L}(E_L - V_i) + \frac{1}{R_a}(V_{i+1} - V_i) \tag{4.1}$$

Due to the capacitive term in this equation, it can be written in a rather general form as

$$\frac{\partial V}{\partial t} = f(V, t) \tag{4.2}$$

Although f really depends on the voltages of neighboring compartments as well, these variables will be neglected in the following considerations about numerical methods for solving first-order ODEs for reasons of simplicity. After introducing these numerical methods, the case of coupled ODEs that take axial currents into account will be considered.

A large number of numerical methods exists for solving such ODEs (Runge-Kutta, multistep methods). Two important groups of methods are *implicit* and *explicit* methods (combinations of these methods exist as well); the difference is that while explicit methods only use old, known values for calculation of the new value, implicit methods make use of both old and new (unknown) values. In general, implicit methods require rewriting the ODE; they have the advantage of guaranteeing numerical stability, while explicit methods often result in unstable behavior like computing values that oscillate values against infinity.

In the following, two simple methods, the explicit Euler and the implicit Euler method, will be introduced and analyzed. The explicit Euler method is based on the first-order correct approximation of the first derivative of a function $V(t)$,

$$V'(t) = \frac{V(t + \Delta t) - V(t)}{\Delta t} \tag{4.3}$$

whereas the implicit Euler method is based on a different approximation of the derivative,

$$V'(t + \Delta t) = \frac{V(t + \Delta t) - V(t)}{\Delta t} \tag{4.4}$$

The real difference between the explicit and the implicit Euler method is the point in time at which the differential equation is defined. Assuming $V(t)$ is known and $V(t + \Delta t)$ needs to be computed, the explicit Euler method defines eq. 4.2 at time t, which, when combined with eq. 4.3, results in

$$V'(t) = f(V, t) \Rightarrow V(t + \Delta t) = V(t) + \Delta t \cdot V'(t) = V(t) + \Delta t \cdot f(V, t) \tag{4.5}$$

whereas the implicit Euler method (eq. 4.4) defines equation 4.2 at time $t + \Delta t$, giving

$$V'(t+\Delta t) = f(t+\Delta t) \Rightarrow V(t+\Delta t) = V(t)+\Delta t \cdot V'(t+\Delta t) = V(t)+\Delta t \cdot f(V, t+\Delta t) \tag{4.6}$$

The implicit Euler method generally results in an implicit equation and requires rewriting the ODE in order to solve for $V(t + \Delta t)$; it can thus not be applied to all classes of ODEs. Specifically, for a $f(V, t)$ that is not known in advance (as is the case in plugin-based neural simulations), eq. 4.6 cannot be solved for $V(t + \Delta t)$ unless the simulation program is able to handle symbolic math or uses e. g. root-finding algorithms such as Newton's method. This restriction is fundamental to the understanding of the following chapters, so a small example will demonstrate the problem. A nonlinear ordinary differential equation such as

$$\frac{\partial V}{\partial t} = V^2$$

can be solved with explicit Euler, resulting in

$$V(t + \Delta t) = V(t) + \Delta t \cdot f(V, t) = V(t) + \Delta t \cdot V(t)^2$$

Obviously, the non-linearity may be given as a black box the program does not need to know anything about; it is simply given the old, known value, $V(t)$ and returns something that is

inserted in the formula above to derive the new value, $V(t + \Delta t)$. Using the implicit Euler method, however, gives

$$V(t + \Delta t) = V(t) + \Delta t \cdot f(V, t + \Delta t) = V(t) + \Delta t \cdot V(t + \Delta t)^2$$

The above equation illustrates the main problem of the implicit Euler method: the equation must be rewritten to solve for $V(t + \Delta t)$, giving

$$(-\Delta t) \cdot V(t + \Delta t)^2 + V(t + \Delta t) - V(t) = 0 \Rightarrow V(t + \Delta t) = \frac{-1 \pm \sqrt{1 - 4 \cdot \Delta t \cdot V(t)}}{-2 \cdot \Delta t}$$

Assuming the simulation program cannot handle symbolic math and using root-finding algorithms is infeasible because of the complexity of f, the implicit Euler method can only be used when f (or at least f's form, e. g. linear with known parameters a and b for $f(V, t + \Delta t) = a \cdot V(t + \Delta t) + b$) is known in advance and the formula can be manually solved for $V(t + \Delta t)$ before implementing the algorithm.

The implicit Euler method, when applied to systems of coupled ODEs, requires a linear system of equations (LSE) to be solved for each time step, whereas the explicit Euler method consists of a matrix-vector-product (and possibly a vector addition) only. This important difference will be illustrated by applying both the explicit and the implicit Euler method to the current balance equation 4.1 introduced at the beginning of this section:

$$C_m \frac{\partial V_i}{\partial t} = \frac{1}{R_a}(V_{i-1} - V_i) + \frac{1}{R_a}(V_{i+1} - V_i) + \frac{1}{R_L}(E_L - V_i)$$

Applying the explicit Euler method gives

$$V_i(t + \Delta t) = \frac{\Delta t}{C_m R_a} V_{i-1}(t) + (1 - 2\frac{\Delta t}{C_m R_a} - \frac{\Delta t}{C_m R_L}) V_i(t) + \frac{\Delta t}{C_m R_a} V_{i+1}(t) + \frac{\Delta t E_L}{C_m R_L}$$

Setting up the equations for all compartments and writing this system in matrix-vector notation gives

$$\mathbf{V}(t + \Delta t) = \mathbf{M} \cdot \mathbf{V}(t) + \mathbf{B}$$

Matrix-vector products are operations that are easy to parallelize - multiplying an $n \times n$ matrix with a vector consists of n independent vector-vector products that may be executed in parallel.

However, applying the implicit Euler method to the current balance equation gives

$$-\frac{1}{R_a} V_{i-1}(t + \Delta t) + (2\frac{1}{R_a} + \frac{C_m}{\Delta t} + \frac{1}{R_L}) V_i(t + \Delta t) - \frac{1}{R_a} V_{i+1}(t + \Delta t) = \frac{C_m}{\Delta t} V_i(t) + \frac{E_L}{R_L}$$

Combining again the equations for all compartments and writing this system in matrix-vector notation gives something fundamentally different:

$$\mathbf{M} \cdot \mathbf{V}(\mathbf{t} + \mathbf{\Delta t}) = \mathbf{B}$$

As can be seen, using the implicit Euler method results in an LSE. Solving an LSE is usually a computationally intensive task with a complexity of $\mathcal{O}(n^3)$ for n coupled equations, and parallelizing this task may be very demanding.

The examples so far illustrated two problems of the implicit Euler method: First, it cannot be simply applied to all forms of ODEs because in most cases, the ODE must be rewritten; second, in the case of a system of coupled ODEs, it results in an LSE. Neural simulations, however, often comprise a great variety of different, complex mechanisms such as the Hodgkin-Huxley model introduced in section 3.3, so rewriting the equation is not an alternative. It might therefore seem inevitable to use the explicit Euler method. Unfortunately, the explicit Euler method is numerically unstable; for too large values of Δt or too small values of Δx (the distance between neighboring compartments, implicitly contained in the value of R_a), it can begin to oscillate against infinity.

Thus, in order to still guarantee numerical stability, the stable implicit Euler method or other stable methods must be used; this is achieved by a trick introduced in section 4.3.

NEURON does not support the explicit Euler method but uses the implicit Euler method as the default method of integration. For the sake of completeness, it will be noted that NEURON also supports a second method for temporal discretization, the second-order-correct Crank-Nicholson scheme. This method is best described as advancing one half-step forward using the implicit Euler method, followed by another half-step forward using the explicit Euler method:

$$V(t + \frac{\Delta t}{2}) = V(t) + f(V, t + \frac{\Delta t}{2}) \tag{4.7}$$

$$V(t + \Delta t) = V(t + \frac{\Delta t}{2}) + f(V, t + \frac{\Delta t}{2}) \tag{4.8}$$

Inserting eq. 4.7 into eq. 4.8 yields

$$V(t + \Delta t) = 2V(t + \frac{\Delta t}{2}) - V(t) \tag{4.9}$$

Obviously, the method can be implemented by an existing implicit Euler implementation with half the specified time step length, $\frac{\Delta t}{2}$, followed by subtracting the old voltage vector from twice the new one; thus, it requires only little changes to existing implicit Euler implementations.

While there exist methods that dynamically adapt Δt during the simulation, these methods are beyond the scope of this work.

4.3 Defining the linear system of equations

In the last section, the compartmental modeling technique and methods for temporal discretization of first-order ODEs were introduced to illustrate some basic properties of these methods. This section will implicitly employ these methods; it will be shown how, given a compartmental model of a neuron and both simple and complex mechanisms such as current injections or Hodgkin-Huxley type channels, a general equation for a compartment can be derived.

The resulting equation forms the basis of NEURON and is introduced in both NEURON's source code as well as the NEURON book [12]. It is used to illustrate basic operations in biophysically realistic neural simulations in general; specifically, the different operations and the order of their execution required for advancing one time step will be identified. Although using

a NEURON specific equation, the concepts derived thereof at the end of this section apply to biophysically realistic neural simulations in general.

Figure 4.3 illustrates the structure of a compartment used in NEURON. Axial currents and capacitive membrane currents are the only part of every compartment in simulations carried out by the core of NEURON. All other transmembrane currents are computed by plugins, so called *mechanisms*. The reason is that users must be able to include all kinds of neural mechanisms in simulations. The big diversity in ion channels or synapses alone, combined with ongoing research in that area, require a general plugin architecture that is able to handle arbitrary, user-defined neural mechanisms.

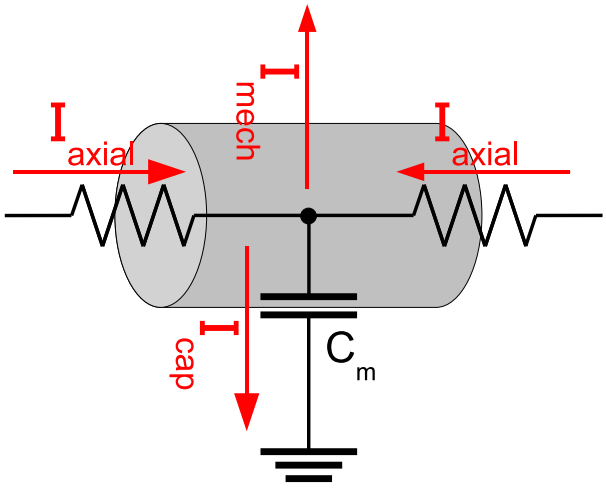


Figure 4.3: Compartment underlying NEURON. Capacitive and axial currents are intrinsic properties of the compartment. All other currents are modeled using plug-ins. These are user-defined functional units that compute the individual current contribution of arbitrary mechanisms, e. g. synaptic currents, channel currents or current injections.

The basic equation for compartment i is a combination of capacitive, axial and mechanism currents:

$$I_{cap,i} + I_{mechs,i} = I_{axial,i} \tag{4.10}$$

Using the implicit Euler method, the equation will be set up at time $t + \Delta t$. The capacitive current is defined as

$$I_{cap,i}(t + \Delta t) = C_{m,i} \frac{\partial V_i(t + \Delta t)}{\partial t}$$

The axial current is the sum of current flowing in from neighboring compartments $j \in Adj_i$:

$$I_{axial,i}(t + \Delta t) = \sum_{j \in Adj_i} \frac{V_j(t + \Delta t) - V_i(t + \Delta t)}{R_{ij}}$$

The current originating from mechanisms, $I_{mechs,i}(V(t + \Delta t))$, is the sum of arbitrary, user-defined functions. For reasons explained below, NEURON computes $\Delta V = V(t + \Delta t) - V(t)$ instead of computing $V(t + \Delta t)$ directly.

As mentioned above, the implicit Euler method cannot be readily applied to arbitrary functions $I_{mechs}(V)$, but is still required to maintain numerical stability. $I_{mechs}(V)$'s purpose is to model arbitrary, mostly highly non-linear neural mechanisms such as ion channels (often described using non-linear systems of differential equations) or chemical synapses (often modeled using underlying postsynaptic first-order kinetics).

NEURON, therefore, approximates $I_{mechs}(V(t+\Delta t))$ in V using the explicit Euler method,

$$I_{mechs}(V(t+\Delta t)) = I_{mechs}(V(t)+\Delta V) \approx I_{mechs}(V(t)) + \Delta V \frac{dI_{mechs}}{dV} \tag{4.11}$$

This is achieved by using another approximation,

$$\frac{\partial I_{mechs}}{\partial V} \approx \frac{I_{mechs}(V(t)+0.001mV) - I_{mechs}(V(t))}{0.001mV} =: \frac{dI_{mechs}}{dV}$$

Experience shows that a value of $0.001mV$ is small enough to get a very good approximation for the first derivative of $I_{mechs}(V)$. This approximation is discussed more thoroughly in section 5.2.3 and in appendix A.

Thus, eq. 4.10 becomes

$$C_{m,i}\frac{\partial V_i(t+\Delta t)}{\partial t} = \sum_{j\in Adj_i} \frac{V_j(t+\Delta t) - V_i(t+\Delta t)}{R_{ij}} - I_{mechs,i}(V(t)) - \Delta V_i \frac{dI_{mechs,i}}{dV_i}$$

Using $V(t+\Delta t) = V(t) + \Delta V$ and applying the implicit Euler method results in the following equation underlying all compartments in NEURON:

$$C_{m,i}\frac{\Delta V_i}{\Delta t} + \Delta V_i \frac{dI_{mechs,i}}{dV_i} - \sum_{j\in Adj_i} \frac{\Delta V_j - \Delta V_i}{R_{ij}} = -I_{mechs,i}(V_i(t)) + \sum_{j\in Adj_i} \frac{V_j(t) - V_i(t)}{R_{ij}} \tag{4.12}$$

or, rewritten to attenuate the structure of the resulting LSE:

$$(\frac{C_{m,i}}{\Delta t} + \frac{dI_{mechs,i}}{dV_i} + \sum_{j\in Adj_i}\frac{1}{R_{ij}})\Delta V_i - \sum_{j\in Adj_i}\frac{1}{R_{ij}}\Delta V_j = -I_{mechs,i}(V_i(t)) + \sum_{j\in Adj_i}\frac{V_j(t)-V_i(t)}{R_{ij}} \tag{4.13}$$

The reason for computing ΔV instead of computing $V(t+\Delta t)$ directly is three-fold; it simplifies integration of the mechanism term into the equation and it does not require capacitive terms to be added to the right hand side (*rhs*) of the equation; third, it simplifies application of the Crank-Nicholson method, as shown below.

The process of advancing from t to $t+\Delta t$ consists of two steps:

1. Setting up the LSE:

$$\mathbf{G}\cdot\Delta\mathbf{V} = \mathbf{rhs},\ \mathbf{G}\in\mathbb{R}^{n\times n}, \Delta\mathbf{V}, \mathbf{rhs}\in\mathbb{R}^n$$

As the off-diagonal elements in $\mathbf{G}$, $-\frac{1}{R_{ij}}$ (the axial conductances between neighboring

compartments), do not change, this reduces the work to computing the diagonal entry and the right hand side of every compartments equation.
Note when using the Crank-Nicholson method, Δt must be replaced by $\frac{\Delta t}{2}$.

2. Solving the LSE, updating the voltage vector:
The code responsible for solving the LSE and updating the voltage vector will be referred to as the *solver*. When using the default method, implicit Euler, updating the voltage vector is performed by

$$V(t + \Delta t) = V(t) + \Delta V \tag{4.14}$$

For the Crank-Nicholson method (see eq. 4.9), eq. 4.14 is replaced by

$$V(t + \Delta t) = V(t) + 2\Delta V \tag{4.15}$$

4.4 Solving the LSE

It was shown in section 4.2.3 that using an implicit method for discretization of the capacitive term for stability reasons results in an LSE to be solved for every cell instead of a matrix-vector-product as for explicit methods. Because a matrix-vector-product for explicit methods is easily parallelized, this thesis concentrates on the much more demanding case of using implicit methods for the temporal discretization. The process of solving arbitrary LSEs with n equations usually has a complexity of $\mathcal{O}(n^3)$ using Gaussian elimination and back-substitution.

Multiple neurons are represented by different LSEs that can be solved in parallel. Although neurons may be semantically connected by either chemical synapses or electrical couplings called gap junctions, these connections are modeled using mechanisms instead of explicitly expressing them in the matrix by adding off-diagonal elements (see sections 4.1 and 4.3 for details). Therefore, each neuron is represented by a separate matrix. The most straightforward way of parallelizing the solver is to distribute the neurons onto different processors such that the *load* of every processor (the number of compartments this processor must solve) is as close to the average load as possible. The time needed for solving the LSE is bounded by the processor with the highest load. This method is called *whole cell balancing* and will be introduced in more detail in section 5.3.1.

However, whole cell balancing usually does not deliver proper load balance when the number of cells is lower than or roughly about the number of cores. It is therefore necessary to find a strategy that allows parallelization of the elimination and back-substitution process, as well. Although LSE setup usually comprises the majority of the runtime of a simulation, as section 5.2 will explain, the influence of load balance problems during solving on the overall runtime will grow with increasing numbers of cores per chip. In addition, as results will show, load balance issues in the solver stage may have a significant impact on mechanism computation and vice versa on separate cache architectures because of cache efficiency issues, a problem whose significance only become clear late while performing benchmark measurements for this thesis. Last but not least, parallelization of the solving stage is the most interesting and demanding part of neural simulations from an algorithmic point of view; therefore, a great part of the time available for

this thesis was spent on strategies for parallelizing Gaussian elimination and back-substitution of LSEs in the neural context.

The following section will show up a very efficient algorithm for solving LSEs underlying a neuron. This algorithm will be explained in detail because it plays an important role in parallelizing the solver.

4.4.1 Structure of the LSE

The matrix $\mathbf{G} \in \mathbb{R}^{n \times n}$ of the LSE,

$$\mathbf{G} \cdot \Delta\mathbf{V} = \mathbf{rhs}$$

has some interesting characteristics. Every compartment is represented by one row of the matrix (and the corresponding element in the right hand side vector, $\mathbf{rhs}$). Equation 4.13 for compartment i with neighboring compartments $j \in Adj_i$ can be written in a simplified form,

$$D_i \cdot \Delta V_i - \sum_{j \in Adj_i} g_{ij} \Delta V_j = rhs_i$$

that attenuates the structure of the matrix, $\mathbf{G}$. Row i, consisting of n elements, is rather sparsely populated, i. e. most elements are zero, with the exception of the diagonal element D_i (the i-th element in this row) and off-diagonal elements, $-g_{ij}$, one for every neighboring compartment $j \in Adj_i$.

In summary, the matrix only contains zeros except for diagonal elements, $\mathbf{G}(i,i)$, and off-diagonal elements, $\mathbf{G}(i,j), i \neq j$ if compartments i and j are neighboring compartments. Figure 4.5 shows an example of how the matrix of such an LSE looks like - entries denoted by X indicate non-zero elements of the matrix, while all other entries are zero.

4.4.2 Iterative Methods

There exist numerous methods for solving LSEs in a more efficient manner than Gaussian elimination and back-substitution, most notably stationary (Jacobi, Gauss-Seidel [52]) and non-stationary (conjugate gradients [22], GMRES [61]) iterative methods. These methods are even more efficient when the involved matrix is sparse and appropriate preconditioning techniques are applied. Most iterative methods have the great advantage of being easier to parallelize than Gaussian elimination as the underlying operations are mostly vector-vector and matrix-vector products. One problem with iterative methods, however, is that the solution found after a limited number of iterations is, in general, only an approximation to the real solution[1].

4.4.3 Gaussian Elimination and Back-Substitution

The default way to solve the LSE $\mathbf{G} \cdot \Delta\mathbf{V} = \mathbf{rhs}$ consists of two parts, Gaussian elimination which brings the matrix into lower triangular form, and back-substitution, which finally computes

[1] For the non-stationary method of conjugate gradients, a solution for an $n \times n$-LSE is guaranteed to be found after n iterations; however, this is infeasible due to the high number of iterations and numerical errors.

the solution. The tree-like structure of neurons can be exploited such as to reduce the complexity of solving the LSE to $\mathcal{O}(n)$. Gaussian elimination and back-substitution will be explained in detail because they are necessary to understand how potential parallelism for these stages can be identified and exploited.

The following explanations about an optimized form of Gaussian elimination and back-substitution for neurons are not results of this thesis but were first described by Michael Hines in 1984 [28] and have been in use for a long time in both NEURON and Genesis. Rather, this thesis introduces a new interpretation of this method; specifically, it will be proven that the directed graphs underlying compartmental models may be interpreted as dependency graphs of the Gaussian elimination and back-substitution stage. Visualizing solving this way allows for an easier comprehension of how parallelism in Gaussian elimination and back-substitution may be identified and exploited.

Standard Gaussian Elimination

In the following, it is assumed that Gaussian elimination is used to create a lower triangular matrix.

Gaussian elimination applied to the LSE $\mathbf{G} \cdot \Delta\mathbf{V} = \mathbf{rhs}, \mathbf{G} \in \mathbb{R}^{n \times n}, \Delta\mathbf{V}, \mathbf{rhs} \in \mathbb{R}^n$ is performed by a function as depicted in figure 4.4.

```
0  int c, r, i; double p;
1  for (c = n - 1; c >= 1; c--)
2     for (r = c - 1; r >= 0; r--) {
3         if (G[r][c] == 0) continue;
4         p = G[r][c] / G[c][c];
5         for (i = 0; i < n; i++) G[r][i] = G[r][i] - p*G[c][i];
6         rhs[r] = rhs[r] - p*rhs[c];
7     }
```

Figure 4.4: Code for standard Gaussian elimination of an LSE with a matrix $\mathbf{G} \in \mathbb{R}^{n \times n}$, resulting in a lower triangular matrix.

The number of floating point operations necessary for Gaussian elimination of an LSE with an $n \times n$ matrix is linear in the product of two variables:

1. The number of elements to be eliminated
   This includes both non-zero elements above the diagonal as well as fill-in elements, i. e. above-diagonal elements that were zero but, because of row subtractions in line 5, became non-zero during elimination. The worst-case number of eliminations necessary is $\frac{n(n-1)}{2}$, the number of elements above the diagonal.

2. The length of a row in the matrix
   The elimination of an element is performed by a row subtraction (figure 4.4, line 5). In

the worst case, i. e. when all elements are or could be different from zero, this requires n scalar subtractions.

The NEURON Way of Gaussian Elimination

NEURON's method [28] reduces the complexity of Gaussian elimination to $\mathcal{O}(n)$ by intelligently making use of the tree structure of a neuron. In a nutshell, the compartments of a neuron are numbered (where the number of the compartment corresponds to the row-number of this compartment's equation) in a way that guarantees that

- there are only n non-zero elements to be eliminated in the matrix when starting the elimination
- no fill-in occurs during Gaussian elimination
- every row subtracted from another row has exactly two non-zero elements

A complexity of $\mathcal{O}(n)$ is much better than both standard Gaussian elimination or iterative methods and probably the best sequential complexity that can be achieved. This work will therefore stick to this method.

It comes, however, at the cost of strong data dependencies, massively complicating parallel implementations of the solver. In order to better understand these constraints and possible approaches to parallel solvers, the method needs to be discussed in full detail before analyzing different approaches of parallelization.

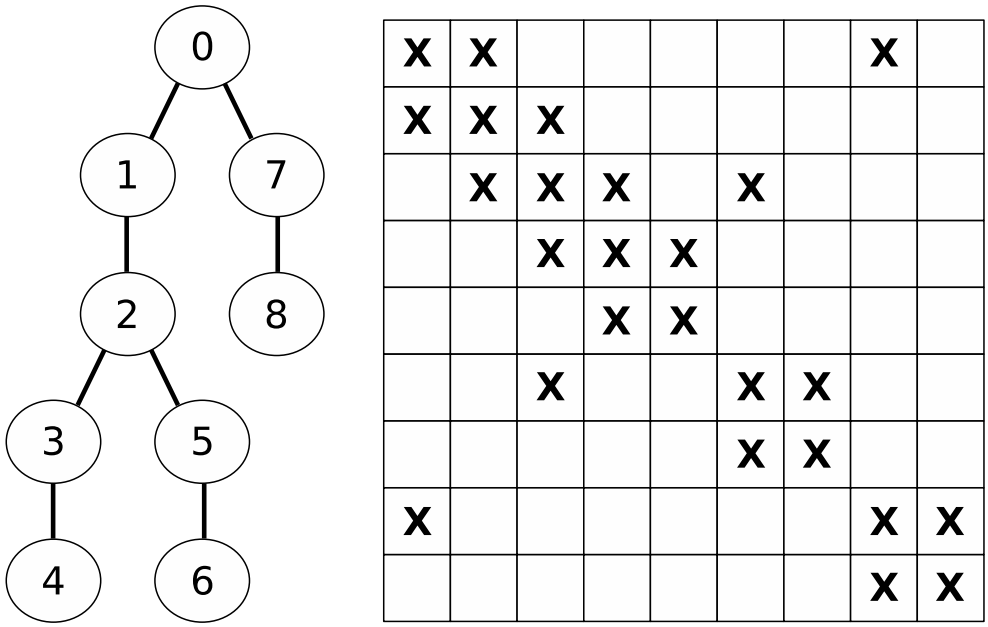


Figure 4.5: Undirected acyclic graph underlying a neuron modeled with 8 compartments and the corresponding adjacency matrix

The matrix of the LSE $\mathbf{G} \cdot \Delta\mathbf{V} = \mathbf{rhs}$ can be interpreted as the adjacency matrix of a graph $G' = (V, E)$ with the set of vertices V being the set of compartments and the set of edges E being the axial electrical couplings between compartments (see figure 4.5, left part). As real neurons are tree structured, i. e. the corresponding graph is acyclic, there exists a numbering scheme $index : V \to \mathbb{N}$ for the vertices such that every vertex u (except for vertex 0) has only one edge to another vertex v where $index(v) < index(u)$ (although it may have many edges to vertices w where $index(w) > index(u)$). This means that a vertex may have several edges to *subordinate* vertices (vertices with a higher index) but at most one edge to a *superordinate* vertex (vertex with a lower index).

Informally, this numbering scheme restricts the layout of the matrix such that

- in every column (except for the first), there is exactly one non-zero element above the diagonal and
- in every row (except for the first), there is exactly one non-zero element left to the diagonal

See figure 4.5 for an example of a graph numbered according to these rules and its corresponding adjacency matrix.

Such a numbering scheme can be achieved by e. g. choosing an arbitrary compartment r as the root of the tree, assigning $index(r) = 0$ and subsequently numbering all other compartments using a depth-first-search (*DFS*). Breadth-First-Search works as well, but for simplicity (the DFS numbering better matches the elimination data path and therefore improves cache efficiency), the following explanations will be based on the assumption of a DFS-numbering.

These rather simple restrictions imposed on graph layout and numbering allow Gaussian elimination to be performed in linear time when starting with the highest numbered compartment, i. e. at the bottom right of the matrix, proceeding to the lowest numbered compartment (*root compartment*) on the upper left, eliminating above-diagonal entries on the way:

- For every row j corresponding to compartment j, there is only one non-zero element above the diagonal element (j, j) that must be eliminated due to the numbering scheme used.
- When eliminating an above-diagonal element $(i, j), i < j$, row j is subtracted in a weighted manner from row i. No fill-in in row i occurs because row j has only two elements:
  1. its diagonal (j, j) that, in a weighted manner, is subtracted from (i, j) for elimination
  2. its left-diagonal element (j, i) that, in a weighted manner, is subtracted from the non-zero diagonal of row i, element (i, i).

  There are no elements right to the diagonal because they have been eliminated before.
- As described above, for every compartment j except for the first, exactly one row j is weighted and subtracted from another row i; row j contains only two elements, so the overall complexity of Gaussian elimination is linear in the number of elements.

Note that in order for Gaussian elimination to produce correct results, the eliminated (off-diagonal) elements do not have to be actually set to zero, because they play no further role in elimination or back-substitution. Therefore, these elements are not changed during elimination; this is also more efficient regarding setup of the matrix.

Besides the two requirements concerning the underlying graph and the numbering scheme, the only additional restriction to the algorithm in order to work was that elimination must start at the lower right element of the matrix and proceed to the root compartment, the upper left element. The only reason for this requirement is that for elimination of an element (i, j), no fill-in in row i may occur, entailing that there are no elements in row j right to the diagonal. Intuitively, this means that all direct subordinate compartments $children(j)$ of compartment j must have been processed before (resulting in eliminating all entries $(j, x), x \in children(j)$). Taking this thought one step further, it becomes clear that it is not necessary to traverse the matrix sequentially from the lower right element to the upper left element; in fact, the real restriction is that compartments may be processed only after all direct and indirect subordinate compartments have been processed.

Therefore, the best way to visualize Gaussian elimination is to think of it as a traversal of the graph from the leaves to the root compartment in an arbitrary order as long as all subordinate vertices of a specific vertex have been processed before the vertex itself is processed.

In other words, when visualizing the undirected graph G' as a directed graph G'_{dir} with edges directed from superordinate to subordinate vertices, the resulting graph G'_{dir} (figure 4.6, left graph) is in fact the data dependency graph for Gaussian elimination, revealing several independent subgraphs that could be solved in parallel.

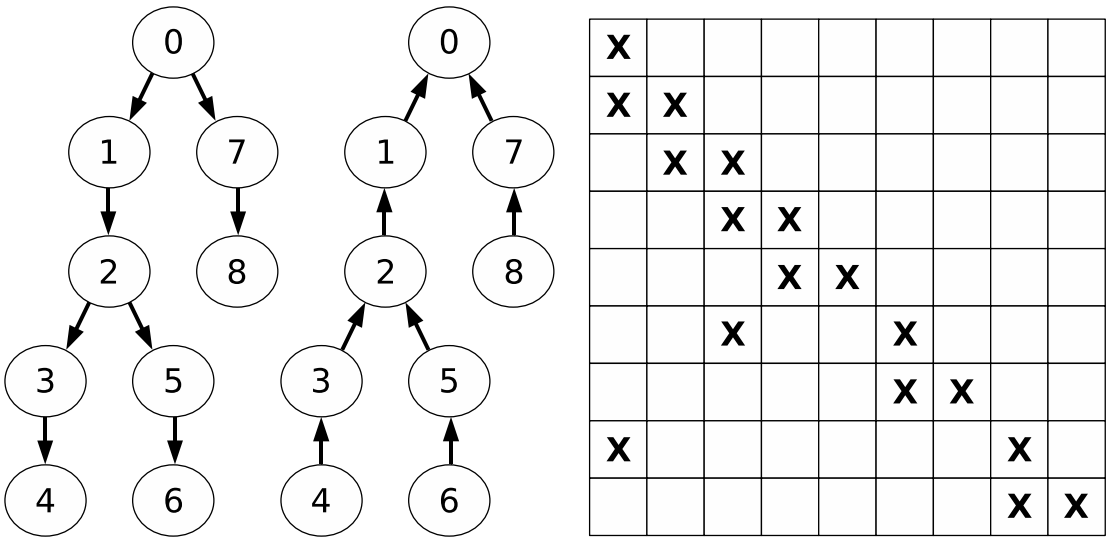


Figure 4.6: Left: Data Dependency Graph of Gaussian elimination. Middle: Data Dependency Graph of back-substitution. Right: Matrix layout after Gaussian elimination, a lower triangular matrix with at most two elements per row.

Back-Substitution

After Gaussian elimination of the LSE $\mathbf{G} \cdot \Delta\mathbf{V} = \mathbf{rhs}, \mathbf{G} \in \mathbb{R}^{n \times n}, \Delta\mathbf{V}, \mathbf{rhs} \in \mathbb{R}^n$, the matrix $\mathbf{G}$ is a lower triangular matrix. The solution vector $\Delta\mathbf{V}$ is computed by a procedure called back-substitution. The general form of the back-substitution procedure is illustrated in figure 4.7. As

```
0  int r, c;
1  for (r = 0; r < n; r++) {
2      v[r] = rhs[r];
3      for (c = 0; c < r - 1; c++) v[r] = v[r] - v[c]*G[r][c];
4      v[r] = v[r] / G[r][r];
5  }
```

Figure 4.7: Code for standard back-substitution of an LSE with a lower triangular matrix $\mathbf{G} \in \mathbb{R}^{n \times n}$

the two nested loops reveal, standard back-substitution has a complexity of $\mathcal{O}(n^2)$.

The NEURON Way of Back-Substitution

In matrices following the structure described in the previous structure, the number of floating point operations needed for back-substitution is linear in n. After Gaussian elimination, the matrix $\mathbf{G}$ has a lower triangular structure, but as no fill-in occurred during elimination, every row (except the first) consists of one diagonal element and one element left to the diagonal (see figure 4.6, right part). Line 3 in the code shown in figure 4.7 therefore requires two floating point operations (one multiplication and one subtraction), resulting in a complexity of $\mathcal{O}(n)$.

Visualizing back-substitution reveals that its data dependency graph is exactly G'_{dir} from Gaussian elimination with all edges having the inverse direction (see figure 4.6, middle part). Finding a good strategy for parallelizing triangularization based on its data dependency graph therefore can also be applied to back-substitution. Thus, the following discussions will focus on Gaussian elimination, only.

Chapter 5

Parallelizing Neural Simulations

This chapter discusses parallelization techniques for neural simulations on shared-memory architectures in general and multi-core architectures in particular. One of the main characteristics of this thesis is that the two parts of advancing a time step, setup of the LSE and solving it, will be handled separately. The first section of this chapter will introduce some parallel-computing specific terms used throughout this chapter. Then, parallelizing LSE setup will be considered in detail, and two approaches will be presented. The next section deals with solving the LSE in parallel. The final section features a comparison of the approaches presented in this thesis to existing techniques and a summary of the novelties introduced in this work.

5.1 Threaded Execution Model and Terminology

This section will introduce the execution model used and explain four terms extensively utilized in the following sections. First, the modified *Fork&Join model* used in this thesis is described. Then, the term *mutex* as a way of thread synchronization is introduced. The other two terms, *mechanism type* and *mechanism instance*, are terms defined in this thesis solely for the purpose of simplifying subsequent mechanism-specific explanations.

5.1.1 The Modified Fork&Join Model

The threaded execution model employed is a slight modification of the *Fork&Join model*, e. g. used by OpenMP [56]. The program is executed in a single-threaded manner by the so-called *primary thread*, except for *parallel regions* of the code. These parts are executed in parallel by the primary thread and the other threads, the so-called *secondary threads*, allowing for parallelization of a program by declaring computationally intensive parts of the program such as mechanism computation as parallel regions.

While the name of the Fork&Join-model implies that secondary threads are created (*forked*) at the beginning of a parallel region and terminated (*joined*) at the end of that region, the implementation in this thesis creates secondary threads only once at the beginning of the program.

The reason is simply that this eliminates the overhead for thread creation and termination at every parallel region.

When the primary thread encounters a parallel region, it notifies the other, waiting threads about what part of the code and what data they must process by passing them the required information via a shared data structure and waking them up. These parallel regions are usually (the exact usage depends on the context) enclosed by a function similar to the barrier operation that, executed by the primary thread, only returns once all secondary threads are idle.

The details of how this model is implemented are further described in section 6.2.

5.1.2 Mutex

Mutual exclusion algorithms (commonly abbreviated with *mutex*) are used in parallel programs to prevent the concurrent execution of certain regions of the code enclosed by special commands. Mutexes may be used to e. g. prevent multiple threads accessing shared variables simultaneously, or, formulated more generally, to combine a list of commands to a region of code that is executed atomically with respect to other threads.

Several algorithms that allow for the implementation of mutexes exist, for instance *semaphores* and *POSIX mutex variables* [35].

5.1.3 Mechanism Type

A mechanism or *mechanism type* comprises the code used for computation of the transmembrane current contributions of this mechanism; it may be interpreted as the equivalent of a class in the object-oriented programming (*OOP*) paradigm.

5.1.4 Mechanism Instance

A *mechanism instance*, in contrary, is the result of an instantiation of a specific mechanism type for a single compartment, encapsulating the data this mechanism needs to compute its compartment specific current contribution. It may be interpreted as the equivalent of an object in the OOP context, using the code of its class (the mechanism) but with data private to the concrete instance (i. e. private to the compartment). The amount of mechanism instance data per compartment is mechanism type dependent and usually ranges from two to more than twenty double precision floating point variables. For example, an instance of the Hodgkin-Huxley mechanism would have to store at least m, h and n for each compartment, but also parameters a user might want to define on a per-compartment level, such as the exact values of ion channel reversal potentials and maximal conductances.

5.2 Setting up the LSE

5.2.1 Introduction

As was indicated in section 4.3, setting up the LSE consists of computing the diagonal elements of the matrix and the right hand side of equation 4.13 for every compartment. The axial conductances, $-\sum_{j \in Adj_i} \frac{1}{R_{ij}}$ are constant and do not change during the simulation. The capacitive terms on the diagonal, $\frac{C_m}{\Delta t}$, are generally constant (unless varying values for Δt are used).

Computing the contributions of mechanisms to the transmembrane current of compartment i,

$$-I_{mechs,i}(V(t)) = -\sum_{k \in mechs_i} I_{mech,i,k}(V_i(t))$$

for the right hand side of every equation and

$$\frac{dI_{mechs,i}}{dV_i} = \sum_{k \in mechs_i} \frac{dI_{mech,i,k}}{dV_i}$$

for the matrix diagonal elements generally comprises the computationally most intensive part of neural simulations. This thesis will address the two main tasks, parallelizing the process of setting up the LSE and solving it, separately. This is, besides the threaded execution model, the main difference to other methods for parallel neural simulations; a more detailed comparison to existing approaches can be found at the end of this chapter in section 5.4.

While the fraction of runtime that is spent on computing mechanism dependent currents strongly varies depending on the kind and number of mechanisms used, it exceeds in most cases 80% of the overall runtime (although it may be as low as 30% when only using the computationally cheap mechanism modeling passive channels and the equilibrium battery). Taking into account the time spent on adding the capacitive term and the axial conductances to the diagonal elements and the right hand sides reveals that in most cases, more than 90% of the overall runtime is spent on the first step of advancing a time step, namely setting up the LSE.

Therefore, it is reasonable to first concentrate on achieving good parallel performance for the process of setting up the matrix before considering the much more complex task of solving the LSE in parallel, although, as results will show, these two steps sometimes strongly influence each other due to high inter-core communication latencies when the cores do not share a common cache.

Any parallel implementation must strive to reduce the overall time spent on setting up the LSE. This is achieved by distributing the existing workload as uniformly as possible onto the available resources, while taking boundary conditions such as cache efficiency and data dependencies into account. This section will first analyze the data dependencies involved in mechanism computation, followed by two multiple approaches for parallelization that were implemented and tested.

5.2.2 Dependency Analysis for Axial and Capacitive Contributions

There do not exist any data dependencies for adding the axial conductances, $\sum_{j \in Adj_i} \frac{1}{R_{ji}}$, and the capacitive term, $\frac{C_m}{\Delta t}$, to the variables representing an equation's diagonal or its right hand side. However, concurrent write accesses by different cores to the same element (diagonal or right hand side) must be prevented by either using mutexes, barrier-like operations or by choosing the strategy of assignment of equations to cores such that two cores never access the same element.

5.2.3 Dependency Analysis for Mechanism Contributions

Mechanism Applications and Heterogeneity in Placement and Complexity

Mechanisms types used in neural simulations cover a wide range of transmembrane currents. Very common types of mechanisms are

- simple leak currents comprised of a constant conductance and a battery: $I_L = g_L(E_L - V)$
- ion currents governed by complex ion channel behavior ranging from relatively simple models such as the Hodgkin-Huxley-model introduced in section 3.3 to detailed models of ion channel characteristics such as calcium channels with a low or a high activation threshold (see e. g. [64]). These kinds of currents typically require solving differential equations that model underlying first-order kinetics and thus require a high amount of floating point operations.
- synaptic currents ranging from gap junctions, modeled by a single constant conductance, to detailed models of chemical synapses where both transmitter release and postsynaptic receptor binding are modeled by complex Markov kinetic models.

Three important heterogeneity aspects of mechanism computation that will play a prominent role in parallelization design decisions must be noted before continuing:

1. Different mechanism types require different amounts of data and computation time per mechanism instance.
2. Different mechanism types have a different number of instantiations. For example, in most simulations, there are only few current injection instances, but a very high number of ion channel instances.
3. Different mechanism types are often instantiated at different locations. For instance, postsynaptic mechanisms are most commonly instantiated for compartments in the dendritic part of a cell, while active ion channel mechanisms are most commonly applied on somatic and axonal compartments.

Mechanism Dependency Analysis

Section 4.3 approximated the current contribution of a single mechanism type by

$$I_{mech}(V(t+\Delta t)) \approx I_{mech}(V(t)) + \frac{dI_{mech}}{dV}$$

In reality, most mechanism type currents are multidimensional functions of not only voltage but also ion concentrations, internal state variables for kinetic schemes, presynaptic voltages in the case of synapses and time in the case of current injections. For reasons of efficiency and simplicity, NEURON only uses this respective compartment's voltage for approximating the second term. The correct form of the approximation actually used is

$$I_{mech}(V(t+\Delta t), other(t+\Delta t)) \approx I_{mech}(V(t), other(t)) + \frac{dI_{mech}}{dV}$$

The use of and justification for this approximation is further discussed in section A.

The most important consequence of this approximation is that the computation of

$$I_{mech}(V(t+\Delta t), other(t+\Delta t))$$

depends only on old values (V, $other$) defined at time t. There are neither inter-compartmental nor intra-compartmental data dependencies for the computation of mechanisms.

Care must be taken, however, when it comes to integrating the results into the LSE. The contributions of all mechanism instances on compartment i must be added to the variables presenting the respective equation's left hand side, lhs_i (specifically, the ith diagonal element in the matrix) and right hand side, rhs_i. This requires using synchronizations in order to avoid concurrent accesses to these variables if one variable is accessed by more than one core. Relatively fast implementations of mutexes exist, yet they still involve core-to-core communication and may have a significant impact on performance if used extensively, especially for cores that do not share a common cache.

The lack of data dependencies during mechanism computation leaves many possibilities for workload distribution strategies. The following sections will show up two very promising approaches for parallel computation of mechanisms together with their advantages and disadvantages.

5.2.4 Parallelization on the Compartment Level

One approach is to split up the set of all compartments into $ncores$ (denoting the number of cores used) disjunct subsets and assign each subset to a different core. Every core then processes all mechanism instances of the compartments in its subset. This approach does not require thread synchronization at all, as no lhs_i or rhs_i is accessed by more than one core. Instead, the whole process of mechanism computation, including all mechanisms, may be defined as one parallel region, avoiding synchronizations within that region.

The identification of these subsets, however, may be a challenging task. As noted in section 5.2.3, the amount of computational work for different compartments may vary strongly depending on both the number and the types of mechanism instances on these compartments. Simply

splitting up the set of all compartments into equally large subsets may therefore lead to workload imbalance for certain models. Rather, the subsets must be chosen so that the workload per set, depending on both the number and the types of mechanism instances in this set, is roughly equal for all sets, i. e. the total workload must be distributed over all sets as balanced as possible.

One approach is implementing some kind of static load balancing by approximating the complexity of each mechanism before starting the actual simulation. This is done by performing a dummy simulation for each mechanism type with e. g. 100 compartments and mechanism instances. The complexity data gathered during these tests may then be used to estimate an approximate complexity for every compartment by summing up the complexities of all mechanisms types used on this compartment.

Approximately determining a per-compartment complexity before starting the simulation was performed by Hines, Eichner and Schuermann in their *splitcell* approach [23] (yet unpublished; see section 5.4 for details). The overall approach in their method was different, however, in that cells were split based on not only mechanism complexity considerations but with taking into account parallelization of the Gaussian elimination as well, rather constricting the effectiveness of mechanism parallelization. It should also be noted that test runs used to approximate mechanism complexity do not necessarily produce realistic results because time for mechanism computation may depend on additional parameters such as presynaptic action potentials in the case of postsynaptic mechanisms.

A simpler and more effective approach is to use a kind of dynamic load balancing by measuring time spent on mechanism computation at the beginning of the simulation and, in case of imbalances, dynamically resize the sets of the different cores during the simulation such that every core has approximately the same amount of work. Such a method was implemented after one model, a *CA1 pyramidal cell*, exhibited a strongly heterogeneous mechanism distribution which rendered the initial, homogeneous distribution of compartments useless. The implementation measures time spent on mechanism computation for each core and each mechanism type. After a fixed number of time steps, e. g. $nsteps = 50$, a per-mechanism-instance complexity mc_i is estimated for every mechanism type i based on the accumulated time spent for this mechanism type on every core c, $t_{mech,i,c}$ and the number of mechanism instances of mechanism type i, n_i:

$$mc_i = \frac{\sum_{c=1}^{ncores} t_{mech,i,c}}{nsteps * n_i}$$

Following this, a per-compartment complexity, co_j is calculated for every compartment j based on the mechanism types used for this compartment, $mechs_j$:

$$co_j = \sum_{k \in mechs_j} mc_k$$

When assigning only consecutively numbered compartments to a set, the first set of m out of n compartments may then be determined by the following formula:

$$\sum_{j=1}^{m} co_j \approx \frac{1}{ncores} \sum_{j=1}^{n} co_j \tag{5.1}$$

The next set is determined in the same manner, starting with compartment $m + 1$. The effectiveness of this approach will be evaluated in chapter 6.

In addition to the advantage of not requiring locks because an equation's variables are accessed by one core only, another advantage of this approach is caching of these variables across mechanisms. When not taking cache evictions into account, the second mechanism instance on each compartment can access already cached values for the above mentioned variables, while approaches where mechanism instances on one compartment are computed by different cores that do not share a cache often lead to inter-core communication.

5.2.5 Parallelization on the Mechanism Type Level

The second promising approach is to first create sets for each mechanism type containing all instances of the respective mechanism, followed by splitting each set into $ncores$ equally large subsets and assigning them to the available cores. While this approach always results in a perfect balance of load, it has a significant drawback.

The sets of all instances of different mechanism types may overlap such that two mechanism instances on the same compartment are computed by different cores. Thus, either mutexes must be used to ensure no variables representing a compartment's equation are accessed by two cores simultaneously, or, the approach chosen in this thesis, every mechanism type computation is declared as a parallel region with the primary thread waiting for the completion of all secondary threads after each mechanism type. However, as load balance is perfect, the time spent on waiting for other cores should be very low; the problem is rather that the synchronization alone may take a significant amount of time on separate cache architectures because of the high complexity of inter-core communication.

In order to identify unnecessary synchronizations, one might check for overlaps in mechanism instances sets during initialization and generate a new order of how mechanism types are processed such that the number of overlaps in consecutive mechanism instance sets (and thus the number of necessary synchronizations) is minimized. However, these modifications are quite complex and could not be implemented and tested in the limited time available.

5.3 Solving in Parallel

This section will show up strategies for parallelizing the solver. The first part of this section will focus on whole cell balancing, while subsequent sections will explore the possibility of parallelizing the process of solving a single neuron and how these findings can be combined with whole cell balancing.

5.3.1 Networks of Neurons

At the beginning of section 4.4, the concept of whole cell balancing was introduced - different neurons may be seen as unconnected graphs, each neuron being described by a separate matrix.

Whole cell balancing is a method that assigns whole neurons to cores that solve the corresponding LSEs such that load balance is ensured.

Whole cell balancing is best done by a static load balancing technique as the workload and number of processors is known in advance and does not change during the simulation. The problem of assigning n neurons with sizes/complexities $size(i)$ to m cores such that the load on each cores is as close to the average load as possible is also known as *Number Partitioning Problem* or simply *Partitioning Problem* [20] and belongs to the class of NP-complete problems.

Therefore, a heuristic algorithm should be chosen for larger numbers of cells and cores; one possibility of assigning neurons to processors is to simply sort the neurons by their size, largest neuron first, and consecutively assign the largest unassigned neuron to the processor with the currently lowest load. This greedy algorithm [49] delivers satisfactory results in most cases; it is also used in the *splitcell* approach, introduced in section 5.4.

This algorithm is employed by the implementation used for performance benchmarks in this thesis.

5.3.2 Single Neurons - Challenges

One problem of whole cell balancing in general is that the method fails to deliver proper load balance for many sets of neurons; most notably when there are more cores than neurons or when the size of neurons varies strongly.

Therefore, it is desirable to not only solve separate neurons in parallel but also to distribute the process of solving single neurons onto different processors. As was indicated in section 4.4.3, there is a high potential of parallelism in Gaussian elimination and back-substitution of neurons. It turns out to be hard to effectively exploit this parallelism for arbitrary numbers of cores, however.

First, four important requirements a parallel algorithm for solving the LSE must meet will be introduced.

1. Load balance between processors must be ensured.
2. Overhead for mutexes and time spent on waiting for other threads must be minimized.
3. Cache efficiency by means of prefetching must be optimized.
4. Cache-inefficiency due to false sharing must be minimized.

In general, requirement 1 and requirements 2-4 are contradictory; while a fine-grained assignment of compartments to processors is necessary in order to achieve a satisfactory balance between processors (see e. g. Hu's algorithm [33]), a coarse-grained assignment is necessary to guarantee cache- and inter-processor communication-efficiency. In addition, it is not clear how these four requirements should be weighted in order to achieve the best possible performance.

Therefore, classical scheduling algorithms for directed acyclic graphs fail; they do not take into account cache efficiency or overhead for inter-core communication but are usually based on the assumption that a vertex in such a graph takes a significant time to be processed. Although a

lot of time during this work was spent on trying to find an algorithm that meets all of the above mentioned requirements, the problem could not be solved.

In the course of this thesis, a second, yet unpublished method for single-neuron parallelization, *multisplit*, was implemented by Michael Hines that might, when employed properly, deliver satisfactory results regarding the four requirements for proper parallel performance (see section 5.4). This method was developed for message passing architectures, however; although applicable to shared memory architectures as well, it requires a very complex implementation and, currently, a high amount of user interaction. Because of this method's complexity and novelty, it was not taken into account for this thesis; in particular, a multithreaded implementation making use of the *multisplit* method probably requires several months worth of implementation[1]. Section 6 will further explain how *multisplit* might solve performance problems for single cell simulations.

Instead, the following sections will focus on a much simpler approach to parallelization, *cell splitting* - splitting the compartment tree representing a neuron at one compartment into as many subtrees as are connected to this compartment. The motivation for this approach is two-fold. First, the largest performance gain in Gaussian elimination is achieved by correctly distributing full neurons onto the available processors; in addition, this problem alone is NP-complete (see section 5.3.1). Secondly, the largest performance gain in neural simulations is achieved by a proper parallelization of mechanism computation as has been noted in section 5.2.1, so putting too much work into parallelizing Gaussian elimination is simply not worth the effort. Of course, for neural simulations with fewer cells than cores and relatively low requirements for mechanism computation, employing cell splitting still has a significant impact on performance.

The following section will explain how the cell splitting approach developed and programmed for this thesis works.

5.3.3 Single Neurons - Cell Splitting

In section 4.4.3, it was shown that numbering the compartments of a neuron starts with choosing an arbitrary compartment as the root compartment of a neuron's dependency tree; the different subtrees of this root compartment may then be first processed in parallel by the Gaussian elimination procedure. The primary thread then waits for all other threads to finish Gaussian elimination before back-substitution in the subtrees may take place in parallel again. Concurrent accesses to the variables representing the root compartment's equation must be avoided by using mutexes. This technique will be called *cell splitting* in the following. The approach of solving subtrees connected to a given compartment in parallel was developed independently by NEURON's main developer, Michael Hines, and the author of this thesis [2]. The main difference is that the algorithm presented below automatically identifies the root compartments of a cell in order to reduce required user interaction to a minimum.

The main question is how to choose a root compartment given a specific neuron. This section will present an algorithm that finds a proper solution for single cell simulations automatically.

[1] private conversation with Michael Hines
[2] private conversation with Michael Hines

The case of multiple cells will be handled in the next section.

Because the size of the largest subtree of the root compartment often governs the maximum workload on a core, it seems reasonable to identify a root compartment whose largest subtree is as small as possible. The algorithm presented here and used for performance benchmarks is to start at an arbitrary compartment and traverse the tree by descending into the largest subtree of each compartment. It stops when the size of the largest subtree of the current compartment is lower than or equal to half of the overall number of compartments. This is the compartment whose largest subtree with size t_k is smaller than or equal to all other compartments' largest subtrees.

Proof by contradiction:

$$t_k \in \{t_1, ..., t_m\}, t_k \leq \frac{n}{2} \Rightarrow \sum_{i \in \{1..m\} \setminus k} t_i \geq \frac{n}{2} - 1 \Rightarrow t_k \leq 1 + \sum_{i \in \{1..m\} \setminus k} t_i$$

Further descending into the largest subtree with size t_k would therefore result in a compartment whose largest subtree has

$$1 + \sum_{i \in \{1..m\} \setminus k} t_i \geq t_k$$

compartments, while descending into another subtree with size t_l would result in a compartment whose largest subtree has at least

$$1 + \sum_{i \in \{1..m\} \setminus l} t_i > t_k$$

compartments. Therefore, a compartment whose largest subtree has at most $\frac{n}{2}$ compartments is a compartment whose largest subtree is smaller than or equal to all other compartment's largest subtrees.

Figure 5.1 illustrates how an unnumbered graph (left) representing a neuron may be numbered such that the size of the largest subtree is minimal (middle). The resulting subtrees (colored, right) are then distributed onto the available cores with the same heuristic method that was introduced in the previous section for whole cell balancing. While Gaussian elimination may proceed simultaneously in the subtrees, all threads (at most three) must access the variables representing the root compartment's equation, which is therefore not assigned to any core and not colored. As has been noted above, this requires using mutexes.

This algorithm does a good job in meeting requirements 2-4 introduced in section 5.3.2. For a small number of cores, it may also deliver a proper load balance.

Figure 5.2 illustrates how a reconstructed VS2 cell, a neuron from the blowfly's lobula plate involved in visual input processing, may be split such that its largest subtree is minimal. The red part shows the approximate position of the root compartment, while the three colored regions show the subtrees that may be solved in parallel. The figure illustrates how the automatic splitting algorithm introduced in this section splits the approximately 5080 compartments into subtrees with about 2170 (yellow subtree), 1930 (green subtree) and 980 compartments (blue subtree), respectively. On examining the cell's structure, it becomes clear that the cell may not be split

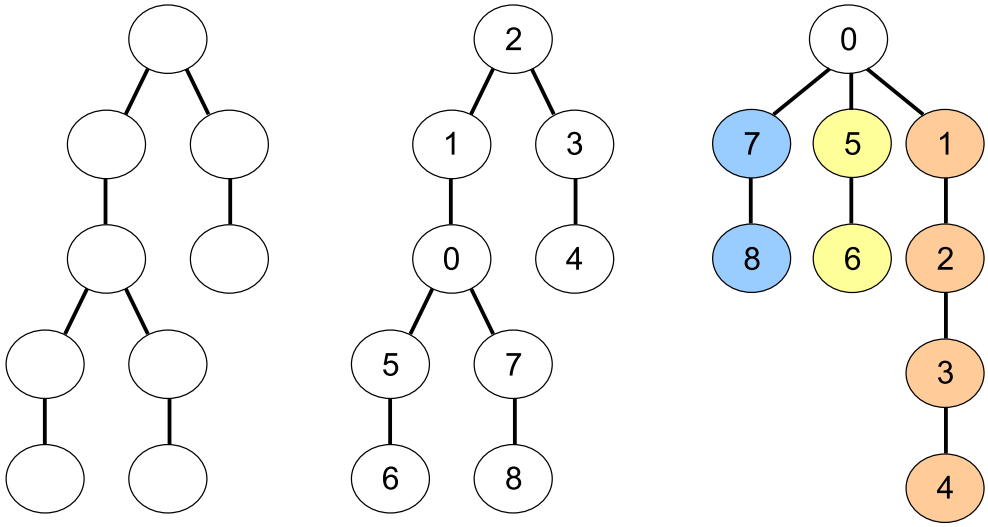


Figure 5.1: Left: Yet unnumbered acyclic graph representing the connectivity of some neuron. Middle: Same graph, numbered such that the largest subtree connected to the root compartment (compartment 0) is as small as possible. Right: Same graph and numbering as for the middle graph, but restructured and colored to emphasize the distinct subtrees that may be solved in parallel.

into two equally large pieces. Applying the algorithm to other cells from the VS system results a root compartment with up to four subtrees.

Before continuing, it should be noted that a lot of effort was spent on parallelizing Gaussian elimination - it became clear early that mechanism parallelization, while nontrivial from a technical point of view, was easy in an algorithmic sense because of missing data dependencies. However, exploiting the undoubtedly high amount of parallelism in tree-shaped data dependency graphs when, in addition to theoretical requirements, taking practical considerations such as cache efficiency and synchronization overhead into account as well, turned out to be a very challenging problem.

5.3.4 Combining Whole Cell Balancing and Cell Splitting

A more realistic scenario is simulating more than just one cell. Then, choosing a root compartment such that single cell Gaussian elimination is as efficient as possible may not be the best global choice, i. e. when taking all other cells and subsequent load balancing of whole cells and subtrees into account. In addition, trees cannot always be split such that the number of subtrees and the subtree sizes connected to the root compartment fulfill a certain requirement of the balancing algorithm used - even the rather simple requirement of choosing a root compartment such that its subtrees may be partitioned into two equally sized sets often cannot be met as figure 5.2 shows.

Due to these problems and the NP-completeness of whole cell balancing, a heuristic approach to combine cell splitting and whole cell balancing seems reasonable. One rather sophisticated

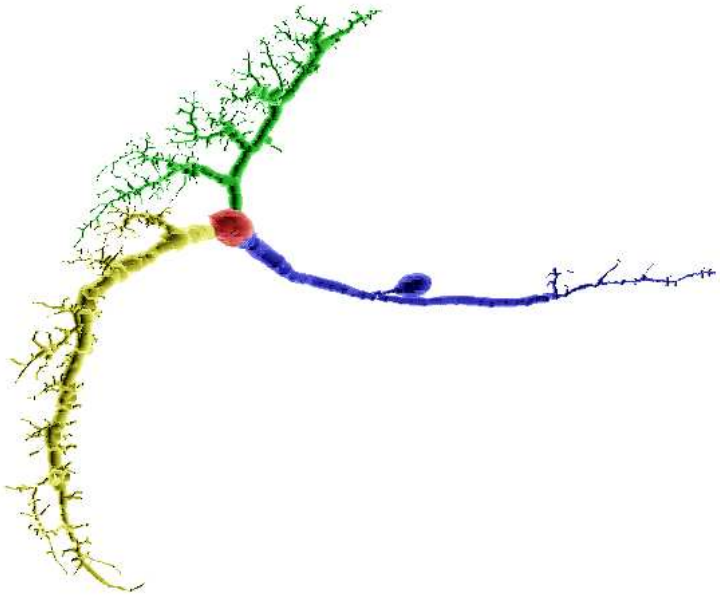


Figure 5.2: A VS2 cell from the blowfly's visual system, split such that the largest subtree is minimal. Red part indicates the approximate position of the root compartment, the blue, green and yellow colored trees mark the three subtrees that may be solved in parallel. The picture was generated by the Max-Planck-Institute of Neurobiology, Department of Systems and Computational Neurobiology.

method is presented in [23] where load balance for Gaussian elimination was of utmost importance because it was applied to mechanism parallelization as well.

The technique presented in this section is a combination of splitting neurons and distributing a number of neurons onto a set of processors. It was developed and used for performance measurements discussed in chapter 6. First, all cells are ordered according to their sizes. Then, the cells are split one after another, largest cell first, until the imbalance resulting from whole cell balancing of the subtrees of split cells and whole cells left is low enough, i. e. below a certain threshold. Here, the imbalance is defined to be the difference between the workloads (number of compartments assigned) of the core with the highest workload and the core with the lowest workload. In the implementation used for performance measurements, a maximal imbalance of 2% of the overall number of compartments was used.

This method makes sure that unnecessary splitting of cells is avoided because every split cell results in additional overhead for thread synchronization as was explained in section 5.3.3. An example of using this method will be shown in section 6.5.

5.4 Comparison to Other Approaches

5.4.1 Classification

To the author's knowledge, this thesis presents the first approach of using a threaded execution model for biophysically realistic neural simulations in parallel. Other implementations were primarily designed for message passing architectures, of course still allowing execution on shared memory architectures by using appropriate message passing implementations like MPICH's [17] `ch_shmem` device [18]. Consequently, one of the most important design goals of these approaches was to minimize inter-process communication. Specifically, the two stages of setting up the matrix and solving it are not treated as independent operations; instead, whole cells or subtrees of cells are assigned to a processor such that this processor both computes all mechanism instances of these cells/subtrees and solves them. Thus, the only data exchanged between processes are synaptic currents and, in the case of split cells, several floating point values concerning the split compartment's equation. The disadvantage of these methods is that potential load balance problems resulting from the rather restricting data dependencies of Gaussian elimination are applied to mechanism computation as well. Computing mechanism contributions for a specific compartment on a processor different from the processor that solves this compartment would require a certain amount of inter-process communication to transfer parts of the matrix between matrix setup and solving and to transfer result values (voltages) between solving and matrix setup for the next time step.

This thesis, however, is based on the assumption that inter-process communication, due to the restriction to shared memory architectures and multi-cores in particular, is fast enough compared to cluster architectures to not be a fundamental restriction for the design of parallel algorithms. The first consequence of this assumption and, to the author's best knowledge, novelty in parallel neural simulations is that parallelizing LSE setup and parallelizing the solver are viewed as independent problems. Therefore, the techniques for mechanism parallelization are novel approaches first presented in this thesis. Concerning parallelization of the solver, whole cell balancing is a technique that has been employed frequently in the last five to ten years. The notion of viewing the acyclic graphs underlying the matrix as dependency graphs of the Gaussian elimination and the possibilities following therefrom were obtained in the course of this thesis but, independently and simultaneously, were employed by Michael Hines in the *splitcell* approach as well (see below). The whole cell balancing technique (sort cells by size and assign them to processor with least load) is a rather common greedy algorithm frequently employed because of the known NP-completeness of number-partitioning.

Parallelizing an **existing** program is, in most cases, much simpler on shared memory architectures because all data (with the exception of *Thread Local Storage* and data on the stack) is global - often, parallelization of a program consists of only modifying a number of loops such that the range of the iteration variable is processed by the available threads, and introduction of semaphores where shared variables are accessed.

When programming message passing architectures, however, care must be taken not only of technical limitations like network bandwidth or latency, but the user must explicitly specify what data to send to which processing node at what point of the program. This can be a rather

challenging task for complex sequential applications and is one reason why the above mentioned approaches require a high amount of user interaction, thus limiting the messages sent between processing nodes to e. g. synaptic currents or synchronization operations.

Most importantly, this thesis presents methods that do not require user interaction but are designed with simplicity in mind like automatic load balancing and automatic cell splitting, while the methods explained in the next section all require a special environment with message passing libraries etc. and a high amount of user interaction for load balancing, specifically manual whole cell balancing and manual specification of split compartments.

5.4.2 Other Approaches

For the sake of completeness and in order to compare the design decisions made in this thesis to other implementations of parallel neural simulators, four different approaches will be presented in the following:

- PGenesis [58]

  Genesis uses *messages* to send axial currents to neighboring departments, to send conductance values of mechanisms to the respective compartments as well as to send spikes from one neuron to another. PGenesis, an enhanced version of Genesis, allows messages to be sent on top of PVM or MPI between different processors instead of just within one process. In addition, Genesis' default method for numerical solution of the ODE for each compartment is an explicit one, not requiring Gaussian elimination or a parallel version thereof (although implicit methods are supported by Genesis, as well). Therefore, it basically allows the user to choose arbitrary parallelization techniques; however, the manual recommends to execute large simulations in parallel only, in such a way that the amount of messages being sent between time steps is minimal, suggesting to simulate networks in a way that places whole subnets on single processors, only sending spikes between the processes ([59], [11]).

  In particular, no threaded execution model is used but only message passing environments such as MPI or PVM are supported. Most tactics introduced in this thesis should be applicable to Genesis, especially parallel mechanism computation and parallel calculation of the matrix-vector product resulting from Genesis's default explicit time stepping method. It must be noted, however, that the author of this thesis is not familiar with the internals of Genesis and the efficiency of its message exchange model.

- NEURON *ParallelNetManager*

  NEURON's first method for parallel neural simulations, *ParallelNetManager* [26], was a message passing based whole cell balancing approach. Specifically, whole cells are assigned to an MPI node; this node is then responsible for both setting up and solving the LSE for his cells. Nodes exchange synaptic currents by message passing.

  The main justification for this tactic is that it was the simplest way to implement parallel implementations and can easily be extended, as the following two implementations, *splitcell* and *multisplit*, will show. The main changes were enhancing the user interface to offer

message passing features to a user's scripts (like determining a node's rank in the network, calling remote script's procedures etc.) and enhancing synaptic mechanisms such that information about synaptic currents could be exchanged via MPI. Load Balancing must be done manually by the user via scripts that, depending on their MPI rank, load certain cells, and subsequently set up synaptic connections that transmit or receive data via the interconnect.

Allowing whole cell balancing only restricts the use of this method to relatively large networks, i. e. with a number of neurons much higher than the number of processors.

- NEURON *splitcell*

  In order to mitigate performance issues of whole cell balancing, an extension to ParallelNetManager, *splitcell* [23], [27], was developed. It allows splitting cells into at most two pieces with the same method that was introduced in section 5.3.3.

  The location where a cell is split must be specified by the user, however (in addition to specifying which cells or which parts of a cell are assigned to what MPI node). An interesting aspect of this method is that in order to properly split and balance cells onto nodes, a complexity proxy is used that estimates the computational requirements of solving a specific cell by performing dummy benchmarks for all mechanisms used before the actual simulation; this technique was discussed in more detail in section 5.2.4.

- NEURON *multisplit*

  multisplit [31], [25] is a very promising approach that is similar to *splitcell* but allows splitting cells into many pieces by using multiple split points. This is achieved by a rather complex modification of the default method for Gaussian elimination that also adds computational complexity. In most cases, the increase in complexity is very low compared to what is gained by much better load balancing. This methods allows for proper load balancing in networks of neurons even when the number of cells is equal to or lower than the number of MPI nodes.

  The method is not yet published, and the modifications to Gaussian elimination are far too complex to describe them in this section.

All these methods are based on message passing architectures and primarily focus on the simulation of large networks. These thoughts are the main reason why the approaches introduced differ rather strongly from the work presented in this thesis.

Chapter 6

Results

This chapter first provides information about practical problems encountered during this thesis and limitations resulting therefrom, in particular why an existing neural simulator, NEURON, had to be used instead of a stand-alone implementation and a short description of problems encountered with NEURON.

The next part of this chapter deals with technical details of the implementation such as thread control and thread synchronization as well as the types of processors, software and techniques used for measurements.

The third part finally presents a number of measurements, each combined with a discussion about the model and the results obtained.

6.1 Practical Considerations

6.1.1 Problems with Using a Stand-Alone Implementation

This thesis concentrates on the algorithmic and time-intensive part of neural simulations. A stand-alone implementation of the numerical core of neural simulations comprising LSE setup and solving is obviously the best way for trying out different parallelization techniques, different data layout tactics and detailed benchmarks. Developing on top of an existing implementation such as NEURON, on the other hand, may be complicated and restricting because of numerous side effects induced even by small changes to one part of the source code. For instance, changing the layout of data structures accessed by multiple parts of the program may require changing access strategies in many places. Accordingly, modifications to the mechanism data allocation scheme to improve cache efficiency turned out to be very difficult. Similarly, changes in a function's parameter list require adaption of all calling definitions of this function, something that may become a real hassle when function pointers are involved that expect a certain interface. Finally, even small changes often propagate through larger parts of the program recursively.

Therefore, a stand-alone implementation, *fNeuron*, of the numerical core of NEURON was programmed first. However, in order to evaluate the different techniques and combinations thereof presented in this paper, realistic, published models had to be simulated to grasp the

wide range of real applications a multithreaded implementation of a neural simulator must be able to handle efficiently. Importing arbitrary or, at any rate, complex models turned out to be impossible. There is no unified language or data format completely describing arbitrary neurons and neural mechanisms, and even if such a language existed, it would be very difficult to write a program being able to handle the full extent of the language's descriptions simply because of the complexity of neurons and neural mechanisms such a language must be able to describe. NEURON uses its own languages to describe models/mechanisms and to allow the user to control the simulation. A neural model description for NEURON consists of two parts:

- A number of files specifying mechanisms in a special language, `NMODL` [29]. NEURON features a compiler, `nrnivmodl`, that takes `NMODL`-descriptions as input files and generates C files which are then compiled with a common C compiler, forming a shared library that is used during simulation. The interface these C files offer and rely on (the function signatures as well as the data structures) are highly NEURON-specific.

- A number of files specifying the morphology of neurons, which mechanisms are to be applied where and further commands controlling the simulation. These files are written using `hoc` [12], a language similar to many programming languages supporting loops, variables, pointers etc.

Due to the complexity of both `NMODL` and `hoc`, it was not possible to implement support for either of them in a stand-alone implementation. An initial attempt to still simulate complex neurons and networks with fNeuron was still performed by modifying NEURON such that after parsing all `hoc`-files, the resulting matrix (representing the exact topology and axial resistances) and additional data like the surface areas of all compartments were printed to a file which could subsequently be imported by fNeuron.

What was missing, however, was the support for arbitrary mechanisms. While three basic mechanisms, namely for current injection, passive ion channels and Hodgkin-Huxley type channels were implemented for fNeuron using the C files emitted by `nrnivmodl` as a basis, it turned out to be too much work to implement more mechanisms for fNeuron. For instance, the large model used in section 6.6 uses about 30 mechanisms. In addition, synaptic mechanisms require a complex spike queue management that fNeuron does not support.

Therefore, after first working on fNeuron, the decision was made to recycle the efforts spent on fNeuron by backporting parallelization techniques developed for fNeuron to NEURON as far as possible. This also corresponded more to a long-term aim of this thesis, multithreading support for NEURON.

6.1.2 Parallelizing Mechanism Code

In order to allow for parallel computation of mechanisms, the C code emitted by `nrnivmodl` must meet two criteria:

1. Support for processing a specified subset of this mechanism type's compartment list, for instance indicated by start and stop indices, as opposed to the single-threaded case where the whole compartment list is processed by a single thread, and

2. Thread safety of functions that are possibly called by different threads simultaneously.

While the first requirement resulted in rather manageable enhancements to `nrnivmodl`, the second requirement, thread safety, turned out to become one of the main problems encountered during the thesis.

The emitted C code of most models makes heavy use of global variables to pass parameters at function calls. In order to allow two or more threads access to the same mechanism type's functions simultaneously, all write and read accesses to global variables of this mechanism type that are modified by a thread had to be eliminated; this required the modification of functions to accept these global variables as parameters passed by the calling function as well as the modification of calling definitions.

The two main problems that made it impossible to implement these far-reaching changes in the limited time available were:

- Global variables and functions in the emitted C code that use these variables are not only due to internals of `nrnivmodl` but, in many cases, defined as global variables by the user in the `NMODL`-file describing the mechanism. Automatic conversion of such variables to local variables by `nrnivmodl` would also require identification and modification of user-defined functions that make use of these variables.

- The `NMODL` language is very feature-rich, and `nrnivmodl`'s source code is, similar to the rest of NEURON, very complex and thus difficult to modify. Significant effort was spent on understanding and modifying the source code of `nrnivmodl`, but implementing the required changes for multithreaded execution by a person not fully familiar with its source code is a cumbersome task that would take a month or more even for a skilled programmer.

Because it seemed nevertheless necessary to simulate a very large model in order to illustrate the performance of the implementation, a different approach was chosen: The changes to the interface of functions that are called by NEURON (support for start and stop indices indicating a range within the compartment indices list) and the loops over the compartments as well as some minor, multithreading independent additions to improve cache efficiency were added to `nrnivmodl`, and the `NMODL` files were compiled in order to gain access to the emitted C code. The C code was then manually modified in order to change global variable references to local ones and convert mechanism-internal functions such that these variables are now passed as parameters.

This procedure was difficult and very time-consuming and is the only reason why only a limited set of models could be simulated and presented in this chapter. However, modifications to `nrnivmodl` are currently planned to use *BLAS* (*Basic Linear Algebra Subprograms* [50], [1]) vector operations for computing a mechanism type's contributions to the LSE. These far-reaching modifications would also require removing write accesses to global variables and, as a side-effect, result in probably full thread-safety of the emitted code.

6.2 Technical Details

6.2.1 POSIX Threads and OpenMP

Thread creation and control was implemented using the NPTL (Native POSIX Thread Library, [15]) implementation of the POSIX threads standard defined in [35], in the following referred to as *pthread* or *pthreads*. One reason for not using OpenMP [56] was that support for it was added to GCC [2] (NEURON's default compiler; see section 6.2.6 for a note about using other compilers) only recently, while due to the timeframe of the thesis, possible shortcomings in GCC's OpenMP implementation *GOMP* could not have been tolerated. Another reason was that using pthreads directly did not add significant complexity to the implementation compared to OpenMP but allowed for more control during development, while GOMP itself maps OpenMP pragmas to `pthreads` functions, anyway [55]. For instance, defining the behavior of secondary threads between parallel regions (sleeping or busy waiting), dynamically adapting the working set for a thread (done during load balancing) or processing parallel regions that exhibit strong data dependencies (such as the solver) instead of simply distributing a loop's iteration variable's range were all strong indicators for using pthreads during the design and development phase. Later versions might, however, use OpenMP exclusively or a combination of OpenMP pragmas and further multithreading-specific functions (such as binding a thread to a specific core) instead for reasons of improved portability.

6.2.2 Thread Creation and Control

An already existing initialization function, `finitialize()`, was modified in order to start the secondary threads and allocate and initialize several multithreading-specific data structures for thread synchronization and workload distribution.

The basic technique employed for thread-level parallelization, a slightly modified Fork&Join model, has already been introduced in section 5.1. Code that is subject to parallelization, a parallel region, was encapsulated in functions (referred to as `f()` in the following) providing a specific interface that allows multithreaded execution of these functions. The primary thread then executes the program in a single-threaded manner; when encountering a parallel region, it invokes the secondary threads such that `f()` is executed in parallel.

Processing such a function `f()` in parallel usually consists of the following steps:

1. The first step, if not done previously, is for the primary thread to wait for all secondary threads to become idle, i. e. having finished computation of any previous parallel functions. This step may be omitted in case it has already been performed, e. g. at the end of the previous parallel region. The exact implementation of this, the third and the last step of this list may have a significant influence on parallel performance and will be discussed more thoroughly in section 6.2.3.

2. For every secondary thread, the primary thread fills a thread-specific shared data structure with a function pointer to the aforementioned parallelized function `f()` and further information about the data the function must process (i. e. information about a set of cells

and subtrees, or variables identifying a range within a list of compartments or mechanism instance indices). The kind of parameters depends on characteristics of the parallel region and is explained directly after this listing The data structures used for exchanging data between the primary thread and the secondary threads are declared as `volatile` to force the compiler to emit code that accesses main memory (or the caches) on every access to these data structures.

3. The primary thread virtually wakes up the secondary threads (see section 6.2.3 for details about this procedure), making these threads call `f()` with the parameters specified in the shared data structure.

4. The primary thread calls `f()` itself, thereby participating in the parallel computation.

5. After the primary thread has returned from computing `f()`, it usually (depending on the concrete context within the program) waits for all secondary threads to finish computation and become idle again.

There are two different kinds of parallel functions. The first one are functions that process a set of compartments but do not impose any order on how these compartments are processed, i. e. there are no inter-compartmental data dependencies. These functions take as parameters an array containing all compartment indices that must be processed in this time step, followed by the start and stop index of the subset in this array the calling thread must actually process. For example, functions computing mechanism current contributions or capacitive contributions belong to this class.

The second kind of parallel functions processes either whole cells or subtrees of a split cell, i. e. functions that operate on a per-cell or per-subtree level. Examples are the triangularization and back-substitution functions; for cache efficiency reasons, however, other functions that do not require working on a per-cell basis may still be executed for whole cells or subtrees only. These functions take as a parameter a pointer to a data structure not only containing cell indices but also flags indicating what cells are split so the function may prevent concurrent accesses to the root compartment by using mutexes.

6.2.3 Implementing Thread Waiting and Thread Notification

The previous section explained the five steps involved in computing a parallel region. In steps one and five, the primary thread waits for all secondary threads to become idle, while in step three, the primary thread transmits information to and wakes up the secondary, idle threads.

These two operations were implemented using a shared synchronization variable (declared as `volatile`) for each secondary thread. These variables indicate the state of secondary threads and are therefore initialized with a value of `IDLE`[1]. All secondary threads are waiting for their variable to get assigned the value of `BUSY` by the first thread, indicating that the function specified in the previously mentioned shared data structure must be processed. After having called

[1] In the source code, numerical values are used directly instead of the BUSY or IDLE macros; these are used here in order to facilitate understanding of the explanations and the source code excerpt below

and returned from this function, a secondary thread resets its synchronization variable to `IDLE`, thereby allowing the first thread to determine that it has finished processing the given function.

An important design decision is how waiting on the synchronization variable, by both the primary and the secondary threads, is exactly performed. There are two solutions that were implemented: busy waiting on the variable and *pthread condition variables* [35].

Busy waiting makes a thread spin-wait on the synchronization variable until it is changed by another thread. Although the waiting thread is actually not performing any useful computation, it still occupies a CPU as much as possible. Busy waiting may consume much more CPU time than necessary, but it has the advantage of being the fastest method because it does not involve any library or operating system calls.

Using the method of condition variables (not to mistake with the actual synchronization variable which is still used), on the other side, notifies the operating system via a library function that the thread is waiting for a specific signal from another thread. The operating system's scheduler may then suspend the execution of this thread, possibly scheduling other processes or threads and thus use a CPU more efficiently. When another thread signalizes a change, the operating system may reschedule one of the threads waiting for this signal. In short, using condition variables is an operating system visible way of thread synchronization allowing for a more intelligent and fairer scheduling. The downside of condition variables is that synchronization overhead is introduced; this is a significant factor if the time spent in parallel regions is rather small, increasing the ratio of synchronization time to parallel computation time.

```
pthread_mutex_lock(mut[t]);
while (sync_var[t] != BUSY) {
    pthread_cond_wait(cond[t], mut[t]);
}
pthread_mutex_unlock(mut[t]);
//process parallel region
...
pthread_mutex_lock(mut[t]);
sync_var[t] = IDLE;
pthread_mutex_unlock(mut[t]);
pthread_cond_signal(cond[t]);
```

Figure 6.1: Code for a secondary thread, waiting for a signal from the first thread using pthread condition variables

Figure 6.1 lists the code approximately how it is used by the secondary thread `t` to wait for a signal from the primary thread with condition variables. Waiting on the shared variable `sync_var[t]` is done in a loop that is left only once the primary thread has changed the value from `IDLE` to `BUSY`. The code of the primary thread for signaling the secondary thread that a parallel region must be processed looks similar and is therefore not listed here.

The code for busy waiting is an exact copy of the code listing in figure 6.1 except that is lacks the function calls for locking or unlocking mutexes and signaling or waiting on condition

variables.

6.2.4 The MC_NRN Environment Variable

All changes to NEURON are enclosed by preprocessor conditionals to allow for both simple identification and activation/deactivation of the mutithreading specific changes. All multithreading specific settings such as the number of threads, the method of thread waiting/notification, the method of mechanism parallelization etc., are specified using an environment variable, `MC_NRN`. This variable contains a comma-separated list of name-value pairs of the form `PARAM=value`, which is parsed during initialization and consists of the following parameters and values:

- `NT` - this parameter allows the user to specify the numbers of threads used, including the primary thread.

- `BW` - setting this parameter to `1` enables busy waiting as the method used for thread notification and waiting, while setting it to `0` makes the program use of condition variables. These two methods were discussed in detail in section 6.2.3.

- `MP` - this parameter, an abbreviation for *mechanism parallelization*, can be set to either `cs` (static compartment level parallelization, that is, without dynamic load balancing), `cb` (compartment level parallelization with load balancing) and `mt` (mechanism type level parallelization).

- `SC` - setting this parameter to `0` disables cell splitting, while using a value of `1` enables cell splitting such that cells, ordered by sizes, are split until a desired workload balance between the cores is achieved or all cells have been split. Details about this algorithm may be found in section 5.3.4. For most measurements, cell splitting was enabled, and apart from negligible overhead during initialization, cell splitting does not influence the runtime negatively even when the algorithm finds that no cells need to be split. It will only be disabled in section 6.4 to illustrate how using cell splitting can improve performance over not using cell splitting at all.

- `UPC`, `MRPC`, `APC` and `AM` - these parameters are specific to NEURON and are only noted for the sake of completeness. In short, they specify whether computing the new voltage vector out of the LSE's result, resetting the matrix diagonal and right hand side to zero and integrating axial conductances into a compartment's equation are performed in parallel on a per-cell basis or by simply splitting up the set of compartments into equally large sets for each core. These parameters are set to values of `1`, `0`, `1` and `0`, respectively, during all simulations; other parameter combinations were tried but found to result in equal or higher runtimes, mainly due to the low computational complexity but significant impact on cache efficiency of these operations.

- `CA` - this parameter, an abbreviation for *core assignment*, allows the specification of the cores (using a colon-separated list of CPU IDs) the threads will be scheduled on. For

instance, specifying `CA=2:0` will schedule the primary thread on CPU 2 and the secondary thread on CPU 0.

All figures in this section will use the above introduced abbreviations (except for `UPC`, `MRPC`, `APC` and `AM` which are fixed) to distinguish performance results. However, specifying directly the IDs of the cores that were used would not be helpful because it does not indicate whether these cores share a cache or not. Therefore, the figures will not indicate the exact thread-to-core assignment but instead note the cache architecture used for tests with the abbreviations `SHC` (*shared cache* - all cores access a common, shared cache), `SPC` (*separate cache* - every core has its own, private cache) and `HYB` (*hybrid* - some subsets of cores share a cache, others do not). Further information about the architecture, such as a more detailed description of which cores share a cache in the `HYB`-case or which cores are located on the same or on different physical chips will, if necessary, be noted when discussing the respective benchmark results.

6.2.5 Measurement Problems

In order to analyze the performance and identify bottlenecks of a multithreaded application on multi-core architectures, it is indispensable to perform measurements on different combinations of cores in order to measure the behavior of the application in dependence of the cache architecture, most notably with respect to inter-core communication.

Especially evaluating the performance of synchronization methods (busy waiting versus condition variables) and the cost of using locking algorithms for cell splitting require manually specifying the desired cache architecture by setting the processe's CPU affinity. This allows, for instance, to compare the cost involved in synchronizing two processes that run on cores with either shared caches, separate caches on the same chip or even when the cores are located on separate physical chips (with separate caches).

After programming the routines for thread waiting and thread notification as explained in section 6.2.3, a major problem with multithreaded applications occurred - under certain circumstances, there were large variances in the execution time of multithreaded applications. Particularly for specific combinations of a model, the number of threads and the specific thread-to-core assignment, the time spent on synchronizing threads varied significantly between consecutive measurements. These problems could be reproduced with a simple test program that may be found in the appendix, section B.

The problems occurred when using the tools `taskset`, `numactl` or specifying the CPU affinity mask of the whole process (a bitmask containing all cores that may be used by the threads in this process) by the primary thread in the source code using the `sched_setaffinity()` function. A solution was finally found - each thread sets is CPU affinity mask separately by specifying only the CPU it is designated to run on.

Extensive measurements not shown here using the test application strongly indicate that the operating system thread dispatcher or scheduler is the reason. Most importantly, observing the CPU usage for each core with `top` while running the test program revealed that certain specified cores are simply not used by the scheduler, e. g. only three of the four cores specified are used. In addition, the problem did not exist when not specifying a CPU mask at all but allowing

the operating system to select arbitrary cores for execution. It was not possible to examine the problem more closely during the thesis.

6.2.6 Hardware and Software Environment

Hardware, Operating System and Compiler

All benchmarks were performed using Ubuntu Linux 7.04 (kernel: `2.6.20-16-server`) on one of the following two computers:

- 2x Intel Xeon X5355 (codename: *Clovertown*; quadcore with 2.66GHz, 2x4MB L2 cache, FSB1333 (333MHz)), 8GB fully buffered DDR2-SDRAM
- 2x AMD Opteron K10 2347 (codename: *Barcelona*; quadcore with 1.910GHz, 4x512KB L2, 1x2MB L3 cache), 8GB fully-buffered DDR2-SDRAM

Time measurements were performed by reading a core's timestamp counter.

The C compiler used is the *GNU C Compiler* [2] (`gcc`, version 4.1.2), the default for compiling NEURON. Using the Intel C compiler reportedly works (although not for compiling NEURON with GUI support) but was only attempted once during this thesis and lead to problems with the build process. The Intel C compiler is known to produce very efficient code; however, it does not help in evaluating the quality of the parallelization approaches tested in this section. Therefore, only little time was spent on trying to fix the Intel C Compiler related issues, and `gcc` was used for all benchmarks presented in this chapter; no other compilers such as the PGI C compiler [63] or the PathScale C compiler [57] were tested.

For all measurements performed in this thesis, ten simulation runs were performed, and the average of these runtimes are displayed in the corresponding figures. No significant variations in runtimes for a specific measurement were observed. It is indicated in the title of each figure which of the two test systems/architectures was used. The two test systems will be referred to simply as *Clovertown* and *Barcelona*, respectively. The correctness of the results obtained with the multithreaded version of NEURON was confirmed by comparing the computed voltage values of some compartment to those obtained with the single-threaded version.

NEURON specific notes

NEURON 6.0.4 was used as basis for the modifications performed. Although NEURON comes with a graphical user interface (GUI), this is an optional part not mandatory for simulations. Rather, the GUI may be used for displaying results during the simulation and for simulation control. The simulation itself is performed independently from the GUI. Therefore, the modified NEURON used in this thesis was compiled without GUI support.

Multithreading independent cache optimizations, partially related to layout of mechanism data (see the `cvode.cache_efficient(1)` `hoc` command and the `REORDER_MECHDATA` macro in the source files) were implemented in NEURON and applied to models before simulation, first. These optimizations were employed for both single-threaded and multithreaded

measurements. In particular, mechanism data for every mechanism type is reallocated such that it is stored in a consecutive manner.[2] Without these optimizations, mechanism data is (depending on the order of compartment creation via `hoc`) often scattered across memory, and it would have been difficult to identify the reason for bad speedups in the development phase.

Results denoting overall execution time refer to the accumulated time spent in `fadvance()`, NEURON's function for advancing one time step including matrix setup and solving the LSE. That is, time spent on the initialization of NEURON or the model, such as parsing the files describing the model, loading mechanisms and allocating or initializing internal data structures was **not considered**; first, these parts are highly NEURON specific; second, they normally constitute only a small part of the overall runtime; third, this thesis focuses on the algorithmic part of simulations, only.

NEURON internally also uses zero-area compartments - compartments without mechanisms and capacitance. In short, these compartments are used for numerical accuracy reasons; details may be found in [12]. The number of compartments used to represent a neuron (and therefore processed during solving) is therefore greater than the number of compartments specified by the user (and therefore, number of compartments with mechanism instances). The number of zero-area compartments is indirectly specified by the user (it is equal to the number of *sections* the user creates, see [12]) and highly model specific; in general, this kind of compartments comprises between 5% and 25% of all compartments. Numbers of compartments specified in this chapter, unless otherwise noted, always refer to the overall number of compartments, including zero-area compartments.

Finally, it is very important to note that the models selected for evaluation of the implemented algorithms naturally comprise a restricted subset of possible neural models, only. Although there are databases that offer a large amount of neural models for NEURON (see [24], [4]), the main problem was that mechanisms had to be manually modified to support multithreaded execution because of the complexity of `nrnivmodl` and the time limit for this thesis (see section 6.1.1). For instance, the cortical column model used in this thesis required manual modifications to about 30 mechanisms which took about five days. Therefore, the models used here were chosen carefully such that a spectrum as wide as possible is covered.

6.3 Influence of Cache Architecture, Cache Size and Model Size

Benchmark results of parallel programs obtained on multi-core architectures must be interpreted with caution. Processor-independent resources like memory bandwidth and I/O performance, do not scale with the number of cores used (as opposed to e. g. cluster architectures). While these natural restrictions of multi-core architectures may not affect many multithreaded programs at all,

[2] This reallocation procedure is not performed for all mechanism types; for two mechanism types, AlphaSynapse and NetStim, this procedure was always disabled because either simulation results were wrong or segmentation faults occurred. The reason for these errors could not be identified; however, only a small number of mechanism instances of these two mechanism types are used in the models presented in this thesis, so the negative effect is probably negligible.

they play a strong role in the area of scientific computing - in design considerations, performance results and, most importantly for this section, a meaningful interpretation of these results.

Shared-cache architectures allow for fast inter-core communication and therefore for a more fine-grained task decomposition, but they are problematic when it comes to comparing single-threaded execution to multithreaded execution because the cache size available to one core is, apart from shared variables and code, virtually divided by the number of cores. This often results in sublinear, unsatisfactory speedups on shared-cache architectures whereas separate-cache architectures, due to the doubling of cache capacity, even allow for superlinear speedups. One might interpret this as a perceptional problem; sub-linear speedups on shared-cache architectures often stem from the fact that the single-threaded runtime used as a reference was achieved by one core using the cache capacity that is normally used by two or more cores.

This section aims to illustrate how cache size, cache architecture, inter-core communication, memory requirements and model complexity interact. Therefore, a series of example models was created that do not represent a real cell or network but are solely used for illustration of how the parameters mentioned in the last sentence interact in general. The model consists of 10 cables without branches with a total of n compartments that were divided up into 10 equally large parts. The simulation lasted for $\frac{1024000}{n}ms$ using a time step of $dt = 0.025ms$. Thereby, the overall computational work required was held approximately constant regardless of the number of compartments used, attenuating the influence of communication costs and memory requirements/cache size. The computational requirements are not totally constant because the overhead for inter-core communication and sequential regions depends on the number of time steps.

Due to the equal sizes of all cells, the cell splitting algorithm did not have to split any cells at all to achieve a proper balance; instead, whole cell balancing delivers a perfect workload balance. The influence of cell splitting on runtime will be discussed in later sections.

Figure 6.2 shows the results of simulation runs of this model type performed on *Clovertown*. The upper part illustrates measurements obtained when simulating computationally demanding Hodgkin-Huxley type channels on every compartment, whereas the results shown in the lower part were obtained by simulating passive channels, only. In addition, computationally cheap current injection mechanisms were applied to each cell. For each model, cache architecture and number of threads, ten simulations were performed, and the averaged value of the time spent in `fadvance()` is displayed in units of seconds (*s*).

The results were obtained using the modified single-threaded version of NEURON (1x4MB L2; black), the multithreaded version with two threads assigned to two cores with a shared cache (`SHC`: 1x4MB L2; blue), with two threads assigned to two cores with distinct caches on the same chip (`SPC, 1 chip`: 2x4MB L2; yellow) and with two threads assigned to two cores with distinct caches on different chips (`SPC, 2 chips`: 2x4MB L2; orange); busy waiting (`BW=1`) was used for thread-waiting and notification. Mechanism computation was parallelized using the compartment level method with load balancing (`MP=cb`) (see the following section about a more thorough discussion of mechanism parallelization).

As noted above, the computational requirements (the model complexity; roughly $n \times steps = const.$) are approximately equal in all cases. Figure 6.2 allows for five important observations:

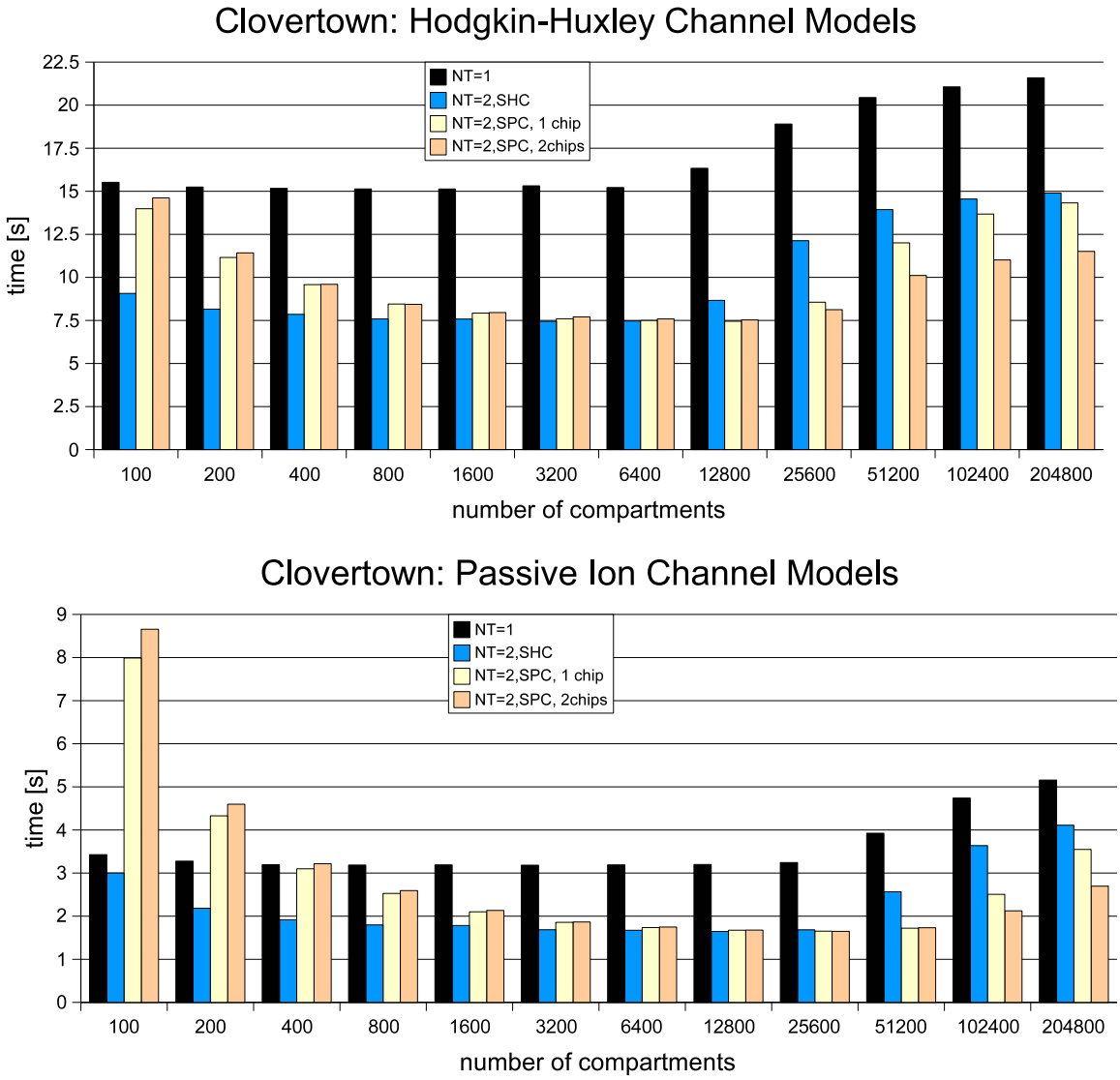


Figure 6.2: Simulation results for a model with ten cells depending on mechanism type, number of compartments and cache architecture/size. Number of time steps was chosen so that computational requirements are constant. Upper graphic shows results for the number of non-zero-area compartments (every model contains 10 zero-area compartments without mechanisms, in addition) indicated on the abscissa, distributed over 10 cells, every compartment assigned a mechanism for Hodgkin-Huxley type channels. Simulations were performed on one core, on two cores with shared caches, on two cores with separate caches on the same chip and on different chips. Lower figure: Same as upper figure, but for passive ion channels, resulting in lower computational and memory requirements for each compartment, attenuating the effect of inter-core communication.

1. The lower the number of compartments, the more time steps are simulated, increasing the amount of synchronizations. At the same time, the overall cache capacity (4MB for two cores with a shared cache, 8MB for two cores with separate caches) does not influence runtime for low numbers of compartments because the whole model and the simulation code fit into the cache(s). Accordingly, the performance of the dual-threaded program on cores with a shared L2 cache is superior to the case of two cores with separate caches, depending on the amount of inter-core communication. This holds true for both the Hodgkin-Huxley model and the passive ion channel model. Overhead for inter-core communication is so high in the case of 100 or 200 compartments that the runtime is even significantly higher for two threads when simulating passive ion channels, only. The difference in runtime between both architectures initially shrinks with increasing numbers of compartments because the amount of inter-core communication is reduced.

2. With increasing model size, i. e. increasing number of compartments and decreasing number of time steps, the amount of inter-core communication decreases, whereas memory requirements increase. This becomes apparent for larger models by the higher parallel performance of two cores with separate caches as compared to two cores with a shared cache - while separate caches result in slower inter-core communication, they practically provide the program with double the cache size, which is more important for larger models than the speed of inter-core communication. This, of course, does not indicate that, for a given cache size, separate-cache architectures are superior to shared-cache architectures for larger models; rather, it emphasizes the importance of cache-capacity in general.

3. The best choice of the underlying cache architecture (if this choice is available to the user in the first place) not only depends on the number of compartments but also on the overall complexity of the model. Particularly, both the computational complexity and the memory requirements for a mechanism instance have an important influence. E. g., the discrepancy between runtime on the two architectures for 12.800 or 25.600 compartments is much clearer for the model using Hodgkin-Huxley because of higher memory requirements and a smaller ratio of communication to computation costs.

4. The fourth observation is that the speedup, the ratio of sequential to parallel runtime, is highly model- and architecture dependent. While in most cases, slightly sub-linear speedups may be observed when comparing the dual-threaded with the single-threaded results, there are also cases of super-linear speedups (when the model fits into two separate caches but does not fit into one cache) as well as situations where, due to extremely high synchronization overhead, the single-threaded results are better than the multithreaded results on two cores with separate caches.

5. The fifth observation that can be made is that cores that are placed on different physical chips allow for better performance for large models because they do not have to share a front side bus. This becomes apparent when comparing the results for the two cases when the cores do not share a cache but are located either on the same chip or on different

chips. At the same time, the difference in runtime for small models shows that inter-core communication between chips on one core is faster than between chips on two cores.

In summary, figure 6.2 illustrates the difficulties in evaluating the quality of a multithreaded implementation of a neural simulator, because other factors such as cache size, cache architecture and the actual model that is used strongly influence performance results. In particular, design decisions often require thoughtful consideration of possible architectural limitations as well as average model complexities.

The following sections will further evaluate the performance of the implementation for different models with different computational requirements and discuss where the characteristics of the given result, be it superlinear speedups or nonsatisfying performance in parallel, stem from. In order to cover the vast area of neural models a parallel neural simulation software should be able to handle, these sections will exemplary examine three model size regimes: *small models* that fit entirely into the cache, *medium size models*, and *large models* where memory requirements are so high that cache size or cache architecture are no more a determinant, but memory bandwidth may become a limiting factor. The notion of small, medium or large models does not follow a strict definition but is solely introduced in this thesis to characterize the three extremes of neural models where either inter-process communication, cache size or processor and memory performance are the major limiting factors.

6.4 Small Models

This section will examine the behavior of the parallel implementation for rather small models where the required data, including code, mechanism data and variables describing the LSE, entirely fit into the L2 cache of a core. Although memory consumption mainly depends on the mechanism types and the number of mechanism instances used, in general models with less than 1000 compartments belong to this class. These models constitute the majority of neural models in reality. Typically, but not necessarily, models that are created manually (as opposed to models that are created in an automatic or semi-automatic manner from photographs of e. g. dye-filled cells) exhibit a rather simple morphology consisting of up to 200 compartments.

Simulating such models is, in most cases, not computationally demanding, and they are therefore not the major target of this thesis. In general, small models are employed in one of the two scenarios denoted below.

- First, such models are mainly used in an interactive manner by users who try to reproduce some previously observed behavior in real cells or try to understand certain physiological characteristics of the original cell. Here, some kind of interpretation of results and/or adaption of model parameters is performed between two simulation runs, and the simulated time interval (and thus the runtime of the simulation) is rather small. In short, time spent on the simulation itself is in general short enough to render a parallelized simulation unnecessary. In most cases, the simulation is so short that the user would not even perceive improvements.

- A computationally more demanding scenario for small models is automatic parameter fitting. Here, a possibly large number of simulations is carried out, dynamically adjusting unknown parameters by reusing results from previous simulation runs. Many optimization algorithms for multi-dimensional, non-linear problems exist, for instance genetic algorithms [16] or simulated annealing [47]. One possible parallelization approach for optimization is to compute several, independent simulation runs in parallel instead of parallelizing single simulation runs; this technique may be applied when there are multiple parameter sets to be evaluated, see e. g. genetic algorithms. When multiple simulation runs cannot be executed in parallel, the efficiency depends on the speed of the parallel neural simulation implementation, however.

Therefore, small models often are not worth being executed in parallel. Still, this section will take a closer look at a rather "big" small model, the simulation of a hippocampal CA1 pyramidal cell consisting of roughly 760 compartments (including about 160 internal zero-area compartments without mechanisms). This NEURON model was used by Migliore et al. [54] to simulate the interaction of back-propagating action potentials and sub-threshold synaptic inputs in the context of a special type of potassium channels (A-type conductances [46]) that are expressed at high density in hippocampal dendrites[3]. The original model [53] was slightly modified in order to enable several multithreading-independent cache optimizations that were not a part of NEURON at the time the model was created. In addition, the `NMODL` files describing the ion channels were slightly adapted to ease the process of making the code thread-safe (changing all `GLOBAL` variables in the `NMODL` file to `RANGE` variables) without changing the semantics of the model. The required data during simulation, i. e. code, data for mechanisms and the LSE, fits into the cache (estimations give about 500KB cache usage). Cell splitting gives three subtrees with sizes 376, 9 and 376 compartments, respectively.

In the context of this thesis, an extremely interesting characteristic of this model is the heterogeneous distribution of mechanisms across the cell. While all compartments feature the mechanism for passive ion channels, only compartments with a diameter bigger than $0.5\mu m$ and a distance from the soma of up to $500\mu m$ also are assigned sodium and potassium channels with voltage dependent conductances. This makes the model an ideal candidate to evaluate the quality of the load balance algorithm proposed in section 5.2.4. In fact, load balancing was not planned to be implemented before using this model simply because no other models simulated during this thesis exhibited a strong enough heterogeneity in spatial mechanism distribution that would have required load balancing.

Figure 6.3 illustrates performance measurements of the CA1 model for the three different types of mechanism parallelization - compartment level parallelization with (`MP=cb`) and without using dynamic load balancing (`MP=cs`), and mechanism type level parallelization (`MP=mt`). All measurements in this and the following sections were performed with (`BW=1`) and without (`BW=0`) busy waiting. At first, it becomes obvious how big the influence of the thread waiting and notification method is for rather small models. Using busy waiting is far better than using condition variables in all cases (in contrast, busy waiting will have nearly no visible effect on runtime in section 6.6 where inter-core communication comprises only a small part of the runtime). This

[3]The cell was simulated with the synaptic conductance set to 0 as shown in figure 1c in [54].

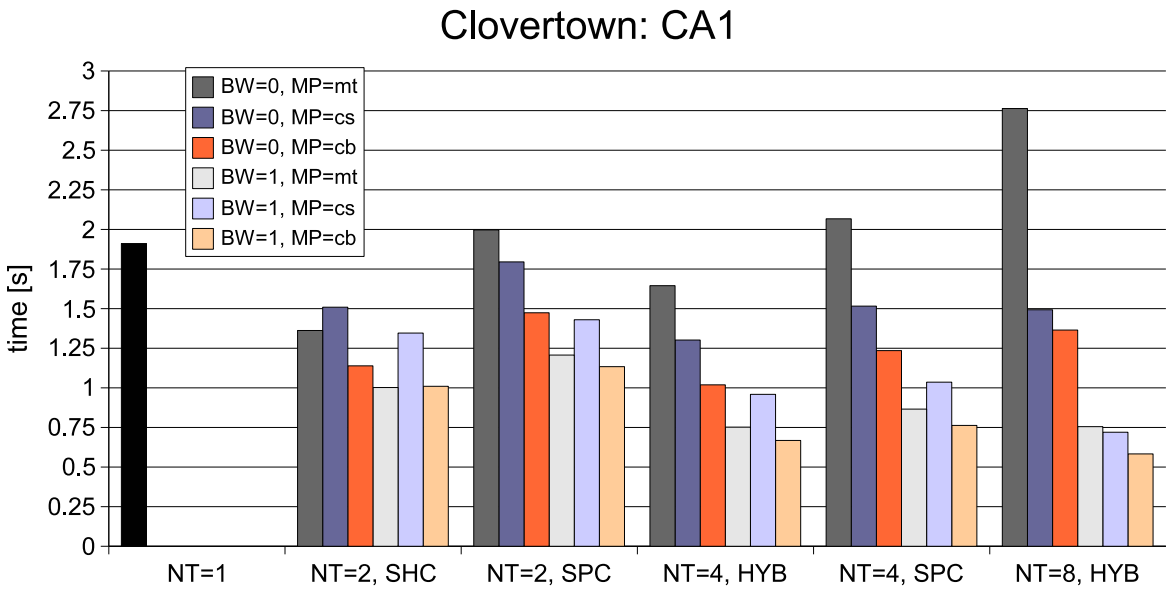


Figure 6.3: CA1 measurements on the *Clovertown* test system with the following three tactics for mechanism computation parallelization: mechanism type level parallelization, compartment level parallelization without load balancing and compartment level parallelization with load balancing.

effect is attenuated when using separate caches instead of shared caches. Increased cache size (as gained by using separate cache architectures), in contrary, does not decrease runtime because the model and the code fit into the cache of a single core, anyway.

One of the most interesting aspects of figure 6.3, however, is the influence of either using or not using the dynamic load balancing implementation of the compartment level mechanism computation stage. Regardless of the underlying architecture (shared or separate caches) and the thread notification/waiting algorithm, dynamic load balancing allows for much lower runtimes. In the case of using two threads with a shared cache and busy waiting, not using load balancing but simply splitting up the set of compartments into two equally large sets leads to such a large imbalance that the first thread spends $780ms$ only on waiting for the second thread. In contrast, using load balancing reduces the time spent on waiting to about $50ms$ for both threads, only; the time spent on load balancing itself is negligible (about $130\mu s$). Consequently, the runtime is reduced from $1350ms$ by about the half of the time spent on waiting, $340ms$, to $1010ms$, which is a nearly linear speedup compared to the sequential runtime of $1910ms$.

The figure also depicts the measurements when not using compartment level parallelization at all but mechanism type level parallelization. The results reveal that compartment level parallelization with load balancing results in runtimes similar to the mechanism type level parallelization approach with shared caches and busy waiting. At the same time, the main disadvantage of mechanism type level parallelization becomes apparent: the increased amount of synchronizations necessary (one after each mechanism type; the model uses 9 distinct mechanism types)

leads to very high runtimes when either condition variables (`BW=0`), separate caches (`SPC`) or a combination of these two are used. For higher numbers of cores, mechanism type level parallelization results in runtimes much higher than the single-threaded runtime. However, when using busy waiting, the results are comparable to those obtained by using compartment level parallelization with load balancing.

In short, in all cases, compartment level parallelization with load balancing provides the lowest runtimes and is the best choice regardless of whether using `BW=0` or `BW=1`.

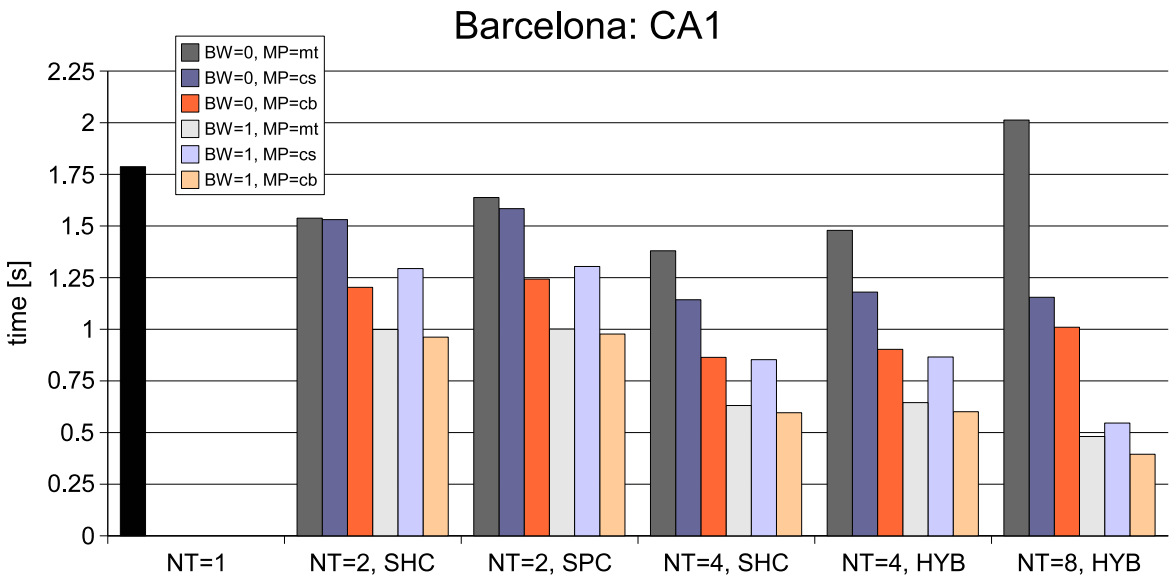


Figure 6.4: CA1 measurements on *Barcelona* with the following three tactics for mechanism computation parallelization: mechanism type level parallelization, compartment level parallelization without load balancing and compartment level parallelization with load balancing.

The same series of benchmarks was also performed on *Barcelona*, depicted in figure 6.4. The main finding from the *Clovertown* benchmark results, the superiority of compartment level parallelization with load balancing, is confirmed. However, for higher numbers of cores, the *Barcelona* architecture delivers much better results, an observation that can be made in the following tests as well. At the same time, the negative effects of inter-core communication on mechanism type level parallelization are not as strong as on *Clovertown*, probably because inter-core communication is faster due to the underlying cache coherency protocol, *MOESI* (see section 6.5).

6.5 Medium Size Models

This section will exemplarily investigate simulator performance for two models. The cells constituting these models can be found in the *lobula plate* in the visual system of blowflies. Compartmental models were reconstructed out of dye-filled cells using two-photon microscopy [14].

The two models used are the *V1* cell and the *VS network* [21] consisting of the ten consecutively numbered *VS cells*, *VS1*, ..., *VS10*. Neighboring cells are connected by electrical synapses [19]. Each of these cells is comprised of about 3600-14500 compartments (including zero-area compartments).

The simulation of V1 will employ the Hodgkin-Huxley mechanism and therefore illustrate the case when mechanism computation constitutes the majority of the runtime.

The VS cells, in contrast, are simulated using passive ion channels only. Although the VS cells have voltage- and ion concentration-dependent ion channels [9], they are thought to encode information using graded potential shifts instead of action potential frequency.

6.5.1 V1

The V1 cell is a spiking neuron responding (with an increase in firing rate) preferably to downward motion in front of its receptive field. Therefore, the reconstructed cell is simulated using the Hodgkin-Huxley mechanism and a current injection. It is a rather detailed model, consisting of about 14600 compartments (including approximately 2100 zero-area compartments) which is split into three subtrees with 7121, 7067 and 415 compartments, respectively, by the automatic cell splitting algorithm.

V1 with 2 Threads

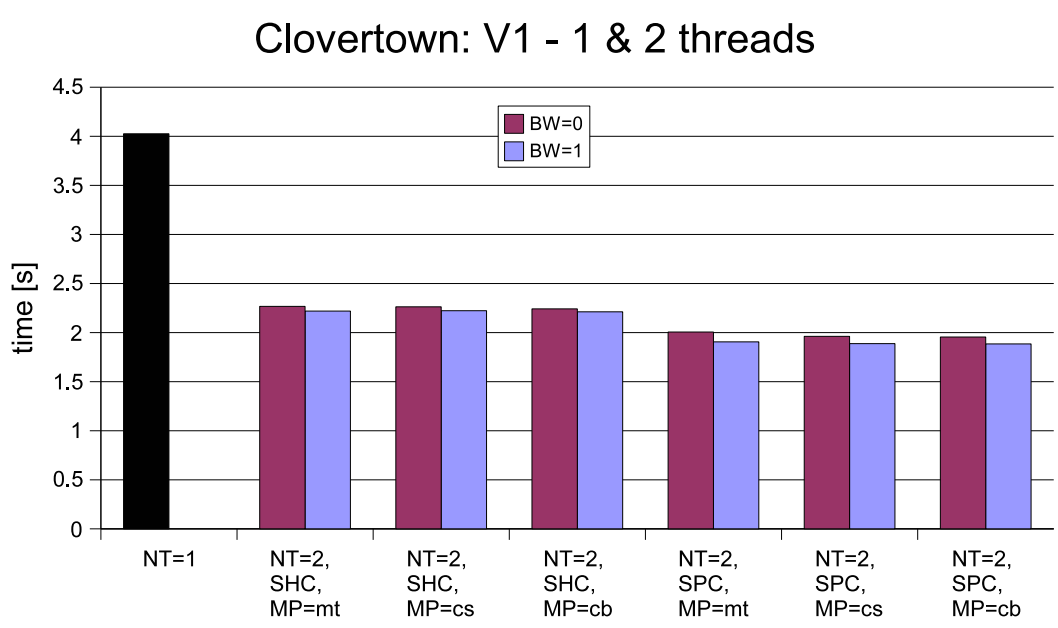


Figure 6.5: V1 simulation for $20ms$, $\Delta t = 0.025ms$. The results show the runtime of the single-threaded version and the multithreaded version using two cores for several combinations of cache architecture and mechanism computation parallelization method.

Figure 6.5 illustrates that mechanism computation load balancing is not necessary for this model but, at the same time, does not influence runtime negatively. In addition, the figure shows that using mechanism type level parallelization does, even for models where only one or two mechanisms are used (here: Hodgkin-Huxley and a current injection), not have any advantages over compartment level parallelization with load balancing. Therefore, in the following sections, only results obtained with compartment level parallelization and load balancing (`MP=cb`) will be shown. Although measurements with the other two methods were also performed, these results will not be shown because the `MP=cb` method proved to be equal or superior to the other two methods in all cases.

Figure 6.5 also illustrates that the choice of using busy waiting or not does not have such a big influence as it had for the small model in the last section; however, cache size does. In short, figure 6.5 confirms that compartment level parallelization with load balancing is equal or superior to all other approaches and how cache size and communication costs influence runtime for larger models.

Cell splitting and Inter-Core Communication

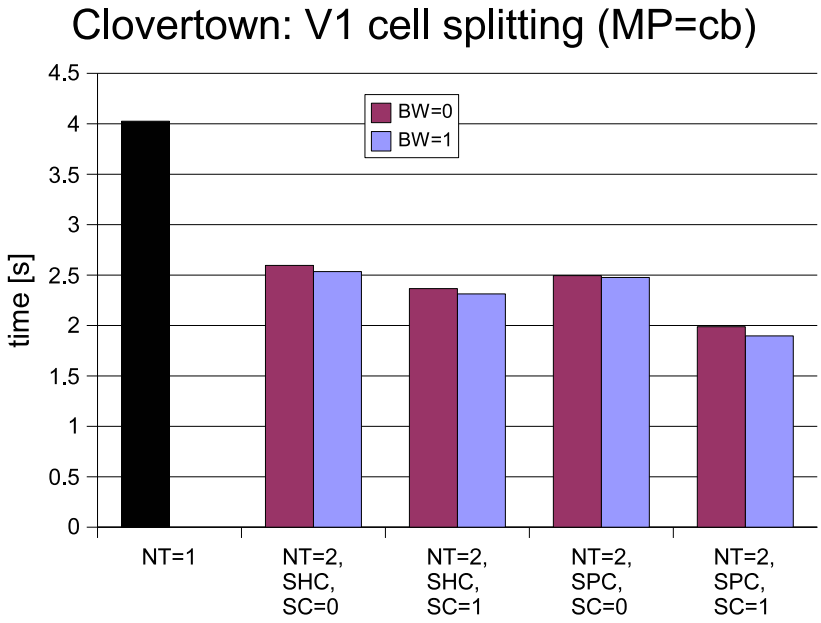


Figure 6.6: Simulation of the V1 model for $20ms$, $\Delta t = 0.025ms$ on two cores with or without using cell splitting. The results show that although solving does not comprise a big part of the runtime, not using cell-splitting still leads to performance degradation when the two cores have separate caches because of inter-core communication.

In the single cell measurement results shown so far (CA1 and V1), the automatic cell splitting algorithm was enabled (`SC=1`) and decided to split the respective cell in order to achieve a

better balance for the solver. Figure 6.6 opposes the benchmark runtimes obtained with and without cell splitting applied to the V1 model. When the two cores share a cache, enabling cell splitting reduces runtime only slightly because the time spent on solving the LSE is rather low compared to what is spent on mechanism computation. Cell splitting is much more important if the mechanism computation stage does not constitute the big majority of the runtime.

If both cores access separate caches, however, cell splitting influences the runtime much more. This demonstrates a basic problem of the principle of decoupling LSE setup and solving - if a compartment's mechanism instances are computed on a different core than where this compartment is processed during solving, and if these two cores do not share a cache, then communication between the caches is required and may strongly decrease performance.

In particular,

- the core that computes the mechanism instances of a compartment for time t usually requires the voltage value of this compartment from time $t - \Delta t$ which is either in main memory or in the cache of the core that processed this compartment during solving, and
- the core that processes the compartment in the solver stage requires the diagonal and right hand side element of this compartment's equation which is either in main memory or in the cache of the core that computed this compartment's mechanism instances.

Off-diagonal elements will be neglected in the following explanations because they are not changed during simulation.

What exactly happens when a core requires data that can be found in another core's cache depends on the underlying cache coherency protocol. For people not familiar with cache coherency protocols used in modern multiprocessors such as the *MESI* [40] protocol employed by Intel processors or the *MOESI* [5] protocol used by AMD's Opteron processors, a short comment about transferring data from one cache to another cache must be made first. If a processor i (i. e. its cache) tries to read a variable whose most up-to-date version is in processor j's cache which is distinct from processor i's cache, processor j will interrupt processor i's load operation from main memory first. When using the MESI protocol, processor j writes back the variable (to be precise, a whole cache line) to main memory and subsequently signals processor i that it may continue loading the value from main memory. If the MOESI protocol is used, processor j transfers the variable to processor i directly without writing it back to main memory. If both processors share a cache, however, no main memory accesses are involved regardless of whether MESI or MOESI is used.

The first three columns of figure 6.7 illustrate how cache architecture and not using cell splitting interact. The left column shows a cell with the root compartment chosen such that two equally large subtrees could be solved in parallel; spatial mechanism distribution is assumed to be homogeneous as in the V1 model. The colors illustrate how compartment level parallelization divides up the compartments into distinct sets when two cores (first core: light and dark blue, second core: light and dark orange) or four cores (each core one color) are used for mechanism computation. The two graphics in the middle illustrate with colors and gradients what cores process and what caches contain the above mentioned LSE data (diagonal elements, right hand side elements, voltage variables) if the cell is **not** split for the cases of two cores sharing a

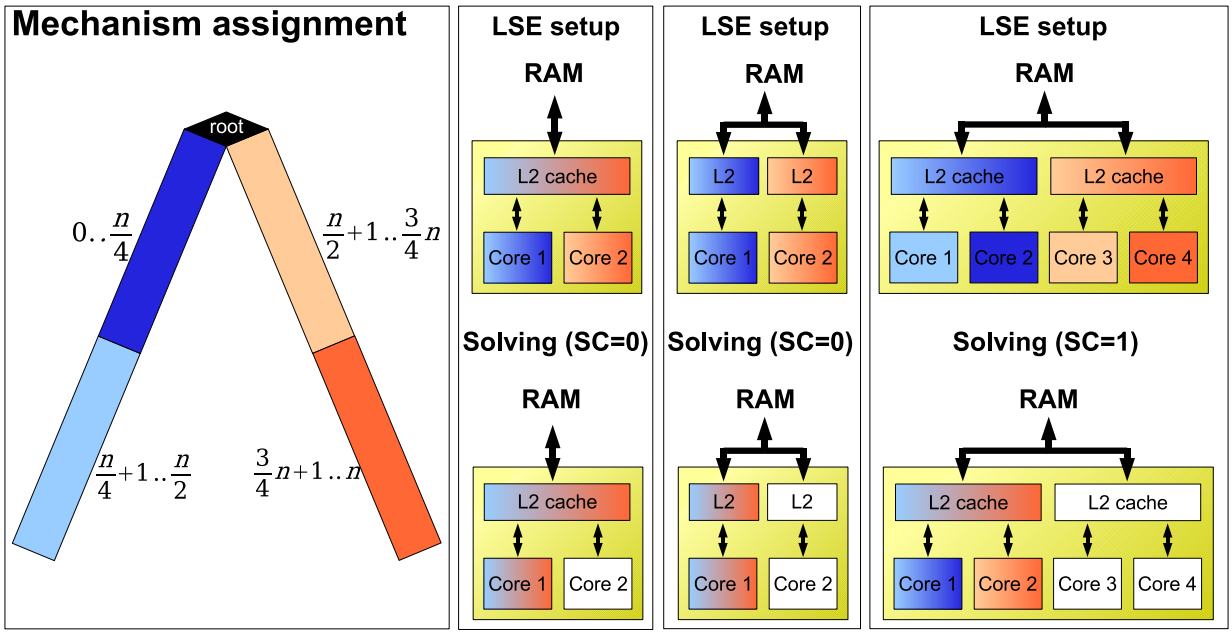


Figure 6.7: Placement of compartment specific data in caches and cores for a single cell (left graphic) when two cores either sharing a cache or having separate caches are used with cell splitting disabled (middle graphics), or four cores with a hybrid cache architecture are used with cell splitting (using a consecutive assignment) of subtrees to cores is enabled (right graphic).

cache or not. When using separate caches, data needs to be transferred between the two caches, something that is not required when both cores share a cache.

This section will not discuss this issue in more depth because the details are quite complicated and strongly depend on the actual cache coherency protocol and its implementation.

Instead, another case that requires a high amount of inter-core communication will be shown. The right part of figure 6.7 illustrates how using cell splitting may still fail to solve the problem for hybrid cache architectures such as a combination of several multi-cores with shared caches like the *Clovertown* processor. The obvious solution to the problem depicted in this graphic would be to schedule the process of solving the second subtree not on the second but on the third core. This requires knowledge of the exact cache topology, however, and still fails if all four cores have separate caches. The next section will examine this optimal assignment.

To make a long story short, the tactic of decoupling LSE setup and solving may lead to problems when

- a compartment's mechanism instances are computed on a different core than where the compartment is processed during solving and when
- these two different cores do not share a cache and therefore (in the case of MESI) have to communicate via main memory, and when
- model complexity is so low that mechanism computation does not comprise the big majority of runtime.

As the following measurements show, this issue does not play a big role for V1 when four threads are used, but measurements not shown here confirm that for single cell simulations with a rather low mechanism complexity (e. g. simulating passive channels only), significant performance problems may occur.

V1 with 4 Threads

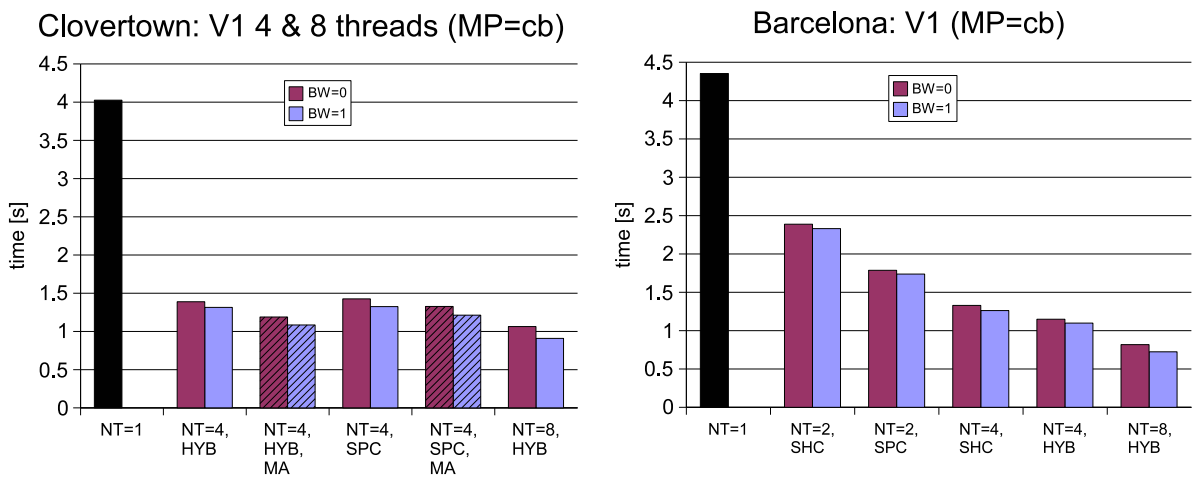


Figure 6.8: Left figure: V1 simulation on *Clovertown* for $20ms$, $\Delta t = 0.025ms$ using one thread, four threads with the default assignment method for mechanism instance subsets and subtrees, four threads with the optimal assignment method implemented manually (MA), and 8 threads. Right figure: Same simulation on *Barcelona* with the default assignment only.

The left part of figure 6.8 opposes the single threaded execution time with measurements obtained with four cores. The first two quad-threaded measurement results show the runtime for the V1 simulation when using four cores such that two cores share a cache, respectively.

The second one of these results (marked with `MA` for *manual assignment*), however, was performed with a patched version of NEURON that simply exchanges the cells assigned to the second thread with the cells assigned to the third thread - the resulting assignment is best illustrated by the right part of figure 6.7 with the workset of the second and the third core being exchanged in the solver stage, only.

As predicted, this manual re-assignment of subtrees allows for a better runtime because inter-core communication is drastically reduced. Even when none of the four cores share a cache (`SPC`), this manual assignment method reduces runtime slightly because the (now inevitable) large amount of inter-core communication is now happening between cores on the same chip instead of cores on different chips.

The *multisplit* approach has already been introduced in section 5.4; this approach has the potential to overcome these problems. However, it is yet unclear how the increase in complexity for the solver will influence runtime and how well automatic cell splitting can be combined with this approach.

Using all 8 cores on *Clovertown* does not increase runtime significantly. Similar benchmarks were performed on *Barcelona*, shown in the right part of figure 6.8. Although the performance in the single-threaded case is not as good as on *Clovertown*, the runtime on *Barcelona* scales better for higher number of cores. It is noteworthy that the L3 cache shared between four cores on the *Barcelona* system does not, contrary to expectations, solve the issue of inter-core communication described above, while using two caches, each being used by two cores, allows for better runtime than using one shared cache. This is probably due to a combination of the increased cache size and the more efficient inter-core-communication scheme on Opteron processors where the MOESI protocol introduced above does not require main memory accesses when exchanging data between cores.

6.5.2 VS Network

The VS network consists of all ten VS cells, giving a total of approximately 60.000 compartments. Although the number of compartments is very high, the model is simulated with passive ion channels only, such that due to the rather low memory requirements, it is characterized as a medium size model. Approximating the memory requirements by taking into account the number of compartments, the memory needed for the LSE, mechanism data and further internal data structures gives about 5.5MB, much less than what the large model presented in the next section requires. Neighboring cells are connected with gap junctions [19]. The model is derived from Cuntz et al. [13]; it is simulated for $50ms$ with a time step length of $\Delta t = 0.025ms$.

Figure 6.9 illustrates performance results of the VS network for various benchmark scenarios. The small difference between using busy waiting and condition variables in the case of two or four threads shows that the computational requirements of the model are very high compared to inter-core communication. For two cores, whole cell balancing gives a ratio of roughly $29900 : 30200$ compartments, and the automatic cell splitting algorithm decides to not split any cells at all.

The low amount of inter-core communication combined with the memory requirement mentioned above (the data probably does not fit into one cache of 4MB but fits into two such caches) of the VS network result in strongly sublinear speedups when two cores with shared caches are used; the *Barcelona* test system achieves much better runtimes in this case. On the other hand, these characteristics of the model also lead to linear or superlinear speedups when running the simulation on two cores with separate caches, regardless of the whether busy waiting is used or not.

In the case of two cores, no cells must be split to obtain a satisfactory load balance. The situation is different when four cores are used; then, whole cell load balancing (i. e. without splitting cells) leads to a balance (measured in compartments) of roughly $16600 : 16200 : 14000 : 13200$ for the four cores, respectively, so cell splitting seems to be more reasonable. Indeed, enabling the automatic cell splitting algorithm results in the two largest cells being split, giving a balance of approximately $15200 : 15100 : 1500 : 14900$. Accordingly, enabling automatic cell splitting allows for slightly lower runtimes, as can be seen in figure 6.9.

When using 8 cores, whole cell balancing results in a load of roughly $9200 : 9000 : 8900 : 7600 : 7200 : 6200 : 6200 : 5800$, so cell splitting becomes even more important. The automatic

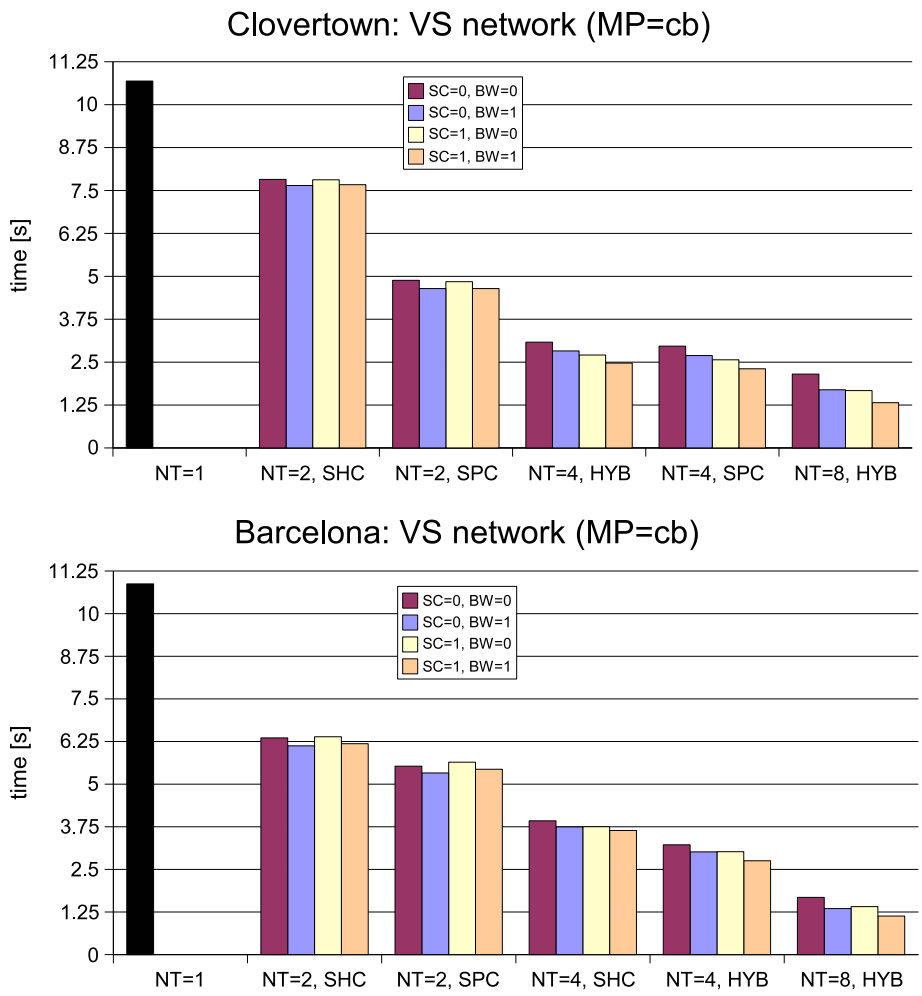


Figure 6.9: Benchmark results for VS network simulations for $50ms$ and a time step length of $\Delta t = 0.025ms$. Upper part shows results obtained on *Clovertown*, lower part shows results obtained on *Barcelona*.

cell splitting algorithm stops after splitting six of the ten cells, reaching a balance of approximately $7900 : 7700 : 7700 : 7700 : 7300 : 7300 : 7300 : 7200$. The advantage of splitting cells is well reflected by measurement results obtained with 8 threads (two cores sharing a common 4MB L2 cache, respectively) - with cell splitting and busy waiting, even using 8 threads results in super-linear speedups.

When comparing the results obtained on *Clovertown* to those obtained on *Barcelona*, it becomes apparent that cell splitting is more important for the *Clovertown* system. The reason is probably that the negative effect of not using cell splitting on inter-core communication (between LSE setup and LSE solving as well as between LSE solving and LSE setup) is more emphasized on *Clovertown* because the MESI protocol requires main memory accesses instead of directly passing the values between the processors as performed on the *Barcelona* architecture where

MOESI is used.

In addition, the *Barcelona* test system exhibits better scaling of the runtimes on 8 cores where a speedup of approximately 9.6 is achieved, whereas the *Clovertown* test system allows for a speedup of approximately 8.1, only.

6.6 Large Models

In order to prove that the algorithms presented in this thesis may be applied to large models as well, the thalamocortical network presented in [65] was simulated. About 30 mechanisms for different kinds of ion channels and synapses are used. The original model [64] consists of 3560 cells and takes about 500s to simulate with a single thread; the model was modified by Michael Hines to support execution with only a tenth of the original number of cells, i. e. 356 cells, in order to allow for faster benchmarks. Both model versions were tried to be run with the multithreaded NEURON version. However, for the original model, calls to the `malloc()` function failed in the spike queue management. The author is neither familiar with the spike queue management, nor did conventional debugging further indicate where the problem stems from exactly. The `errno` variable only indicated that the system was out of memory, although enough physical memory was available.

Therefore, the small version of the model was used, which is comprised of about 50.000 compartments. The automatic cell splitting algorithm finds satisfactory balances without splitting cells at all.

Figure 6.10 shows the results for several combinations of number of cores and cache architecture on *Clovertown*. The first thing one can notice is that the runtime does not scale properly with the number of processors. This is due to the high memory requirements of the model in combination with bandwidth restrictions of a chip's front side bus. There are several indications for this. First, using either shared or separate caches for two threads on the same chip does not make a big difference in runtime, but using two cores with separate caches on different chips results in a much higher performance than using two cores with separate caches on the same chip. Basically the same observation can be made for four cores: Cache size, again, does not play an important role - using 2x4MB L2 cache for four cores, two cores on each chip, or using 4x4MB L2 cache for four cores, two cores on each chip, does not have a great influence on runtime. However, using 2x4MB L2 for four cores, all located on the same chip, results in a performance even below that of two cores on different chips.

Another indicator for a chip's front side bus as the limiting factor is that while significant speedups may still be observed when comparing single-threaded runtime with dual-threaded runtime and dual-threaded runtime with quad-threaded runtime in several cases, there is no real improvement in using 8 cores over 4 cores unless the cores used for the quad-threaded measurements are all located on one chip. The very small differences between using busy waiting or condition variables shows that the problem is not related to inter-core communication.

It is possible that main memory bandwidth in general, i. e. the bandwidth the northbridge can provide to both chips, is also a limiting factor. Estimating the required amount of memory by taking into account space required for mechanism data, the LSE and internal data structures which

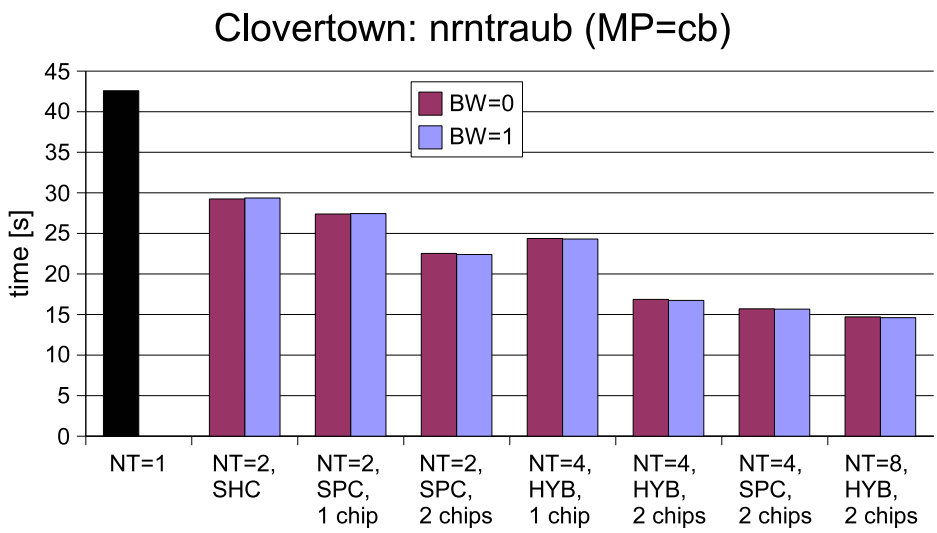


Figure 6.10: Simulation runtimes of the reduced Traub model for $10ms$ with $\Delta t = 0.025ms$ for various combinations of number of cores and cache architecture on *Clovertown*. The results indicate that the front side bus shared between cores on the same chip is a limiting factor and that neither cache size nor busy waiting play an important role because of the model size.

are all required in one time step, and dividing this by the amount of time spent for one time step gives an approximate average bandwidth of 1.3GBytes in the single-threaded case. Considering four or more threads, however, and the fact that the above mentioned value is only an average over time and not a peak bandwidth approximation indicates that main memory bandwidth also plays a significant role.

However, the limited time for this thesis did not allow for more detailed benchmarks.

Performance measurements were performed on the *Barcelona* test system as well; these results are illustrated in figure 6.11. Again, the *Barcelona* architecture allows for better scaling of the runtime with the number of cores, although the performance in the single-threaded case is not as good as that of the *Clovertown* test system. The most interesting result shown in figure 6.11 is the significant increase in runtime when using eight instead of four cores, something the *Clovertown* system could not provide. One might first assume that the NUMA architecture of *Barcelona* is the reason for these results. Both chips have their own memory controller and memory modules, although a core on a chip may access remote memory modules by communicating with the remote memory controller via a *HyperTransport* [34] connection between the chips.

However, exploiting the advantages of NUMA systems usually requires a special kind of memory allocation scheme that takes care of data being placed on memory modules connected to the chip that accesses the data most frequently. Such a NUMA-aware allocation of mechanism and LSE data was not programmed. The Linux operating system also supports the so-called first-touch policy that places specific pages of memory in the vicinity of the core that first accesses a previously allocated page. However, parts of the data of a mechanism are usually initialized (i.

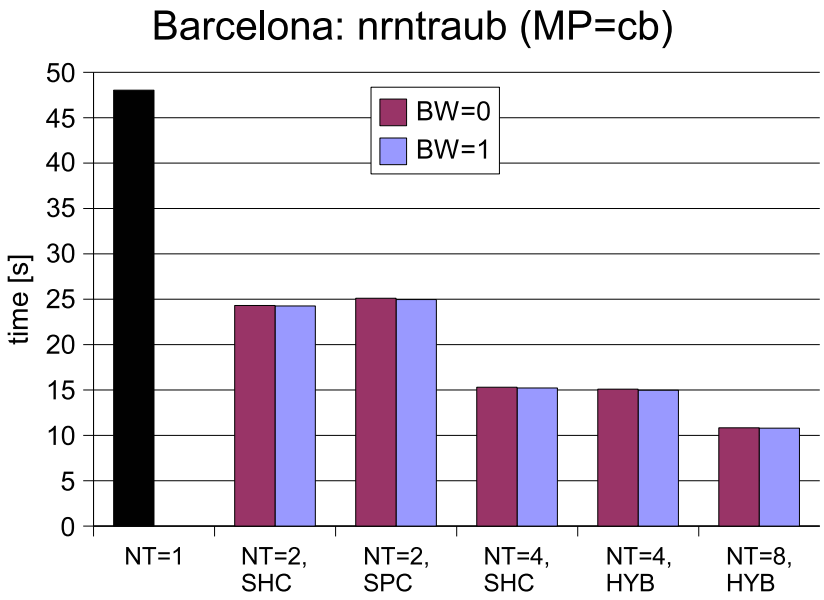


Figure 6.11: Simulation runtimes of the reduced Traub model for $10ms$ with $\Delta t = 0.025ms$ for various combinations of number of cores and cache architecture on *Barcelona*.

e. touched for the first time) by the primary thread after reallocating it which leads to most of the mechanism data being placed on a memory module connected to the chip/core the primary thread runs on.

Another reason why the NUMA architecture is unlikely to be the reason for much better scaling as obtained on *Clovertown* is that there is practically no difference in the runtime of the two cases when two threads were located on either one or two chips. The fact that using separate caches even results in slightly higher runtimes although time spent on inter-core communication is very low indicates that one core indeed has to perform many memory accesses on a RAM module that is connected to the other chip. There is also no difference in execution time when four threads are either all placed on one chip or two threads are scheduled on each of the two chips, another indicator that the NUMA architecture is not the reason for the speedup between using 4 and 8 threads.

These measurements were performed only at the end of the time available for this thesis, and the author did therefore neither program NUMA-aware allocation nor examine the exact data placement more closely. In addition, a conclusion originally drawn after performing benchmarks on *Clovertown*, namely that memory bandwidth did not allow for runtimes significantly lower than $15s$, was refuted by the much better results obtained on *Barcelona* because the memory bandwidth of these two systems does not differ enough to allow for such big differences.

Many points of the above discussion therefore remain speculation and should be confirmed or refuted by more detailed measurements and changes to the mechanism data allocation scheme.

Chapter 7

Conclusions

This thesis presented several new approaches to parallel simulation of biophysically realistic neural networks on shared-memory architectures with a focus on chip-level multiprocessors and using implicit integration methods for the ODEs involved. The following list will summarize the main propositions and findings:

- If inter-core communication facilities allow for low latency and high bandwidth, most notably by employing shared caches, or if the computational complexity of the model in one time step is so high that overhead for inter-core communication is negligible, then the process of setting up the linear system of equations (LSE) in parallel and the process of solving it in parallel may be regarded as independent tasks. Previous approaches like *splitcell* [27] or *multisplit* [25] (see section 5.4) that focus on message-passing architectures instead strive to reduce the amount of inter-process communication and therefore do not decouple these two tasks in order to minimize the amount of data comprising the matrix that must be transferred between compute nodes.

- Dynamically resizing the sets of compartments assigned to cores based on measurements of the actual workload on these cores works surprisingly well and allows for proper workload balance of the LSE setup stage between cores even when the overall complexity of the model is distributed over the compartments in a highly inhomogeneous way. Such a dynamic load balancing technique is especially suited for shared-memory architectures, whereas resizing working sets on message-passing architectures usually is a complex task.

- Solving the LSE may be parallelized by splitting up the cell at arbitrary compartments, a finding also presented in [23] and exploited in the *splitcell* approach [27]. This thesis introduces an algorithm to automatically identify compartments where a cell should be split based on the objective of minimizing the size of the largest resulting subtree of this compartment. Automatically splitting single cells is combined with a simple algorithm that automatically distributes the resulting chunks (subtrees and whole cells that were not split) onto the available cores, with the aim of reducing the total workload imbalance.

- For most models, busy waiting on shared variables delivers significantly lower runtimes than using operating system visible synchronization methods like condition variables. This

is because the parallelization techniques that were used, while allowing for a fine-grained task decomposition, require a high amount of inter-core communication for thread notification, waiting and synchronization.

All these proposals were implemented by enhancing an existing program for neural simulations, the popular software NEURON, and the effectiveness of these algorithms and their implementations were evaluated by a number of simulations. The simulated models were chosen with the aim of covering a range of neural models as vast as possible by simulating rather small, medium size and extremely complex models, and discussing the results obtained from these measurements.

The proposed algorithms and their implementation were shown to work well for nearly all models within the limits of multi-core architectures such as memory bandwidth and cache size (per core) restrictions.

One problem with the approaches presented in this thesis was shown in section 6.5, when a number of subtrees lower than the number of cores was assigned to these cores in a manner that lead to cache-inefficiencies/a high amount of inter-core communication. This is a quite complex issue which requires a more sophisticated method of splitting cells such that neurons may be split at multiple compartments. The *multisplit* approach has the potential to overcome these problems; however, *multisplit* requires a very complex implementation and new algorithms for automatically identifying compartments where a cell may be split. In a private conversation, NEURON's main author, Michael Hines, mentioned that a future multithreaded implementation of NEURON is likely to employ the *multisplit* technique; however, the changes required for such an implementation will take several months or longer.

Chapter 8

Acknowledgments

Both working and finishing this thesis would not have been possible without the help of many people, friends and colleagues alike. Foremost, I would like to thank my two advisors, Prof. Alexander Borst at the Max-Planck-Institute of Neurobiology, who strongly supported me and my work in the last two years in both a professional and a personal way, and Tobias Klug at the Technical University of Munich who dedicated an unusual high amount of his time to helping me with technical problems as well as giving me helpful advice for the written part of this thesis. I am also deeply indebted to Michael Hines, the author of NEURON, who supported my work by giving technical advice about NEURON internals, and to the whole *MMI* (*Munich Multicore Initiative* [8]) team at the Technical University of Munich that supplied me with both the hardware and useful tips that made it possible to test and evaluate my algorithms: Michael Ott, Dr. Carsten Trinitis and Dr. Josef Weidendorfer.

Last but not least, I would like to thank my parents for both financing and strongly supporting me and my studies in the last five years.

Appendix A

Mechanism Computation

In section 4.3, eq. 4.11 introduced one way for computing a mechanism's current contribution at time $t + \Delta t$:

$$I_{mech}(V(t + \Delta t)) \approx I_{mech}(V(t)) + \Delta V \frac{dI_{mech}}{dV}$$

with

$$\frac{dI_{mech}}{dV} = \frac{\partial I_{mech}}{\partial V} \approx \frac{I_{mech}(V(t) + 0.001mV) - I_{mech}(V(t))}{0.001mV}$$

The above approximation results in a mechanism having to compute $\frac{dI_{mech}}{dV}$ and $I_{mech}(V(t))$. This is best done in one function so the value of $I_{mech}(V(t))$ needed for both terms must be computed only once.

The reason for choosing the above approximation was that while the underlying implicit Euler method requires the mechanism currents to be defined at time $t + \Delta t$, rewriting the equation in order to solve for $V(t + \Delta t)$ is impossible when arbitrary mechanisms I_{mech} must be supported. The author feels this approximation should be introduced only together with a short introduction of alternatives. This thesis uses a combination of implicit methods for the capacitive term and axial currents, while an explicit method is used for mechanism current approximation. The following two sections will consider the two cases when only explicit or only implicit methods are used.

The third section will discuss another approximation - equation 4.11 assumes that $I_{mech}()$ is a function of one variable, $V(t)$, only, while it really is usually a multi-dimensional function.

A.1 Using only Explicit Methods

The reason why I_{mech} needs to be computed for time $t + \Delta t$ was the choice of using implicit methods for the discretization of the capacitive term, so axial currents are computed based on voltage values from time $t+\Delta t$. While explicit methods such as Runge-Kutta methods or explicit Euler are subject to numerical instabilities in general, this is a particular problem in the context of neural simulations because the underlying system of equations is usually *strongly coupled*, that is, changes in membrane potential propagate quickly between neighboring compartments.

Such strong coupling often causes numerical instabilities when using explicit methods unless Δt is extremely small.

A more detailed analysis is out of the scope of this work. NEURON and this thesis use a combination of implicit methods and explicit methods. While most numerical instabilities are avoided by using the most up-to-date voltage values for axial current computation, mechanisms are approximated based on old voltage values. This may lead to problems in a very small number of cases, most notably when gap junctions (electrical synapses) are simulated[1]; all other mechanism types, as far as the author knows, do not exhibit to stability problems.

A.2 Using only Implicit Methods

Implicit methods are nevertheless often applied to more complex equations that can not be rewritten because of their complexity or because they are not known at the time of implementation. Consider an ODE such as

$$\frac{\partial x}{\partial t} = f(x, t)$$

with f being an arbitrary, possibly highly nonlinear function. Then, the implicit Euler method may be applied and $x(t + \Delta t)$ may be found by iteratively searching a solution to the equation

$$\frac{x(t + \Delta t) - x(t)}{\Delta t} - f(x(t + \Delta t), t + \Delta t) = 0$$

This can be achieved by using root-finding algorithms such as Newton's method. There are several reasons, however, why this approach is not suitable for neural simulations; the most important one being that many iterations may have to be performed in order to find the root or a solution close to it. Each iteration requires the computation of f, however, and f, comprising mechanism currents, is the most expensive part of neural simulations.

In addition, for coupled equations, a root-finding algorithm is needed that does not only find the root of one equation but the roots of all equations, a possibly difficult task because these root cannot be searched independently. The author is not familiar with using root-finding algorithms for coupled equations, however.

A.3 Mechanisms as Multi-Dimensional Functions

Coming back to eq. 4.11, an important correction must be made. I_{mech} may be a function of many variables, depending on the mechanism type. While arbitrary variables may be used in order to compute I_{mech}, most notable are membrane voltages (this compartment's membrane voltage for ion channels, but also other compartment's membrane voltages for synapses or gap junctions),

[1] NEURON supports modeling gap junctions with implicit methods as well by integrating them into the matrix as off-diagonal connection coefficients (see LinearMechanism, http://www.neuron.yale.edu/neuron/docs/help/neuron/neuron/classes/linmod.html); this prohibits the use of the efficient solver scheme, however, because the result LSE may not tree-structured, anymore.

followed by ion concentrations (for e. g. synaptic transmitter release or different kinds of ion channel conductances) and the current time (for e. g. simulating time-varying current injections or voltage clamps). Therefore, the approximation applied really is

$$I_{mech}(\mathbf{V}(\mathbf{t}+\mathbf{\Delta t}), \mathbf{C}(\mathbf{t}+\mathbf{\Delta t}), t+\Delta t, ...) \approx I_{mech}(\mathbf{V}(\mathbf{t}), t, \mathbf{C}(\mathbf{t})) + \Delta V \frac{dI_{mech}}{dV} \tag{A.1}$$

where thick elements indicate a list of variables; $\mathbf{C}$ refers to a list of (intracellular and extracellular) concentrations of different kinds of ions.

The correct, explicit multidimensional first-order approximation using the Jacobian matrix would be

$$I_{mech}(V_1(t+\Delta t), V_2(t+\Delta t), ..., C_1(t+\Delta t), C_2(t+\Delta t), ..., t+\Delta t, ...) \approx$$

$$I_{mech}(V_1(t), V_2(t), ..., C_1(t), C_2(t), ..., t, ...) + \begin{bmatrix} \frac{\partial I(V_1(t))}{\partial V_1} \\ \frac{\partial I(V_2(t))}{\partial V_2} \\ \cdots \\ \frac{\partial I(C_1(t))}{\partial C_1} \\ \frac{\partial I(C_2(t))}{\partial C_2} \\ \cdots \\ \frac{\partial I(t)}{\partial t} \\ \cdots \end{bmatrix}^T \begin{bmatrix} \Delta V_1 \\ \Delta V_2 \\ \cdots \\ \Delta C_1 \\ \Delta C_2 \\ \cdots \\ \Delta t \\ \cdots \end{bmatrix}$$

There are several reasons that justify using the approximation from eq. A.1, instead:

1. Only the unknown changes in the membrane voltage, ΔV_i, are computed with the LSE .
2. Computing the (approximative) derivatives, $\frac{\partial I(x)}{\partial x}$, is computationally expensive.
3. Experience shows that for most mechanisms, V_1 has the greatest influence on changes in mechanism currents. In particular, while many mechanism currents also depend on ion concentrations (and thus, changes thereof) to recompute the equilibrium potential of the ion species they use, the absolute value of these changes within a time step is usually very small and does therefore not have to be taken into account when approximating the first derivative.

Appendix B

Strong Variations in Measurement Results

The following program may be used to reproduce strong variations in runtime that were observed during this thesis when manually specifying certain CPU affinity masks using either the tools `taskset` or `numactl` or when specifying a global CPU mask in the main function. The problem was finally fixed by scheduling each thread separately. This technique may be enabled in the program listed below by setting the `PER_THREAD_SCHEDULING` macro to `1`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <asm/msr.h>
#include <sched.h>


/* Sync-Variable; sync_flag[t]!=0 means: Thread t busy.
 * Write accesses are ordered by using mutexes. */
volatile int sync_flag[NTHREADS];

#if !BUSY_WAITING
pthread_mutex_t mut[NTHREADS];
pthread_cond_t cond[NTHREADS];
#endif

/* For wasting cycles */
#define ROUNDS 100
#define EXPONENTIAL 2400000
#define BASE 1.0000001


#define CPU_MHZ 2666
```

```
#define __USE_GNU
#define PER_THREAD_SCHEDULING 0

int cores[NTHREADS];

void *worker_function(void *arg) {
   int my_id = *((int *)arg);
   int i, t;
#if PER_THREAD_SCHEDULING
   int mymask =  1<<cores[my_id];
   sched_setaffinity(0, 4, &mymask);
#endif

   for (i=0; i<ROUNDS; i++) {
      int j;
      double a = BASE;


      if (my_id == 0) {
         for (t=1; t<NTHREADS; t++) {
            //wait for slaves, tell them to continue
#if !BUSY_WAITING
            pthread_mutex_lock(mut+t);
#endif
            while(sync_flag[t] != 0) {
#if !BUSY_WAITING
               pthread_cond_wait(cond+t, mut+t);
#endif
            }
            sync_flag[t]=1;
#if !BUSY_WAITING
            pthread_mutex_unlock(mut+t);
            pthread_cond_signal(cond+t);
#endif
         }
      } else {
         //wait for message from master
#if !BUSY_WAITING
         pthread_mutex_lock(mut+my_id);
#endif
         while (sync_flag[my_id] == 0) {
#if !BUSY_WAITING
            pthread_cond_wait(cond+my_id, mut+my_id);
```

```
#endif
         }
#if !BUSY_WAITING
         pthread_mutex_unlock(mut+my_id);
#endif
      }


      // calculate redundant stuff, BASE^EXPONENTIAL, to waste
      // some cycles
      for (j=0; j<EXPONENTIAL-1; j++) {
         a *= BASE;
      }


      // slave tells master he's ready
      if (my_id != 0) {
#if !BUSY_WAITING
         pthread_mutex_lock(mut+my_id);
#endif
         sync_flag[my_id] = 0;
#if !BUSY_WAITING
         pthread_mutex_unlock(mut+my_id);
         pthread_cond_signal(cond+my_id);
#endif
      }
   }
   return NULL;
}

int main(int argc, char **argv) {
   clock_t t1, t2;
   unsigned long long tt1, tt2;
   int param_m, param_s[NTHREADS];
   pthread_t thread[NTHREADS];
   int t;


#if PER_THREAD_SCHEDULING
   int my_mask;

   if (argc!=NTHREADS+1) {
      fprintf(stderr, "Usage: ./%s core1 ... core%d\n",
```

```
                        argv[0], NTHREADS);
      exit(1);
   }
   cores[0] = atoi(argv[1]);
#endif

   for (t=1; t<NTHREADS; t++) {
#if !BUSY_WAITING
      pthread_cond_init(cond+t, NULL);
      pthread_mutex_init(mut+t, NULL);
#endif
      sync_flag[t]=0;
#if PER_THREAD_SCHEDULING
      cores[t] = atoi(argv[t+1]);
#endif
   }

   // measure both cpu time and real time
   t1=clock();
   rdtscll(tt1);


   for (t=1; t<NTHREADS; t++) {
      param_s[t] = t;
      pthread_create(thread+t, NULL, worker_function, param_s+t);
   }

   param_m = 0;
   worker_function(&param_m);

   t2=clock();
   rdtscll(tt2);
   fprintf(stderr, "clock=%f tsc=%lld\n",
                   ((double)(t2-t1))/CLOCKS_PER_SEC,
                    (tt2-tt1)/CPU_MHZ);
   return 0;
}
```

Bibliography

[1] BLAS (Basic Linear Algebra Subprograms) homepage. http://www.netlib.org/blas/.

[2] GCC, the GNU Compiler Collection. http://gcc.gnu.org, 2007.

[3] GENESIS home page. http://www.genesis-sim.org, 2007.

[4] ModelDB homepage. http://senselab.med.yale.edu/modeldb/, 2007.

[5] Advanced Micro Devices. AMD64 Architecture Programmer's Manual - Volume 2: System Programming, p. 168. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf, September 2007.

[6] Advanced Micro Devices, Inc. AMD Opteron Processor Family. http://www.amd.com/opteron, 2007.

[7] AntiqueTech. Texas Instruments TMS1000. http://www.antiquetech.com/chips/TMS1000.htm, 2004.

[8] Bode, A.; Trinitis, C.; Weidendorfer, J.; Klug, T. Munich Multicore Initiative. http://mmi.in.tum.de.

[9] Borst, A.; Haag, J. The intrinsic electrophysiological characteristics of fly lobula plate tangential cells: II. Active membrane properties. *Journal of Computational Neuroscience*, 3:313–336, 1996.

[10] Bower, James M. *The Book of GENESIS: Exploring Realistic Neural Models with the General Neural Simulation System*. Springer, 1998.

[11] Bower, James M. *The Book of GENESIS: Exploring Realistic Neural Models with the General Neural Simulation System*, chapter 21. Springer, 1998.

[12] Carnevale, Ted; Hines, Michael. *The NEURON book*. Cambridge University Press, 2006.

[13] Cuntz, H.; Haag, J.; Foerstner, F.; Segev, I.; Borst, A. Robust coding of flow-field parameters by axo-axonal gap junctions between fly visual interneurons. *PNAS*, 104:10229–10233, 2007.

[14] Denk, W.; Strickler, J. H.; Webb, W. W. Two-photon laser scanning fluorescence microscopy. *Science*, 248(4951):73–76, April 1990.

[15] Drepper, U.; Molnar, I. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`, February 2005.

[16] Goldberg, David E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman, January 1989.

[17] Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[18] Gropp, William; Lusk, Ewing. Installation and User's Guide to mpich, a Portable Implementation of MPI. Version 1.2.5. The ch_shmem device for Shared Memory Processors. `http://www-unix.mcs.anl.gov/mpi/mpich1/docs/mpichman-chshmem/mpichman-chshmem.htm`.

[19] Haag, J.; Borst, A. Neural mechanism underlying complex receptive field properties of motion-sensitive interneurons. *Nature Neuroscience*, 7:628–634, 2004.

[20] Hayes, Brian. The Easiest Hard Problem. *American Scientist*, 90(2):113–117, March-April 2002.

[21] Hengstenberg, R.; Hausen, K.; Hengstenberg, B. The number and structure of giant vertical cells (versus) in the lobula plate of the blowfly calliphora erytrocephala. *Journal of Computational Physiology*, 149:163–177, 1982.

[22] Hestenes, Magnus R.; Stiefel, Eduard. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 46(6), December 1952.

[23] Hines, M.; Eichner, H.; Schürmann, F. Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. 2008.

[24] Hines, M.; Morse, T.; Migliore, M.; Carnevale, N. T.; Shepher, G. M. ModelDB: A Database to Support Computational Neuroscience. *Journal of Computational Neuroscience*, 17(1):7–11, 2004.

[25] Hines, Michael. multisplit programming reference. `http://www.neuron.yale.edu/neuron/docs/help/neuron/neuron/classes/parcon.html#multisplit`.

[26] Hines, Michael. Parallel NetManager programming reference. `http://www.neuron.yale.edu/neuron/docs/help/neuron/neuron/classes/parnet.html`.

[27] Hines, Michael. splitcell programming reference. `http://www.neuron.yale.edu/neuron/docs/help/neuron/neuron/classes/parcon.html#splitcell`.

[28] Hines, Michael. Efficient Computation of Branched Nerve Equations. *International Journal of Bio-Medical Computing*, 15:69–76, 1984.

[29] Hines, Michael; Carnevale, Ted. Expanding NEURON's repertoire of mechanisms with NMODL. *Journal of Neural Computation*, 12:995–1007.

[30] Hines, Michael; Carnevale, Ted. The NEURON simulation environment. *Journal of Neural Computation*, 1997.

[31] Hines, Michael; Markram, Henry; Schürmann, Felix. Fully Implicit Parallel Simulation of Single Neurons.

[32] Hodgkin, A. L.; Huxley, A. F. A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve. *Journal of Physiology*, 117:500–544, 1952.

[33] Hu, T. C. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9(6):841–848, November 1961.

[34] HyperTransport Consortium. HyperTransport Consortium - Low Latency Chip-to-Chip and beyond Interconnect. `http://www.hypertransport.org`.

[35] IEEE Portable Applications Standards Committee, The Open Group, ISO/IEC Joint Technical Committee 1. IEEE Std 1003.1, 2004 Edition - Standard for Information Technology - Portable Operating System Interface (POSIX) - System Interfaces, 2004.

[36] Intel Corporation. Intel Core Microarchitecture. `http://www.intel.com/technology/architecture-silicon/core/`.

[37] Intel Corporation. Intel NetBurst Architecture. `http://softwarecommunity.intel.com/articles/eng/3084.htm`.

[38] Intel Corporation. Teraflops Research Chip. `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`.

[39] Intel Corporation. Intel 4004. `http://www.intel.com/museum/archives/4004.htm`, 2007.

[40] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1, pp. 10-11,10-12. `http://www.intel.com/design/processor/manuals/253668.pdf`, November 2007.

[41] Intel Corporation. Intel Xeon Processor 5000 Sequence. `http://www.intel.com/products/processor/xeon5000/specifications.htm?iid=products\_xeon5000+tab\_specs`, 2007.

[42] International Business Machines Corp. Cell Broadband Engine Technology and Systems. *IBM Journal of Research and Development*, 51(5), 2007.

[43] Johnston, Daniel; Wu, Samuel Miao-Sin. *Foundations of Cellular Neurophysiology*, chapter 2.4. The MIT press, 1995.

[44] Johnston, Daniel; Wu, Samuel Miao-Sin. *Foundations of Cellular Neurophysiology*, chapter 2.2. The MIT press, 1995.

[45] Johnston, Daniel; Wu, Samuel Miao-Sin. *Foundations of Cellular Neurophysiology*, chapter 4. The MIT press, 1995.

[46] Johnston, Daniel; Wu, Samuel Miao-Sin. *Foundations of Cellular Neurophysiology*, chapter 7.4.3.1. The MIT press, 1995.

[47] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[48] Koch, Christof. *Biophysics of Computation*, chapter 6. Oxford University Press, 1999.

[49] Korf, Richard E. A Complete Anytime Algorithm for Number Partitioning. `http://web.cecs.pdx.edu/~bart/cs510cs/papers/korf-ckk.pdf`, 1997. section 2.5.

[50] Lawson, C. L.; Hanson, R. J.; Kincaid, D. R.; Krogh, F. T. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[51] Markram, Henry. The blue brain project. *Nature Reviews Neuroscience*, 7:153–160, 2006.

[52] Meister, Andreas. *Numerik linearer Gleichungssysteme*. Vieweg, 3 edition, 2007.

[53] Migliore, M.; Hoffman, D. A.; Magee, J. C.; Johnston, D. Hippocampal CA1 pyramidal neuron model from the paper M.Migliore, D.A Hoffman, J.C. Magee and D. Johnston (1999) Role of an A-type K+ conductance in the back-propagation of action potentials in the dendrites of hippocampal pyramidal neurons, J. Comput. Neurosci. 7, 5-15. `http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=2796`.

[54] Migliore, M.; Hoffman, D. A.; Magee, J. C.; Johnston, D. Role of an A-Type K^+ Conductance in the Back-Propagation of Action Potentials in the Dendrites of Hippocampal Pyramidal Neurons. *Journal of Computational Neuroscience*, 7:5–15, July 1999.

[55] Novillo, D. OpenMP and automatic parallelization in GCC. `http://people.redhat.com/dnovillo/Papers/gcc2006.pdf`, 2006.

[56] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface Version 2.0. `http://www.openmp.org/mp-documents/cspec20.pdf`, March 2002.

[57] PathScale LLC. PATHSCALE COMPILER SUITE. `http://www.pathscale.com/`, 2007.

[58] Pittsburgh Supercomputing Center. Parallel GENESIS Homepage. `http://www.psc.edu/Packages/PGENESIS/`.

[59] Pittsburgh Supercomputing Center. Suitable Models for PGENESIS. `http://www.psc.edu/Packages/PGENESIS/suitable.html`.

[60] Romain Brette, Michelle Rudoph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Frederick C. Harris Jr., Milind Zirpe, Thomas Natschlaeger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P. Davison, Sami El Boustani, Alain Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23(3), December 2007.

[61] Saad, Yousef; Schultz, Martin H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[62] SGI. SGI Altix Family. `http://www.sgi.com/products/servers/altix/`, 2007.

[63] The Portland Group. PGI Fortran, C and C++ Compilers and Tools. `http://www.pgroup.com/`, 2007.

[64] Traub, R. D.; Contreras, D.; Cunningham, M. O.; Murray, H.; Lebeau, F. E.; Roopun, A.; Bibbig, A.; Wilent, W. B.; Higley, M.; Whittington, M. A. A single column thalamocortical network model. `http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=45539`.

[65] Traub, R. D.; Contreras, D.; Cunningham, M. O.; Murray, H.; Lebeau, F. E.; Roopun, A.; Bibbig, A.; Wilent, W. B.; Higley, M.; Whittington, M. A. A single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles and epileptogenic bursts. *Journal of Neurophysiology*, 93(4), November 2004.

[66] Voßen, Christine; Eberhard, Jens P.; Wittum, Gabriel. Modeling and simulation for three-dimensional signal propagation in passive dendrites. *Computing and Visualization in Science*, 10:107–121, 2007.