Lukas Johannes Breitwieser, BSc

BioDynaMo: A New Platform for Large-Scale Biological Simulation

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Institute for Technical Informatics

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dr. Fons Rademakers (CERN)

November 23, 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

This thesis is dedicated to my parents Monika and Fritz Breitwieser

Without their endless love, support and encouragement I would not be where I am today.

Acknowledgment

This thesis would not have been possible without the support of a number of people.

I want to thank my supervisor Christian Steger for his guidance, support, and input throughout the creation of this document.

Furthermore, I would like to express my deep gratitude towards the entire CERN openlab team, Alberto Di Meglio, Marco Manca and Fons Rademakers for giving me the opportunity to work on this intriguing project. Furthermore, I would like to thank them for vivid discussions, their valuable feedback, scientific freedom and possibility to grow.

My gratitude is extended to Roman Bauer from Newcastle University (UK) for his helpful insights about computational biology detailed explanations and valuable feedback.

Furthermore, I would like to show my appreciation to all remaining partners in Russia, Newcastle and CERN for their input and help.

Abstract

Computersimulationen werden auf Grund der Komplexität von biologischen Systemen immer öfter zu Forschungszwecken eingesetzt. Diese Arbeit stellt die ersten Schritte des BioDynaMo-Projekts zur Entwicklung einer allgemeinen Plattform für biologische Simulationen vor. Das Projekt wird am CERN gemeinsam mit der Universität in Newcastle (Großbritannien), Innopolis Universität (Russland) und der staatlichen Universität Kasan (Russland) erarbeitet. Diese verteilte Entwicklung stellt zusätzliche Herausforderungen dar und benötigt deshalb einen gründlichen Entwicklungsprozess um eine hohe Softwarequalität zu erreichen. Der Ausgangspunkt von BioDynaMo ist die Software Cortex3D, welche neuronale Entwicklung simuliert. Cortex3D ist allerdings in der Programmiersprache Java geschrieben, welche nicht für Hochleistungsanwendungen geeignet ist. Weiters ist die Architektur nicht flexibel genug um Simulationen in anderen Bereichen durchzuführen. Um ambitionierte Forschungsfragen beantworten zu können wird eine skalierbare und effiziente Implementierung benötigt. Deshalb wurde zuerst der Quellcode von Java in C++ übersetzt und anschließend ein Prototyp entwickelt, der die verschiedenen Parallelisierungsmöglichkeiten moderner Hardware nutzt.

Abstract

To deal with the complexity of biological systems, researchers increasingly rely on computer simulation. This thesis presents the early steps of the BioDynaMo project, which aims to build a general platform for biological simulation. This project is developed at CERN together with partners at Newcastle University (GBR), Innopolis University (RUS) and Kazan Federal University (RUS). Distributed development poses additional challenges and requires a rigorous engineering process to foster high quality of code. Best practices and development tools are presented in this thesis. The starting point for BioDynaMo was a neural simulator called Cortex3D. This software is written in Java, which is not suited for high-performance computing and is not flexible enough to perform simulations in different areas, such as immunology or oncology. Addressing ambitious research questions requires a scalable and efficient implementation. Consequently, the first step was to translate the code base from Java to C++, followed by development of a 'proof of principle' to show how multiple levels of parallelism in today's hardware can be fully utilized.

To accomplish great things we must dream as well as act.

ANATOLE FRANCE

Contents

Figures

Tables

Listings

Abbreviations

AOS	array of structures
AOSOA	array of structure of arrays
AVX	advanced vector instruction
Cx3D	Cortex3D
CI	continues integration
CISQ	consortium for IT software quality
CPC	currently ported class
CPI	cycles per instruction
CSE	computational science and engineering
fMRI	functional magnetic resonance imaging
HBP	Human Brain Project
HEP	high-energy physics
HPC	high-performance computing
IDE	integrated development environment
IM	instant messaging
ISA	instruction set architecture
JDC	Java-defined class
JIT	just in time
JNI	Java native interface

LOC	lines of code
MIMD	multiple instruction multiple data
MPI	message passing interface
NDC	native-defined class
OS	operating system
PMU	performance monitoring unit
SIMD	single instruction multiple data
SOA	structure of arrays
SSE	streaming SIMD extension
TCO	total cost of ownership
TDD	test-driven development

1. Introduction

Computer simulations have become an important tool in science to study systems too complex for a purely analytical or experimental solution [7]. Computing models enable studying phenomena with variable time scales. From processes inside a particle detector to the formation of galaxies. Furthermore, it is possible to perform parameter sweeps, which means systematically running the simulation with different input parameters, resulting in a cost-effective alternative to producing large amounts of prototypes. Within the medical and life science domain, computer simulations also facilitate research in areas that would not be possible from an ethical standpoint.

A prerequisite to using simulations is a proper understanding of the underlying process and development of a mathematical model. Further challenges include verifying correctness of obtained results and dealing with the computational complexity associated with large-scale simulations. The importance of verification was shown in a recent paper about inflated false-positive rates in functional magnetic resonance imaging (fMRI) studies [8]. The authors found a 15-year old software bug in one of the most popular tools that could have an impact on thousands of research papers. Moreover, the design of simulations became increasingly difficult due to the breakdown of single processor core performance improvements around 2004 [9]. Thereafter, industry shifted to more and more parallelism which increases software complexity.

Consequently, research teams engaging in biological simulations are facing big challenges. While they have to stay on top of their fields in order to produce meaningful research results, increasing knowledge in computer science is needed to build these systems. Furthermore, the scientific environment has little incentive to put efforts into the development of a general-purpose simulation tool. Hence, research labs often develop customized software for their research and spend a considerable amount of time working on problems that have already been solved elsewhere.

1.1. BioDynaMo

To solve these issues, the BioDynaMo project was initiated. BioDynaMo stands for Biology Dynamic Modeller. Its goal is to build a general-purpose platform for large-scale biological simulations hiding the computational complexity associated with parallel and distributed computing and promoting reproducibility of results from shared open access data [7].

The following itemization gives more details:

- **General-purpose**
  BioDynaMo must be designed in a flexible, generalizable way to run simulations from different specialities with requirements that are possibly quite distinct. In neuroscientific simulations neurons have long ranging connections which are absent in immunology models. The longest axons can be found in the sciatic nerve which is composed out of fibers from spinal segments L4 to S3, continuing on the back of the leg to reach, after its division, the toes [10].

- **Large-scale**
  Neuroscientists from the Newcastle University in UK estimated that in order to tackle complex challenges, such as schizophrenia a simulation of the whole cortex, which corresponds to 100m - 10bn neurons would be required. This would even exceed the capabilities of a high-performance computing (HPC) cluster and could be a scenario for cloud resources, federated clouds, or a hybrid HPC and cloud environment. The first step is to develop an efficient implementation that fully utilizes parallelism on one node.

- **Reproducibility**
  One of the core principles of science is the reproducibility of results; yet surprisingly, it is sometimes difficult to recreate research findings. This does not necessarily mean that the researcher acted in bad faith and withheld information. Sometimes information is omitted because it seems too trivial. Consequently, a common platform could improve this situation by providing a controlled environment where researchers can release their models, thus enabling colleagues to easily validate their results.

BioDynaMo started as a code modernization project at CERN openlab together with Newcastle University and Intel. Within the framework of CERN openlab, “CERN collaborates with leading ICT companies and research institutes" resulting in a “unique public-private partnership that accelerates the development of cutting edge solutions" [11]. Shortly after, two Russian universities joined the consortium: Innopolis University and Kazan Federal University.

Code modernization was performed on Cx3D [12], a neurodevelopmental simulator developed at ETH-Zurich and Uni Zurich. Figure 1 shows the simulation result of a cultured neural network. It approximates cell bodies with spheres and neurites with a chain of cylinders. Cx3D is capable of simulating mechanical interactions, extra- and intracellular diffusion as well as user defined biological behaviour. Cx3D is written in Java, a programming language not suited for HPC application, is single threaded, and is not flexible enough to simulate models from different specialities, such as immunology or oncology.

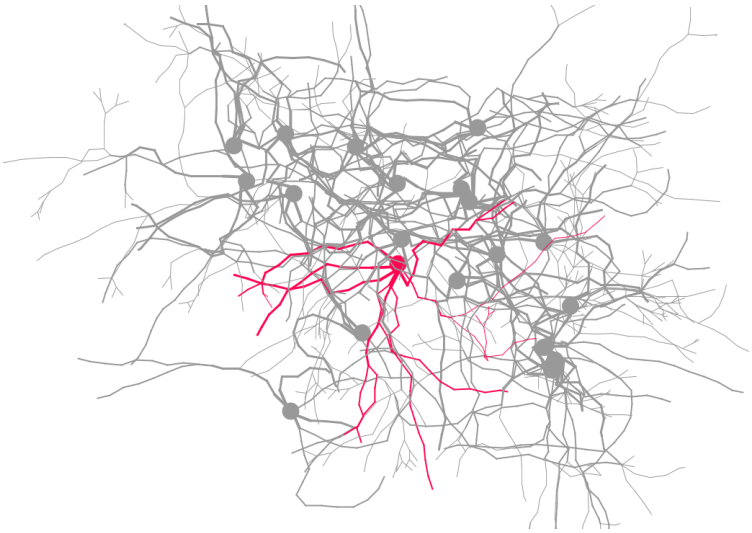


Figure 1.: Simulation Outcome of a Cultured Neural Network. Excitatory neurons (grey) and one inhibitory neuron (pink) are placed on random positions and begin to grow neurites. “An attractive force between the cell elements induces a tendency to fasciculate". After the growth stage has been completed, synapses are formed if an axon and dendrite are in close proximity. [12]

1.2. Structure and Scope of the Thesis

Out of the many goals that the BioDynaMo consortium wants to achieve, this thesis will focus on setting up a sustainable software development workflow and on different aspects of code modernization. Code modernization subsumes a variety of activities. In this thesis I use the term for efforts to port the application from Java to C++ and to change the applications architecture to enable parallel execution and a more modular design. Figure 2 gives an overview about the main contributions of this thesis to the BioDynaMo project.

This thesis is organized as follows. Chapter 2 discusses other simulation software packages. Chapter 3 outlines the importance of sustainable software development, mentions problems in the scientific domain and discusses best practices from literature and industry. Sustainable software [13] is defined as code that is usable for the expected lifetime. The implementation section of this chapter evaluates different combinations of project management tools and evaluates them with regard to benefit, cost, risk and flexibility in the context of the BioDynaMo project. Several development aspects are described in more depth before the chapter is concluded with a short discussion and an outlook of future work.

Chapter 4 describes the whole porting process from Java to C++. It outlines differences between these two languages and gives the reasons for the decision to select C++ as foundation for BioDynaMo. The design section explicates different porting strategies and discusses their advantages and

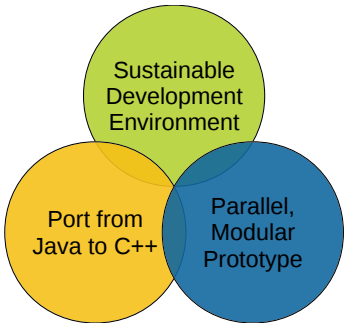


Figure 2.: Main Contributions of this Thesis to the BioDynaMo Project.

disadvantages. Based on this analysis one approach was selected and refined. The implementation section describes this workflow in great depth. Finally, performance of the developed C++ version is evaluated.

Chapter 5 explains the development of a modular and parallel prototype. It clarifies industries shift to more and more parallelism together with the implementation in today's hardware. This chapter gives a more detailed introduction to vectorization, an extension to x86 instruction set architecture (ISA) to execute the same operation on multiple data elements, and evaluates various options to utilize this feature in an application. Furthermore, the chapter gives an overview of different memory layout techniques which strongly influence vectorization. Parallelization is discussed in Section 5.5.3 which contains an introduction to Amdahl's law. Subsequently, the importance of performance profiling is laid out in Section 5.4 introducing perf and Intel VTune Amplifier XE.

The design section of Chapter 5 presents two different options to achieve a customizable design and evaluates their strength and weaknesses. Furthermore, this section benchmarks the SIMD library Vc against the linear algebra package Eigen and presents a vectorization architecture used in the particle simulator GeantV. Moreover, the importance of a common task interface to parallelize execution is emphasized. The final subsection unifies the considerations and introduces the envisaged BiodynaMo package structure with a thin core and speciality specific extensions (e.g. Neuroscience or Immunology) which can be customized by the biologists.

The implementation part, Section 5.6, takes a closer look at non trivial details. Template metaprogramming techniques such as type traits and conditional compilation as well as transformation of conditional statements for vectorized code are explained. It presents results from an initial scaling benchmark, points out performance issues and shows how these hotspots can be eliminated.

The last section in Chapter 5 shows several benchmarks and explains unanticipated results obtained from different vector instruction sets.

2. Related Simulation Software

This chapter gives an overview about different biological simulation packages. This overview contains contents from discussions with Roman Bauer (Computational Neuroscientist at Newcastle University UK) and Marco Manca (Medical Doctor and Senior Research Fellow at CERN Medical Applications).

2.1. Cortex3D

The starting point for BioDynaMo was Cx3D [12]. Its main target is to study brain development, by simulating physical processes, cell cycles and gene expression [12]. The simulator is also able to form synapses, but they are not functional yet, since electrophysiology has not been implemented yet. Biological behaviour is encoded as abstract heuristics without simulating chemical pathways. To give an example, a scientist could define a high level rule that a cell moves into the direction of the gradient of a certain chemical cue [12]. Application for other specialities is limited due to its monolithic software architecture. Another disadvantage is the lack of parallel execution capability and the programming language Java which is not very well suited for HPC applications.

Cx3D is divided into four layers: cell, biology, physical and spatial. Figure 3 shows this separation together with the most important classes in (A), color coded by layer. The neuroscientists specifies the behaviour of the cell and its elements using `CellModule` and `LocalBiologyModule`. These act as the genetic code of the virtual neuron. C to F illustrates how this behaviour gets replicated on cell division or neurite bifurcation. Figure 4 shows the dependency diagram of the spatial organization layer in more detail. It is the most complex part and is mostly hidden from the computational scientist. It calculates a Delaunay triangulation which defines neighboring relations between cells and their elements. In 2D space a triangulation is a subdivision of space into triangles (in 3D it is tetrahedrons). A Delaunay triangulation adds an additional constraint that has to be fulfilled and leads to a result that maximizes the minimum angle of the triangulation [14]. Figure 4 shows the large number of dependencies between classes and high level of abstraction in the design.

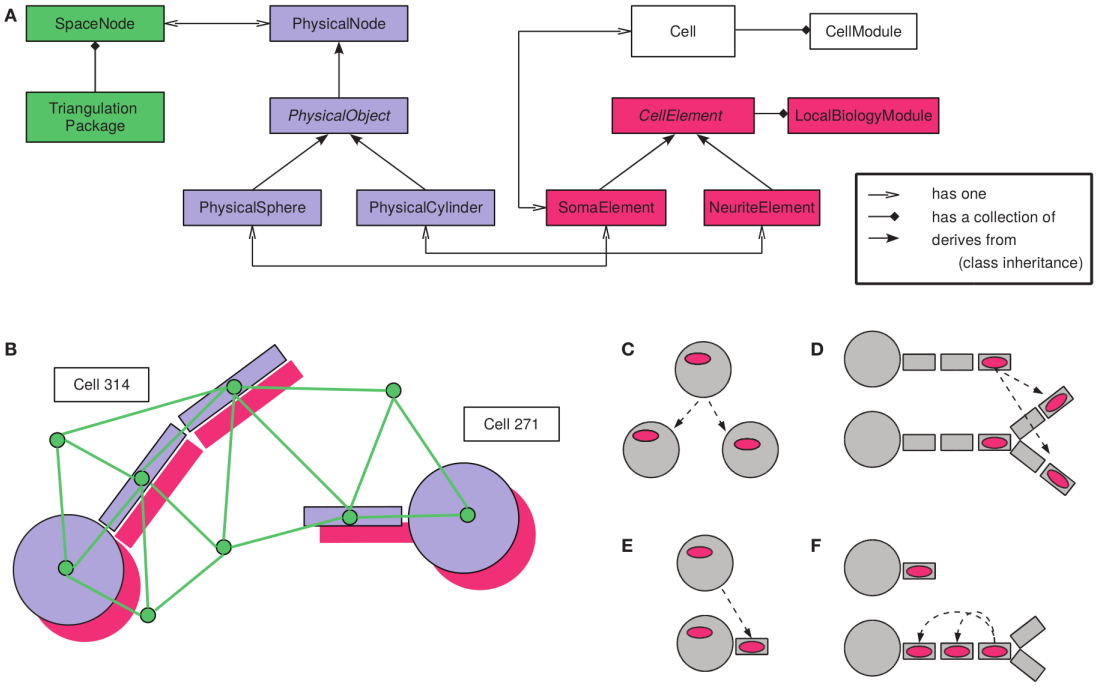


Figure 3.: Architecture of Cx3D: (A) shows an overview of the most important classes. The color shade corresponds to the different layers illustrated in (B): cell (white), biology (purple), physics (pink) and spatial organization layer (green). (C-F) describes different replication strategies of `LocalBiologyModule` or `CellModule` in the event of cell division, neurite elongation and bifurcation. Figure taken from [12].

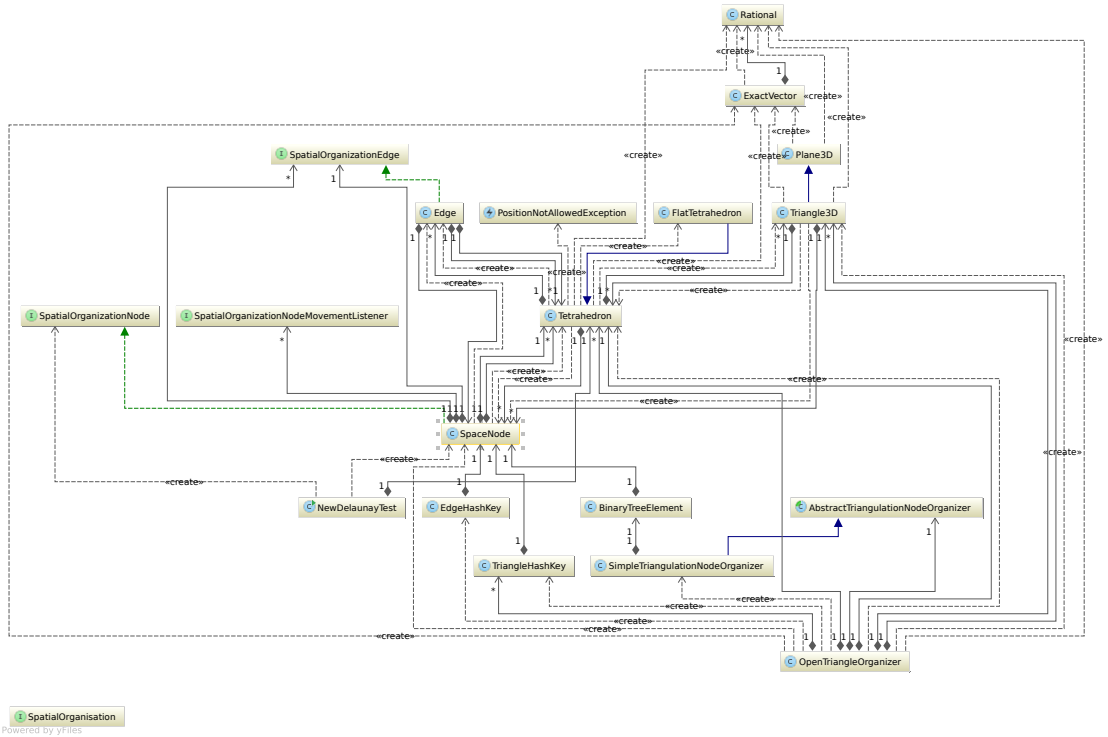


Figure 4.: Dependency Diagram Spatial Organization Layer of Cx3D

2.2. Cortex3D parallel – Cx3Dp

An attempt to increase computational potential of Cx3D has been made by Andreas Hauri who developed a parallelized version of Cx3D called Cx3Dp [14]. In his dissertation he used the simulator to model cortical lamination, i.e. the development of the outermost layer of the brain associated with high level functions such as cognition and behaviour.

A schema of the architecture is shown in figure 5. The principles of the predecessor Cx3D (blue box) remained largely unchanged. Separation into layers (biology, physics and spatial organization) and representation of cell bodies and neurites using spheres and cylinders have been retained. The biggest change was the replacement of the underlying spatial organization layer used to determine neighbors of an object. Delaunay triangulation was replaced with an octree (figure 6). For a scalable parallel execution it is important to keep overhead as small as possible. Communication between execution units should therefore be kept at a minimum. Therefore, Delaunay triangulation is not well suited because local changes can modify distant parts [14].

An octree is a data structure to partition 3D space. On each level a compartment is divided into eight equal sized parts. A volume can be subdivided to adapt to a heterogeneous distribution of elements in space, thus, keeping the number of elements in each leaf nearly constant. The transition

to octrees requires also changes to the diffusion calculation, due to the close relationship between these two components.

The ability to run the simulation in parallel requires management components to ensure consistency of the model and to separate administrative tasks from simulation [14]. This distinction reduces complexity for biologists. Figure 5 shows an overview about system maintenance components. Simulation objects that cannot be divided are called particles. Each compute node maintains a global particle list. In Cx3Dp there are three different particle types: soma, neurite segment and diffusion volume. The scheduler divides the list of particles into work packages and adds them to a queue [14]. Subsequently, the work manager assigns them to free workers threads [14]. The parallelization framework also divides tasks into scientific and system maintenance which are handled by complex workers and simple workers, respectively (figure 5) [14]. Load balancing among mutliple compute nodes is achieved through assignment of subvolumes of the simulation space [14]. Therefore, the load balancer will assign a larger subspace to more powerful servers.

For the agent-based approach a particle has data dependencies on its neighbors. In a distributed setting, where particles are stored on different machines, this will require communication between nodes to exchange information about the margin regions (figure 7). The required interface for data transfer is provided in the communication framework component.

2.3. LAMMPS

The software project LAMMPS, which stands for Large-scale Atomic/Molecular Massively Parallel Simulator, is a software to simulate molecular dynamics. The main range of application lies in simulating metals, semiconductors, biomolecules or polymers [16]. Its functionality can also be applied to objects with larger scales. Lykov et al. for example used it to simulate blood flow in capillaries [17]. Although, LAMMPS is a mature, parallel, open source software package, which is actively maintained, it is not possible to simulate morphological growth and biological behaviour (e.g. gene expression).

LAMMPS' architecture is illustrated in figure 8. Class names written with a blue font are core classes with global visibility [18]. Classes written in red are parent classes that define a common interface [18]. To ease access to core classes, every class except `LAMMPS` itself, derives from `Pointer` [18]. Similar to Cx3Dp, LAMMPS calculates neighbors (class `Neighbor`), divides space into subvolumes (class `Domain`), and transmits margin regions to neighboring compute nodes (class `Comm`) [18]. Data transmission is performed with message passing interface (MPI).

Simulation elements are approximated as point mass. Possible variations are: atom, group of atoms, or larger particles [19]. Calculation of inter-particle forces are decisive for a molecular simulator and account for about 80% of total CPU time [19]. Class `Force` calculates a variety of

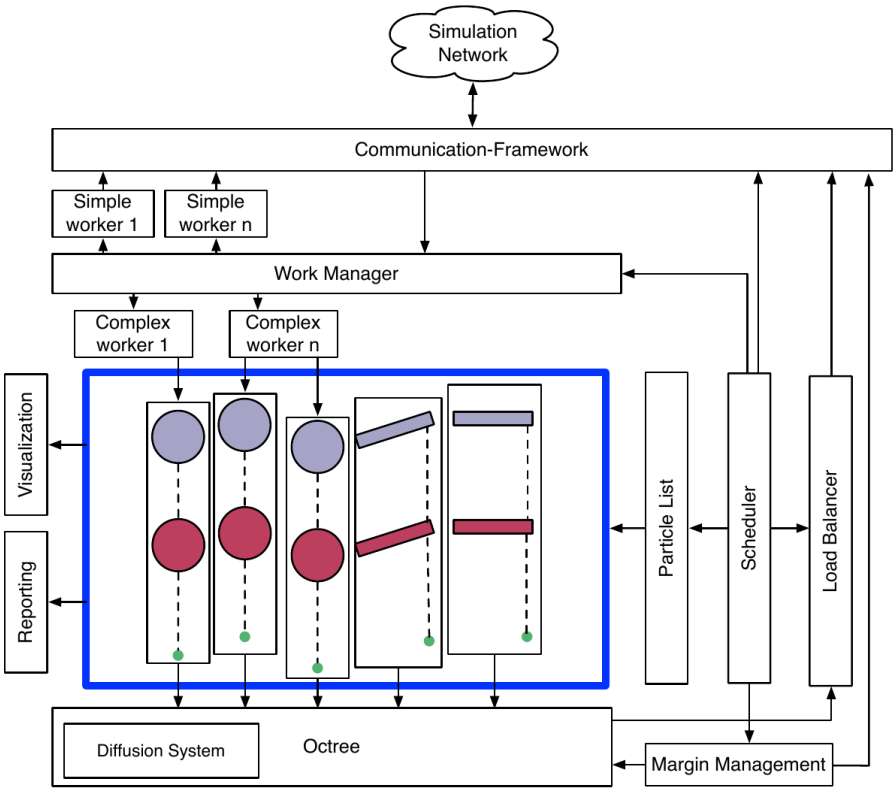


Figure 5.: Architecture of Cx3Dp: Clear distinction between simulation model (within blue box) and maintenance components. Each part fulfils a certain function. Interactions between components are illustrated with arrows. Figure taken from [14].

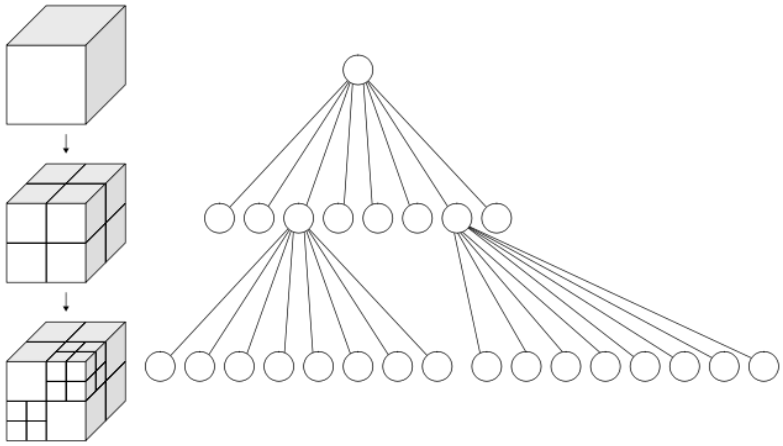


Figure 6.: Octree: Data structure to divide space. The number of children is fixed to eight. Leafs are not required to have the same depth. Figure taken from [15] used under CC-BY-SA 3.0.

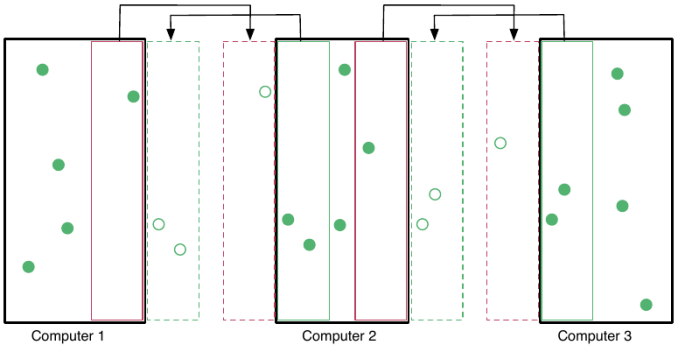


Figure 7.: Margin Management: The properties of a cell in the next time step depends on all its neighbors. If calculation is distributed among several compute nodes, neighbors might be stored on two different machines. Therefore, the runtime environment has to transfer these margin regions. Figure taken from [14].

interactions including non-bonded pair wise, bonded, and long-range Coloumbic forces [18]. In order to save computing time a cut-off for short-range forces can be defined. This introduces an error, but can be a reasonable trade off. Consequently, the list of fixed-radius near neighbors must be calculated. LAMMPS avoids calculating neighbors in every time step using Verlet lists (figure 9), thus improving simulation runtime even further [19]. This method calculates two separate lists every N time steps. One list contains particles within the selected cut-off distance and another one with elements in a buffer zone, called “skin region". Time steps that recalculate the Verlet lists only take the inner region into account [19]. For the remaining iterations, also the buffer region must be considered [19]. The skin region and number of iterations N must be chosen such that [20]:

$$r_m - r_{cut} > N\|v_{max}\|\Delta t$$

Therefore, particles which are not in the Verlet list cannot reach the inner circle within N time steps. LAMMPS rebuilds the list if any atom moves more than half the skin region [19].

The simulator offers two different options to calculate the next time step (class `Update`): time integrators or energy minimizers [18]. Logic to customize the time step calculation are added as specialization of class `Fix` and stored inside class `Modify` [18].

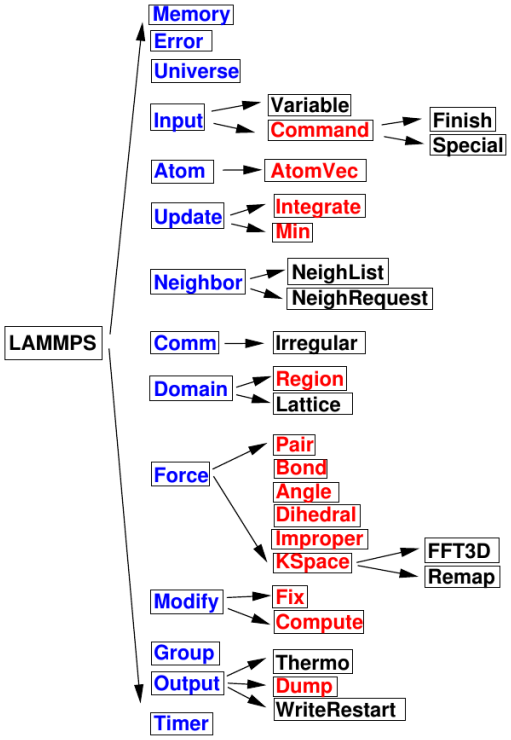


Figure 8.: LAMMPS Class Structure: Classes written in blue are core classes with global visibility. Class names with red font are parent classes and define a common interface. Figure taken from [18].

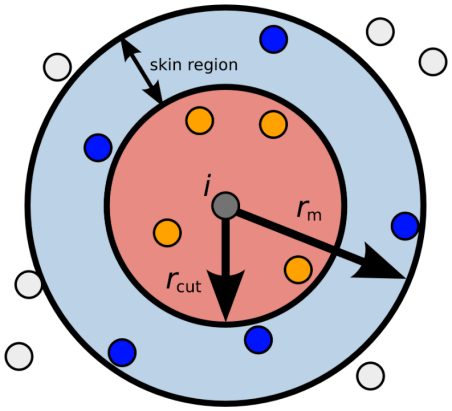


Figure 9.: Verlet Lists: Data structure in molecular dynamics to avoid calculation of neighbors in each time step. In addition to neighbors within distance r_{cut}, elements in the skin region are stored. The width of the blue buffer zone and recalculation time must be chosen in a way that outside particles moving with v_{max} cannot reach the inner circle. Figure taken from [20].

2.4. Others

NETMORPH was built to generate large scale neural networks with realistic neural morphologies [21]. The framework models, elongation, branching and turning of neurites in a stochastic manner resulting in 3D structures which were validated against in vivo and in vitro data from neuromorpho.org [21, 22]. This is similar to the functionality of Cx3D. However, NETMORPH, does not simulate cell proliferation or gene expression. The simulator is released under an open source license. Its latest release dates back to 2011.

CellModeller is a framework developed at the University of Cambridge to simulate multi-cellular systems [23]. This software package is written in Python, but leverages GPUs using PyOpenCL interface. It is possible to simulate populations of more than 100,000 cells and has been used to better understand biofilms [24].

iDynoMiCS is another Java representative. Similar to CellModeller, iDynoMiCS targets microbial communities. It is not parallelized and therefore not suitable for large scale models.

Biocellion, the outcome of a collaboration between the Pacific Northwest National Laboratory and the Institute for Systems Biology, followed a similar approach to BioDynaMo with respect to liberating biologists from dealing with intricacies of parallel and distributed computing [25]. It follows a discrete agent based approach. Parallelism is achieved using Intel TBB (threading builing blocks) for shared memory environments and MPI to communicate among different processes or nodes.

NEST is one of the leading simulators for electrical activity in neuronal networks and is capable of simulating large scale models [26]. It has been mentioned in more than 40 peer reviewed publications in 2015 alone. However, NEST does not offer realistic morphology or other physical processes apart from electrical activity.

A successful commercial physics simulator is COMSOL. It is capable to simulate electrical, mechanical, chemical and flow processes [27]. However, COMSOL is using a mesh-based approach. For our intended use case where the number of objects fluctuates (cell mitosis / apoptosis) and where objects move continuously this is not very well suited. Also Cx3D used a Delaunay triangulation, but this was replaced in Cx3Dp, because it hinders parallelization [14].

With exception of COMSOL, none of the introduced systems, was built to run on a cloud environment and are therefore not prepared to run simulations in a failure prone setting. Furthermore, many of the introduced frameworks cover only the requirements of a specific subgroup. Thus, these software packages find little application in other fields of computational biology.

Lastly, the Human Brain Project (HBP) aims at simulating a human brain from ion channels all the way up to cognitive behaviour. In 2015, Markram et al. published results of reconstruction and simulation of the somatosensory cortex of juvenile rat [28]. To simulate spontaneous activity they adapted the NEURON simulator to run on supercomputers [28]. Nonetheless, HBP uses a static model which is in contrast to the developmental approach BioDynaMo is following. Hence, BioDynaMo is complementary and does not compete with HBP.

3. Sustainable Software Development

Code that is usable for the expected lifetime is defined as sustainable software [13]. Therefore, this term is closely related to software quality. Quality is characterised by consortium for IT software quality (CISQ) with four attributes: reliability, efficiency, security and maintainability [29]. Reproducibility of results is an important aspect in science and must therefore be added to this list.

Software quality and maintainability are of paramount importance for any long term and especially large scale project. Software industry has come a long way since developers exchanged code by sending patches by e-mails. It has never been easier to develop software in a distributed team than it is now. There are modern code repositories, testing frameworks, continues integration (CI) servers, build tools, issue trackers and project management tools. Success of agile development techniques such as Scrum and Extreme Programming has facilitated development of these tools.

Beck [30] describes a software project based on four variables: cost, time, quality and scope. He claims that external forces choose three of those values while the remaining one can be influenced by the development team. Due to the environment, software that is developed in the scientific domain has to cope with issues that do not exist in industry.

The situation for PhD students based on Becks' model will be examined in this paragraph. Cost is a constant. Usually, there are no further funds to hire additional software engineers that assist development. Typically, time cannot be influenced as well since submission deadlines are set by conferences. However, there is some leeway for submission of the dissertation. The most important assignment for a doctoral student is producing high impact research results. Consequently, the focus will lie on scope at the expense of software quality. An important factor for an academic career are publications and citations. Therefore, in a system with many doctoral students and few tenure and full professor positions one should not expect that huge effort is put into producing high quality software. If the developed software is not in a state where it can be used by others it will soon end up unmaintained. Consequently, if a piece of code is not updated it will soon be hard to build and execute it.

Heroux [13] is stating that "just desiring improved quality is not sufficient". He suggests that external forces like funding agencies, publishers and employers should "raise their expectations for software quality". If software quality metrics are part of the evaluation process to receive a

research grant or play a role if a paper will get accepted than behaviour will likely change.

Another fact that should not be neglected is that many researchers that develop software do not have a formal background in computer science, but come from engineering, physics or life science. They usually learn software development best practices on the job. This underlines the importance to set up a solid development process for BioDynaMo to guide new project members.

3.1. Best Practices

Before a team starts to write code it should define or adopt a language style guide, a contract which defines best practices and stylistic rules. Ideally, code developed by two different persons is undistinguishable. In this regard it reduces the number of freedoms for the developer. The C++11 standard for example allows following class names: (i) can be composed out of characters that fall into certain ranges of the ISO 10646 encoding (e.g. letter, digits, underscores) (ii) must not start with a digit (iii) must not be a reserved keyword e.g. `public, const, volatile, ...`. More details can be found in chapter 2.11 of the C++11 standard [31]. Therefore, the following class name will be accepted by the compiler: `mY_clAs5Name`. One possible rule in the style guide could restrict class identifiers to CamelCase (e.g. `MyClassName`). Code is more often read than written. Consequently, Google's style guide optimizes for the reader [32]. An example [32] is the "All parameters passed by reference must be labeled const." rule, which is equivalent to "Parameters must not be passed as non-const reference". Code Listing 3.1 demonstrates how the required address-of operator makes it explicit that `foo` can be modified inside `DoSmth` without knowing the function definition.

Listing 3.1: Googles no non-const Reference Rule

```
1  class Foo { ... };
2
3  void DoSmth(Foo* foo) {
4    // modify foo
5  }
6
7  int main() {
8    Foo foo;
9    DoSmth(&foo);
10 }
```

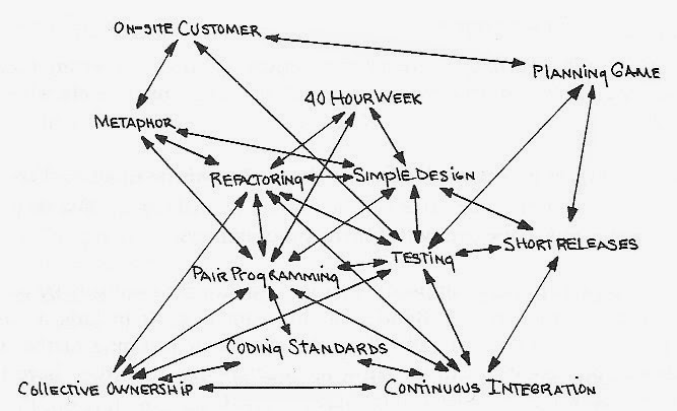


Figure 10.: Connection between Different Software Development Practices in Extreme Programming. Figure taken from [30].

Ideally, a style guide has associated tools to assist the developer comply with these rules. Also, they make code reviewer's life easier and can be used to enforce adherence. This can be achieved by using a check before code is added to the repository (also referred to as a precommit hook). These tools have different complexity. Whitespace rules or checks for include guards can be performed with simple string operations. On the other hand, the appearance of variable or function names can only be reliably asserted if the tool has access to compiler information i.e. abstract syntax tree [33].

One of the cornerstones of agile development is automated tests. Even if a team develops code in a more traditional methodology, automated tests are essential for code quality. Although, they do not guarantee an error-free product, using this technique is associated with a number of benefits [30, 34]:

- Early discovery of bugs
- Increased trust into code
- More modular design
- Living documentation
- Foundation for other techniques like CI

The earlier software errors are detected the easier it is to fix them. Furthermore, their impact (e.g. on research results) is limited. Developers will trust code more which is backed by automated tests and will not fear that they unknowingly introduced bugs [30]. This enables code refactorings, since engineers are not afraid of editing complex code that was written by someone else [30]. In test-driven development (TDD), unit tests are written before the actual functionality is implemented. This leads to a more modular design and reduces complexity, because it is easier testable.

Furthermore, testing code is a kind of living documentation. While API documentation can get out of synchronization because people forgot to update it, this cannot happen for test cases (they would not compile or pass). Other techniques like CI or short releases rely on the existence of automated tests. Figure 10 shows the central role in extreme programming. Lastly, testing manually is simply not a viable option taking the complexity and size of a usual application into account.

Continues integration takes testing one step further. It automatically executes the test suite on a dedicated server and reports if the tests also pass on a different machine — possibly using different operating systems and various compilers. This is important for portability and reports additional errors. If the developer forgot to add a source file to the code repository, for example, the application will compile locally, but will fail on the CI build. CI and automated tests together ensure that the repositories' master branch always works. This enables short releases, because it is not required to "go through a lengthy test cycle" [30].

Refactoring refers to code changes to improve or simplify the design without changing behaviour. Restructuring long methods or removing code duplication is important to reduce technical debt [35]. Thus, improving maintainability or flexibility of the code base. Refactoring can lead to many code changes across the whole application. Automated tests give the required confidence that the applications behaviour indeed did not change after the procedure has been finished.

Communication between project members in a distributed environment is certainly a key challenge. We are using a mix of synchronous, asynchronous, low and high bandwidth channels to coordinate our efforts. For the open source project OpenMRS, Burke Mamlin wrote an association which communication channel should be used in a certain situation [36]. Asynchronous tools (e.g. mailing list) can be used to speak to the community. Low bandwidth real time conversations (instant messaging (IM)) are suitable for brief discussions that require immediate feedback. Finally, high bandwidth applications like Skype or Google Hangouts are an ideal tool for presentations among a subgroup or more detailed discussions.

3.2. Implementation for BioDynaMo

This chapter describes the evaluation and selection of project management tools, more detailed development aspects and discusses open points and future work.

3.2.1. Selection of Project Management Tools

To decide which combination of tools we want to use, we started by listing common options for each category (Table 1). In a second step, we analysed combinations of these setups which we call software stacks (Table 2).

OpenMRS is a medical record system built to improve health care in developing countries. It has been adopted by many hospitals and ambulances around the world [37]. Not only the users of the software, but also the developers work from different continents. Therefore tools for distributed development, effective communication and documentation are required. OpenMRS counts on the Atlassian software products for CI, project management, issue tracking and documentation. Atlassian is one of the leading vendors covering the whole development process and is used by more than 50.000 companies according to their website [38]. Hence, it is a mature software, provides enterprise-grade support and offers a large ecosystem of plugins. Unfortunately, it has a proprietary license, but open source projects that fulfil a number of criteria, can use their products for free. However, people reported that the software is bloated and complicated [39].

With more than 38 million hosted repositories Github is the Facebook of software development. In contrast to Atlassian which is fully configurable to ones needs, Github convinces with simplicity. Moreover, it minimizes the barrier for contribution since a large number of developers is already accustomed with their platform and workflow. Furthermore, its success has attracted many other companies to seamlessly integrate with their services. CI service Travis [40], for example, can be configured with a few lines of code for a standard application (code example 3.2.

Listing 3.2: Travis-CI Configuration for a Java Maven Project

```
1 language: java
2 jdk:
3   - oraclejdk8
```

Tuleap is a relatively young open source project that develops a one stop shop for application lifescyle management [41]. It integrates a number of successful software projects like Git, Gerrit and Jenkins and supplements it with a home grown project management software. Their focus is to integrate all these solutions and connect developers, product owners, DevOps, project managers and customers [41]. The project is backed by the company Enalean. Among their clients are companies like Orange, Airbus and Renault. [42].

The individual software stack gives freedom to choose software tools independently of other categories. Therefore, it can be tailored to the projects needs. Table 2 shows a substitutional combination of applications. On the downside, this leads to problems with integration and higher maintenance effort. For instance, single sign-on, to connect all those tools must be added by the adopting team.

Table 3 shows the final evaluation score on four dimensions. The following enumeration gives more details:

- **Benefit:**
  The benefits for each of the options has been evaluated under the current situation of BioDynaMo. At the moment the number of developers is still small. Hence, Github's simplicity still meets all demands. Tuleap and the individual solution do not integrate with communication tools like Slack or Gitter and therefore do not reach full score. While the individual stack meets all demands for each separate tool, it will not reach the full potential, because they are not well integrated with each other.

- **Cost:**
  Github has the least total cost of ownership (TCO) - it is free for public projects and does not require complicated setup. CERN has already a license for Atlassian products and a team which manages its installation. Therefore, BioDynaMo only has to configure it. Tuelap has a cloud offering, but they do not offer a free plan for open source projects. Therefore, we would need to pay a monthly fee or install it on premise and perform maintenance ourselves.

- **Risk:**
  For the OpenMRS stack risk is minimal. Even in case Atlassian discontinues their free open source license, it would not increase the projects cost due to CERN's license. Given the adoption rate and size it is very unlikely that they will go out of business. The same holds true for Github, where free access for public repositories is an essential part of their business plan in order to attract millions of developers. There are tools to export data e.g. to Atlassian, in case that Github's features do not longer fulfil all requirements. Tuleap on the other side is a rather new player on this market. Therefore, this is associated with higher risk. Lastly, the discontinuation of one product which is part of the individual stack would require to migrate to another tool.

- **Flexibility:**
  This is clearly the strength of the individual software stack. Tools can be replaced individually based on changing requirements. Atlassian offers many plugins and configuration possibilities, while Github can be used as is, but integrates with many services.

Category	**Options**
Code Repository	Github, Gitlab, Atlassian BitBucket, Google Code, Tuleap, ...
Project Management Tool and Issue Tracker	Atlassian JIRA, Redmine, Tuleap, ...
Continues Integration	Jenkins, Travis-CI, Atlassian Bamboo, Gitlab, ...
Code Review	Gerrit, Github, Gitlab, ...
Documentation	MediaWiki, Atlassian Confluence, DokuWiki, TWiki, ...
Mailing List	CERN e-groups, Mailman, Google Groups, FreeList, ...
IM	IRC, Gitter, Slack, ...
Conference Calls	Jitsi, uberconference, Skype, Google Hangouts, ...

Table 1.: Development Tools: Available Options

Category	**OpenMRS**	**Github**	**Tuleap**	**Individual**
Code Repository	Github	Github	Git	Github
Project Management and Issue Tracker	Atlassian JIRA	Github Issues	Tuleap	Redmine
Continues Integration	Atlassian Bamboo	Travis-CI	Jenkins	Travis-CI
Code Review	Github	Github	Gerrit	Gerrit
Documentation	Atlassian Confluence	Github Wiki	phpWiki	MediaWiki
Communication	Discourse	External	Mailman	Discourse

Table 2.: Software Stack Comparison

	OpenMRS Stack	**Github**	**Tuleap**	**Individual Stack**
Benefit				
Cost				
Risk				
Flexibility				

Table 3.: Software Stack Evaluation (white circle: 0%, black circle: 100%)

As a result, we agreed on using the “Github Stack" as long as it fulfills our requirements. For communication we use CERN’s e-groups as mailing list, a Slack channel and Skype. This setup does not lead to additional costs or maintenance effort.

3.2.2. Development Details

At the beginning of the project we have decided to adopt Google’s C++ style guide with minor modifications. It was selected because it comes bundled with a ready to use code formatting definition for eclipse and cpplint, a tool that checks source code if it complies with the rules. We loosened the restriction on maximum line length and on allowed C++11 headers. Conventions and information which is not part of a language standard has been put into the “BioDynaMo Developers Guide" (see appendix). It introduces newcomers to the project, its members and vision. Furthermore, it mentions our Git workflow, conventions for commit messages, how to set-up the development environment and walks through the development steps from selecting an issue towards merging into master.

We use CMake to build C++ code. It abstracts platform and compiler specific details and can be used to locate required libraries and headers [43]. With about 200 lines of code (LOC) it is possible to detect all required dependencies, build a shared library, a test executable, demo applications, Doxygen API documentation, create a coverage report and check for memory leaks – for two operating systems (Ubuntu, OSX) and different compilers (g++ and clang).

Option	**Default Value**	**Description**
`test`	`on`	build a test executable precondition for `valgrind` and `coverage`
`valgrind`	`on`	enable memory checks
`coverage`	`off`	creates a make target to generate a HTML report indicating which parts of the code are covered by automatic tests (Figure 11)

Table 4.: CMake Options [5]

In our implementation we define three options (see Table 4). Based on those flags CMake generates the following make targets [5]:

- `make test`: execute all tests
- `make check`: execute all tests and show test output on failure
- `make clean`: clean all targets including external projects
- `make bdmclean`: only clean `libbiodynamo` and `runBiodynamoTests` targets
- `make testbdmclean`: only clean `runBiodynamoTests` target
- `make doc`: generate Doxygen documentation in directory `build/doc`. It contains a html and latex version. To view the html version open `html/index.html` in the browser.
- `make coverage`: create coverage report in `build/coverage`

First, testing verifies that all test cases pass and performs a memory leak check on the same executable afterwards (`make check`). This is important since C++ does not have a garbage collector and memory must be freed by the developer. This can be the source for many issues and is therefore analysed. This is done by the tool valgrind which instruments the code and thus increases runtime up to a factor of 50 based on own measurements. Therefore, some long running tests are disabled for valgrind to reach reasonable execution times. Besides memory leak checks, valgrind also reports usage of uninitialized memory, invalid frees and more [44]. If the option `coverage` is switched on, an additional make target is created that will produce a report which shows the code lines that have been executed by one of these tests. It can serve as code metric indicating which classes need more testing (Figure 11).

The development workflow is organized as follows:

1. Select a task from the issue tracker
2. Make changes in a feature branch or separate fork
3. Execute code formatter and checker
4. Run test suite and make sure all tests pass
5. Check if code changes have sufficient code coverage and are well documented
6. Commit
7. Send pull request
8. Code Review
9. Merge into master branch

3.2.3. Discussion and Future Work

Although we have established a solid development environment which fosters best practices, there is still work to do and decisions to be made.

Google's style guide does not distinguish between members and static members of a class. We are currently discussing whether to introduce a prefix for static data members. Furthermore, I have mentioned that we relaxed the maximum line length to 120 characters. It seems that an 80 character limit is outdated in the era of widescreen displays. However, it improves productivity if editors can be displayed side by side. Astonishingly, there are many discussions on the internet about this question [45]. People claim that long lines are more difficult to read and probably try to perform too many instructions. Someone also pointed out that code is also displayed in other applications than the editor: diff tools, bug tracker or code review tools and should therefore be kept short [45].

Closely related to style guide are code formatter and checker. So far we have been using Eclipse code formatter, but recently came across clang-format. Performing a few tests showed superior performance compared to Eclipse and we will therefore switch soon. An example can be seen in the code listing below. Also clang-format has a built in definition of Google's style guide.

Current view: top level - src		Hit	Total	Coverage
Test: coverage.info.cleaned	Lines:	96	104	92.3 %
Date: 2016-09-13 13:34:08	Functions:	25	27	92.6 %

Filename	Line Coverage		Functions	
color.cc	87.5 %	14 / 16	75.0 %	3 / 4
math_util.cc	96.4 %	27 / 28	100.0 %	9 / 9
random.cc	97.6 %	41 / 42	100.0 %	8 / 8
string_builder.cc	77.8 %	14 / 18	83.3 %	5 / 6

(a) Overview

Current view: top level - src - math_util.cc (source / functions)		Hit	Total	Coverage
Test: coverage.info.cleaned	Lines:	27	28	96.4 %
Date: 2016-09-13 13:34:08	Functions:	9	9	100.0 %

```
      Line data    Source code
    1            : #include <param.h>
    2            : #include "math_util.h"
    3            :
    4            : #include "crlibm.h"
    5            :
    6            : namespace bdm {
    7            :
    8      2325951 : double MathUtil::exp(double d) {
    9      2325951 :   return exp_rn(d);
   10            : }
   11            :
   12       209176 : double MathUtil::cbrt(double d) {
   13       209176 :   return pow_rn(d, 1.0 / 3);
   14            : }
   15            :
   16    155778891 : double MathUtil::sqrt(double d) {
   17    155778891 :   return pow_rn(d, 1.0 / 2);
   18            : }
   19            :
   20       328786 : double MathUtil::cos(double d) {
   21       328786 :   return cos_rn(d);
   22            : }
   23            :
   24       329182 : double MathUtil::sin(double d) {
   25       329182 :   return sin_rn(d);
   26            : }
   27            :
   28           18 : double MathUtil::asin(double d) {
   29           18 :   return asin_rn(d);
   30            : }
   31            :
   32         3614 : double MathUtil::acos(double d) {
   33         3614 :   return acos_rn(d);
   34            : }
   35            :
   36           17 : double MathUtil::atan2(double y, double x) {
   37           17 :   if (x > 0) {
   38            5 :     return atan_rn(y / x);
   39           12 :   } else if (x < 0 && y >= 0) {
   40            3 :     return atan_rn(y / x) + Param::kPi;
   41            9 :   } else if (x < 0 && y < 0) {
   42            0 :     return atan_rn(y / x) - Param::kPi;
   43            9 :   } else if (x == 0 && y > 0) {
   44            1 :     return Param::kPi / 2;
   45            8 :   } else if (x == 0 && y < 0) {
   46            3 :     return -Param::kPi / 2;
   47            :   } else {
   48            5 :     return 0;
   49            :   }
   50            : }
   51            :
   52        50003 : double MathUtil::log(double d) {
   53        50003 :   return log_rn(d);
   54            : }
   55            :
   56            : }  // namespace bdm
```

(b) Source Code View

Figure 11.: Sample Code Coverage Report

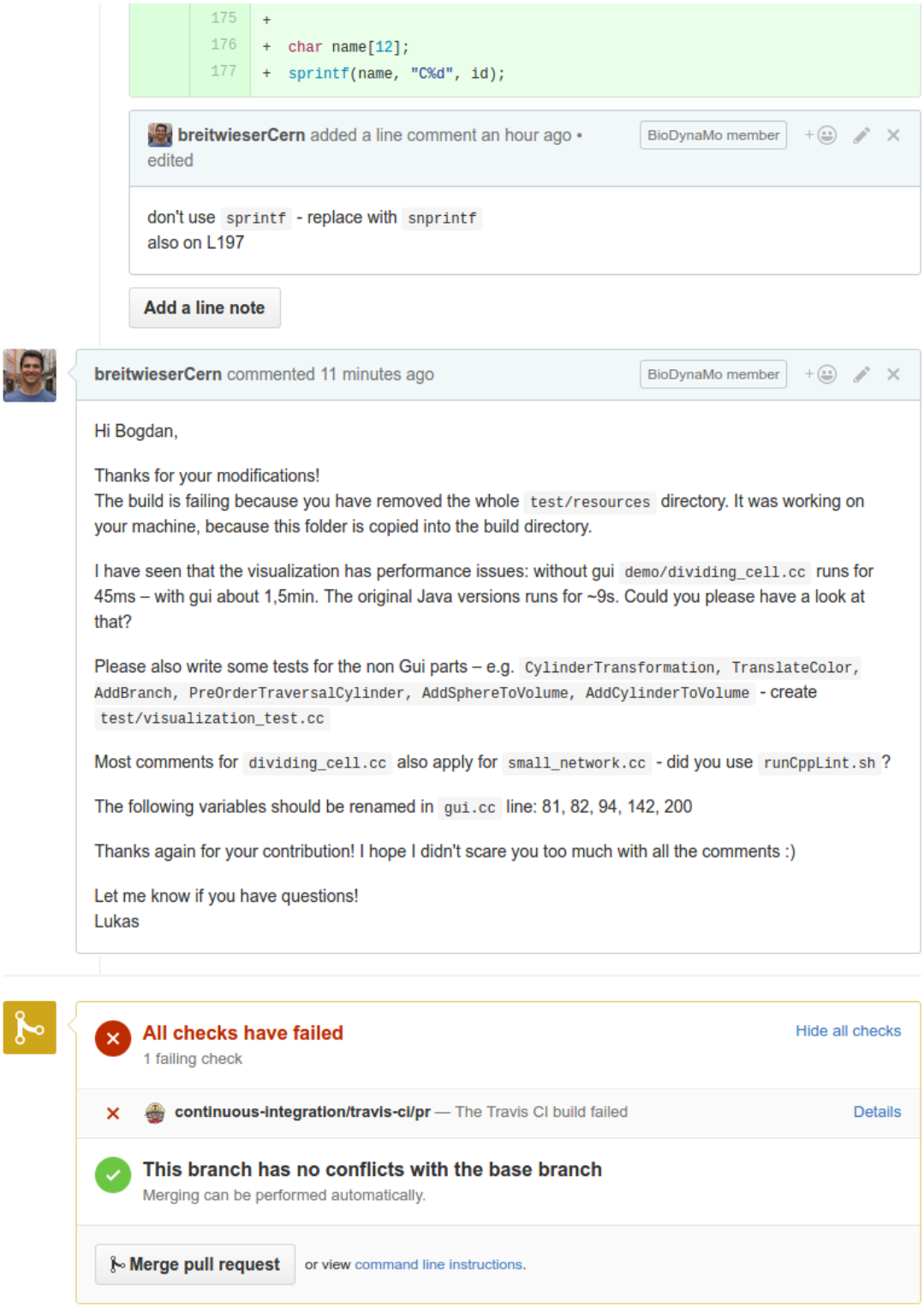


Figure 12.: Review Example for Visualization Pull Request.

Listing 3.3: Diff between Eclipse Formatter and clang-format

```
1  std::array<double, 3> PhysicalSphere::transformCoordinatesLocalToGlobal(const std::array<double, 3>& pos) const {
2  -   std::array<double, 3> glob { pos[0] * x_axis_[0] + pos[1] * y_axis_[0] + pos[2] * z_axis_[0], pos[0] *
       ↪ x_axis_[1]
3  -       + pos[1] * y_axis_[1] + pos[2] * z_axis_[1], pos[0] * x_axis_[2] + pos[1] * y_axis_[2] + pos[2] *
       ↪ z_axis_[2] };
4  +   std::array<double, 3> glob{pos[0] * x_axis_[0] + pos[1] * y_axis_[0] + pos[2] * z_axis_[0],
5  +                              pos[0] * x_axis_[1] + pos[1] * y_axis_[1] + pos[2] * z_axis_[1],
6  +                              pos[0] * x_axis_[2] + pos[1] * y_axis_[2] + pos[2] * z_axis_[2]};
7    return Matrix::add(glob, mass_location_);
8  }
```

Our code checker cpplint does not verify naming conventions for classes, methods or variables, because it does not have access to compiler information. In order to improve the review process and make sure that style guide errors do not arrive at the repository, a more advanced tool should be added to complement cpplint. At CERN, the static analysis suite [46] has been built for this purpose. It is a wrapper around clang compiler and runs in parallel to compilation [46].

4. Port from Java to C++

C++ is superior compared to Java for the application in high-performance computing. Java code runs in a virtual machine, thus adding additional overhead. Just in time (JIT) compilation that takes Java bytecode as input and generates native machine code improves this situation.

Java and C++ have been designed with different goals in mind. In Java, every member function in a class is `virtual` in contrast to C++ where the developer has to explicitly add this keyword to the function declaration. A virtual function is not bound at compile time, but dynamically during runtime. This requires a virtual function table (vtable) that contains information which implementation should be executed. This design decision has two implications: (i) calling a virtual function requires an additional jump to the vtable which is not cache friendly and (ii) virtual functions cannot be inlined. Another major difference is the C++ feature called template meta-programming. This part of C++ is executed at compile time and in itself Turing-complete.

Also, the ecosystem that has been built around a language was part of the decision to opt for C++. Tools, libraries and runtimes like OpenMP, CUDA, OpenCL, MPI, Vc, Intel TBB, and many more are predominantly built for Fortran, C, or C++. Although, Java bindings for several of these add-ons exist, they are not very common.

The following two sections indicate the influence of different C++ programming styles on performance and reveal considerable differences.

Performance Impact of Virtual Functions

Virtual functions enable polymorphism, but their use has a negative impact on performance. How substantial this degeneration can be shows code Listing 4.1. It calculates the dot product of 512 pairs of two dimensional vectors and repeats this a million times. Execution on A takes 2.5s for `VirtualVector` and 1.0s for `Vector`. The example was chosen as it is close to the use case in BioDynaMo where an operation iterates over a set of elements (e.g. cells) and performs certain operations. Nevertheless, it should be noted that performance degradation depends on the actual application. In this example there is no work done inside the function bodies, therefore, overhead is significant. To compile the non-virtual vector version omit the `-DVIRTUAL` flag from line 1.

Listing 4.1: Performance Evaluation of Virtual Function Calls

```
1  // g++ --std=c++11 virtual_functions.cc -o virtual_functions -O2 -DVIRTUAL
2  #include <iostream>
3
4  struct VirtualVector {
5    int x, y;
6    VirtualVector() : x{1}, y{1} {}
7    virtual int GetX() const { return x; }
8    virtual int GetY() const { return y; }
9  };
10
11 struct Vector {
12   int x, y;
13   Vector() : x{1}, y{1} {}
14   int GetX() const { return x; }
15   int GetY() const { return y; }
16 };
17
18 int main() {
19   const size_t N = 2 << 10;
20 #ifdef VIRTUAL
21   VirtualVector v[N];
22 #else
23   Vector v[N];
24 #endif
25   double sum = 0;
26   for (int i = 0; i < 1e6; i++) {
27     for (size_t j = 0; j < N; j += 2) {
28       sum += v[j].GetX() * v[j + 1].GetX() + v[j].GetY() + v[j + 1].GetY();
29     }
30   }
31   std::cout << sum << std::endl;
32 }
```

C++ Template Metaprogramming Example

Code example 4.2 shows the benefits of template metaprogramming by means of Fibonacci sequence calculation. Function `Fibonacci` will be evaluated at compile time while `FibonacciNaive` is calculated at runtime. Executing line 28 is immediate and equivalent to the code line `std::cout << 20365011074 << std::endl;` – the result of the calculation. The result can be found in the assembly code generated by the compiler. To see the assembly output add the `-S` flag for g++. The dynamic calculation on line 29 takes about 34s. Hint: The dynamic Fibonacci version can be sped up tremendously by using dynamic programming ($O(n)$ instead of $O(2^n)$). Due to simplicity of this example the naive implementation was shown.

Listing 4.2: Template Metaprogramming Example

```
// compile with: g++ Fibonacci_tmp.cc -o Fibonacci_tmp -O3

#include <iostream>

template <long N>
long Fibonacci() {
  return Fibonacci<N - 1>() + Fibonacci<N - 2>();
}

template <>
long Fibonacci<0>() {
  return 1;
}

template <>
long Fibonacci<1>() {
  return 1;
}

long FibonacciNaive(long n) {
  if (n <= 1)
    return 1;
  else
    return FibonacciNaive(n - 1) + FibonacciNaive(n - 2);
}

int main() {
  std::cout << Fibonacci<50>() << std::endl;
  std::cout << FibonacciNaive(50) << std::endl;
  return 0;
}
```

4.1. Design

Our porting approach has been strongly influenced by the fact that Cortex3D did not have automated tests. This was a major challenge since we had to ensure that the ported C++ version produces the same results as the original Java simulator. Porting will inevitably introduce bugs. It is crucial that mistakes are detected as early as possible. Finding the reason(s) for numerical differences in 15k LOC is a veritable nightmare for a developer. Consequently, this scenario must be prevented. This requirement translates into a workflow where a small piece of code is translated and immediately validated. Once correctness has been verified, the developer proceeds with the next part. There are two options to perform validation: write unit tests for every single class, or test the whole simulator.

The highly interdependent architecture of the spatial organization layer (Figure 4) would require many code changes to separate components. This would be necessary to test individual functions. Furthermore, option number one requires a very detailed understanding of the whole

implementation to select test cases that cover a meaningful part of the input space.

In contrast, testing the whole simulator is a lot easier. The only obligatory code changes are adding serialization to objects that contain simulation state. After this has been completed, existing demo simulations can be transformed into test cases and executed on the original Java application. The results are persisted to disk and form the target simulation state which must also be obtained from the new C++ version. The most important requirements for serialization of the simulation state are as follows: [1]

- Serialization should not contain implementation details e.g. which specific data structure was used, or the state of a lock.
- It must be possible to generate the serialization in Java and C++ with the same result.

The issue with testing the whole simulator, is that executing the tests is only possible once all code has been ported, thus invalidating one of our specifications. This can be mitigated by creating an executable Java / C++ hybrid and gradually replacing Java code with its C++ representation. Hence, a prerequisite for the proposed approach is two-way communication between these two languages. Java must be able to call functionality implemented in C++ and vice versa. Java contains Java native interface (JNI), a low level mechanism that is able to perform this task for C code. Figure 13 visualizes the just described porting workflow and will be explained in greater depth in Section 4.2.

After examination of the advantages and disadvantages of the two options, we decided to opt for the second choice. The decisive reason was that it also ensures that the interaction of many components produces the original results, something that unit tests on its own cannot warrant. Furthermore, we expected a higher development speed due to the aforementioned reasons.

Even in the iterative porting workflow that translates one class in each step, debugging tools are required because some classes are more than 1000 LOC long.

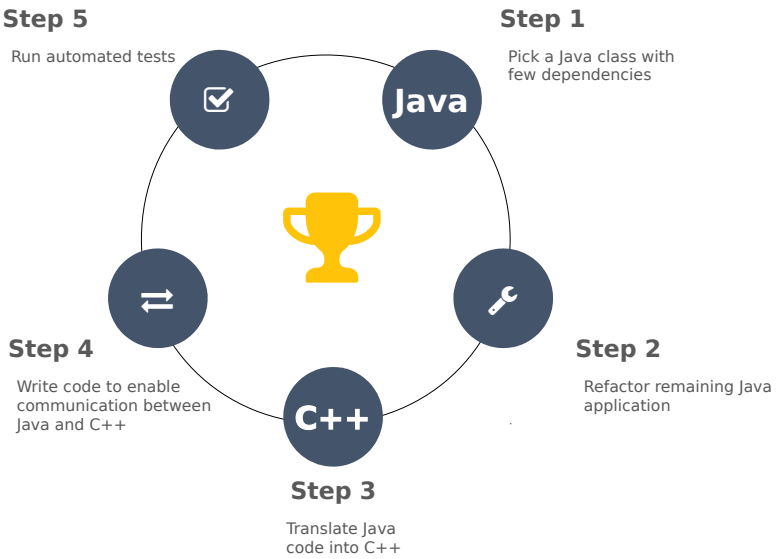


Figure 13.: Iterative Porting Workflow: Java classes are translated to C++ one at a time. Selecting a class with few dependencies reduces development and runtime overhead. The remaining Java application must be refactored to use a common interface of this class. This allows switching between native and Java implementation which is important for debugging purposes. After the class has been translated to C++, code has to be added to enable communication between languages. The last step is to execute the automated tests. If they pass, the next Java class will be ported.

4.2. Implementation

4.2.1. Obtaining Simulation Target Values

As described in the design part, we transformed existing tutorial simulations and two simulations from [12] into test cases. They cover the most important features like mechanical interactions, extra- and intracellular diffusion, creation of synapses and definition of custom behaviour. Based on the requirements from Chapter 4.1 we decided to develop a custom solution to serialize class members to JSON format and further to obtain target values of the simulation outcome. Listing 4.3 shows this process for class `PhysicalNode`. More advanced techniques that automatically serialize classes did not comply with above rules. To perform the JSON comparison we used the library `Gson`. A common base class `BaseSimulationTest` was created which sets up the environment, runs the test, asserts the result and tracks performance . This makes it very easy to transform a tutorial simulation. The only thing that must be done is replacing one line of code: changing the function signature of `main` to `public void simulate()` [1].

The result of performance tracking for `IntracellularDiffusion` can be observed in Figure 14. For each test the system logged execution times of the last commit. This should not be seen as a very accurate measurement since they are not normalized. Therefore, different development hardware or simply different utilization of the machine leads to different wall clock time measurements. Initially this functionality was added to assert that performance improves with each commit. `BaseSimulationTest` was designed to fail a test if the runtime was higher than the last commit including some margin. Looking again at Figure 14 it clearly shows that we had to change this very quickly. The assert was removed, but logging execution times remained active. Spikes in the plot are closely related to the overhead due to interlanguage communication between Java and C++. Performance degraded if there was a pair of classes which communicated intensively with each other and were implemented in different languages. One measurement was even ten times slower than at the beginning. After all classes were available on the native side the spike disappeared.

4.2.2. Iterative Porting Workflow

In the design part we evaluated different options and decided to select a workflow that tests the whole simulator and replaces Java classes step by step. This approach is illustrated in Figure 13 and will be described in more depth in this chapter.

The process starts by selecting a Java class with few dependencies. The less interactions the currently ported class (CPC) has the easier becomes the whole process. In general, interactions mean method calls between CPC and the remaining Java application with all method calls crossing

Listing 4.3: Code Sample Demonstrating JSON String Generation

```
@Override
public StringBuilder simStateToJson(StringBuilder sb) {
  sb.append("{");

  SimStateSerializationUtil.keyValue(sb, "ID", ID);
  SimStateSerializationUtil.keyValue(sb, "idCounter",
    idCounter.get());
  ...
  SimStateSerializationUtil.keyValue(sb, "soNode", soNode);

  SimStateSerializationUtil.removeLastChar(sb);
  sb.append("}");
  return sb;
}
```

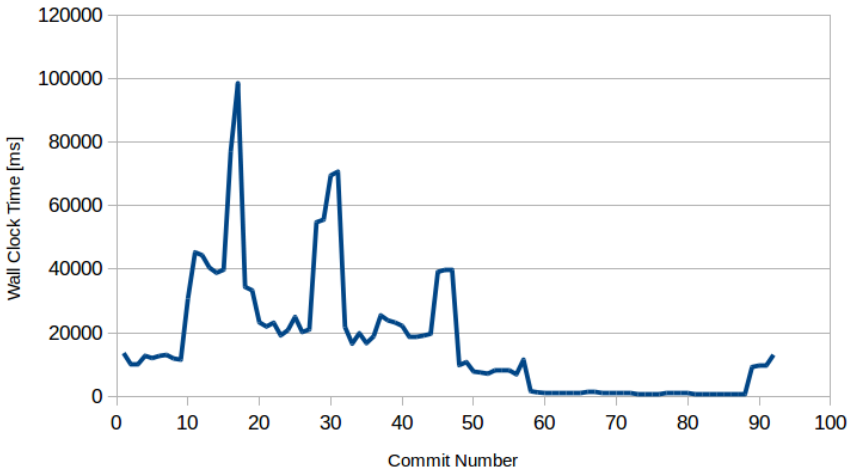
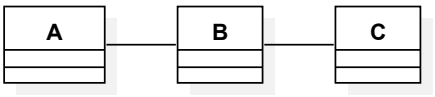


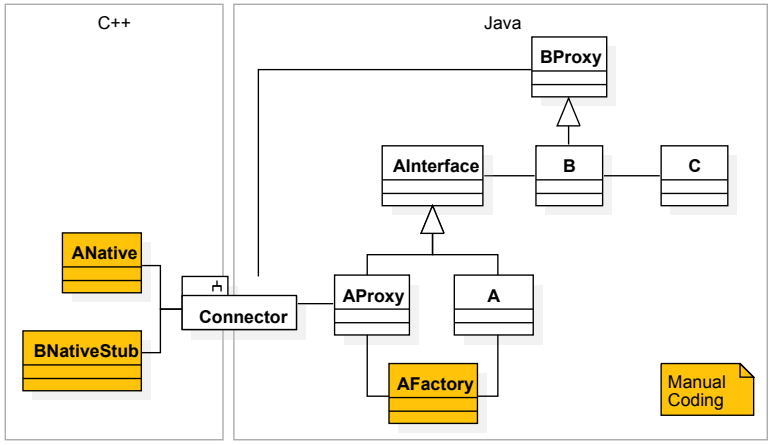
Figure 14.: `IntracallularDiffusion` Performance Monitoring

the language border. Hence, transformation rules for all parameter and return types have to be created and special code has to be be inserted for each Java class that the native implementation has to communicate. Furthermore, runtime overhead will be substantial as it can be observed in the spikes of Figure 14. Therefore, I strongly recommend to start with a class that minimizes all these negative effects. Taking the spatial organization layer (Figure 4) as an example, class `Rational` is a very good starting point opposed to `Tetrahedron` or `SpaceNode`. Additionally, `Rational` has the rare benefit that it does not call methods of other classes, thus, minimizing the required development overhead even further.

I want to explain the next steps by means of a simple example. Figure 15 shows an application composed of three classes `A`, `B` and `C`. For the first iteration we chose to port class `A`. Our next task is to “refactor the remaining Java application". This includes extracting an interface for `A`, replacing all types of `A` with its interface in the remaining application (class `B` and `C`) and creating a factory that creates new instances of `A`. The first two tasks can be performed semi-automatically



(a) Initial Example Application



(b) Architectural Changes after Porting Class `A`

Figure 15.: Porting Example

with modern development environments like Eclipse or Jetbrains IntelliJ. The factory will become important for debugging to quickly switch between Java and native implementation. Debugging is described in more detail in Chapter 4.2.5. Although not present in the overview, this would be a good time to execute all tests, especially if many classes have been affected by replacements. If they pass, I would also recommend to commit this intermediary result. Step three is the actual core assignment: translating the Java code to C++ (class `ANative`). Methods of `B` that are called from `A` must also be declared in C++, but do not require an implementation. Afterwards, code must be written to enable communication between languages. In UML diagram 15b this corresponds to classes `AProxy`, `BProxy` and the package `Connector` (see Chapter 4.2.3). Lastly, we execute all tests using native version of `A` and commit our changes if they produce the correct result.

4.2.3. Connecting Java and C++

At the beginning of Chapter 4 it was mentioned that Java provides JNI to connect both languages. However, JNI is considered low level and it would be very beneficial if `Connector` and proxy classes (Figure 15b) could be generated automatically from the declaration of `ANative`.

Fortunately, a tool called SWIG [47] exists, which connects native code with many high level languages including Java. It can be seen as a compiler that takes C++ header files and SWIG customizations as input and generates a couple of Java and C++ files. It is used in production, the

most prominent project that uses SWIG is probably Tensorflow — Google's distributed machine learning framework [48]. In retrospective it was a good decision to use this tool since the generated C++ code alone exceeded 35k LOC. However, during the whole process we also encountered its limitations, but always found a workaround.

Now I would like to examine the tool SWIG in more depth by explaining Figure 16. The labels in parenthesis draw the connection to the class diagram in Figure 15. Generated Java files have a yellow background while C++ output is blue. Contrary to the class diagram the names are all the same (`A`). Distinction is possible, because they all belong to different packages / namespaces. SWIG is organized around modules. The developer defines a set of classes that are bundled together. Modules from BioDynaMo are similar to the four layer architecture, but biology and cell have been merged. For every input class or struct SWIG generates a proxy with the same method signature as the original Java implementation. Therefore, they can both implement the same interface (`AInterface`) and become easily exchangeable. A standard proxy has only one data member, the pointer to the native object that gets created when the proxy is constructed. It is also possible to create a proxy if the pointer already exists. The method bodies of the proxy functions forward the call to the related static native function declaration in `moduleJNI.java`. They prepend the pointer of the native object as additional parameter. Due to the specified `native` keyword this function does not require a definition. The virtual machine now looks for a method with the following name to execute [49]: `Java_{mangled fully-qualified class name}_{mangled method name}`. Mangling of method name can be observed in Figure 16: The underscore of `A_foo` in `moduleJNI.java` got replaced with `_1` in the native source file.

Once the runtime found this method, execution continues in C/C++. The generated method casts the pointer passed as `long` to `A*` and lastly calls the native user defined function `foo`. The described procedure is visualized in the sequence diagram in Figure 17: "Calling Native Code from Java". The subcomponent `Connector` from the class diagram consists of four components: `moduleJNI.java`, `module.java`, `moduleJAVA_wrap.h` and `cxx`. The generated file `module.java` which has not been explained yet can be used to hold custom Java code, but is empty by default.

For the opposite calling direction let us assume that `ANative` needs to call methods from `B` which are still implemented in Java. In our terminology we call `B` a Java-defined class (JDC) — `A` would be a native-defined class (NDC). The developer needs to declare only the methods that are called by `A` and leave the body empty as it will never get executed (`BNativeStub`). Afterwards, `B` needs to be defined as JDC for SWIG. Hence, SWIG will subclass `BNativeStub`. Therefore, any call from `ANative` to `BNativeStub` will end up in the generated sub class called `SwigDirector_BNativeStub` which in turn contains code to call a static method inside `moduleJNI.java`. It prepends the corresponding Java proxy object as additional parameter which allows `moduleJNI.java` to forward the call to `BProxy`. Since `B` has been updated to subclass `BProxy` the call ends up at the intended destination due to polymorphism.

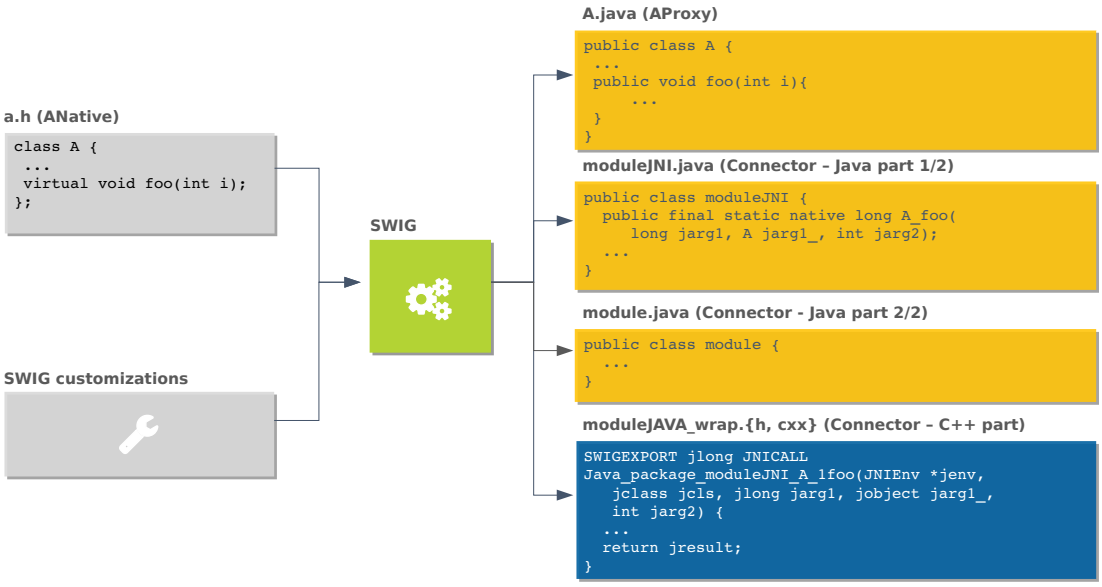


Figure 16.: Schema of the Tool SWIG: Based on the native header file and SWIG customizations, SWIG generates a set of Java and C++ source files. On the Java side a thin proxy is created, a file that contains native method declarations, and a file (`module.java`) which is empty by default, but can be used to inject custom code. On the native side two files are generated that contain methods called by JNI which forward the call to the actual implementation.

In a more complex example where a JDC is passed as an argument, this solution does not suffice (e.g. `jdc->equalTo(other_jdc)`). For parameters SWIG uses the pointer of `other_jdc` passed as `long` instead of the Java proxy object. Therefore, on the Java side, we need an association that returns the Java object given the pointer. This is implemented using a HashMap. After construction of the Java object `B`, which in turn creates an instance of `BNativeStub` this key-value pair is added to the map.

The complexity of these call sequences increases for class hierarchies (Figure 18). All displayed classes are implemented in Java. The initial situation is shown within the left rectangle. `PhysicalObject` subclasses `PhysicalNode` and is derived by `PhysicalSphere`. Let us assume that `PhysicalNode` has already been ported while `PhysicalObject` is a JDC. Once we finish porting of `PhysicalObject` it is possible to switch to the native implementation, thus, removing it from the class hierarchy. Finding bugs, requires quickly switching between native and Java implementation and turning on debugging output. Doing so for the Java version requires another type of call hierarchy (fourth column). Therefore, a mechanism is needed which rewires the classes based on SWIG's configuration. The implemented solution adds an empty nested class called `PhysicalSphereBase` inside `module.java`. `PhysicalSphere` is changed to subclass it. This base class is controlled by SWIG and can be modified to extend the required parent. Beside increased complexity, changing the class hierarchy has another nega-

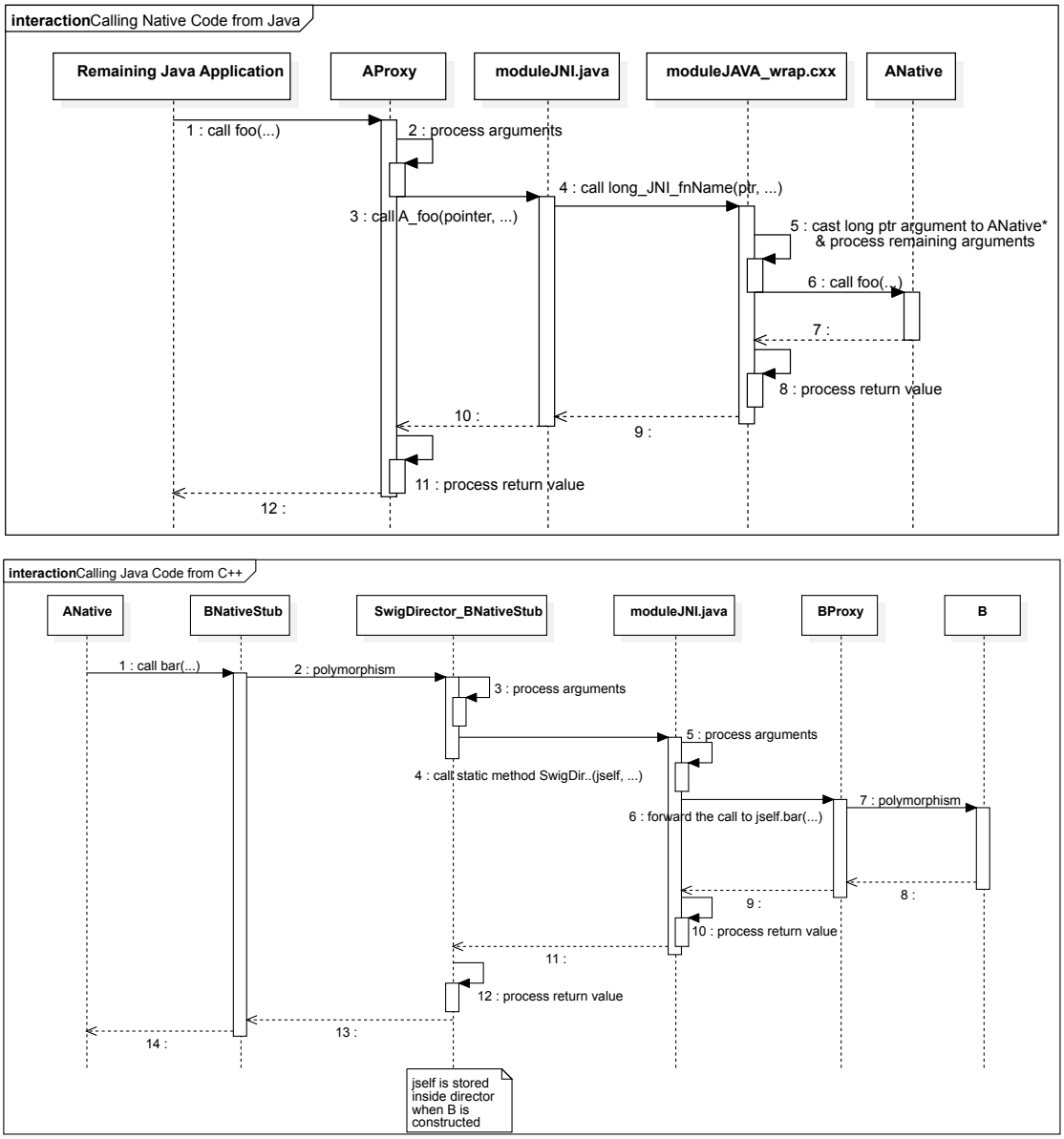


Figure 17.: Sequence Diagrams to Call Native Code from Java and Vice Versa.

tive aspect. A call inside `PhysicalSphere` to `super` might end up at the wrong destination. Unfortunately, SWIG does not allow to insert arbitray code into a proxies' method body. Therefore the only solution left is to add additional functions that redirect to the correct destination. An example of this can be seen in code Listing 4.5 on line 18 – 21 and 29 – 32. It adds the function `superSuperSimStateToJson`. Consequently, all calls in `NeuriteElement` to `super.simStateToJson(...)` must be changed to `superSuperSimStateToJson`.

It should be noted that although `PhysicalNode` has already been ported it is not strictly a NDC, but also needs attributes from a JDC due to the fact that it is subclassed by a JDC.

Using thin proxies on the Java side that are created and destroyed frequently, breaks code which assumes that the identity of an object stays the same over its lifetime (e.g. `a == b`). Even if they both point to the same C++ object the comparison will fail if the proxy objects are different. As mitigation, all these comparisons should be replaced with `java.util.Objects.equals(a, b)` and the proxies' `equals` method must be implemented accordingly. This is also of great importance for Collection classes like `Vector` and `HashMap` which assume a correct implementation to function properly.

SWIG customizations

The last component from SWIG's schema which has not been described yet is "SWIG customizations". They are used to influence code generation and are clearly separated from C++ source files. Explaining all its aspects would go beyond the scope of this chapter. Therefore, I would like to point the interested reader to SWIG's documentation [47]. Especially the chapters: Introduction, SWIG Basics, Typemaps and Java Support [1]. In general it gives the developer many possibilities to alter the output of the compilation process and is essential for the following tasks:

- Type and value transformation
- Special code needed for JDC's and NDC's
- Code to automatically load the native library on startup

Files containing SWIG code have the file extension `*.i`. It is possible to define macros which are processed by SWIG's preprocessor and helps to avoid code duplication. Code Listing 4.4 shows parts of the module definition for the biology module. Line 1 defines the module and switches the director feature on which is required for cross language polymorphism. Line 3 and 4 include SWIG files that contain macro definitions and a central definition if a class should run in native or Java mode and whether debugging is turned on. Line 6 to 18 will be added verbatim to the generated C++ files and are necessary to tell SWIG for which classes it should generate code. General code modifications and dependant modules are defined between line 20 and 38. All adaptations for a

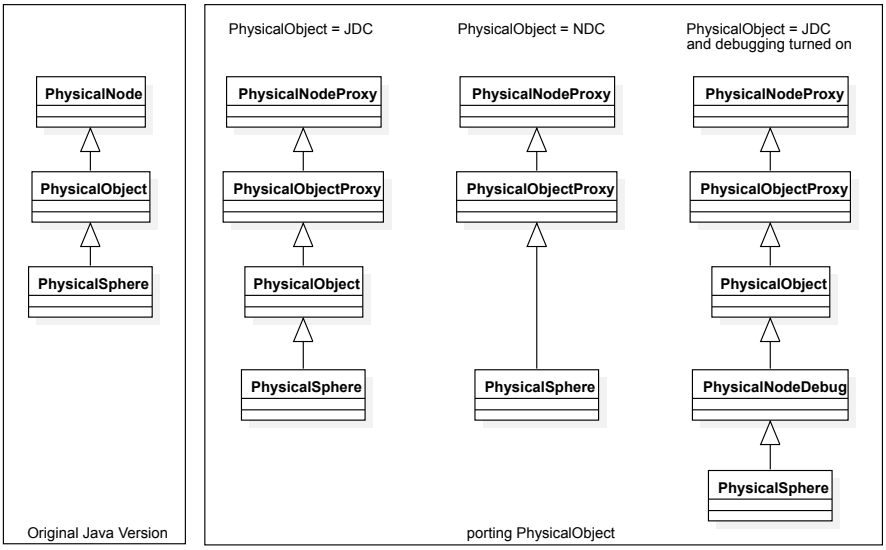


Figure 18.: Dynamic Class Hierarchy: Class hierarchy depends on the configuration of the class. Therefore, a mechanism is required, that allows SWIG to rewire the inheritance path

specific class are collected in a separate file which keeps the module code clean and readable. An example for `NeuriteElement` is shown in code example 4.5.

Line 5 – 24 define partial macro applications by fixing the value of a number of arguments. Therefore, these macros are easier to use. Customizations are applied from line 40 onwards. They issue the required code changes to use `NeuriteElement` inside a C++11 shared pointer, array and list. Furthermore, they add imports to the generated proxy class and make sure that it implements the `NeuriteElement` interface. Lastly, based on the configuration it alters the code to run the native or Java version (line 41 – 44).

4.2.4. Build Setup

Chapter 4.2.3 introduced SWIG as a precompilation step that generates Java and C++ files. The schema in Figure 19 outlines the resulting build process. Before the porting process has been started, `Cx3D` has been transformed into a Maven project. Maven is a Java build tool and dependency manager [1]. A very convenient feature is dependency management. Libraries are defined inside configuration file (`pom.xml`). Maven downloads the requested files and automatically adds it to the class path. It follows the principle “convention over configuration". Hence, the folder structure in every Maven project looks the same (see Figure 20a).

Listing 4.4: SWIG Module File

```
%module(directors="1") biology

%include "util.i"
%include "config.i"

%{
#include <memory>
#include "color.h"
#include "local_biology/abstract_local_biology_module.h"
#include "local_biology/cell_element.h"
...
using namespace cx3d::local_biology;
using namespace cx3d::cells;
using cx3d::physics::PhysicalObject;
using cx3d::physics::PhysicalSphere;
using cx3d::physics::PhysicalCylinder;
using cx3d::Color;
%}

// import depending modules
%import "cx3d.i"

// transparently load native library - convenient for user
%include "load_library.i"
JAVA_LOAD_NATIVE_LIBRARY(cx3d_biology);

// typemap definitions, code modifications / additions
%include "primitives.i"
%double_stdarray_array_marshalling(biology, 2);
%double_stdarray_array_marshalling(biology, 3);
%include "color_typemap.i"
%color(biology);
%pragma(java) jniclassimports="import ini.cx3d.swig.NativeStringBuilder;
import ini.cx3d.swig.biology.CellElement;
import ini.cx3d.swig.biology.LocalBiologyModule;
import ini.cx3d.swig.physics.PhysicalObject;
import ini.cx3d.swig.physics.PhysicalSphere;
import ini.cx3d.swig.physics.PhysicalCylinder;"

// class modifications
%include "class_customization/local_biology/cell_element.i"
%include "class_customization/local_biology/local_biology_module.i"
...
```

Listing 4.5: `NeuriteElement` Class Customizations

```
%include "util.i"
%include "cx3d_shared_ptr.i"
%include "std_list_typemap.i"
%include "std_array_typemap.i"

%define %NeuriteElement_cx3d_shared_ptr()
  %cx3d_shared_ptr(NeuriteElement,
                   ini/cx3d/localBiology/interfaces/NeuriteElement,
                   cx3d::local_biology::NeuriteElement);
%enddef

%define %NeuriteElement_java()
  %java_defined_class_add(cx3d::local_biology::NeuriteElement,
                          NeuriteElement,
                          NeuriteElement,
                          ini.cx3d.localBiology.interfaces.NeuriteElement,
                          ini/cx3d/localBiology/interfaces/NeuriteElement,
                          public NativeStringBuilder superSuperSimStateToJson(
                            NativeStringBuilder sb) {
                            return super.simStateToJson(sb);
                          });
%enddef

%define %NeuriteElement_native()
  %native_defined_class(cx3d::local_biology::NeuriteElement,
                        NeuriteElement,
                        ini.cx3d.localBiology.interfaces.NeuriteElement,
                        NeuriteElement,
                        public NativeStringBuilder superSuperSimStateToJson(
                          NativeStringBuilder sb) {
                          return super.simStateToJson(sb);
                        });
%enddef

...

/**
 * apply customizations
 */
%NeuriteElement_cx3d_shared_ptr();
#ifdef NEURITEELEMENT_NATIVE
  %NeuriteElement_native();
#else
  %NeuriteElement_java();
#endif
%NeuriteElement_stdlist();
%typemap(javaimports) cx3d::local_biology::NeuriteElement %{
  import ini.cx3d.swig.NativeStringBuilder;
  import ini.cx3d.swig.biology.LocalBiologyModule;
  import ini.cx3d.swig.physics.PhysicalObject;
  import ini.cx3d.swig.physics.PhysicalCylinder;
%}
%typemap(javainterfaces) cx3d::local_biology::NeuriteElement
 "ini.cx3d.localBiology.interfaces.NeuriteElement"
%NeuriteElement_array(biology, 2);
```

Listing 4.6: Common Maven Commands [1]

```
# remove all build artefacts
mvn clean
# compile project
mvn compile
# compile project and run unit tests
mvn test
# compile project and run a specific test
mvn -Dtest=IntracellularDiffusionTest test
```

The non trivial build steps (Figure 19) were integrated into Maven. As a result, the standard build commands from code Listing 4.6 can be used to build the whole hybrid simulator. On the native side, cmake build system was used to detect and execute SWIG and to compile the native shared libraries. For debugging purposes two cmake derivatives were created to build the system without SWIG (`cmake_wo_swig`) and to compile a native only application (`cmake_standalone`).

Native source files and SWIG customizations were integrated into the existing `src` folder structure: `src/main/cpp`. SWIG generated files are stored in `src/main/cpp/java/ini/cx3d/swig/`. The final shared library is stored in directory `src/main/resources`. The same structure applies also for the test branch.

4.2.5. Debugging

Although classes are ported incrementally, number of code changes during one iteration can still be substantial for important classes. Inevitably, errors will be introduced. The debugging frameworks helps the developer to fix issues as quickly as possible. First I want to give an overview about types of errors and their mitigation measures taken from [1]:

- **Java cannot load the native library or crashes while creating the first object**
  Test constructing this object on the C++ side in a main method and use `cmake_standalone` to compile it. The implementation of a method or constructor might be missing.

- **JVM failure**
  The generated file `hs_err_pid*.log` provides more information. It points to the function causing the issue. If the issue is in the glue code, try to fix it there directly and compile using `cmake_wo_swig`. After it has been fixed, integrate it into the SWIG code generation process.

- **Simulation outcome is different or throws Exception**
  Use debugging framework to identify the issue.

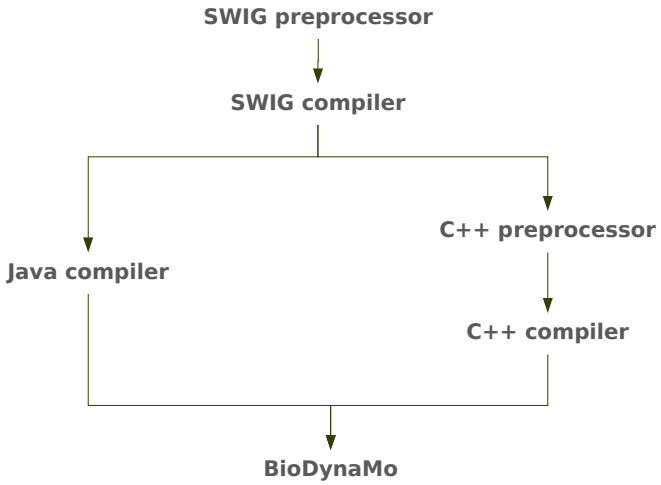
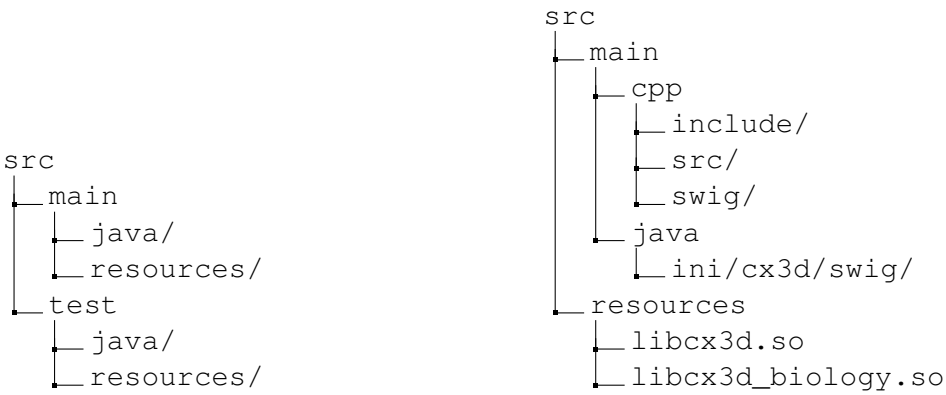


Figure 19.: Build Steps Schema



(a) Maven `src` Folder Structure **(b)** Native Code Integrated into Maven Structure

Figure 20.: `src` Folder Structure [1]

The hardest to find issues are simulation results that differ from the target. Tracking down the point in time when simulations started to diverge requires tools, because often the first difference occurs after thousands of method calls between the languages. The idea is to log all relevant information and compare the Java and C++ output to find the first difference. Hence, the feature to quickly switch between native and Java implementation is of utter importance. In Figure 21 this approach has successfully revealed a bug in the C++ `equals` method which returned a wrong result.

A generic solution that does not miss any disparity needs to log the following data [1]:

- All method calls with parameters
- Inner state before method call
- Inner state after method call
- Method return value
- (All calls to other objects from within the CPC with parameter and inner state)

If the first four bullet points are implemented, the last requirement can be achieved by turning on debugging output for all classes that are called by CPC [1].

The first version of the debugging framework was using dynamic Java proxies to perform this logging. The clear advantage lies in the fact that it is a generic solution that uses Java reflection mechanism to wrap and intercept arbitrary objects. Unfortunately, this solution had to be replaced, because it could not capture C++ to C++ calls, did not capture nested method calls and was lost sometimes during execution if the wrapped object called a method and passed itself (using `this`) as argument. Similar to the serialization of the simulation state, the second release of the debugging tools used a custom implementation. A debugging subclass is generated that produces the required outputs and forwards the call to the implementation. Listing 4.7 shows one example. Fortunately, modern integrated development environments (IDEs) are able to create most of it automatically. Post processing to eliminate errors in the generated code, was done manually.

Listing 4.7: Debug Output Generation Example [1]

```
1 bool isInsideSphere(const std::array<double, 3>& point) override {
2   logCall(point);
3   auto ret = Tetrahedron<T>::isInsideSphere(point);
4   logReturn(ret);
5   return ret;
6 }
```

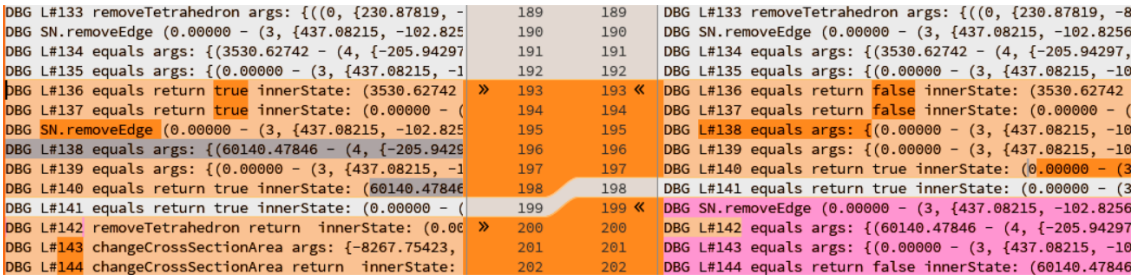


Figure 21.: Error Detection with Debugging Framework: A wrong return value of method `equals`, indicated by a difference in debugging output, leads to a diverging simulation state.

Usage and Limitations

The subsequent code example demonstrates the usage of the debugging framework under the assumption that `IntracellularDiffusionTest` fails

Listing 4.8: Debugging Framework Usage [1]

```
1 mvn -Dtest=IntracellularDiffusionTest test | grep DBG >java
2 # change implementation in config.i and rerun
3 mvn -Dtest=IntracellularDiffusionTest test | grep DBG >cpp
4 # find first difference
5 debugging_solution/find_first_diff.sh java cpp
```

The files `java` and `cpp` are usually to large to process in a diff tool. Therefore `find_first_diff.sh` splits them up into pages of 100k lines.

One of the big issues with this setup is noise due to false positives. Sometimes, language differences lead to distinct outputs although the result is correct (Figure 22). Furthermore, switching between native and Java implementation is usually limited to the CPC, but this was never an issue.

4.2.6. Numerical Instabilities

Simulations run for many hundreds of iterations. Even small rounding differences affecting only the least significant digit can amplify to an extend that validation of simulation correctness is not possible any more. Using simple JSON comparisons it is infeasible to distinguish rounding differences from actual software errors. In one of those tests (`SomaClusteringTest`), these differences where as large as a factor of two compared to the target value. The problem is that the IEEE 754-2008 standard for binary floating-point arithmetic only *recommends* correct rounding for transcendental and algebraic functions (e^x, $sin(x)$, $log(x)$, ...) [50].

Listing 4.9: Original Java Version

```
1           double[][] positions = new double[][] {
2                   adjacentNodes[0].getPosition(),
3                   adjacentNodes[1].getPosition(),
4                   adjacentNodes[2].getPosition(),
5                   adjacentNodes[3].getPosition() };
6
```

Listing 4.10: Ported C++ Code

```
1           for (size_t i = 0; i < adjacent_nodes_.size(); i++) {
2                 positions[i] = adjacent_nodes_[i]->getPosition();
3           }
4
```

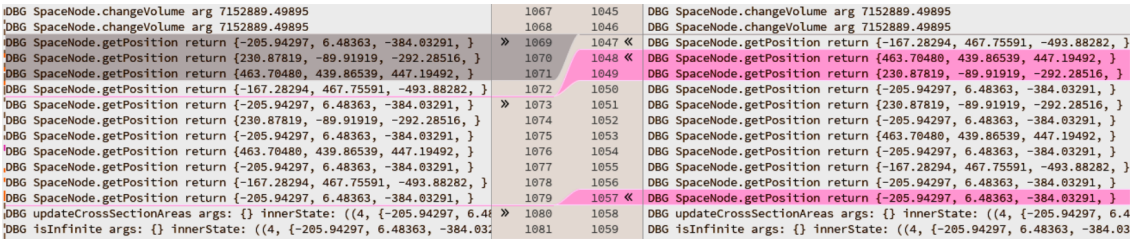


Figure 22.: Spurious Debugging Differences [1]: Variation in debugging output although Java and C++ version are equivalent. This is caused by different execution order.

In a discussion with Roman Bauer he pointed out that if these small differences have such a huge impact on the final result, than the underlying biological model is not plausible. However, our test cases are simple demonstration simulations that do not lay claim of biological relevance.

LHC@home project had similar issues and published their findings [51]. They are using a library called `crlibm` which stands for correct rounding math library. Adopting this library solved this issue, but increased runtime.

4.3. Evaluation

After porting has been finished a benchmark between the new C++ version and original Java application was performed. The results can be found in Figure 23. The term “unoptimized" in the title of the plot refers to the state of the C++ version after porting. The application is almost a one to one copy of the Java version. It is still single threaded, does not support vectorization and its architecture has not been optimized yet. Most of the tests run about 1.6 times faster. The figure clearly indicates two outliers, one in each direction. `DividingCell` simulation runs 4.8 times faster, but `SomaRandomWalkModule` improved only by a factor of 1.1x. This results were obtained on environment A with five repetitions for each measurement.

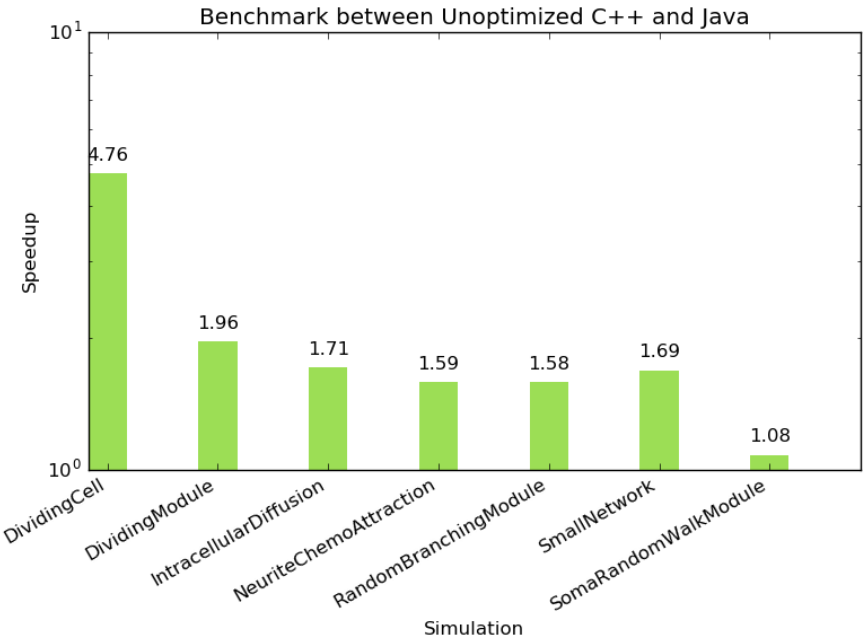


Figure 23.: Benchmark between Unoptimized C++ and Java

An investigation about the underlying cause of the outliers showed that speedup strongly depends on how much time is spent within the spatial organization layer. The C++ version of the highly abstract architecture (Figure 4) runs slower than the Java equivalent. It seems that this is related to the use of shared and weak pointers which are required by the given design. Experiments with different simulation runtimes (approximately one, two and four minutes for each test) showed that speedups are stable proving that runtime complexity remained unchanged. We did not analyse this in more depth, because we already knew that this part of the application has to be replaced, since Delaunay triangulation is not well suited for parallel and distributed environments (Chapter 2.2).

5. Parallelization – Proof of Principle

Although, porting has been finished successfully, the current C++ version of BioDynaMo does not fully utilize capabilities of modern hardware. Therefore, it is not yet possible to simulate ambitious models of e.g. epilepsy. Besides, parallelism, a second goal is to create a more flexible architecture that allows scientists to easily modify simulation objects. This chapter explains how to reach these goals.

Since 2004, industry has shifted to more and more parallelism to deliver exponential performance gains in new processor generations [9]. Before this transition, the main performance indicator of a processor was its clock speed since it increases the number of instructions executed over time. To harvest the gains of a new processor generation an application did not require any code changes.

Patt Yale described computer science as a series of layers to transform a problem defined in natural language into movements of electrons (problem $\rightarrow$ algorithm $\rightarrow$ program $\rightarrow$ ISA $\rightarrow$ microarchitecture $\rightarrow$ circuits $\rightarrow$ electrons) [52]. Applications could exploit performance gains, since changes to increase the frequency were hidden below an unchanged ISA.

From a software developer's perspective an ideal situation, but unfortunately this era is over. Chip manufacturers reached physical limitations because higher clock speed correlates with higher power consumption and heat generation. Therefore, processor vendors have shifted to more and more parallelism to push the performance envelope while frequency almost remained unchanged. This had a tremendous impact on software engineering, since hardware changes are exposed to software. Code changes are required to tap the unused potential. Benefits of code modernization efforts are shown in Schema 24. Time t_0 marks the transition from frequency to parallel driven performance. Parallelized applications remain on the peak performance line while others only benefit marginally. Let us assume that we invest in code modernization at point t_1. Usually, the graph is read in a vertical manner. Once refactoring has been finished performance increases by Δp with the benefit of fully exploiting future performance gains. Furthermore, on the horizontal time line, this means that suddenly the application is as performant as it would have been after many new processor generations t_2.

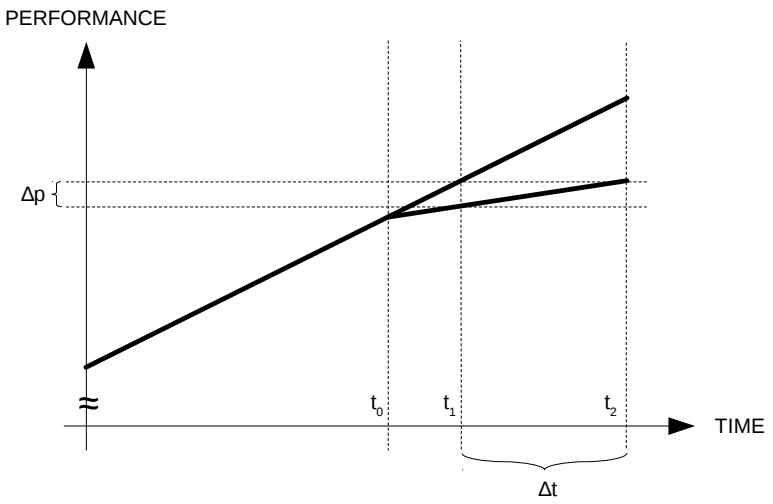


Figure 24.: Benefits of Investing in Code Modernization

Figure 25 shows the latest result after being already a decade into this transition. It is a die picture of the recently released Xeon Phi Knights Landing coprocessor. It is equipped with up to 72 cores with four threads each and AVX-512 vector instructions [53].

Parallelism is integrated on many different levels. Some are transparent to the application while others require an “enabled" program (Figure 26). Usually compute nodes in a data center are equipped with more than one processor. Each of them has a number of physical cores and hardware threads. Chip vendors replicate a number of registers in order to let these threads run concurrently. This can lead to performance improvements since many processors are superscalar. This means, that each physical core accepts more than one instruction per cycle. The current Intel Skylake architecture features seven execution units, each accepting a subset of the instructions [54]. Therefore, cycles per instruction (CPI) can be smaller than one. Instruction pipelining is used to increase parallelism inside an execution unit by separating different phases (i.e. fetch, decode, execute). The goal is to complete one instruction per cycle even if a single instruction would take multiple cycles. The last two techniques are called instruction level parallelism. They can be performed from a serial stream of instructions and are therefore transparent to software. Vectorizations are instructions that perform one operation on more than one operand at the same time. Due to the hierarchical structure of those elements, performance gains are multiplicative.

5.1. Vectorization

Vector instructions and registers have been added to processors as a means of fine grained data parallelism. In Flynn’s taxonomy of parallel machines vector instruction belong to the single instruction multiple data (SIMD) group [55]. Depending on the length of the registers, one

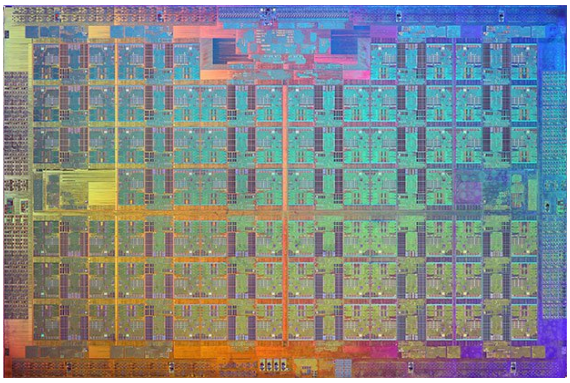


Figure 25.: Die Picture of the Xeon Phi Coprocessor with 72 cores [2].

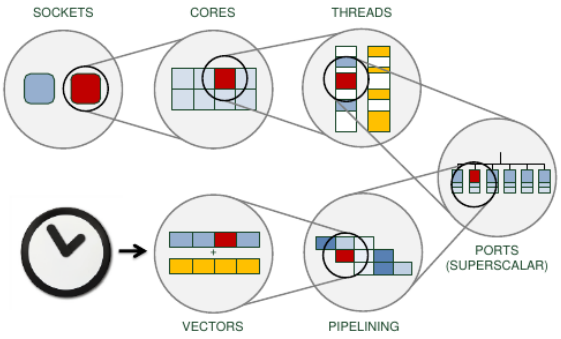


Figure 26.: Different Levels of Parallelism in Today's Modern Hardware (taken from [4]).

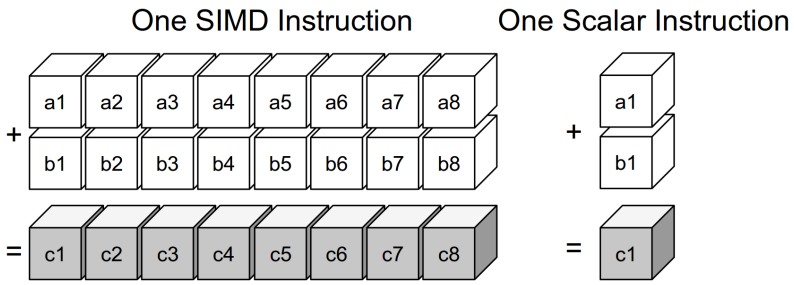


Figure 27.: Principle of Vector Instructions

operation can be executed on 16 single precision floating point values at the same time (AVX512). This is visualized in Figure 27. SIMD extensions continuously evolved from MMX with eight 64 bit wide registers that only supported integers to AVX512 with 32 512 bit wide registers [4].

There are many different ways to utilize these capabilities [3]:

- Auto vectorization
- Compiler pragmas
- SIMD library
- Compiler intrinsics
- Inline assembly

Figure 28 gives a short code example for each of these options. Due to portability and programmability using compiler intrinsics or inline assembly is not a viable option. The advanced vector instruction (AVX) extension alone has 292 instructions [56] which the programmer would have to cope with. Even if the team completes an application using one of the two options, it will only run on AVX enabled hardware. It will not run on systems with streaming SIMD extension (SSE) only and will not be ready for future standards. Lastly, code readability will suffer due to the verbose programming style. However, analysis for the LHCb experiment at CERN showed that intrinsics outperforms auto vectorization by a factor of 1.2x [57]. The most promising solution seems to be auto vectorization: gain benefits while delegating heavy lifting to the compiler. Together with explicit compiler annotations 28b it shares the disadvantage that the results will be highly compiler dependant. Although it can give good results, the process of generating vector code based on a scalar implementation is fragile. In example 28a the compiler has to check if [4]:

- `N`, `a`, `b` and `c` are loop invariant
- `c` is aliased with `a` or `b`
- `N` is large enough
- Elements are contiguous in memory → is `i` incremented in unit stride?
- Vector code is expected to be faster on the given platform (based on heuristics)

During these checks the compiler has to take a conservative standpoint and has to assume the worst case if no definitive decision can be made. As a mitigation, the compiler can insert a dynamic check to defer decision towards runtime (e.g. if it is save to execute the vectorized version), but this introduces a small overhead [58]. The most common obstacles in auto vectorization are [4, 59]:

- Mixed data types
- Data dependencies (aliasing and overlaps)
- Non contiguous memory access
- Memory alignment
- Exceptions in loop body
- Program flow statements (conditionals, loops)

Annotations are a way for the developer to convey additional information to the compiler to obtain better results. `#pragma ifdep` tells the compiler that memory regions do not overlap, the keyword `restrict` added to pointer definitions conveys that they are pointing to different memory regions and `_declspec(align(16, 8))` ensures correct memory alignment [4].

SIMD libraries like Vc [60] or UMESIMD [61] have been developed due to the shortcomings of the so far described vector solutions. They provide an easy to use interface which hides the complexity of many different vector extensions across compilers, have overloaded operators to use them like primitives and are well tested.

5.2. Memory Layout

Referring back to the vectorization analysis, data elements have to be contiguous in memory to benefit from vector instructions. Otherwise data elements cannot be loaded into the register with one instruction. In most cases however, data members in object oriented applications are laid out in the so called array of structures (AOS) format (Listing 5.1). Consequently, an operation that should be carried out only on values of `x` cannot be vectorized, because they are separated in memory. This can be remedied by using structure of arrays (SOA), but is less cache friendly. Therefore, array of structure of arrays (AOSOA) should combine the advantages of the aforementioned options.

5.3. Parallelization

While vectorization applies one instruction to multiple data elements, multi and many-core processors are able to execute multiple (different) instructions on multiple data streams MIMD [55] and are thus more flexible. Problems can be categorized by how well they are parallelizable:

- Embarassingly parallel:
  no communication between sub tasks required (e.g. event reconstruction in high-energy physics (HEP))

- Parallelizable / (tightly) coupled
  requires communication between subtasks (e.g. BioDynaMo)

- Not Parallelizable
  calculation of the Fibonacci series (see code Listing 4.2)

BioDyanMo falls into the second category. It follows an agent based approach where objects can only interact with its local environment [12, 14]. Hence, if the simulation volume is split among two different threads, cores, processors, or nodes, communication in neighboring regions is required.

After it has been clarified in which category the problem falls and thus if it makes sense to continue, it is helpful to evaluate the achievable speed-up. Therefore, I want to introduce Amdahl's law (Figure 29) which describes the maximum obtainable improvement. It is calculated with $\frac{1}{S+\frac{P}{N}} = \frac{1}{(1-P)+\frac{P}{N}}$, where P refers to the parallel portion of the code, S the serial part and N to the number of parallel execution units. Serial portions are introduced through synchronisation like locks or barriers. Consequently, if 10% of the application is serial, maximum speedup is limited to a factor of 10.

```
1  const size_t N = 8;
2  float a[N], b[N], c[N];
3  for (int i = 0; i < N; i++) {
4    c[i] = a[i] + b[i]
5  }
```

(a) Auto Vectorization

```
1  const size_t N = 8;
2  float a[N], b[N], c[N];
3  #pragma omp simd
4  #pragma ivdep
5  for (int i = 0; i < N; i++) {
6    c[i] = a[i] + b[i]
7  }
```

(b) Compiler Pragmas

```
1  float_v a, b, c;
2  c = a + b;
```

(c) SIMD Library

```
1  __m256 a, b, c;
2  c = _mm256_add_ps(a, b)
```

(d) Compiler Intrinsics

```
1  __asm {
2    vmovaps ymm0, a
3    vmovaps ymm1, b
4    vaddps ymm2, ymm0, ymm1
5    vmovaps c, ymm2
6  }
```

(e) Inline Assembly

Figure 28.: Different Vectorization Techniques [3, 4]

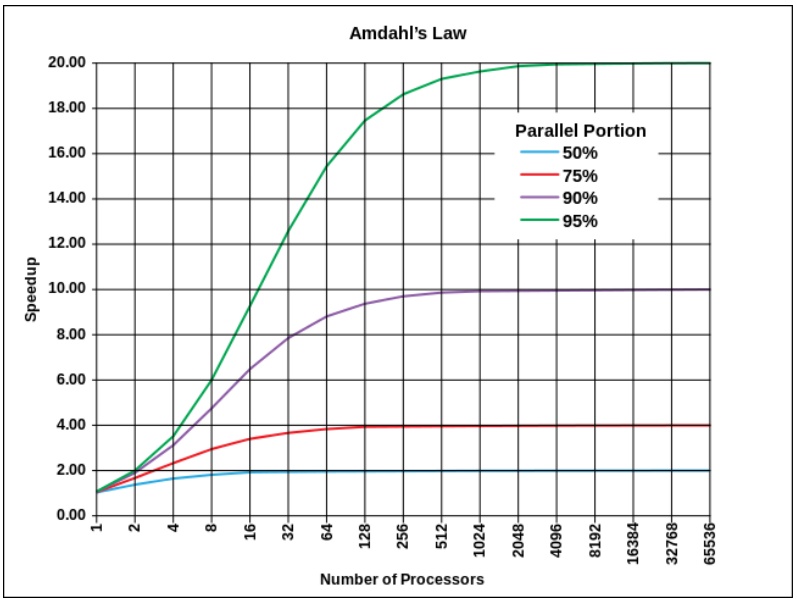


Figure 29.: Amdahl's law illustrates the maximum achievable speed-up based on the serial part of the application. Figure taken from [62] used under CC-BY-SA 3.0.

Listing 5.1: Different Data Layouts (AOS SOA AOSOA)

```
// AOS - Array of Structures
struct Point {
  float x, y, z;
};

Point points[64];             // allocate 64 points
float x_11 = points[11].x;    // single element access

-------------------------------------------------------------------------------
|  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | ..
-------------------------------------------------------------------------------
| x0 | y0 | z0 | x1 | y1 | z1 | x2 | y2 | z2 | x3 | y3 | z3 | x4 | y4 | z4 | ..
-------------------------------------------------------------------------------

// SOA - Structure of Arrays
struct Point {
  float x[64], y[64], z[64];
};

Point points;                 // allocate 64 points
float x_11 = points.x[11];    // single element access

-------------------------------------------------------------------------------
|  0 |  1 |  2 | .. | 63  | 64 | 65 | 66 | .. | 127 | 128 | 129 | 130 | 131 | ..
-------------------------------------------------------------------------------
| x0 | x1 | x2 | .. | x63 | y0 | y1 | y2 | .. | y63 |  z0 | z1  | z2  | z4  | ..
-------------------------------------------------------------------------------


// AOSOA - Array of Structure of Arrays
struct Point {
  float x[8], y[8], z[8];
};

Point points[8];              // allocate 64 points
float x_11 = points[1].x[3];  // single element access

-------------------------------------------------------------------------------
|  0 |  1 | .. |  7 |  8 |  9 | .. | 15 | 16 | 17 | .. | 23 | 24 | 25 | 26  | ..
-------------------------------------------------------------------------------
| x0 | x1 | .. | x7 | y0 | y1 | .. | y7 | z0 | z1 | .. | z7 | x8 | x9 | x10 | ..
-------------------------------------------------------------------------------
```

Parallel computing within the same address space is possible using multiple threads. A common low level implementation is pthreads (POSIX threads) which is implemented in most operating systems. Other tools like OpenMP or Intel TBB build upon this thin operating system (OS) wrapper and provide more functionality. OpenMP for example provides compiler annotations to parallelize a for loop. Intels template library offers a `parallel_for` construct which executes a provided lambda function in parallel.

Listing 5.2: OpenMP and Intel TBB Example

```
1 // OpenMP example
2 #pragma omp parallel for
3 for (size_t i = 0; i < N; i++) {
4   DoSmthUseful(i);
5 }
6 // TBB example
7 tbb::parallel_for(0, N, 1, [=](int i) {
8   DoSmthUseful(i);
9 });
```

5.4. Performance Profiling

Given the complexity of today's computer systems, understanding runtime dynamics of an application is crucial for optimization and parallel computing to answer the following questions:

- Which methods have the highest execution time?
- What are the performance bottlenecks?
- (How) can they be eliminated?

Performance depends on many aspects in the application itself and the underlying microarchitecture. Therefore, a developer relies on measurements to gain insights about the most important problems. Below is a non exhaustive enumeration of these manifold issues:

- Unnecessary copies of objects
- Unnecessary heap allocations
- Last level cache misses
- Branch mispredictions
- False sharing in multi-threaded applications

- High load latency
- Synchronisation (e.g. locks)
- Many AVX - SSE transitions

Performance data can be obtained through code instrumentation and sampling. The former solution can record detailed metrics, but slows down the application significantly (up to a factor of 100 [4]). On the other hand, event based sampling generates a statistical approximation. It periodically takes a snapshot of the execution state (stacks, registers) to calculate application metrics and to associate hardware events obtained from performance monitoring unit (PMU). This approach is possible without modifying the binary. Moreover, hardware counter do not add execution overhead and can measure a large variety of events — e.g. number of last level cache misses, number of stores. The number of available hardware performance counters varies, but most modern systems provide at least four [4]. A common tool to acquire performance data is `perf` (performance analysis tools for Linux). Code Listing 5.3 demonstrates its usage.

Listing 5.3: `perf` Usage

```
# list available hardware events
perf list
# measure fraction of last level cache misses
# outputs only counter values
perf stat -e LLC-load-misses,LLC-loads ./binary
# to see which code parts caused them, record samples with
perf record -e LLC-load-misses,LLC-loads ./binary
# and browse the results
perf report
```

One commercial tool is Intel VTune Amplifier XE [63] which is part of Intel's Parallel Studio. It follows the same sampling approach to avoid impact on execution, but offers more powerful functionality and reporting features. One of their predefined analysis types is “general exploration". It uses hardware counter multiplexing to record more than 60 performance events and aggregates them into a one page summary [64] (Figure 30).

Furthermore, it provides analysis types for hotspots, memory accesses, locks and waits, and lets the user define custom ones. This is very useful to quickly test assumptions. In one case I wanted to see whether transitions between AVX and the older standard SSE are the cause for performance issues. Within five minutes it was possible to setup a new test type and obtain results.

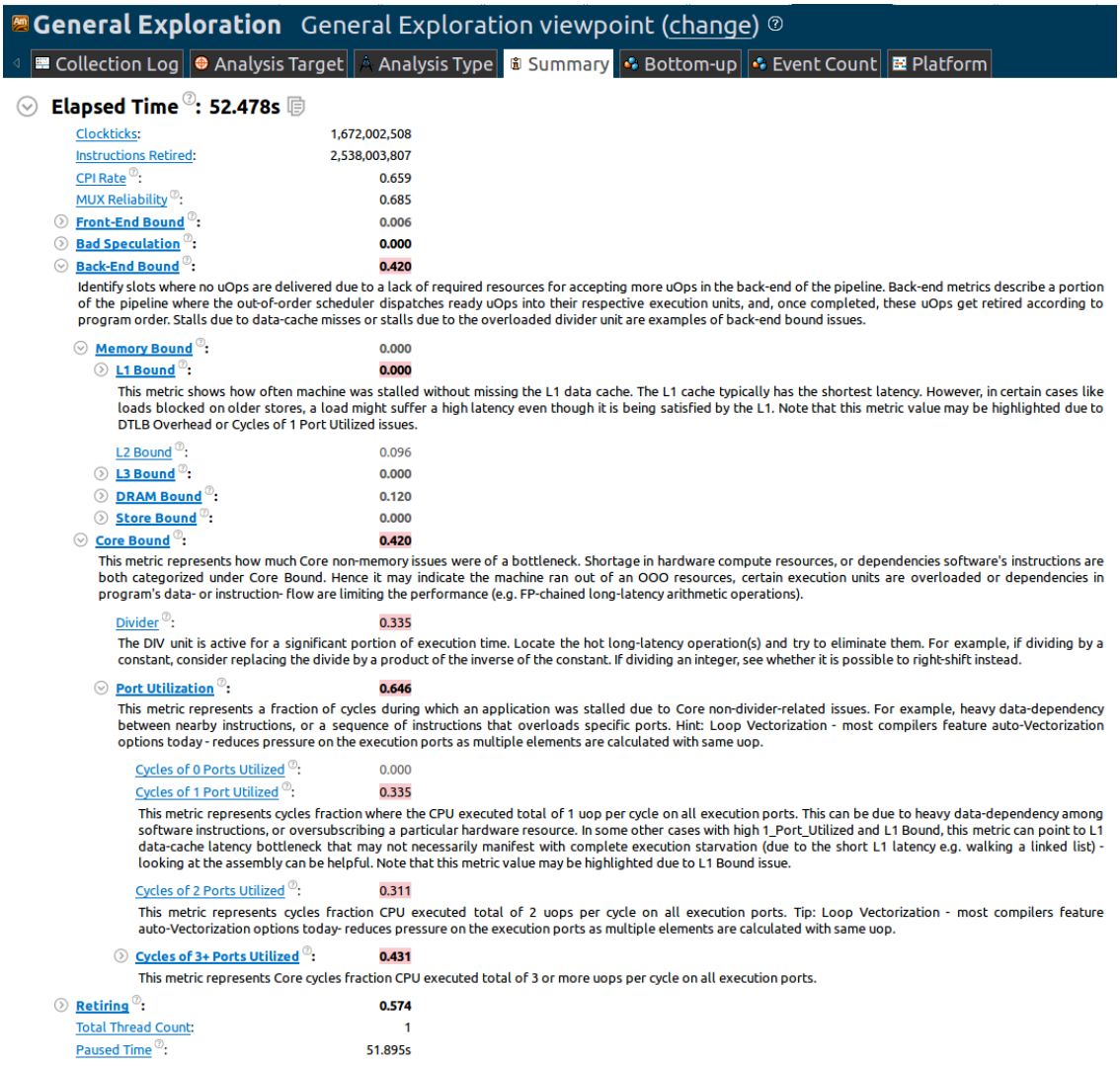


Figure 30.: Example Summary of VTune's General Exploration Analysis. VTune uses multiplexing to record about 60 different performance counters [64]. These measurements are aggregated into the displayed metrics.

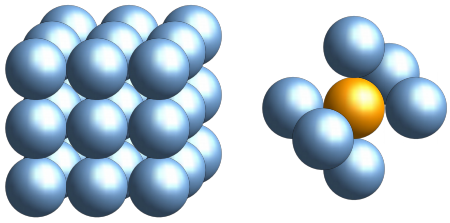


Figure 31.: Initial Simulation Situation (left), Cell Neighbors (right)

5.5. Design

Going back to the beginning of Chapter 5, our requirements for the new system are efficient utilization of multi level parallelism and a more generalizable architecture which allows scientists to quickly add their modifications.

Quick iterations are crucial to quickly converge on an efficient design with high initial uncertainty. Therefore, I decided to develop a separated proof of principle prototype. The remaining functionality will be transferred after early defects in the architecture have been eliminated. The simulation which should be carried out by the simplified simulator takes a three dimensional grid of cells as input, grows each cell and displaces them based on the mechanical forces that are exerted by its neighbors. The initial simulation state and a visualization of a cell's neighbors are shown in Figure 31. Therefore, the whole simulation translates into three main tasks: find neighbors, grow cells, calculate displacement.

While the old four layer architecture (Figure 3) which separates cells, biology, physics and spatial organization is a good choice for didactic reasons it adds unnecessary complexity and runtime overhead. Therefore, I removed this distinction and put the required data members in one object.

5.5.1. Flexibility

The desired workflow from the biologists point of view is as follows:

1. Select simulation entities (e.g. neuron, glia, plasma cell)
2. Select operations (physical interactions or biological behaviour)
3. Define initial state and parameters
4. Run simulation

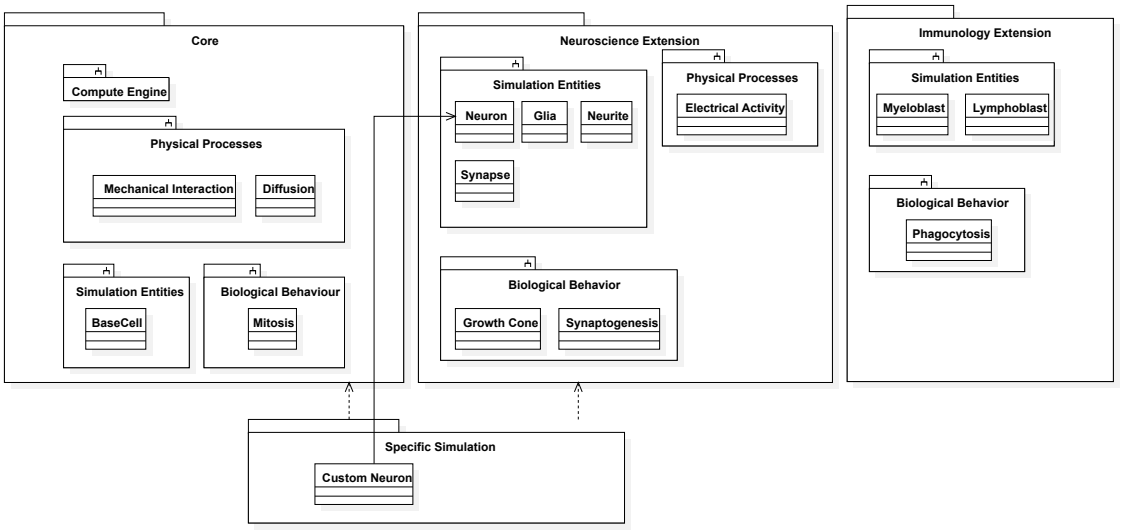


Figure 32.: Envisaged Package Structure for BioDynaMo: The core package contains the distributed compute engine and simulation components that are shared across all domains. Domain specific functionality is combined and forms an extension. For a specific simulation, the scientists can use/modify classes from core and various extensions or add completely new components.

In future, it is our goal to separate BioDynaMo into packages (Figure 32). The core will contain the compute engine as well as simulation entities, physical processes, and biological behaviour which are shared among all specialities. Functionality which is specific to a field will be released as an extension. Computational scientists can use them as is or adapt them to their needs. The neuroscience package would probably contain object definitions for neurons and glia, electrical activity as additional physical interaction and predefined biological behaviours like a neurite's growth cone.

Adding data members or functions to Objects can be done at compile or runtime. Code Listing 5.4 explores the option of runtime modifications. Class `Type` is templated with the type of the parameter. It inherits from `BaseType` to have a common interface which allows that different `Type` templates can be stored in the same container. The object which should be modified needs two: a static one which holds a template for each data member with its default value and a non static version for each instance of `Object`. Usage is shown in function `main`. Firstly, an instance for each data type is created and optionally a default value specified. Then they are added to `Object` by invoking its static `AddAttribute` function. These types are now stored in `s_attributes_`. Once an instance of `Object` is created, all types inside `s_attributes_` are copied to `attributes_` (line 26-30). They can now be retrieved from the outside application using the `Get` method specifying the parameter id as attribute and the type as template parameter.

Alternatively, data members and functions can be added at compile time. Listing 5.5 shows a sample implementation. As previously explained, it divides functionality, into core, a neuroscience

extension, and customizations. Core defines class `BaseCell`. The Neuroscience package adds an additional object `Neurite` and adds it as member to a templated parent class by means of class `Neuron`. For the actual simulation, the biologist decides to add an additional data member `foo_`. Therefore, she writes class `NeuronExtension` (line 40). The function `coreOp` demonstrates that customized objects can be passed to predefined functionality. Its parameter `cell` is templated. Hence, the compiler will accept all objects as long as they have a function `GetPosition`. This can be seen as a form of compile time duck-typing. This solution uses variadic template parameters, a feature added in C++11, to solve the construction problem. The subclasses remove their parameters and forward the remaining ones to the parent. Also types of data members can be templated which makes it possible to modify them as well and inject them into the owner (see function `main`). Default template parameters and typedefs ensure that code remains readable.

Now let us compare the two options. For the dynamic solution we expect a performance penalty imposed by the detour via the container. To quantify it I carried out a performance benchmark. It retrieved a member of type double a million times and added it to a running sum. Baseline is a class with a private double member and a get function. Compiling with `-O0` indeed decreased performance by a factor of four. However, from `-O1` onwards there is no difference in obtaining the data member. This example was deliberately written with performance in mind. It uses a fixed size array and a `short` as data member id. Although, readability and flexibility suffers as a result, the compiler is able to optimize the extra function calls out. Out of curiosity, I have also benchmarked a modification which uses a `unordered_map` with a string as key instead of an array. This analysis showed that runtime is 20 times slower even with `-O3`. All these tests were carried out on A. Although, obtaining data members performed on par with a classic implementation one big disadvantage remains. Each data member for each object is allocated on the heap. This would lead to a huge overhead if many objects are created. The compile time solution not only avoids this overhead, but is also superior in terms of readability, flexibility and robustness. The templated `Get` method requires specification of the data member type, currently using a magic number as id. Furthermore, how to add a method (`Object::AddMethod`) has not been explored. Applications are more likely to break because each constructor has to ensure that `s_attributes_` is copied accordingly. The compiler will not issue an error if this part is missing. Consequently, the clear recommendation is to use the compile time solution shown in code Listing 5.5.

Listing 5.4: Add Data Members at Runtime

```
struct BaseType {
  virtual ~BaseType() {}
  virtual BaseType* Clone() {}
};

template <class T>
class Type : public BaseType {
  T data_;

 public:
  virtual ~Type() {}
  BaseType* Clone() override {
    auto other = new Type<T>();
    other->data_ = data_;
    return other;
  }
  void Set(const T& data) { data_ = data; }
  const T& Get() const { return data_; }
};

class Object {
  static std::array<BaseType*, 3> s_attributes_;
  std::array<BaseType*, 3> attributes_;

 public:
  static void AddAttribute(short id, BaseType* type) { s_attributes_[id] = type; }
  Object() {
    for (size_t i = 0; i < s_attributes_.size(); i++) {
      attributes_[i] = s_attributes_[i]->Clone();
    }
  }
  virtual ~Object() {
    for (size_t i = 0; i < s_attributes_.size(); i++) {
      delete attributes_[i];
    }
  }
  template <typename T>
  const T& Get(short id) {
    auto base_type = attributes_[id];
    return static_cast<Type<T>*>(base_type)->Get();
  }
};

int main() {
  // add data members to class Object
  Type<std::array<double, 3> > position;
  position.Set({1, 2, 3});
  Type<Foo> foo;
  Object::AddAttribute(0, &position);
  Object::AddAttribute(1, &foo);

  // use object
  Object o;
  std::cout << o.Get<std::array<double, 3> >(0)[1] << std::endl;  // 2
  o.Get<Foo>(1).Bar();                                           // foobar
}
```

Listing 5.5: Modify and Add Data Members at Compile Time

```
// core library only defines minimal cell
class BaseCell {
  std::array<double, 3> position_;

 public:
  explicit BaseCell(const std::array<double, 3>& pos) : position_(pos) {}
  BaseCell() : position_{0, 0, 0} {}
  const std::array<double, 3>& GetPosition() const { return position_; }
};

template <typename Cell>
void CoreOp(Cell* cell) {
  std::cout << cell->GetPosition()[2] << std::endl;
}

// ----------------------------------------------------------------------------
// libraries for specific specialities add functionality - e.g. Neuroscience
class Neurite {};

// adds Neurites to BaseCell
template <typename Base, typename TNeurite = Neurite>
class Neuron : public Base {
  std::vector<TNeurite> neurites_;

 public:
  template <class... A>
  explicit Neuron(const std::vector<TNeurite>& neurites, const A&... a)
      : neurites_{neurites}, Base(a...) {}
  Neuron() = default;
  const std::vector<TNeurite>& GetNeurites() const { return neurites_; }
};

// ----------------------------------------------------------------------------
// code written by life scientists using package core and Neuroscience extension
template <typename Base>
class NeuronExtension : public Base {
  double foo_ = 3.14;

 public:
  template <class... A>
  explicit NeuronExtension(double foo, const A&... a) : foo_{foo}, Base(a...) {}
  NeuronExtension() = default;
  double GetFoo() const { return foo_; }
};

template <typename Cell>
void CustomOp(const Cell& cell) {
  std::cout << cell->GetNeurites().size() << "-" << cell->GetFoo() << std::endl;
}

int main() {
  typedef NeuronExtension<Neuron<BaseCell>> CustomNeuron;
  // note: also Neurites can be modified and then inserted into Neuron
  // typedef NeuriteExtension<Neurite> CustomNeurite;
  // typedef NeuronExtension<Neuron<BaseCell, CustomNeurite>> CustomNeuron;
  CustomNeuron cell2(1.2, std::vector<Neurite>{Neurite()},
                     std::array<double, 3>{1, 2, 3});
  CoreOp(&cell2);    // prints: 3
  CustomOp(&cell2);  // prints: 1-1.2
}
```

5.5.2. Vectorization

Another option to facilitate vector hardware which has not been part of the analysis presented earlier in the chapter is the software package Eigen [65]. It is a full linear algebra package with explicit vectorization (currently for SSE and ARM NEON) which offers much more than SIMD libraries. Google added the Tensor module to Eigen which is the cornerstone for Tensorflow. To evaluate if it is useful for BioDynaMo I implemented a simplified version of force calculation between spheres (without conditionals) in Vc and Eigen and run a benchmark. For Eigen I implemented three different versions: One using variable size two dimensional tensors (heap allocated matrix), fixed size two dimensional tensors (stack allocated matrix) and finally fixed size one dimensional tensors (stack allocated vector). Figure 33 shows the arithmetic mean out of five executions on A for each of those setups for SSE and AVX. The y-axis is using a logarithmic scale due to the poor performance of tensor options. The reason behind the long runtime of variable size tensors is heap allocations. For SSE performance is on par between the vector tensor version and Vc for optimization flags larger than one. All implementations use single precision floating point values. For AVX tests I initialized fixed size tensors with size eight. Results for all Eigen implementations were much worse than the scalar version, while Vc gives better performance compared to SSE. According to Eigen's website there is currently no AVX implementation.

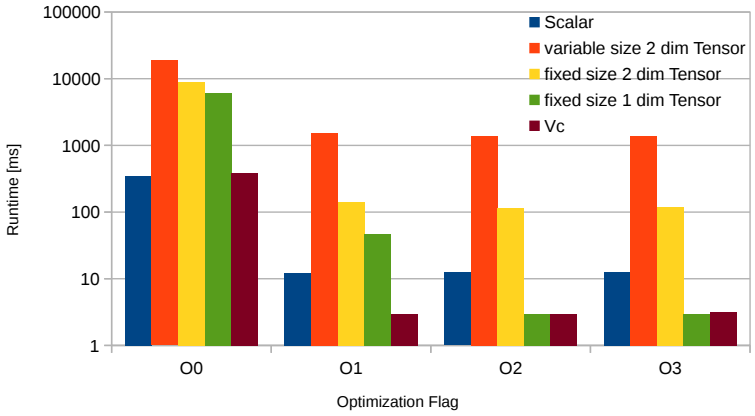
Based on the obtained results and earlier analysis about vectorization, we decided to use pure SIMD libraries which abstract the complexity of different vector extensions, are well tested and give reliable results across compilers. Using the library Vc a class can be defined in a SOA way:

```
1 struct Sphere {
2   Vc::double_v diameter_;
3 };
```

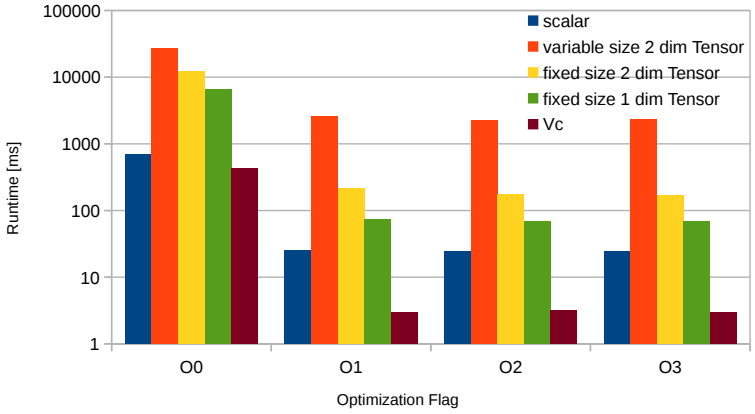
The number of elements inside `double_v` depends on the available hardware (e.g. two for SSE and four for AVX). If `Sphere` is stored inside a container with contiguous memory (e.g. array or vector), it turns into an AOSOA memory layout.

VecGeom [66] is a subpackage of the detector simulation software GeantV. It introduces an additional abstraction to exchange different SIMD libraries. They refer to them as different backends. BioDynaMo has adopted this idea. The following code example shows a minimal Backend definition in BioDynaMo:

```
1  struct VcBackend {
2    typedef Vc::double_v;
3    static const size_t kVecLen = Vc::double_v::Size;
4  }
5
6  template <typename Backend>
7  struct Sphere {
8    typename Backend::double_v diameter_;
9  };
10
11 Sphere<VcBackend> sphere;
```



(a) SSE



(b) AVX

Figure 33.: Benchmark Results Comparing Different Force Calculation Implementations.

5.5.3. Parallelization

The requirement to execute user-defined physical processes and biological behaviour requires a uniform interface such that the compute engine can process them. This becomes even more important once BioDynaMo has grown to a mature software which has a distributed runtime and is able to execute operations on heterogeneous hardware. In this scenario the runtime has to answer many questions to make an efficient scheduling decision.

- Which device class is most optimal?
- How much memory will be required?
- How large are the input parameters?

A common interface helps to gather metadata although implementation is unknown. The explicit definition of tasks or operations can be found in Intel TBB (`Task` class), Tensorflow (`Op`) or Java's `Runnable` interface.

5.5.4. Unification

The design in class diagram 34 has been distilled from all considerations so far. The main components are:

- Simulation Objects have an associated backend are stored inside containers and passed to operations as parameters.
- Backend is the abstraction to use different kind of vectorization libraries. Also contains a ScalarBackend which will be described in more detail in Section 5.6.
- Containers store simulation objects based on a vector template. They contain functionality to transition between scalar and vector representation.
- Operations contain the implementation for physical processes and biological behaviour under a uniform interface.
- Classes in package benchmark are used to obtain and output runtime data for post processing.

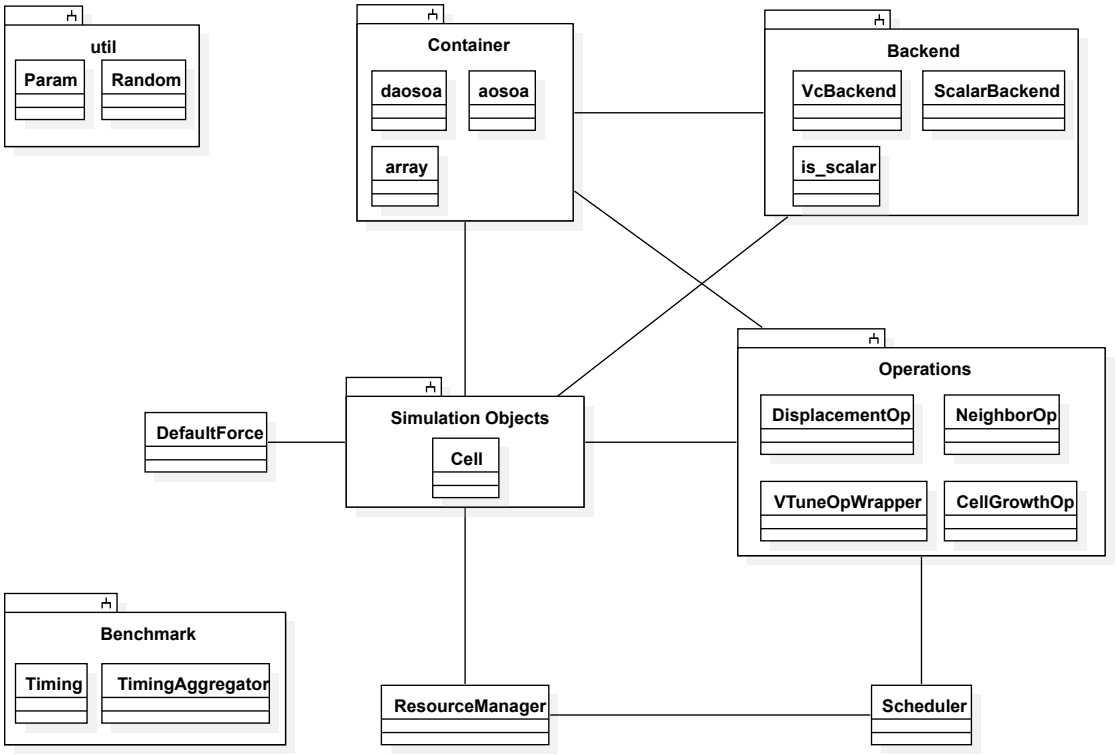


Figure 34.: Class Diagram Proof of Principle

5.6. Implementation

In this chapter I will present a selection of interesting aspects in the implementation of the proof of principle.

Although, simulation objects are predominantly processed in vector mode, sometimes it is necessary to access a single element. This involves for example gathering neighbors for a cell which are spread among different vectors. This can be done by defining a scalar backend. The principle of backends has already been shown in code sample 5.5.2. Container `daosoa`, which stands for dynamic AOSOA, is a specialized to hold classes with backends. It adds value by providing methods that ease the transition between scalar and vector. Therefore, it is possible to append get and set both of these types. Consequently, `daosoa` needs two different implementations for `push_back`: one for vector value types and one for scalars. Using template metaprogramming we need a type trait `is_scalar` to decide which of the two versions should be compiled for a certain type combination. Type traits allow to retrieve type characteristics at compile time. C++11 already defines many of them in header `<type_traits>` (e.g. `is_array`, `is_move_constructible`). Here the implementation for `is_scalar`:

```
1 template <typename T>
2 struct is_scalar {
3   static const bool value = false;
4 };
5
6 template <template <typename> class SimulationObject>
7 struct is_scalar<SimulationObject<ScalarBackend> > {
8   static const bool value = true;
9 };
```

First, we define a templated structure which has a static boolean member set to false. In a partial template specialization, we define the case for the scalar backend where `value` should be set to true. Thereby, we have to use a template template parameter since `ScalarBackend` is itself a template parameter for its enclosing type. The alternative would be a definition like this:

```
1 template <>
2 struct is_scalar<Cell<ScalarBackend> > {
3   static const bool value = true;
4 };
```

This is using the concrete type of simulation object `Cell`. Consequently, if we would add a different simulation object, we would need to define `is_scalar` again—otherwise, it would give the wrong result. This is very error-prone and should therefore be avoided. Now that the desired type trait exists we can use it in `daosoa`.

```
1 template <typename T1>
2 typename std::enable_if<is_scalar<T1>::value &&
3                          !std::is_same<value_type, T1>::value>::type
4 push_back(const T1& value) { ... }
```

This code example shows one of the two implementations for `push_back`. It only gets compiled if `T1` is scalar and the class template `value_type` and `T1` are not the same. The underlying mechanism is based on SFINAE (substitution failure is not an error). If a failure occurs during template argument substitution the compiler removes this entity from the candidate set and does not throw an error [31]. Let us dissect this example. Line two and three define the return type of `push_back`. Simplified, the statement looks like this:

```
1 typename std::enable_if<expression>::type
```

`type` is only defined if `expression` evaluates to true. Hence, if `expression` is false it leads to a substitution error and is removed. The keyword `typename` is required, because `type` is not a static variable name, but a type definition. As mentioned above type traits are needed to evaluate the expression at compile time.

Working with vector types is straight forward. They overload all math operators and can therefore be used in the same way as primitives. Special attention must be given to conditionals. Remember, that SIMD instructions always apply the same operation to multiple data elements. Therefore, conditionals require code changes. In Vc comparison operators are implemented in a way to return a bit mask. This allows evaluation of an if-expression for multiple elements.

```
1 Vc::float_v x((const float[]){1, 2, 3, 4});  // assumes vector size of 4
2 auto x_lt_3 = x < 3;  // [1, 1, 0, 0]
```

Vc also has an API that generalizes the ternary operator to vectors.

```
1 double result =  boolean_expression ? true_value : false_value;  // ternary operator
2 // equivalent to
3 if (boolean_expression)
4   result = true_value;
5 else
6   result = false_value;
7
8 // for vectors
9 Vc::double_v result = Vc::iif(bit_mask, true_value_v, false_value_v);
```

If the operation inside the conditional has a neutral element (e.g. zero for addition), then a conditional can be transformed as follows:

```
1 // original scalar code
2 if (x > 9)
3   a += 3.14;
4
5 // vector version - assuming vector length of 4
6 auto float_v summand = 3.14;      // [3.14, 3.14, 3.14, 3.14]
7 auto x_gt_9 = x > 9;              // based on x - let us assume [1, 0, 0, 1]
8 summand.setZeroInverted(x_gt_9);  // [3.14, 0, 0, 3.14]
9 a_v += summand;
```

More complex situation could require that both branches are evaluated and merged in the end using `Vc::iff`. This of course adds an overhead. Another problem is early exits from a function.

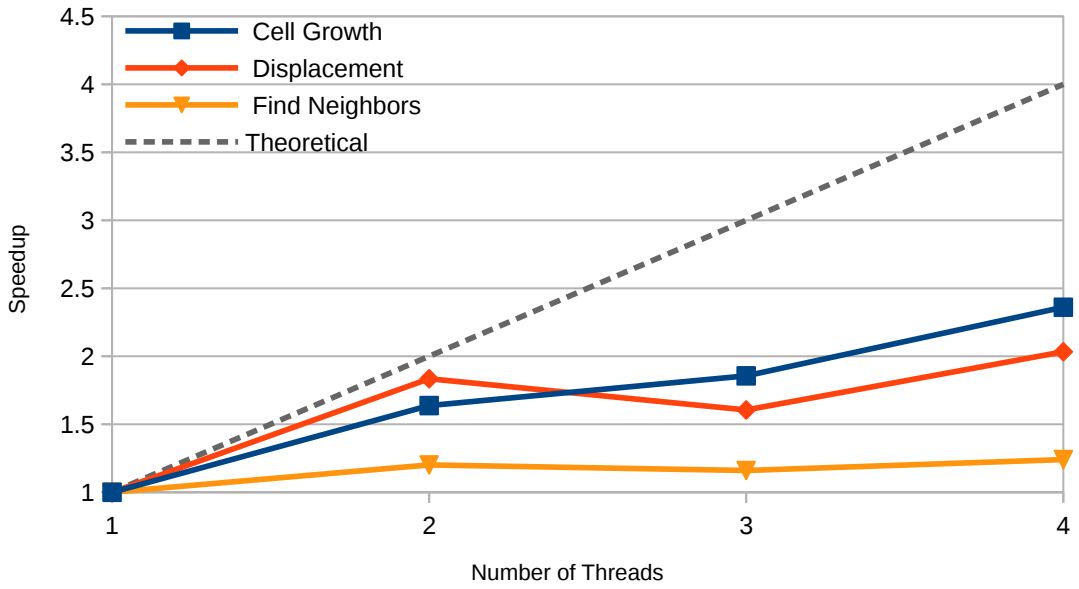
```
// scalar version
if (x)
  return 0.0f;
expensiveCalculation();
...

// vector version
if (x.isFull())       // early exit is only possible if x is 1 for all elements
  return float_v(0);
expensiveCalculation();
```

The more elements in a vector, the less likely it is that all would exit early. In general, functions with many branches are hard to handle with vectorized code.

Adding multithread support is easy due to the designed architecture (Figure 34). Operations iterate over all cells and are independant of each other for one discrete time step. Therefore, it is sufficient to add a OpenMP pragma above the main for loop of an operation:

```
template <typename daosoa>
Vc_ALWAYS_INLINE void Compute(daosoa* cells) const {
#pragma omp parallel for
  for (size_t i = 0; i < cells->vectors(); i++) {
    ...
  }
}
```

After a first version of the proof of principle prototype had been finished, I carried out a scaling analysis (Figure 35) on a Lenovo Thinkpad X1 Carbon with an Intel i7-5500U processor. More details can be found in appendix A. The CPU features two physical cores with two hardware threads each. The “Find Neighbor" operation is only partially multi-threaded and only added for completeness. The parallel part only accounts for about a quarter of the total runtime of this operation. Consequently, maximum speedup is 1.33 according to Amdahl. For this prototype I have used the library nanoflann to perform a fixed-radius near neighbor search. Another team is working on a parallel version. The behaviour of the remaining two operations (cell growth and displacement) looks promising, but looking at the raw data in Figure 35 it is striking that calculating the displacement of cells based on collisions is very time consuming. Further analysis, using Intel VTune unveiled a number of hotspots in the code base (Figure 36a). Astonishingly, the most time consuming function is random number generation. Furthermore, the top down function view (Figure 36b showed that obtaining neighbors accounts for 45% total CPU time. The actual calculation of mechanical forces and the resulting movement of cells is almost negligible in this listing. The runtime of `GetNeighbors` is particularly alarming, because it does not calculate the neighbors, but takes the predetermined Cell identifiers and returns a copy.



Threads	Cell Growth [ms]	Displacement [ms]	Find Neighbors [ms]	Setup [ms]
1	69	1653	4044	68
2	42	901	3368	68
3	37	1030	3487	68
4	29	813	3260	68

Figure 35.: Initial Scaling Analysis on A

Top Hotspots

This section lists the most active functions in your application.

Function	Module	CPU Time
bdm::Random::next	libbiodynamo.so	8.583s
operator new	libstdc++.so.6.0.21	4.813s
__libc_free	libc-2.21.so	2.867s
bdm::Random::nextDouble	libbiodynamo.so	2.283s
__cbrt	libm-2.21.so	1.376s
[Others]	N/A*	26.872s

*N/A is applied to non-summable metrics.

(a) Hotspots

Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-dow

Grouping: Call Stack

Function Stack	CPU Time: Total — Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spin Time	Ov.. Time
Total	4.679e+01s	0.0%	0.0%
_start	4.679e+01s	0.0%	0.0%
__libc_start_main	4.679e+01s	0.0%	0.0%
main	4.679e+01s	0.0%	0.0%
execute	4.675e+01s	0.0%	0.0%
bdm::DisplacementOp::Compute<bdm::daosoa<bd	4.102e+01s	0.0%	0.0%
GOMP_parallel	4.099e+01s	0.0%	0.0%
bdm::DisplacementOp::Compute<bdm::daosoa<	4.088e+01s	0.0%	0.0%
bdm::Cell<VcBackend>::GetNeighbors	1.859e+01s	0.0%	0.0%
bdm::Cell<VcBackend>::GetForceOn	1.852e+01s	0.0%	0.0%
bdm::DefaultForce<VcBackend>::forceBetwee	1.848e+01s	0.0%	0.0%
bdm::Random::NextNoise<VcBackend>	1.384e+01s	0.0%	0.0%
_ZN4Vc_1dvIdNS_9VectorAbi3SseENS_6Vect	1.061e+00s	0.0%	0.0%
_ZN4Vc_1mlIdNS_9VectorAbi3SseENS_6Vect	9.557e-01s	0.0%	0.0%
Vc_1::sqrt<double>	4.568e-01s	0.0%	0.0%
_ZN4Vc_1plIdNS_9VectorAbi3SseENS_6Vectc	4.146e-01s	0.0%	0.0%
_ZN4Vc_1mlIdNS_9VectorAbi3SseENS_6Vect	3.192e-01s	0.0%	0.0%
Vc_1::iif<Vc_1::Mask<double, Vc_1::VectorAb	2.771e-01s	0.0%	0.0%
Vc_1::sqrt<double>	1.365e-01s	0.0%	0.0%

(b) Top-Down View

Figure 36.: Intel VTune Advanced Hostpots Analysis

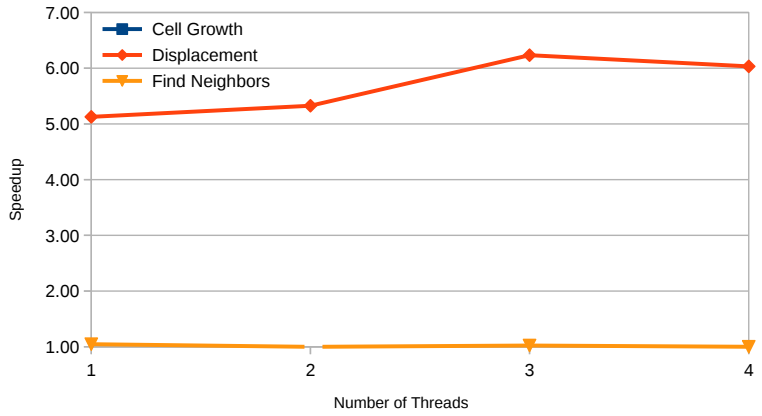


Figure 37.: Speedup Due to Improvements in `GetNeighbors`

The following changes were made to improve runtime:

- Avoid unnecessary random number generation
- Add missing `&` to function parameter
- Add `CopyTo` method to copy data members from one vector into another - originally, this was done with an intermediate scalar object
- Use vector's `reserve` method instead of `resize` - to avoid pushing back an empty cell - saves one copy constructor invocation
- Remove all heap allocations by using fixed size arrays

In one of the standard test cases with about two million cells, a missing reference for a function parameter can impact performance negatively. In this case it lead to two million unnecessary heap allocations. The last bullet point to remove heap allocations would crash the program in case there are more neighbors than slots. This is of course not acceptable. This measure should only reveal the performance penalty of dynamic containers. Furthermore, the negative aspect can be avoided. Inspired by Google Tensorflow, the neighbor array will be replaced with an inline vector. The first `N` elements, specified as template parameter, are stored on the stack. Only if additional data elements should be appended, heap space will be allocated. Based on these changes I achieved a speedup of around 5.5x (Figure 37).

Profiling has been a great help to identify these issues. To get more targeted results, VTune offers an API to pause and resume data taking in certain sections and is able to annotate tasks. Therefore, I have added the class `VTuneOpWrapper`. Annotations can be seen in Figure 38.

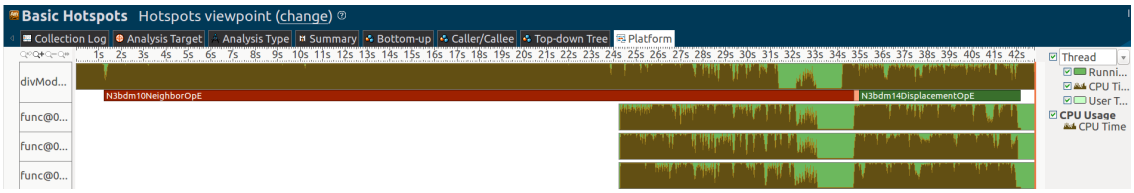


Figure 38.: Annotations Created Using `VTuneOpWrapper`. These annotations, for example, make it easier to analyse the CPU utilization of certain parts of the application.

5.7. Evaluation

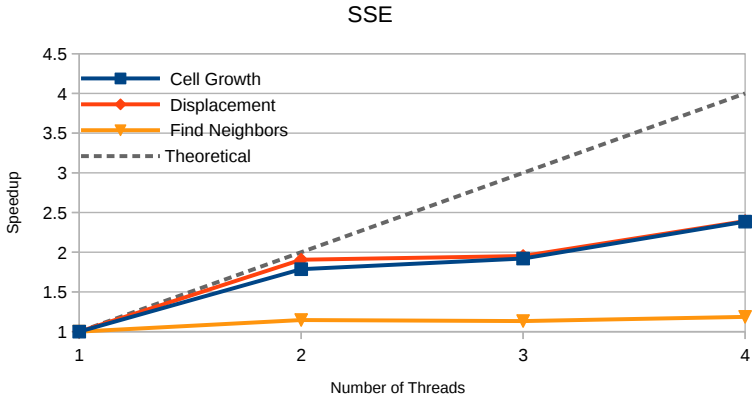
After improvements have been applied, I ran the scaling benchmark again and obtained better results. Runtime does not improve after the number of physical cores is exceeded, because the processor's floating point unit is already saturated. A comparison between AVX and SSE only showed modest improvements. Intuitively, one would expect a maximum speedup of 2x, due to the doubled register length. Let us investigate why we are not achieving better results and if this is a reasonable assumption.

Computationally, cell growth is very lightweight. A considerable part in is not vectorized, because Vc does not offer a vector version for cubic roots yet. However, an issue has already been filed and it seems someone is already working on it [67]. Therefore, our implementation has to use a for loop to iterate over the vector elements. For displacement calculation the situation is more difficult. Let us start by evaluating the maximum speedup that we can expect. The majority of cells in the three dimensional grid have six neighbors. Hence, we need three vectors for SSE versus two for AVX to store them. Therefore, speedup is already limited to a factor of 1.5x. Furthermore, force calculation accounts only for about 50% of wall clock time. The rest is spent on gathering neighbors. However, the absolute number of neighbors does not change. Hence, AVX will not improve this part. The following equation shows the maximum speedup for our sample simulation (Amdahl's law):

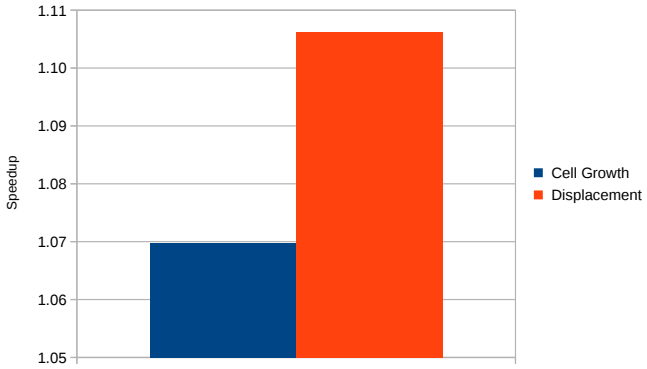
$$\text{max speedup} = \frac{\text{runtime for SSE}}{\text{runtime for AVX}} = \frac{\text{neighbors} + \text{forces}}{\text{neighbors} + \frac{\text{forces}}{1.5}} = \frac{1}{0.5 + \frac{0.5}{1.5}} = 1.2 \tag{5.1}$$

This means, we still do not have an explanation for the missing 10%. Further analysis revealed that corresponding instructions between SSE and AVX have different latency on Intel i7-5500U (see Table 5). Reciprocal throughput is defined as "average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread" [6]. For these instructions, although vector length doubled for AVX, increased latency cancels out the improvements.

To test if the analysis was correct, I modified the code and executed the benchmark again (Figure 40). I removed the non vector cubic root in cell growth, replaced the division operator with an



(a) Scaling (2M cells)



(b) SSE vs AVX (512 cells)

Figure 39.: Analysis on an Intel i7-5500U CPU with two Physical Cores

Instruction	Vector ISA	Latency	Reciprocal Throughput
DIVPS	SSE	10-14	4-5
VDIVPS	AVX	17	10
DIVPD	SSE	10-14	8
VDIVPD	AVX	19-23	16
SQRTPD	SSE	15-16	8-14
VSQRTPD	AVX	27-29	16-28

Table 5.: SSE and AVX Instruction Comparison for Intel Broadwell Architecture [6]

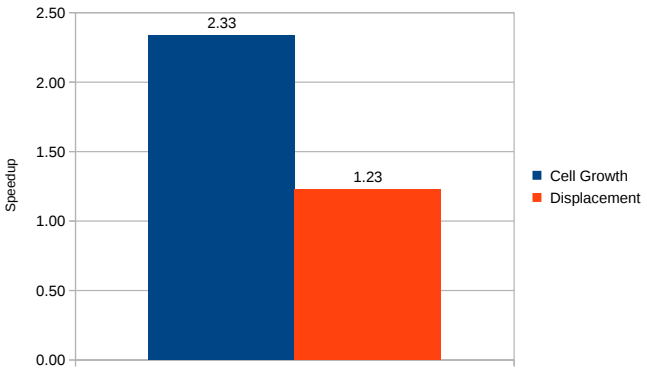


Figure 40.: Hypothetical Scenario to Compare SSE and AVX (512 cells)

```
1  movapd  -16(%rax), %xmm0
2  cmplepd %xmm7, %xmm0
3  andpd   %xmm6, %xmm0
4  mulpd   %xmm5, %xmm0
5  addpd   (%rax), %xmm0
6  movapd  %xmm0, %xmm1
7  cmpltpd %xmm3, %xmm1
8  movapd  %xmm1, %xmm2
9  andnpd  %xmm0, %xmm1
10 andpd   %xmm3, %xmm2
11 movapd  %xmm1, %xmm0
12 orpd    %xmm2, %xmm0
13 movaps  %xmm0, (%rax)
14 mulpd   %xmm4, %xmm0
15 movaps  %xmm0, -16(%rax)
```

(a) SSE

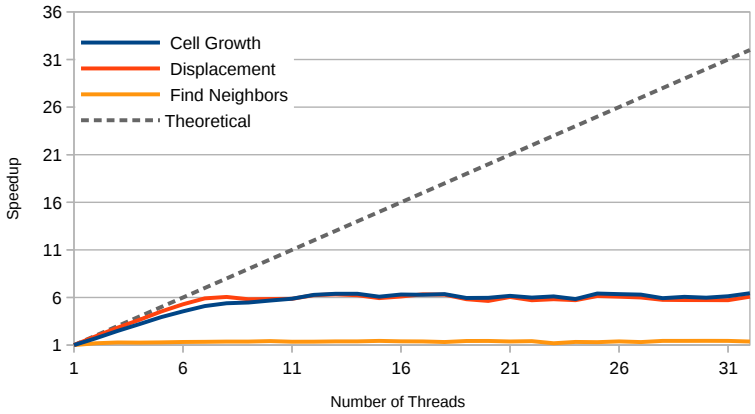
```
1  vmovapd    -32(%rax), %ymm0
2  vcmppd     $2, .LC56(%rip), %ymm0, %ymm0
3  vandpd     .LC57(%rip), %ymm0, %ymm0
4  vmulpd     .LC25(%rip), %ymm0, %ymm0
5  vaddpd     (%rax), %ymm0, %ymm0
6  vcmppd     $1, .LC58(%rip), %ymm0, %ymm1
7  vblendvpd  %ymm1, .LC58(%rip), %ymm0, %ymm0
8  vmovapd    %ymm0, (%rax)
9  vmulpd     .LC59(%rip), %ymm0, %ymm0
10 vmovapd    %ymm0, -32(%rax)
```

(b) AVX

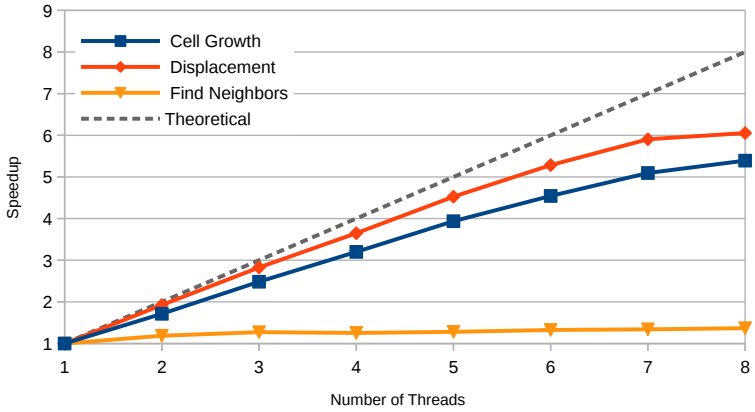
Figure 41.: Cell Growth Assembly Code

addition which has the same latency on Intel Broadwell architecture for SSE and AVX and removed square root calculations. Of course the simulation yields wrong results after these changes, but this is to test whether there are further issues in the design. This time we get much better results. Displacement reaches the estimated maximum and execution for cell growth is 2.3 times faster. A speedup exceeding a factor of two is certainly suspicious. Hence, I performed an investigation to see if this result is plausible. Analysing the generated assembly code for SSE and AVX, we see twice as much moves for SSE. This is due the new three operand form that preserves register values. Furthermore, line 8 to 12 on SSE is done in one instruction in AVX (`vblendvpd`). Based on these results, a speedup of 2.3x seems logical.

Finally, I executed the benchmark with more than 160 million cells (512 cells per dimension) on an a server with eight physical cores with four threads each (Intel Xeon E5-2690) and 64GB of memory — more details in A. This test consumes 20.8GB of main memory. After yielding good results up to the number of phyiscal cores, speedup saturates as expected.



(a) All



(b) Details up to the Number of Physical Cores

Figure 42.: Benchmark of ∼160M Cells on Intel Xeon E5-2690

6. Conclusion and Future Work

This thesis presented the first steps to build a large-scale platform for biological computer simulation. It emphasized the importance of a rigorous development environment to foster high code quality, demonstrated how to successfully port an application from Java to C++ and introduced a parallelized proof of principle prototype.

Chapter 3 is devoted to development best practices and explored reasons that hinder sustainable software development. Furthermore, we have compared different software stacks against each other to select a combination that fits our requirements and presented the process implementation for BioDynaMo. Working with this setup has shown that some modifications will be necessary in the future. This includes adjustments of the coding style guide, using a superior code formatter and lint tool, adding a git commit hook to enforce checks and automatically upload documentation to the projects website.

The chapter about porting the application from Java to C++ outlined benefits of a native implementation and introduced an iterative porting workflow. Although, development of intermediate Java / C++ hybrid added a significant overhead, it ensured that errors can be revealed quickly.

The last part of this thesis explained how to utilize multiple levels of parallelism in today's modern hardware. A proof of principle has been developed that simulates cell growth of a three dimensional grid and displaces cells according to mechanical forces. Benchmarks have shown good scaling behaviour across physical cores and speed-ups based on utilization of vector instructions. Further benchmarks need to be executed on the latest processors and a backend for GPU's should be evaluated. Increasing complexity of the simulation should stress the design and reveal possible issues. Another important step is to replace neighbor calculations with a more optimal implementation.

Once these tasks have been completed, the next big milestone is to execute on many compute nodes. We are especially targeting a hybrid HPC on cloud environment. Simulating e.g. the entire cortex to better understand neurodevelopmental diseases exceeds the capabilities of some clusters. Therefore, compute resources should be extended with cloud infrastructure. Another reason behind targeting HPC on cloud is democratization of computational resources, since not every research team has access to a supercomputer.

Although, cloud computing is very successful due to on-demand, elastic resources, its disadvantages have limited widespread adoption in scientific computing. Especially, high network latency limit scalability for tightly coupled compute intensive applications. Furthermore, cloud data centres are built using commodity hardware. Consequently, failures become a norm not the exception. Therefore, further research is required to compensate for these drawbacks.

References

[1] L. J. Breitwieser, R. Bauer, M. Manca, and F. Rademakers, "Porting a Java-based Brain Simulation Software to C++," Sept. 2015.

[2] HardwareZone, "Meet Knights Landing, Intel's 2nd gen Xeon Phi coprocessor!." http://www.hardwarezone.com.sg/tech-news-meet-knights-landing-intels-2nd-gen-xeon-phi-coprocessor. (Accessed on 10/11/2016).

[3] A. Gheata, "The GeantV project." Intel Modern Code Developer Challenge, 2016.

[4] A. Nowak, "Introduction to Efficient Computing." thematic CERN School of Computing, 2016.

[5] "BioDynaMo Github Project." https://github.com/BioDynaMo/biodynamo. (Accessed on 10/10/2016).

[6] F. Agner, "Instruction tables." http://www.agner.org/optimize/instruction_tables.pdf. (Accessed on 10/20/2016).

[7] L. Breitwieser, R. Bauer, A. Di Meglio, L. Johard, M. Kaiser, M. Manca, M. Mazzara, F. Rademakers, and M. Talanov, "The BioDynaMo Project: Creating a Platform for Large-Scale Reproducible Biological Simulations," *4th Workshop on Sustainable Software for Science*, 2016.

[8] A. Eklund, T. E. Nichols, and H. Knutsson, "Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates," *Proceedings of the National Academy of Sciences*, 2016.

[9] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[10] W. Kahle and M. Frotscher, *Taschenatlas Anatomie, Band 3: Nervensystem und Sinnesorgane*. Georg Thieme Verlag, 2013.

[11] "CERN openlab." http://openlab.cern/. (Accessed on 10/24/2016).

[12] F. Zubler and R. Douglas, "A framework for modeling the growth and development of neurons and networks," *Frontiers in computational neuroscience*, vol. 3, p. 25, 2009.

[13] M. A. Heroux, "Sustainable & Productive: Improving Incentives for Quality Software," 2016. WSSSPE4.

[14] A. Hauri, *Self-construction in the context of cortical growth*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 21173, 2013, 2013.

[15] Wikipedia, "Octree." `https://en.wikipedia.org/wiki/Octree`. (Accessed on 11/02/2016).

[16] "LAMMPS Molecular Dynamics Simulator." `http://lammps.sandia.gov/`. (Accessed on 10/21/2016).

[17] K. Lykov, X. Li, H. Lei, I. V. Pivkin, and G. E. Karniadakis, "Inflow/outflow boundary conditions for particle-based blood flow simulations: Application to arterial bifurcations and trees," *PLoS Computational Biology*, vol. 11, no. 8, 2015.

[18] "LAMMPS Developer Guide." `http://lammps.sandia.gov/doc/Developer.pdf`. (Accessed on 11/01/2016).

[19] S. J. Plimpton, "Overview of Molecular/Particulate Dynamics." `http://lammps.sandia.gov/tutorials/sor13/SoR_01-Overview_of_MD.pdf`, 10 2013. Annual Society of Rheology meeting in Montreal –(Accessed on 11/01/2016).

[20] J. Niskanen and H. Henschel, "Lecture notes in Molecular Dynamics Simulations." `http://www.courses.physics.helsinki.fi/fys/moldyn/lectures/L3.pdf`, September 2013.

[21] R. A. Koene, B. Tijms, P. van Hees, F. Postma, A. de Ridder, G. J. Ramakers, J. van Pelt, and A. van Ooyen, "NETMORPH: a framework for the stochastic generation of large scale neuronal networks with realistic neuron morphologies," *Neuroinformatics*, vol. 7, no. 3, pp. 195–210, 2009.

[22] G. A. Ascoli, D. E. Donohue, and M. Halavi, "NeuroMorpho.org: a central resource for neuronal morphologies," *The Journal of Neuroscience*, vol. 27, no. 35, pp. 9247–9251, 2007.

[23] "CellModeller." `https://haselofflab.github.io/CellModeller/`. (Accessed on 10/21/2016).

[24] T. J. Rudge, P. J. Steiner, A. Phillips, and J. Haseloff, "Computational modeling of synthetic microbial biofilms," *ACS Synthetic Biology*, vol. 1, no. 8, pp. 345–352, 2012.

[25] "Biocellion." `http://biocellion.com/#`. (Accessed on 10/21/2016).

[26] "NEST Simulator." `http://www.nest-simulator.org/`. (Accessed on 10/21/2016).

[27] "COMSOL Multiphysics® Modeling Software." https://www.comsol.de/. (Accessed on 10/21/2016).

[28] H. Markram, E. Muller, S. Ramaswamy, M. W. Reimann, M. Abdellah, C. A. Sanchez, A. Ailamaki, L. Alonso-Nanclares, N. Antille, S. Arsever, *et al.*, "Reconstruction and simulation of neocortical microcircuitry," *Cell*, vol. 163, no. 2, pp. 456–492, 2015.

[29] C. C. for IT Software Quality, "CISQ Code Quality Standards." http://it-cisq.org/standards/. (Accessed on 10/11/2016).

[30] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[31] "Working Draft, Standard for Programming Language C++." http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2011/n3242.pdf. (Accessed on 10/03/2016).

[32] "Google C++ Style Guide." https://google.github.io/styleguide/cppguide.html. (Accessed on 10/03/2016).

[33] D. Piparo, "Static Analysis Suite." https://github.com/dpiparo/SAS. (Accessed on 11/07/2016).

[34] Wikipedia, "Unit testing." https://en.wikipedia.org/wiki/Unit_testing. (Accessed on 10/11/2016).

[35] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[36] B. Mamlin, "Why does Darius dislike OpenMRS Talk?." https://talk.openmrs.org/t/why-does-darius-dislike-openmrs-talk/276/12. (Accessed on 10/11/2016).

[37] "OpenMRS Atlas." https://atlas.openmrs.org/. (Accessed on 10/09/2016).

[38] Atlassian, "Software Development and Collaboration Tools." https://www.atlassian.com/. (Accessed on 10/09/2016).

[39] "On the quest for the right project management tool: Jira, Trello, Asana, Redmine : programming." https://www.reddit.com/r/programming/comments/4p4782/on_the_quest_for_the_right_project_management/. (Accessed on 10/27/2016).

[40] "Travis-CI." https://travis-ci.com/. (Accessed on 10/09/2016).

[41] "Tuleap." https://www.tuleap.org/. (Accessed on 10/09/2016).

[42] "Enalean Customers." https://www.enalean.com/customers/. (Accessed on 10/09/2016).

[43] "CMake." https://cmake.org/. (Accessed on 10/10/2016).

[44] "Valgrind Documentation: Memcheck." http://valgrind.org/docs/manual/mc-manual.html. (Accessed on 10/10/2016).

[45] "Does column width 80 make sense in 2014?." https://www.reddit.com/r/programming/comments/2nkntp/does_column_width_80_make_sense_in_2014/. (Accessed on 10/10/2016).

[46] "Static Analysis Suite Github Repository." https://github.com/dpiparo/SAS. (Accessed on 10/11/2016).

[47] "Simplified Wrapper and Interface Generator." http://swig.org/. (Accessed on 10/11/2016).

[48] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[49] Oracle, "Design Overview." https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html. (Accessed on 10/11/2016).

[50] J. M. Arnold, "An Introduction to Floating-Point Arithmetic and Computation." CERN openlab Mini-Workshop on Floating-Point Computation, 2016.

[51] E. Mcintosh, F. Schmidt, F. de Dinechin, *et al.*, "Massive Tracking on Heterogeneous Platforms," in *2006 ICAP Conference in Chamonix, France*, 2006.

[52] Y. Patt, "Requirements, bottlenecks, and good fortune: Agents for microprocessor evolution," *Proceedings of the IEEE*, vol. 89, no. 11, pp. 1553–1559, 2001.

[53] "Intel Data Center Update." CERN openlab openday, 2016.

[54] Intel, "Intel® 64 and IA-32 Architectures Optimization Reference Manual." https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. (Accessed on 10/12/2016).

[55] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sept 1972.

[56] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté, "Boost. simd: generic programming

for portable simdization," in *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pp. 1–8, ACM, 2014.

[57] F. Lemaitre, "Vectorization of the solve parabola function in the SciFi tracking: Proof of concept for LHCb code acceleration." https://indico.cern.ch/event/434843/contributions/1080927/attachments/1129883/1614586/slides.pdf. LHCb Core Software Meeting (Gaudi) (Accessed on 10/17/2016).

[58] "Programming Guidelines for Vectorizing C/C++ Compilers." http://www.drdobbs.com/programming-guidelines-for-vectorizing-c/184401611. (Accessed on 10/17/2016).

[59] S. Blair-Chappell, "The significance of SIMD, SSE and AVX." http://www.polyhedron.com/web_images/intel/productbriefs/3a_SIMD.pdf. (Accessed on 10/17/2016).

[60] M. Kretz, *Extending C++ for explicit data-parallel programming via SIMD vector types*. PhD thesis, 2015.

[61] P. Karpinski, "UMESIMD Project on BitBucket." https://bitbucket.org/edanor/umesimd/overview. (Accessed on 10/17/2016).

[62] "Amdahl's law." https://en.wikipedia.org/wiki/Amdahl%27s_law. (Accessed on 10/18/2016).

[63] "Intel® VTune™ Amplifier." https://software.intel.com/en-us/intel-vtune-amplifier-xe. (Accessed on 10/20/2016).

[64] "Understanding How General Exploration Works in Intel® VTune™ Amplifier XE." https://software.intel.com/en-us/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe. (Accessed on 10/18/2016).

[65] "Eigen." http://eigen.tuxfamily.org/index.php?title=Main_Page. (Accessed on 10/19/2016).

[66] "VecGeom GitLab." https://gitlab.cern.ch/VecGeom/VecGeom. (Accessed on 10/20/2016).

[67] "Add Vc::pow and Vc::cbrt functions VcDevel/Vc." https://github.com/VcDevel/Vc/issues/116. (Accessed on 10/20/2016).

A. Environment

Lenovo Thinkpad X1 Carbon

CPU	Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
Physical Cores	2
Logical Cores	4
Cache	4 MB
Memory	8 GB
OS	Ubuntu 15.10
g++ version	5.2.1
Vc version	commit 6c46d143f77e67881b93f8bd0998e38a201ebe82
CMake version	3.2.2

Server

CPU	Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz
Physical Cores	8
Logical Cores	32
Cache	20 MB
Memory	64 GB
OS	Scientific Linux CERN SLC release 6.8
g++ version	5.2.0
Vc version	commit 32fa97314429f055a8f8c4a5b53cf7d3555ea8a7
CMake version	3.6.1

BioDynaMo/biodynamo

BioDynaMo Developers Guide

by Lukas Breitwieser - CC BY-SA 4.0

Welcome to BioDynaMo!

A very warm welcome to BioDynaMo and thank you for your interest! This guide will help you get started. It introduces you to the project, the community and our software development approach. It was inspired by the OpenMRS Developers Guide as well as conventions and best practices used in the software industry.

Who Should Read This Guide?

The target audience for this document is anyone who wants to build, contribute or learn more about BioDynaMo. Everyone has a different background, you might be new to C++ programming, simulation software or software development in general. This doesn't mean you cannot be a valuable contributer! This guide will help you fill in the blanks.

We did our best to be as concise as possible. If you make it to the end of this guide you will…

- understand the vision of BioDynaMo
- understand the basic architecture of BioDynaMo
- have a working development environment
- know the workflow to contribute code
- know where to ask for help

Feedback

This guide is an evolving resource. If you have difficulties in some steps and feel that parts could be described better or any other kind of comment please reach out to `lukas.johannes.breitwieser _at_ cern.ch`. Thank you for your feedback!

98% Finished Projects

In the open source world sometimes it happens that people work on a feature for weeks or month and leave after it has been finished for 98%. Although this 2% don't look like a big issue, usually that means that all your work doesn't make it into the production code. Normally, other developers are busy and don't have the time to dive into your work and find the pieces that are missing or not working yet. This situation would be a waste of your precious time. We bet that it is way more satisfying if your contribution makes it into production and will be used by scientists around the world.

For a contribution to be considered 100% complete, it must (be)

- comply to our coding guidelines,
- unit tested,

- well documented
- include a demo / screencast in certain cases.

Therefore, we want to encourage you to reserve enough time in the end where you don't code. We do our best to support you!

Project

- What is it?
- Our vision
- Why simulation software?
- Who is involved?

BioDynaMo is a developmental biological tissue simulation software. More precisely, it is possible to grow sophisticated structures emerging from simple rules. This rules represent the genetic code of our virtual cell. Simulation software has established itself as another pillar of science. The possibility to carry out virtual experiments has many advantages. It is possible to get results quickly, more cost efficient and enables testing a vast amount of parameters.

CERN openlab brought together partners from academia and industry:

- Newcastle University, UK
- Innopolis University
- Kazan Federal University
- Intel

Together we strive to create the leading simulation software for biological developmental processes.

Community

Our Team is spread across the world. Therefore we need tools that enable us to efficiently communicate and collaborate.

Communication

- Mailing List - community discussion that everybody sees = speak to the community - biodynamo-dev, biodynamo-talk
- Development Chat on Slack - real time communication
- Skype calls - discussions for subgroup - presentations
- Gitbooks - Documentation

Architecture

to be added soon

Start Contributing

Code Quality

At BioDynaMo we are aiming to develop a high quality software product. The following practices help us to achieve this goal:

- C++ Style Guide
- Doxygen Documentation
- Git Conventions
- Test driven development (TDD)
- Continues Integration (CI)

A coding standard is a set of guidelines and best practices which improve readability and maintainability of a code base. Code is more often read than rewritten. Therefore it is important that a developer quickly understands a piece of code even if it was written by someone else. A coding standard helps to achieve that. We are using the Google C++ Style Guide

Doxygen is used to generate documentation from comments in the source code.

TDD and CI are two practices from agile development. Test Driven Development is a special way of using unit tests. A unit test is a piece of code that tests a certain functionality of our application. If we make some changes in the code and at the end all unit tests pass, we most likely did not break something. This increases confidence in the code and reduces the fear to "touch" others code. Continues Integration takes all this automated tests and executes them on every change to the code repository. We are using Travis-CI

More information:

Git Conventions

Git Workflow:
We are following the Git Workflow proposed by Vincent Driessen in his blog post: A successful Git branching model with the modification that his `develop` branch is our `master` branch and his `master` branch will be replaced with `release`

Git Commit Message Guide:
Taken from a great blog post from Chris Beams

1. Separate subject from body with a blank line
2. Capitalize the subject line
3. Do not end the subject line with a period
4. Use the imperative mood in the subject line
5. Use the body to explain what and why vs. how
6. Limit the subject line to 50 characters
7. Wrap the body at 72 characters

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
```

```
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequenses of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

Use `git commit` without the `-m` switch to write a commit body.

Master Branch
Each commit of the master branch should pass the CI build. Therefore all development should be carried out in a feature/hotfix branch. Once development has been finished and the build passes, it can be merged back into `master`.

Merging should always be done without fast-forwarding to preserve history

```
git merge --no-ff
```

Branch Name Convention
Name the branch after the ticket you are working on (e.g. `123`).

Setting-Up the Development Environment - Ubuntu

This section will help you to set-up the development environment. Please also have a look at README.md which contains the most up to date information.

1. Install prerequisites

```
sudo apt-get install git g++ cmake valgrind doxygen
```

This command installs `git` code repository client, `g++` compiler, `cmake` build tool, `valgrind` a tool to debug and profile programs and `doxygen` used to generate documentation from source code comments.

```
wget https://root.cern.ch/download/root_v6.06.04.Linux-ubuntu14-x86_64-
gcc4.8.tar.gz 2> /dev/null
tar zxf root_v6.06.04.Linux-ubuntu14-x86_64-gcc4.8.tar.gz > /dev/null
cd root
. bin/thisroot.sh
```

Download, unzip and set environment variables for ROOT

2. Set-up git

```
git config --global user.name "Awesome Developer"
git config --global user.email awesome.developer@example.com
```

3. Get the code

```
git clone https://github.com/BioDynaMo/biodynamo.git
cd biodynamo
```

4. Compile it

```
mkdir build && cd build
cmake ..
make
```

5. Execute all tests (this step will take ~30 min)

```
make check
```

Most of the time is spent on valgrind which tests if BioDynaMo is leaking memory. If you are only interested if the unit tests produce the required results run:

```
./runBiodynamoTests
```

But: before you commit you must run `make check`!

Development Workflow

Finally, we are ready to get some work done!

1. Go to "Issues" and select a ticket that you want to work on and assign it to yourself

   Let's assume it is `123`

2. Checkout your `master` branch

```
git checkout master
```

3. Get latest version of `master`

```
git pull --rebase origin master
```

4. Create the feature branch

```
git checkout -b 123
```

5. Make your changes
6. Get updates from the remote `master` branch that have been added in the meantime

```
git pull --rebase origin master
```

7. Run the tests

```
make check
```

8. Commit (have a look at the commit message convention)

```
git add -i
git commit
```

9. Push your changes or go back to point 5

```
git push origin 123
```

10. If Travis-CI reports a passing build it is ready to be merged into `master`. Therefore ping me on our dev chat.

Checklist:

- Sign up for the `biodynamo-dev@cern.ch` mailing list
- Request membership for our git repository
- Bookmark the development chat on Slack
- Introduce yourself - short bio and picture
- Participate in discussions
- Start contributing on introductory tickets