JOHANNES KEPLER
UNIVERSITÄT LINZ

Efficiency and Quality Control for Particle Simulations

Dissertation
zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften
im Doktoratsstudium der
Technischen Wissenschaften

Eingereicht von
DI Richard Berger

Angefertigt am
Department of Particulate Flow Modelling

Erstbeurteiler
Assoz. Univ.-Prof. Dr. Stefan Pirker

Zweitbeurteiler
Dr. Axel Kohlmeyer

Mitbetreuung
Dr. Christoph Kloss
Dr. Thomas Lichtenegger

Januar 2016

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

This work describes the efforts of improving the efficiency and quality of the LIGGGHTS open-source software package for Discrete Element Methods (DEM). It explains the core algorithms present in this code and how a test harness was developed to support refactoring by automated testing and validation. Because of this testing facility, iterative refactoring of the code base could take place without damaging existing functionality.

Major modifications were made to the code base to simplify the common task of creating new force models both for particle-particle and particle-wall contacts. This paved the way for further advances such as multi-threaded OpenMP versions of all granular force kernels. Along the way many improvements were made to the core functionality in LIGGGHTS. Through measuring and profiling, performance bottlenecks in the existing implementation were found and improved.

Finally, an MPI/OpenMP hybrid parallelization was developed which extends the existing MPI parallelization of the code. Problems encountered and the solutions implemented to achieve scalable performance using both parallelization models are explained in this text. Three case studies, including two real-world applications with up to 1.5 million particles, were evaluated and demonstrate the practicality of this approach. In these examples, better load balancing and reduced MPI communication led to speed increases of up to 44% compared to MPI-only simulations. This shows that by using the hybrid parallelization of LIGGGHTS, current and future computing resources can be used more effectively.

Kurzfassung

Diese Arbeit beschreibt die Verbesserung der Effizienz und Qualität des LIGGGHTS Open-Source Software Paketes für Diskrete Elemente Methoden. Sie erläutert zunächst die Kern–Algorithmen und die prinzipielle Funktionsweise dieses wissenschaftlichen Codes. Um das Überarbeiten des Codes durch automatisierte Tests und Validierung zu unterstützen, wurde eine Test Umgebung eingeführt. Mithilfe dieser konnte die Codebasis über mehrere iterative Schritte verbessert werden, ohne dabei die bestehende Funktionalität zu gefährden.

Die Erstellung neuer Kraftmodelle für Partikel-Partikel- und Partikel-Wand-Interaktionen wurde durch umfassende Änderungen vereinfacht. Dies bereitete den Weg für weitere Optimierungen, wie das Ausnutzen mehrerer Threads innerhalb aller granularer Kraftberechnungen mittels OpenMP. Im Zuge dieser Arbeiten kam es auch bei den Kern-Funktionen von LIGGGHTS zu zahlreichen Verbesserungen. Durch Laufzeitmessung und Profiling konnten Engpässe festgestellt und behoben werden.

Schlussendlich führten diese Arbeiten zur Entwicklung einer MPI/OpenMP Hybrid Parallelisierung von LIGGGHTS, welche die bestehende MPI Parallelisierung erweitert. Es werden die dabei auftretenden Probleme und entsprechenden Lösungen dokumentiert, welche für einen skalierbaren Code notwendig waren. Drei Fallstudien, inklusive zweier realer Problemstellungen mit bis zu 1.5 Millionen Partikeln, wurden evaluiert und demonstrieren die praktische Anwendung dieser Methode. Bei diesen Beispielen werden im Vergleich zu reinen MPI Simulationen, aufgrund besserer Lastverteilungen und reduzierter MPI Kommunikation, die Berechnungszeiten um bis zu 44% reduziert. Damit wird gezeigt, dass mithilfe der Hybrid Parallelisierung von LIGGGHTS aktuelle und zukünftige Rechenkapazitäten effektiver genutzt werden können.

Acknowledgements

I want to thank everyone that supported me during my work on this thesis and the last three years of study at Johannes Kepler University.

First of all, I want to thank Christoph Kloss, the original author of LIGGGHTS and original head of the Rock'n Roll Group at PFM, for giving me this opportunity to work on his code and supervising me during my first two years of study. I believe we achieved great things together by coordinating our efforts.

Second, I want to thank Axel Kohlmeyer for supporting me in this PhD and letting me test-drive new waters by inviting me to help and teach at workshops in Trieste, Italy. Many lessons were learned during those stays in Italy. Both professionally and personally. He also continued to be a voice of reason in turbulent times.

I also want to thank Stefan Pirker, the head of PFM, for giving me free reighn over what I was doing, especially in the last year. He saw the need for the investment in this area and provided the working environment for this type of research. At the same time I want to thank our partners voestalpine, Primetals (formally Siemens VAI), and Land Oberösterreich for funding my work. I thank DCS Computing for allowing me to work directly with them and funding parts of the test harness development.

I would also like to thank my other colleagues. Gerhard Holzinger was the one who recruited me in the beginning by pointing out the available position at PFM. Daniel Queteschiner and I worked together during the creation of the second installment of the test harness. Without him I would not have had much tests. In addition to my supervisors, both were also kind enough to read drafts of this work and provided valuable feedback in the final weeks. Thomas Lichtenegger became the new head of Rock'n Roll in the last year and inherited me and my PhD. He allowed me to continue on my path and conclude my work, even though I wasn't aligned to his focus of model development. I additionally thank Stefan Amberger, Andreas Aigner, Philippe Seil and Luca Benvenuti for the work collaborations, travels and fun times we had together.

I also want to acknowledge the people from the Institute of Fluid Mechanics and Heat Transfer. I've learned many practical lessons in CFD from Michael Krieger and Andreas Löderer. Also Martin Barna and Thomas Hoanzl were the ones keeping our Gollum Cluster up and running, which endured my profiling and benchmarking sessions.

I also thank the LAMMPS community, especially Steve Plimpton, for their continued work. Thanks also to Josef Kerbl for providing the initial version of the Mixing process test case. Also many thanks to the developers of Zoltan for their very useful library.

Last, but not least, I want to thank my family for their continuous support during my studies. I thank my parents for enabling me to study at university and allowing me to pursue my dreams. Finally I thank my sister for keeping an eye on her big brother, enduring my ups and downs, and helping me in so many different ways.

Richard Berger

Contents

1 Introduction **1**
- 1.1 Granular media and bulk solids . . . 1
- 1.2 Methods for describing granular flow . . . 2
  - 1.2.1 Eulerian Methods . . . 2
  - 1.2.2 Lagrangian Methods . . . 2
- 1.3 Similarities to Molecular Dynamics . . . 3
- 1.4 LIGGGHTS . . . 4
- 1.5 Outline of this work . . . 5

2 Core Algorithms **7**
- 2.1 Basic integration loop . . . 7
  - 2.1.1 Setup . . . 7
  - 2.1.2 Update Forces . . . 8
  - 2.1.3 Integration of equations of motion . . . 8
- 2.2 Velocity-Verlet integration loop . . . 9
- 2.3 Neighbor Lists . . . 11
  - 2.3.1 Cell Linked-List algorithm . . . 12
  - 2.3.2 Verlet List algorithm . . . 12
  - 2.3.3 Reducing work by applying Newton's third law . . . 13
- 2.4 Spatial Sorting . . . 13
- 2.5 Parallelization . . . 15

I Quality Control **18**

3 Software Quality **19**
- 3.1 Quality of code from a physical standpoint . . . 19
- 3.2 Quality from a software engineering perspective . . . 20
- 3.3 Errors in DEM simulations . . . 21
  - 3.3.1 Modeling errors . . . 21
  - 3.3.2 Discretization errors . . . 21

3.3.3 Round-off and truncations errors . . . 22
3.3.4 Programming errors . . . 22
3.4 Testing . . . 24
3.4.1 Forms of testing . . . 25
3.4.2 Challenges of testing LIGGGHTS . . . 26

4 Acceptance and regression testing of LIGGGHTS 27
4.1 Structure of the Test Harness . . . 27
4.2 Building binaries . . . 28
4.3 Running test cases . . . 29
4.3.1 Structure of a LIGGGHTS test case . . . 30
4.3.2 Configuration of a test case . . . 30
4.3.3 Dependencies between run configurations . . . 31
4.3.4 States of a test case run . . . 32
4.3.5 Post-Processing of a LIGGGHTS run . . . 33
4.4 Version Comparison . . . 35
4.4.1 Comparing plots of data . . . 36
4.4.2 Defining custom plots . . . 38
4.4.3 Comparing reported text and numerical results . . . 39
4.5 Building a collection of test cases . . . 40
4.6 Web Frontend . . . 41
4.6.1 Overview of test cases . . . 41
4.6.2 Test case run information . . . 42
4.6.3 Version Consistency . . . 44
4.7 Continuous Integration with Jenkins . . . 45

5 Structure of LIGGGHTS 2.x 47
5.1 Basic Concepts . . . 47
5.1.1 Extension points in LIGGGHTS . . . 47
5.1.2 Extension Mechanism . . . 48
5.2 Pair Styles . . . 49
5.2.1 Force Loop . . . 50
5.2.2 Granular force kernels in LIGGGHTS . . . 50
5.2.3 Properties . . . 54
5.2.4 Contact History . . . 55
5.3 Walls . . . 56
5.3.1 Primitive Walls . . . 56
5.3.2 Mesh Walls . . . 56
5.4 Limitations of Extension Points . . . 57

6 New Structure of LIGGGHTS 3.x 59
- 6.1 Granular Force Kernels 60
- 6.2 Contact Models 62
  - 6.2.1 Responsibilities of a contact model 63
  - 6.2.2 Structure of a contact model 64
  - 6.2.3 Force evaluation of a contact model 65
  - 6.2.4 Settings & Flags 65
- 6.3 Global Properties 66
  - 6.3.1 Contact History 68
- 6.4 Compile-time Assembly of Contact Model Variants 69

II Efficiency 71

7 Measuring performance and efficiency 72
- 7.1 Single Processors 72
- 7.2 Multi-threading and Multi-core 74
- 7.3 Profiling applications 77
  - 7.3.1 Instrumentation 77
  - 7.3.2 Sampling 78
- 7.4 Case Study: Profiling packed insertion of LIGGGHTS 78
- 7.5 Case Study: Investigating MPI bottlenecks using HPCtoolkit 82

8 Core Optimizations 85
- 8.1 Optimization of dynamic mesh transformations 85
  - 8.1.1 Avoiding expensive operations in critical paths 86
  - 8.1.2 Limiting collective communication and unnecessary work to minimum . . 88
- 8.2 Optimization of the mesh neighbor list building 89
  - 8.2.1 Existing implementation 89
  - 8.2.2 Improved contact detection algorithm 89
- 8.3 Case Study: Optimizing the insertion of particles 92
  - 8.3.1 Original implementation 92
  - 8.3.2 Improved implementation 92

9 Parallelization of LIGGGHTS 94
- 9.1 Forms of Parallelism 94
  - 9.1.1 Distributed Memory Programming 95
  - 9.1.2 Shared Memory Programming 96
  - 9.1.3 Mixed-mode Programming 98
- 9.2 Limits of Parallelization 99
- 9.3 MPI Parallelization of LIGGGHTS 100

9.3.1 Static Domain Decomposition . . . 100
9.3.2 Load-balancing in DEM . . . 105
9.3.3 Dynamic MPI Domain Decomposition . . . 106
9.4 MPI/OpenMP Hybrid Parallelization of LIGGGHTS . . . 107
9.4.1 Related Work . . . 108
9.4.2 Outline of the hybrid parallelization . . . 110

10 OpenMP parallelization of LIGGHTS 112
10.1 Shared-memory programming using OpenMP . . . 114
10.1.1 Compiler Support . . . 114
10.1.2 OpenMP Directives . . . 114
10.1.3 Work-sharing . . . 115
10.1.4 Reductions . . . 116
10.2 Challenges in multi-threaded programs . . . 117
10.2.1 Data Races . . . 117
10.2.2 False Sharing . . . 117
10.2.3 Data Locality . . . 118
10.2.4 Non-deterministic results . . . 119
10.3 Multi-threading of particle-particle computations . . . 120
10.3.1 Data races in granular pair styles . . . 120
10.3.2 Methods to handle data races . . . 122
10.3.3 Avoiding data races through partitioning . . . 125
10.3.4 Dynamic Partitioning and Load-Balancing using the Zoltan Library . . . 128
10.4 Multi-threaded neighbor list building in LIGGGHTS . . . 131
10.5 Multi-threading in particle-wall interactions . . . 133
10.5.1 Mesh wall computations with OpenMP . . . 133
10.5.2 Mesh Neighbor List building with OpenMP . . . 135
10.5.3 Load-balanced particle partitions for particle-wall contacts . . . 138
10.6 Summary of the MPI/OpenMP Hybrid Parallelization . . . 140

11 Validation of the Hybrid Parallelization 141
11.1 Box-in-a-box . . . 143
11.2 Silo discharge . . . 148
11.3 Mixing process . . . 151

12 Conclusion 156
12.1 Impact & Outlook . . . 157

Chapter 1

Introduction

1.1 Granular media and bulk solids

Granular materials play an essential role in many industries, such as chemical, pharmaceutical, food and metalurgy industry [1]. Processes occurring in steel making inside a blast furnace involve granular media such as coal. The production of powders or pills in the pharmaceutical industry involves the mixture of granular raw materials. In food production many of our day-to-day products are a direct or indirect result of granular processes. An example of this is the production of corn flour. Removing impurities and avoiding contamination are important goals in this process. Corn mills store, transport and process the granular corn through multiple steps. This process depends on the flow of the material from the storage silo to the grinding process.

Many advances in these kinds of industries have been achieved through experimental research. The behavior of granular materials has been studied for many decades and has led to the discovery of some unique properties.

Granular flow can exhibit unexpected results. Take, for instance, the flow of granular material out of a hopper. The mass flow is largely determined by the shape and size of the orifice as well as the particle size and shape. However, the mass flow can be increased by simply attaching a standpipe to the orifice, thereby extending the hopper. This is due to pressure difference which builds up due to the interaction of the particles with the surrounding gas [2].

Another property of granular media is the segregation of sizes. This can be observed in the shipments of nuts from overseas [3]. The arriving shipments always ended up containing the small nuts at the bottom of a can, while the larger nuts, such as the Brazil nut, were on top. Due to these observations, this size segregation effect is also commonly known as the Brazil nut effect.

Unlike fluids, bulk solids are also capable of transmitting shear stresses. A silo filled with a fluid will have to endure its highest pressure at its bottom. This is due to the hydraulic pressure building up. Storing granular material in the same silo, however, exhibits a different behavior. Due to wall support the pressure is distributed among the walls. This limits the total pressure on the lower part of the silo [4].

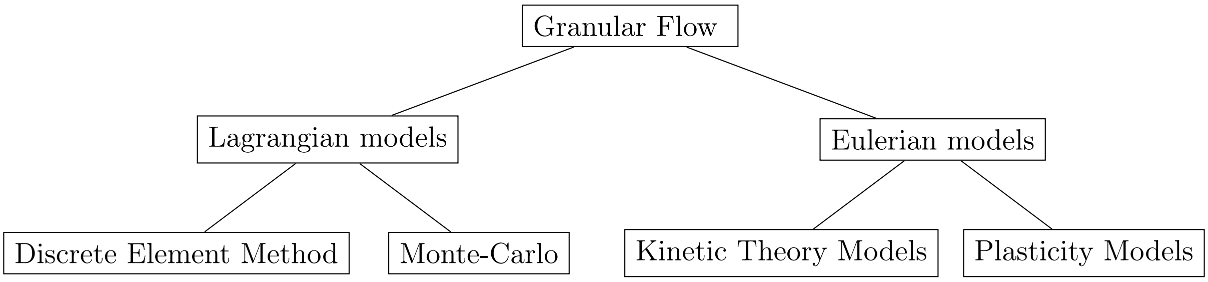


Figure 1.1: Methods for describing granular flow

These examples illustrate that *particle technology* [1], which is the study of granular material and bulk solids, is by itself a complex and interesting field of study. As with any industry, there is high demand for faster and cheaper ways of optimizing and analyzing new designs for industrial processes. Computer-based methods have emerged in the past decades to support these studies and reduce the cost of research & development.

1.2 Methods for describing granular flow

There are two main modeling approaches to tackle granular flow.

- Eulerian methods
- Lagrangian methods

1.2.1 Eulerian Methods

Eulerian methods utilize classical computational fluid dynamics (CFD) [5] where instead of resolving the entire simulation in full detail, the simulation is based on a mesh or grid. The cells of these meshes are much larger than particle diameters. Based on this mesh averaged transport equations are solved. Particles are treated as artificial continuum [6, 7, 8] which leads to a system of coupled partial differential equations which can be solved efficiently. The main benefit of Eulerian methods is that they are computationally less expensive than fully resolved methods.

1.2.2 Lagrangian Methods

Lagrangian methods, such as the Discrete Element Method (DEM) [9], resolve the individual particle trajectories and numerically integrates their equations of motion (EOM). This accuracy comes at the cost of computational overhead.

DEM starts from the equations of motion of each individual particle. In addition to external forces such as gravity acting on a particle, collision forces between particles are computed by force models.

The necessary granular force kernels are applied between both particles and walls. With all force relations known, the resulting system of coupled ordinary differential equations is then discretized in time. This produces algebraic equations which are then used to numerically integrate the equations and generate particle trajectories.

Contact forces in DEM are often computed based on the soft-sphere model [10]. In this model particles can overlap to a certain extent. It assumes that particles deform on impact which is the source of stresses, forces and torques.

Granular force kernels are important key ingredients of any DEM code. These computations are numerically expensive and responsible of up to 80% of the total computation time in DEM simulations.

The soft-sphere approach employed by these methods allows particles to overlap and exhibit collision forces on each other. The overlap models the elastic deformation of the particles. Each particle can also be in contact with multiple collision partners at once. This is in contrast to the hard-sphere approach where binary collisions are assumed.

Forces between two soft spheres are usually modeled using a system of springs and dash-pots. While springs model the transformation of kinetic energy to potential energy, dash-pots introduce damping into the system and therefore energy dissipation.

Both normal and tangential forces are a result of their own spring and dash-pots. The coefficients used by these are computed by a set of sub-models. Material properties such as Young's modulus, Poisson ratio, coefficients of friction of each participating collision partner influence these models.

While parts of these computations can be speeded up through the usage of look-up tables, each collision event in a DEM simulation must be considered unique. Particles of different materials or size might collide at any given time.

In addition to forces due to springs and dash-pots, further force components might be modeled to include effects of past collisions. Forces dependent on information of past events must store state or a history. One example of this is the accumulating shear stress during collision. Another more subtle force is the formation of liquid bridges between particles. Even though the collision might have ended, the liquid bridge might still have a cohesive effect.

1.3 Similarities to Molecular Dynamics

DEM is closely related to a much older group of simulation techniques. The field of Molecular Dynamics (MD) has been performing similar simulations for the last decades. This method tracks and computes the trajectories of individual atoms in a system. Atoms are under the influence of a variety of forces. These can be categorized into short-range and long-range forces. E.g., influences of electromagnetic fields fall into the category of long-range forces. In MD forces usually are the result of potentials. One of the most prominent potentials is the Lennard-Jones potential [11].

One property of MD is that the numerics assume that the system is Hamiltonian, there is no dissipation and energy is conserved. This has led to the development of efficient integration schemes such as the Velocity-Verlet integration which provide stable numeric results while keeping the numerical error low.

While the assumptions of MD do not hold for DEM anymore, the algorithms and techniques developed for this field share many similarities and have been applied with great success [12].

One major difference between MD and DEM is that the computational cost of each individual collision computation is more expensive in DEM due to the complex granular force kernels. What also makes computations more difficult is that unlike MD regular structures are not typical in DEM. Due to particles flowing from one part of the domain to another, the densities of particles will vary greatly throughout the domain. This influences the amount of collisions taking place at any given point in time, which is proportional to the total amount of work in a simulation. In MD, on the other hand, the amount of neighboring atoms is often constant, even enforced by a chosen cutoff radius of a given force potential.

Finally, since DEM simulations do not only involve interactions with other particles, but also with parts of machines and other geometries, the workload of MD and DEM further diverges in this regard. To conclude, while MD and DEM share the same basic principles, their workload can differ considerably.

1.4 LIGGGHTS

In 2009 Christoph Kloss developed a new Open Source DEM code for his PhD thesis. It was based on the Open Source MD code called LAMMPS [13], which stands for Large-scale Atomic/Molecular Massively Parallel Simulator. Since this new DEM code improved this original code in many regards for granular simulations and added heat transfer computation, the name LIGGGHTS was born. This new acronym stands for LAMMPS Improved for General Granular and Granular Heat Transfer Simulations.

LIGGGHTS extended LAMMPS with granular force kernels and added capabilities such as heat transfer computations among particles and walls [14]. It also allows adding static or dynamic mesh geometries into a simulation, which is a prerequisite for any industrial applications. Further additions include the ability to define templates of particles which can be used to insert packs or streams of particles into the domain.

The new DEM code benefited greatly from the rich set of existing functionality of LAMMPS. The extensibility of this code base allowed a diverse group of people from different fields to work together and extend its functionality over time, including Stefan Amberger, Andreas Aigner and Philippe Seil.

This has led to large success of the code internationally. Industrial partners started to benchmark it against commercially available DEM products. The conclusion was that not only is the code more efficient, but the Open Source nature of the code benefited the model developers since they could verify and understand how the code operates.

While the number of industrial partners and applications continued to grow, other fields also took notice and applied it in new ways. The granular extensions were for example used by the Jet Propulsion Labs of NASA to simulate the rocky surface of Mars and test drive the wheels of one of their rovers [15].

Another application in recent years was the usage during the Rosetta and Philae lander mission. Apparently LIGGGHTS was used during simulations for their planning [16].

Every year more and more researchers discover this code and try to apply it for their field of study. In combination with CFDEMcoupling, a code which allows LIGGGHTS to cooperate with OpenFOAM simulations, the code has rapidly grown its community. The Open Source effort has been orchestrated into one large project known as CFDEMproject[1].

In 2012 DCS Computing was founded by Christoph Kloss and Christoph Goniva. With this company they are now dedicated to the future development of the CFDEMproject, which includes the LIGGGHTS and CFDEMcoupling codes.

1.5 Outline of this work

This work was started in 2013 and began an effort to build bridges between the engineering and physical modeling groups with the computer science community. While the developed codes in the CFDEMproject have been highly successful, further investments in code development were necessary in order to continue growth.

The main focus on a code such as LIGGGHTS is that it produces physically sound results. Well established and tested models build the basis for this. Experimental validation of simulation results add further reassurance. With rising complexities the code was however slowly approaching a maintenance crisis. What lessons could be learned from the computer science field? How can they be utilized in this applied science code? How can we write fast and efficient code, while ensuring the quality of results?

This requires tearing down some pre-established walls. Code was made to work. It's the physics that mattered at the beginning. But an important step of software engineering was neglected for a long time. Retrospection. Dogma was introduced to ensure new code follows the same old standards. Code portions were labeled as "do not touch" because it was too complicated to understand what was going on in the code. This level of uncertainty needed to be gradually brought under control. By no means has this fully been achieved. But the foundations have been laid out.

There are two main parts of this work: Quality Control and Efficiency. Prior to these, Chapter 2 first introduces the basic operation of LIGGGHTS and the main algorithms which enable it.

It is then followed by the Quality Control part of this work. Chapter 3 describes what software

---

[1] www.cfdem.com

quality in the domain of DEM simulations means. Chapter 4 then shows how the consistency of the code is ensured through a newly developed test harness. Chapters 5 and 6 finally outline how the quality of the code was gradually improved through refactoring.

The Efficiency part begins with Chapter 7 introducing the topic of software efficiency. How can it be measured? What are existing inefficiencies and defects in the code? Where is computational time wasted and what can be done about it? Based on the new structure of the code and a way of measuring quality, the efficiency could then gradually be improved. Chapter 8 describes the optimizations of the core algorithms which were performed. Building on these, a new parallelization strategy based on multi-threading was developed using OpenMP. This new parallelization of the code allowed to tackle some pending issues in larger DEM simulations. Parts of this work have been published in [17]. Chapter 9 introduces the existing parallelization strategies and challenges in LIGGGHTS and outlines the hybrid parallelization which was implemented. It is followed by Chapters 10 which describes the OpenMP implementation in detail. This work is finally validated using several larger case studies in Chapter 11.

The impact of this work and future developments are sketched out in Chapter 12. It describes logical next steps moving the code and its ecosystem forward and concludes this work.

Chapter 2

Core Algorithms

LIGGGHTS is a simulation engine which operates on a well defined integration loop to compute the state of a particle system one time step after another. The following section will describe the building blocks which make up this loop and explain the different responsibilities of each block.

2.1 Basic integration loop

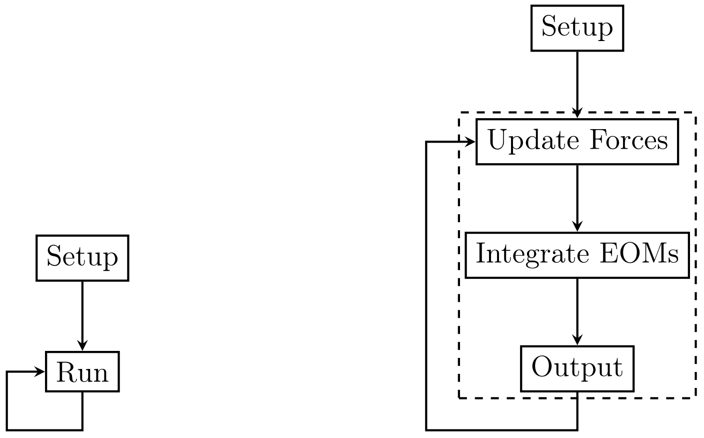
Any simulation software package consists of at least two execution phases: Setup and Run. While the setup phase is used to create an initial state of the simulation universe, the run phase will then transform this system from one state to the next state, one time step at a time. In DEM the simplest run phase consists of computing new forces, integrating the equations of motions of all the particles and finally writing data on screen or to storage.

2.1.1 Setup

The setup phase requires the following steps to define an initial state of a system:

- Define bounding box of the simulation domain
- Define geometry of the simulation (e.g. through meshes)
- Define material properties of particles and walls
- Define global properties such as time step
- Define initial condition and boundary conditions
- Setup which data should be dumped (e.g., written to disk) during computation

In LIGGGHTS this setup is done using a text-based input script. Through a series of commands the state of the system is adapted until it matches a desired initial state. A `run` command then launches the system and executes the desired amount of time steps.



(a) Basic structure (b) Main blocks of integration loop

Figure 2.1: Structure of a basic DEM integration loop

2.1.2 Update Forces

At each time step all forces acting on particles are computed. These are the result of particle-particle interactions, particle-wall interactions and external influences such as gravity.

$$\mathbf{F} = \sum \mathbf{F}_{p-p} + \sum \mathbf{F}_{p-w} + \mathbf{F}_{ext}$$

This computation makes up the most time-consuming part of the simulation.

2.1.3 Integration of equations of motion

Discretizing and numerically integrating the equations of motions of particles is an initial value problem. Given the forces acting on each particle and its initial conditions such as $x(t=0)$, $v(t=0)$, a DEM simulation computes particle trajectories $(\mathbf{x}(t), \mathbf{v}(t))$ by integrating Newton's equation of motion.

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{v}\\ \dot{\mathbf{v}} &= \frac{1}{m}\mathbf{F}\end{aligned}$$

Time discretization leads to a computation of $(\mathbf{x}(t), \mathbf{v}(t))$ at discrete times $t = n\Delta t$. There are many discretization schemes which could be used to perform the calculation with varying accuracy and computational cost. The simplest time discretization is the Euler method which uses a forward difference to approximate the first time derivative.

$$\begin{aligned}x(t+\Delta t) &= x(t) + \frac{dx}{dt}\Delta t + \dots\\ \frac{dx}{dt} &\approx \frac{x(t+\Delta t) - x(t)}{\Delta t}\end{aligned}$$

Using this time discretization the EOMs can be transformed into algebraic equations. Rearranging them yields a scheme to compute the position and velocity of the next time step.

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t$$
$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t$$

The acceleration of each particle $a(t)$ is computed from the forces acting on them using Newton's law:

$$\mathbf{a}(t) = \frac{1}{m}\left(\sum \mathbf{F}_{p-p}(t) + \sum \mathbf{F}_{p-w}(t) + \mathbf{F}_{ext}(t)\right)$$

Output data

Once the new state of the simulation is computed this information can be used to periodically save or output data of the current time step. Dumped data could then be further post-processed to facilitate interpretation of results.

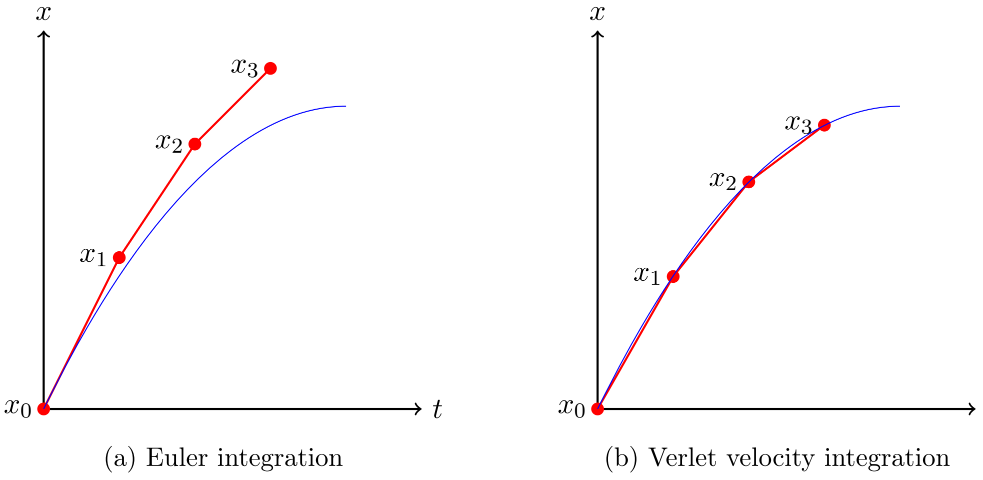
2.2 Velocity-Verlet integration loop

The Euler scheme is not widely used in molecular dynamics or DEM simulations. This is because it is only a first-order discretization scheme and therefore produces large errors. It can also be numerically unstable. Both LAMMPS and LIGGGHTS therefore use the Velocity-Verlet [18, 19] integration scheme by default.

$$x(t+\Delta t) = x(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2$$
$$v(t+\Delta t) = v(t) + \frac{a(t)+a(t+\Delta t)}{2}\Delta t$$

This is second order discretization scheme, with local errors of order 4 and an global error of order 2. It also exhibits excellent numerical stability. The differences between an Euler and Velocity-Verlet integration result can easily be illustrated by integrating one particle with constant acceleration $a(t) = -\frac{1}{2}$ and initial conditions $x(0) = 0$ and $v(0) = 2$. The resulting analytical solution is a parabola. Figure 2.2 shows the resulting discrete positions with a time step of $\Delta t = 1$. While the Euler scheme quickly diverges from the analytical solution with this time step, the Velocity-Verlet scheme still provides reasonable results.

Apart from improved accuracy, this integration scheme also has other important properties for physical systems such as time-reversablity and being a symplectic integrator. They are more relevant for MD simulations because they ensure that energy is well conserved in Hamiltonian systems.



(a) Euler integration (b) Verlet velocity integration

Figure 2.2: Comparison of Euler and Velocity-Verlet integration of a constant acceleration trajectory with $a(t) = -\frac{1}{2}$ and $\Delta t = 1$.

Implementation

The standard implementation of the Velocity-Verlet integration scheme decomposes the calculation into two velocity updates using half time steps and one position update. The algorithm is summarized as follows:

1. Calculate: $\mathbf{v}\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}(t) + \mathbf{a}(t)\frac{\Delta t}{2}$
2. Calculate: $\mathbf{x}\left(t + \Delta t\right) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2$
3. Derive $\mathbf{a}(t + \Delta t)$ from the forces
4. Calculate: $\mathbf{v}(t + \Delta t) = \mathbf{v}\left(t + \frac{\Delta t}{2}\right) + \mathbf{a}(t + \Delta t)\frac{\Delta t}{2}$

This algorithm can be further improved by reformulating the second step:

$$\mathbf{x}\left(t + \Delta t\right) = \mathbf{x}(t) + \underbrace{\mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2}_{\mathbf{v}\left(t + \frac{\Delta t}{2}\right)\Delta t}$$

This way the result of the velocity update in the first step can be reused in the second step. The final algorithm implementing this scheme can therefore be written as:

1. Calculate: $\mathbf{v}\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}(t) + \mathbf{a}(t)\frac{\Delta t}{2}$
2. Calculate: $\mathbf{x}\left(t + \Delta t\right) = \mathbf{x}(t) + \mathbf{v}\left(t + \frac{\Delta t}{2}\right)\Delta t$
3. Derive $\mathbf{a}(t + \Delta t)$ from forces
4. Calculate: $\mathbf{v}(t + \Delta t) = \mathbf{v}\left(t + \frac{\Delta t}{2}\right) + \mathbf{a}(t + \Delta t)\frac{\Delta t}{2}$

Using Velocity-Verlet integration replaces a single integration step in the integration loop by two steps which surround the force computation. One step performing step 1 and 2 of the scheme, while a second step perform steps 4 after forces have been updated. Figure 2.3 depicts the modifications needed in the integration loop.

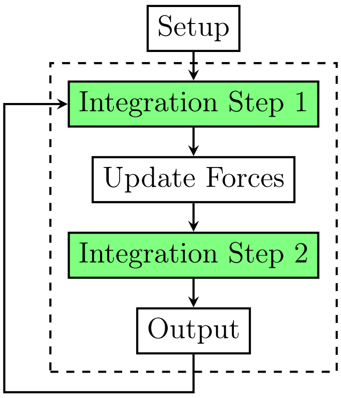


Figure 2.3: Integration loop with Velocity-Verlet integration scheme

2.3 Neighbor Lists

One of the traditional molecular dynamics algorithms present in LIGGGHTS is the generation of neighbor lists. These data structures are used to reduce the total amount of work needed to calculate collisions by reducing the number of potential collision partners. Without them collisions between particles would need an algorithm with a run-time complexity of $O(n^2)$. The neighbor list generation in LIGGGHTS is a combination of the cell list algorithm and Verlet lists [20], which reduces the total run-time complexity of pair collisions down to $O(n)$. Neighbor lists are valid for several time steps. If they have to be rebuilt, their construction occurs between the first Velocity-Verlet integration step and the force computation as illustrated in Figure 2.4.

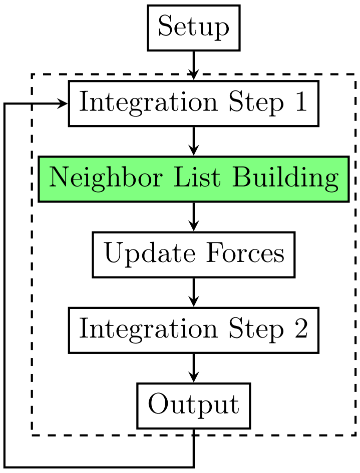


Figure 2.4: Velocity-Verlet integration with neighbor lists

2.3.1 Cell Linked-List algorithm

Cell Lists are created by decomposing the simulation domain into a uniform grid of smaller cells. Each cell is larger than individual particles. In LAMMPS/LIGGGHTS terminology these cells are called bins. The process of assigning each particle to one of these cells/bins is called *binning*. Once all particles are assigned to a bin, the cell algorithm assumes that any neighboring particle is either in the same bin of a given particle or in one of the surrounding 26 bins. This reduces the amount of lookup to 27 bins for which a stencil algorithm can be used.

Figure 2.5 illustrates how a rectangular domain could be decomposed in 2D. A domain with an extent of 100x100 could be decomposed into 10x10 bins. However, the actual number of bins is later modified to extend outside of the borders of the domain for round-off safety and to ensure stencil coverage.

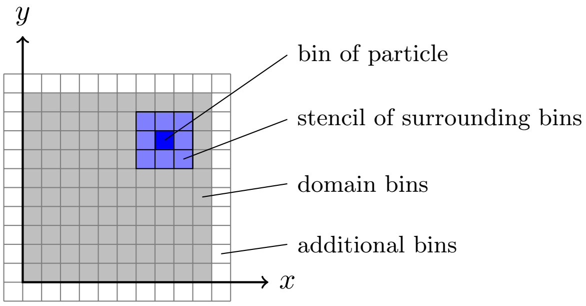


Figure 2.5: 2D Binning of the domain (100×100) with 10×10 bins. Each particle is assigned to a bin. A stencil of neighboring bins is searched for neighboring particles. To avoid corner cases additional bins are added to the data structure which allows using the same stencil for all bins.

2.3.2 Verlet List algorithm

By additionally using the Verlet-List algorithm the amount of neighboring particles around each particle is further reduced by ignoring neighbors outside of a cutoff region. It ensures that only neighbors are included which can interact with the particle in a reasonable amount of time before the next rebuild. Neighbor list rebuilds are triggered if any particle has traveled beyond a trigger distance, which is dependent on the cutoff radius. This radius is chosen in a way which strikes a balance between the amount of neighbor list rebuilds necessary and the size of the lists.

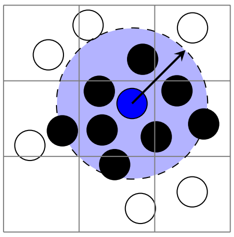


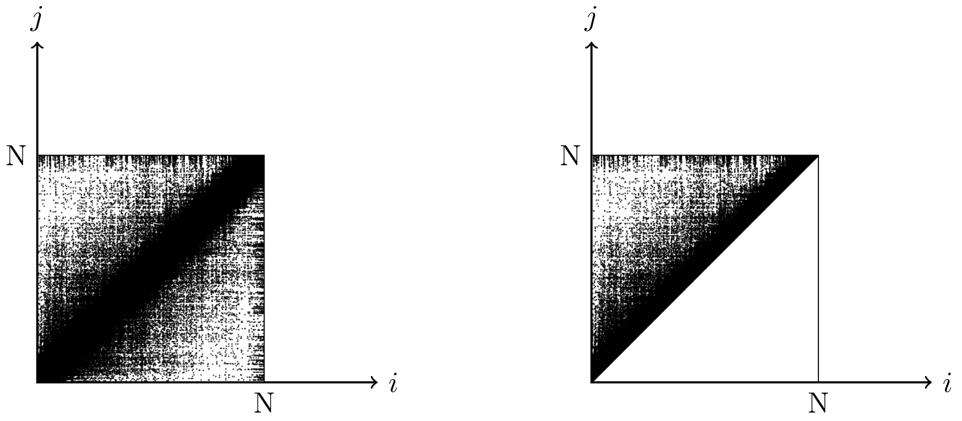
Figure 2.6: Cutoff of Verlet List

2.3.3 Reducing work by applying Newton's third law

Combining both Cell Linked-List and Verlet Lists to a single algorithm creates a list of neighbors for each particle which are within the 27 bins and inside of a cutoff region. Figure 2.7a shows the contents of an actual neighbor list by illustrating which particle pairs (i, j) are visited during iteration.

The total size of neighbor lists can now be further reduced by applying Newton's third law. Since particles will compute forces between them and their neighbors, neighboring particles will not have to recompute the same forces again. Instead the force is computed once and applied to both contact partners with opposite orientation. This cuts the total size of neighbor lists in half. Figure 2.7b visualizes how the iteration space stored in the neighbor lists is cut in half. Only the upper triangle of the force computation remains.

In Chapter 10 it will, however, be shown that this optimization introduces problems for multi-threaded code, since data of more than one particle is manipulated.



(a) Full neighbor list

(b) Half neighbor list due to Newton's third law

Figure 2.7: Iteration space (i, j) visited by particle neighbor list. These figures show the contents of a real neighbor list from a rotary kiln simulation (see Chapter 10). The concentration along the diagonal is due to the geometric proximity of particles inserted at the same time step.

2.4 Spatial Sorting

In particle simulations any particle may interact with other particles at some point in time. As described earlier, as particles get close to each other they will recognize themselves as neighbors. Neighbor Lists are used to reduce look-up times of neighboring particles during collision checks and computation. However, reducing the amount of checked particles is not enough for an efficient implementation of these algorithms. Accessing particle data and neighbor data in general leads to random access of memory. The data of neighboring particles may be scattered throughout many memory pages, which leads to degrading performance characteristics of both neighbor list building and traversal.

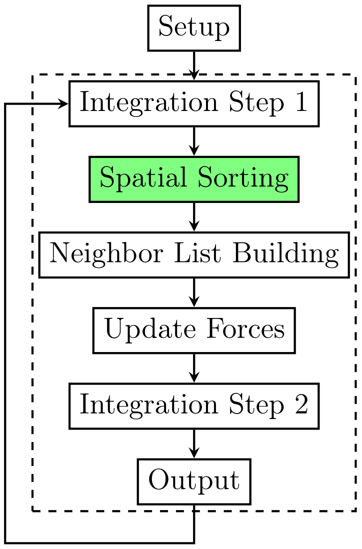


Figure 2.8: Adding spatial sorting to the integration loop

Memory subsystems in modern CPU architectures achieve the highest throughput if memory is not accessed at random locations, but in an orderly, sequential way [21]. One of the most expensive penalties of a memory-bound application is a page miss. Random memory accesses not only lead to cache misses, which requires reloading data from RAM, but also lead to a page miss.

CPUs will try to compensate expensive memory accessed through pipelining and out-of-order execution of instructions. While waiting for memory accesses to complete, other instructions may be executed. This situation, where a processor no longer executes new instructions, but has to wait for pending operations to complete is called a CPU stall. While a CPU could be crunching numbers, it will instead idle.

Cache-misses, page-misses and CPU stalls have to be minimized to utilize the hardware effectively. For this the locality of data has to be improved. Memory regions which are often accessed together should be placed in proximity to each other. This reduces the probability of causing a page miss. If data can be moved close enough, it can even avoid cache misses.

For LIGGGHTS this means that particle data of neighboring particles should be in proximity to each other in memory. Because of this, particle data is periodically sorted based on its spatial information. Using previous neighbor list information, particles can be rearranged so that each particle is followed by its neighboring particles.

The algorithm which creates and applies this permutation of particles in memory is called *spatial sorting* [20]. It improves the cachability of particle data and reduces the amount of page-misses significantly. In turn it also reduces the amount of CPU stalls. Figure 2.9 illustrates the effect of spatial sorting to the iteration space stored in the neighbor lists. Prior to this optimization contact pairs were spread across the entire upper force triangle which leads to random access of memory. Through spatial sorting data is reordered in such a way that neighbors are close in memory, forming a band above the diagonal in the iteration space. The width of this

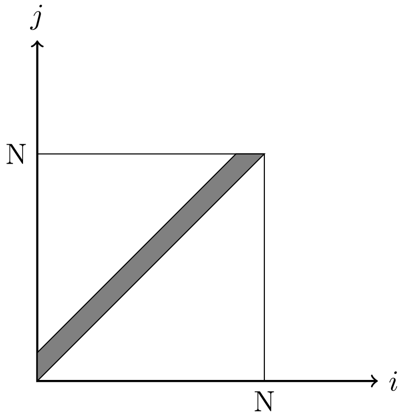


Figure 2.9: Contents of neighbor list after spatial sorting. Because particle neighbors are in nearby memory locations the visited (i,j) pairs are limited to a band above the diagonal.

band is proportional to the maximum amount of neighbors of a single particle.

Spatial sorting is only applied periodically because moving particle data in memory is an expensive operation. It brings the system into more cachable state which is refreshed after a few thousand time steps. An important side effect of this is that particle data can move over time. Its location in memory is subject to change depending on how the simulation progresses.

2.5 Parallelization

As the number of particles and size of geometry increases, running DEM simulations requires more computational resource to finish simulations in a reasonable time frame. A natural approach to solve this problem is to take a given computational load and distribute among many processors and/or machines. This introduces a communication problem into the simulation. The global state of a simulation is no longer located in one computational node, but distributed among many. Depending on the strategy used to do this decomposition of work, the amount of information which has to be synchronized between processors will vary.

In the particle subset method, particles in a simulation are distributed among processors. Because particles in one processor might interact with particles in another processor, force computation will require synchronization of particle information such as position and velocity. In general this requires communication of each processor with all others. To reduce the amount of communication graph-based algorithms have been used to find particle subsets with minimal links between them [22]. While particles are nodes in this graph, force computations between two particles are represented by edges. Cutting an edge introduces synchronization between particles. The algorithm therefore tries to find particle subsets by cutting the least amount of edges and thus minimizing synchronization.

LAMMPS and LIGGGHTS employ a simpler method of distributing work. Instead of distributing particles, this method splits the entire simulation domain into multiple 3D rectangular boxes, each of which is assigned to one of the processors. A processor is then responsible for all calculations inside this subdomain. This method is called *spatial-decomposition* or *domain-decomposition* [13]. By introducing a regular grid of subdomains and assigning it to processors, the amount of communication between each compute node can be minimized. Instead of costly collective communication between all nodes, spatial-decomposition minimizes communication of each processor to only its surrounding neighboring processors. This direct form of communication is called *point-to-point* communication. Figure 2.10 shows at which point this synchronization occurs in the integration loop.

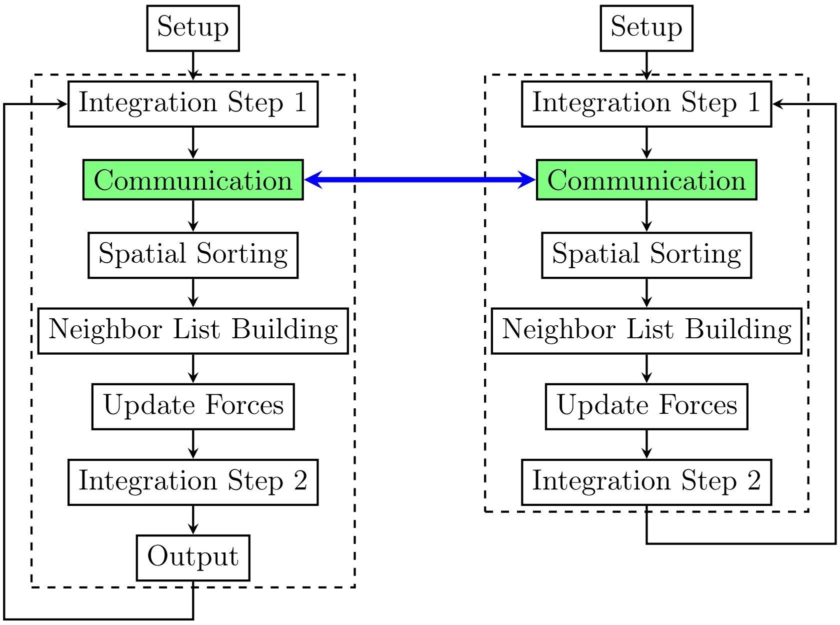


Figure 2.10: Communication between parallel running integration loops

Communication is further reduced by acknowledging the fact that inside of a subdomain, calculations are independent of any surrounding particles in other processors. The only need for communication is inside of a small border region surrounding each subdomain. For this reason processors limit their synchronization to so-called halo regions surrounding their domains. Halo-regions extend beyond subdomain boundaries and into a neighboring processor's subdomain. Particles inside these halo regions are called ghost particles. While they are visible to a processor, they are not part of the *owned* particles which a processor is responsible for. During computation ghost particles are synchronized with minimal information, such as position and velocities.

Over time, as particles move in a simulation, particles which leave a processor subdomain will enter the subdomain of another processor and therefore migrate to this processor. All of this can occur in parallel, only involving processors which are directly neighboring each other therefore reducing the contention on the messaging subsystem.

The described procedure of communicating systems and processor can be implemented in many ways. But there is a general pattern in these types of parallel computations, which is the necessity of communicating with other processors through messages. For HPC applications this common application pattern has been standardized in the Message Passing Interface (MPI).

This standard defines a general interface for point-to-point and collective communication between systems and has become the de-facto standard for writing HPC applications. Chapter 9 explains the MPI parallelization of LIGGGHTS in further detail.

Part I

Quality Control

Chapter 3

Software Quality

3.1 Quality of code from a physical standpoint

The simulations typically performed using DEM are an attempt to model reality and capture behavior which would be difficult to aggregate otherwise. As in any numerical approximation, the results of such computations is influenced by many factors. First of all it requires that we sufficiently understand the physical phenomena in order to select a proper model which can capture the involved effects. The model itself will need to be validated and calibrated using some experimental or analytical data.

While a model could be suitable for a given subset of problems, it might not be valid for other cases. Even if the correct model has been selected it must also be set up using meaningful initial and boundary conditions.

Given all of these preconditions, the final source of error is the numerical computation itself. Due to limited number resolution in computers the discretization of a given problem will always lead to round-off errors.

With so much uncertainty in any simulation, how can the quality of any simulation code be kept in check? Many of the developments of a code like LIGGGHTS are done by individual researchers, each working on a problem for a limited amount of time. Theories are created, hypotheses made. Together with experimental results a numerical model is then developed to capture the essence of what has been observed. The goal of course is to gain new insights into the problem and avoid costly and time consuming experimental setups in the future. Often phenomena are only investigated in lab-scale experiments, because of real-world experiments would be too costly.

Once properly configured and calibrated, a numerical method will produce a set of results which can be easily reproduced. The validation of a model requires a comparison to hypotheses, experimental or historic data using a test case. Software alone is not able to add any physical meaning to results.

Validation requires monitoring physical quantities during simulation which are expected to be influenced by the given model. Due to the inaccuracies of numerical discrete computations results can and should not be checked for equality. It is unlikely to know the exact numerical results due to the uncertainties of floating-point computations. Therefore, indirect methods are required. These include ensuring that physical results are withing an ϵ-region around the expected result or applying statistical methods. A given quantity should not match an exact value but be within a certain range.

Similar guarantees must exist for the prerequisites of any given model. This includes physical quantities which might be computed before a given model becomes active, such as quantities resulting from combinations of material types. The validity of other parts of the simulation system must be assumed. Ideally these have been tested and validated indepentently.

Once validated such models must be checked for consistency in any future versions of the code. New code, new features or models might influence or even alter its results. This must be detected and avoided, unless this change of behavior is intentional.

3.2 Quality from a software engineering perspective

Implementing a physically sound algorithm to simulate technically relevant processes is the primary goal of performing DEM simulations. Many man hours go into the process of developing and validating models. Sometimes these are supported by building expensive experiments to collect real-world data for comparison.

This time consuming process leads to an aggregation of knowledge. Mathematician, physicists, material scientists and other specialists all working together to expand their understanding of the physical phenomena. Their knowledge is finally documented and put into use by developing new code in simulation engines.

As these codes evolve, preservation of this knowledge is one of the top priorities of continued development. Ensuring a high quality of code becomes a necessity to preserve the integrity of such a long term investment.

One way of quantifying the quality of a code therefore can be seen as how well it captures the intent of a model. It is a well known fact that code is more often read than written [23]. A quality sign of a well written code therefore is how well it transports a given idea and concept to its readers.

Only after the functionality has been developed and proved to be correct should it be optimized for performance.

Building robust, sustainable and extensible software systems goes beyond simply programming sequential sequences of arithmetic and functions of code. In software engineering many techniques [24], best practices [23], software design patterns [25] and programming methodologies [26, 27] have evolved to support software developers in this task. Applying them can improve the readability and maintainability of the code as well as reduce the amount of programming errors over time.

3.3 Errors in DEM simulations

3.3.1 Modeling errors

The largest errors in DEM are introduced during the modeling of a physical problem. Assumptions and hypotheses shape the outcome of any numerical simulation. Model developers must be aware of the limitations and applicability of their approach.

Classic DEM simulations are primarily done using soft-spheres which elastically deform during impact and regain their original shape afterwards. These ideal shaped particles simplify computation but also limit the applicability. The non-sphericity of a particle is accounted for by additional force models. Such models may include history effects, additional forces such as cohesion or rolling friction. They might also model the effect of surface roughness. However, at some point spherical models are no longer sufficient and simulations have to be done by approximating the real shape of a particle. This can be done, for instance, by interpolating a shaped particle through multiple spheres bound together to one rigid body.

Another common artificial error introduced on purpose is not using the exact material properties for calculation, but replacing them with values which reproduce the same bulk behavior. Experimental results can be performed to determine characteristic properties of granular material such as the angle of repose. Simulations of the same setup can then be used to find properties which fit the experimental result. One outcome of these investigations is that the Young's modulus used in a simulation can be orders of magnitude smaller for a wide range of applications. This allows using much higher time steps and longer lasting simulations.

In simulations which would require a large amount of real size particles, it can be beneficial to perform calculations using coarse graining. The simulation is then not done with actual particles, but larger particles which represent a collection of them. While this method is crude, it is computationally efficient and can give first approximations of the actual result.

The quality of any simulation is only as good as the modeling behind it. Results with a large modeling error might be sufficient as long as the user is aware of the mistakes which are made. In this sense, similar to CFD simulations which are sometimes nicknamed Colorful Fluid Dynamics, the correct interpretation of the results is of great importance. It can hardly be automated and requires great experience and intuition.

3.3.2 Discretization errors

Numerically integrating the equations of motion of all particles introduces errors caused by time discretization. The size of these errors depend on the used integrator, which in case of LIGGGHTS is the Velocity-Verlet integration scheme. This scheme has a global error of order 2 which means that the reduction of the used time step by a factor of 2 can reduce the error by a factor of 4. At the cost of additional computational overhead. These discretization errors not only affect individual particle trajectories, but because of the coupling between them through collision forces, these errors propagate and accumulate over time across particles.

Additional discretization errors are introduced by walls defined through mesh geometries. Meshes represent arbitrary geometries through a collection of small primitive shapes such as triangles. These mesh elements form an interpolation of the original geometry. The more elements are used for this interpolation, the higher the accuracy of the results which depend on this shape's information. Finer meshes however also introduce additional cost in computation and memory usage.

3.3.3 Round-off and truncations errors

Numeric simulations not only introduce errors by discretization of their problems, but also by the limited resolution of computers to represent numbers. While a computer can represent numbers with an arbitrary resolution if needed, fast computations can only be achieved through a number format which is natively supported by the hardware.

Floating-point numbers are the most common number formats used in numeric computation. They have been standardized in IEEE Standard 754 [28] and are widely adopted in modern hardware. Both single-precision (32bit) and double-precision (64bit) floating-point formats are usually supported by processor floating-point units (FPUs). These two formats only differ in the amount of bits allocated to the mantissa and exponent of the given number. As its name suggests, the double precision format can represent numbers more accurately due to its higher number of bits available to store digits.

Usage of floating-point arithmetic introduces a new kind of error into any computation. These are commonly referred to as round-off and truncation errors. Every stored number in a floating-point format is truncated to the closest number available in the chosen format. This leads to numbers being slightly larger or smaller than the actual result.

One scenario of such round-off errors can be observed by summing up floating-point numbers of different orders of magnitude. If the difference between the two numbers is too large, cutoff errors will accumulate. In the worst case the contribution of the smallest number will be neglected completely because it is outside of the available precision.

Floating-point arithmetic is also not commutative. Any result is dependent on the order of the operations. These properties make comparison of floating point numbers difficult. It is unlikely to predict the exact outcome of a sequence of floating-point operations unless the same algorithm is used. Therefore, it is necessary to compare floating-points using a valid range of values.

3.3.4 Programming errors

A final source of errors can be a faulty implementation of the numerical method itself. While syntax errors are detected during compilation of a program, semantic errors or logic errors can not be detected easily.

Usage of uninitialized memory, calling functions in the wrong order and introducing unintended side-effects are all examples of potential programming errors. In many occasions programming errors are not immediately visible. Listing 3.1 shows a C++ code snippet which was present multiple time in the LIGGGHTS code base.

Listing 3.1: MIN and MAX macros found in LIGGGHTS 2.x

```
#define MIN(A,B) ((A) < (B)) ? (A) : (B)
#define MAX(A,B) ((A) > (B)) ? (A) : (B)
```

While these `MIN` and `MAX` macros seemed harmless in many cases, this chunk of code contained an inherent flaw which introduced a bug in one of the force computations. Since the C++ preprocessor only performs text replacements, the MIN macro is expanded into its definition. Precautions were made in the definition of the macro to ensure expressions are evaluated in the intended order. However, not enough. Listing 3.2 shows how using this macro generates faulty code in a seemingly harmless expression. The macro is used as part of an arithmetic expression which divides the result by a number. However, because of the expansion and the operator precedence rules the division will only take place if $b \leq a$, which is not the intent of this piece of code.

Listing 3.2: Expansion of th MIN macro creating a fault expression

```
result = MIN(a,b) / c;

// expands to
result = ((a) < (b)) ? (a) : (b) / c;

// due to operator precedence in C/C++ this is equivalent to
if (a < b)
  result = a; // missing division!
else
  result = b / c;
```

Such an error is hard to track down. It was identified by single-step debugging and monitoring the variable states after all computations. Depending on how the error manifests at a later stage it can waste hours of debugging time and may force hours or days of simulation results to be discarded. Only after finding such a bug they become self-evident.

These type of errors are more likely to happen if an implementation is not validated through tests. E.g., tests which ensure that both cases $a < b$ and $a \geq b$ return reasonable results would have pointed towards this error. The assumption that code is correct is not a replacement to actually executing available code paths.

3.4 Testing

Testing is an essential part of the software development cycle. Without testing any functionality which was once added to a software package is in an undefined state. Successful code compilation and execution are no guarantee that a given feature is doing what it is supposed to do. It requires an elaborate amount of testing of each existing feature. Testing is about removing assumptions and replacing them with verified behavior.

Once tests are defined, running them and verifying behavior during development allows identifying errors more quickly. Tests document the integrity of a feature and expected behavior. If new developments break documented behavior, tests will fail and give immediate feedback to the developer. Solving new problems is much faster if it occurs during the development of a feature because at that point in time most relevant information is already known to a developer. Trying to solve the same problem at a later point will take additional effort. Either other developers will have to first understand what the code is doing or the original implementer no longer remembers the details.

Tests also remove *fear*[23]. Code which may work today can quickly become bug-prone due to some newer changes. This is often called *software entropy* [29, 30]. Any code which is under active development suffers from entropy which must be actively addressed. The poor man's solution often becomes to not changing existing code at all, but working around it. Teams which label parts of a code with "It's working, do not change!" have an impending maintenance problem at their hands[1]. Such code areas are a major source of entropy. They accumulate fear of change and so-called *technical debt* [31, 32], which is the increasing amount of workarounds and code duplication.

If technical debt accumulates, refactoring of code is a fearful activity. It is avoided because of fear of breaking anything. Refactoring becomes a special activity which only occurs when the pain of technical debt has accumulated to level which is no longer bearable. At that point, however, the work necessary and the complexity of solving the problems in the code have risen dramatically.

Unfortunately testing is often considered a laborious task by many programmers. This view does not take into account the amount of work associated with returning to the code at a later point in time and either adding new functionality or fixing bugs. In either case the existence of tests gives reassurance that even after larger changes the remaining system is still intact and a new feature is doing what it is supposed to.

For this reason this work set out to introduce testing into the LAMMPS/LIGGGHTS code base. It is the one major step to improve the quality of the code in the long run. There is a large amount of technical debt accumulated in the existing code base. It was also not designed with testing in mind. Nevertheless, LAMMPS has succeeded thanks to the great amount of eyes looking at the code and people testing it in production. In this setting, without tests, any

---

[1] the mesh implementation in LIGGGHTS is an example of such a code base. It is often avoided because of its high complexity and the fear of adding regressions.

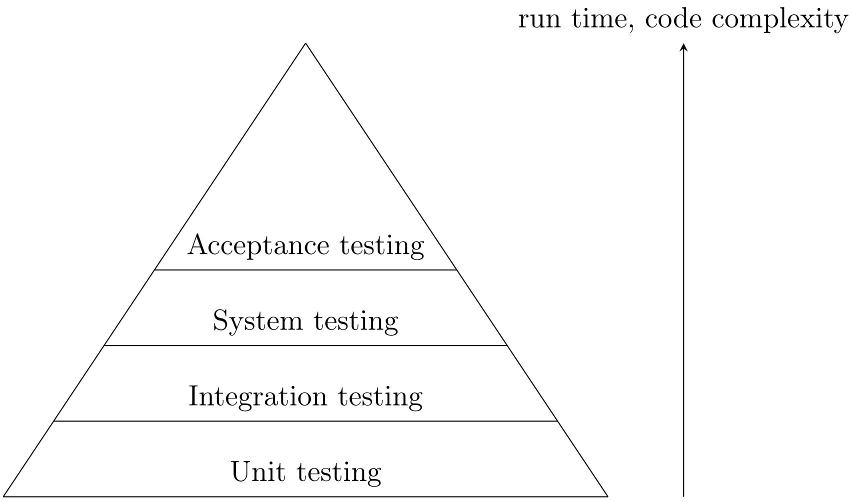


Figure 3.1: Levels of software testing

development is solely dependent on the heroics of the core developers of the code. With tests we can reduce the workload on maintainers and simplify the task in the long run. It also reduces the dependence on individual developers becoming experts in the entire code base.

3.4.1 Forms of testing

Any software test consists of the following steps: (a) specification of the test inputs, (b) running of the test and (c) validation of the test using the resulting output.

Testing can be subdivided into two categories: black-box testing and white-box testing. In black-box testing the software tests do not make any assumptions about how the software tested operates internally. Such tests only operate using the exposed interface and verify what is returned. White-box testing, sometimes also called glass box testing, uses knowledge of the internal implementation and verifies that implemented algorithms perform as intended. E.g., if all control flow paths are taken.

These tests are grouped into multiple levels of testings:

Unit-testing: Any functional unit, such as a function or a class, is tested in isolation from other code. This kind of tests verifies the behavior of objects and effects of functions. It is also known as component testing.

Integration Testing: Interactions between different units using their interfaces are tested with integration tests. E.g., can one object process the returned data from another object.

System testing: Tests which test the full integration of all components into the whole system.

Regression testing: Tests which validate if the software package still operates the same way it did prior to changes. If these tests fail, behavior will have been modified intentionally or unintentionally.

Acceptance Testing: Tests which verify if the business needs of the users are met. This validates if the functionality promised to users is functional using the interface available to the user.

3.4.2 Challenges of testing LIGGGHTS

LIGGGHTS has a large legacy code base which stems from the LAMMPS MD code. While the code is very extensible, easy to read, and highly performant it lacks a set of properties which make it testable. A large set of core code relies on missing encapsulation. It favors simple code and duplication. Given its background from Fortran, many functions and classes directly access the global state of the application. Pointers to global data are copied and held by each class which becomes part of the system.

Core classes have grown larger and have multiple responsibilities. Side effects are spread across the entire code base. The structure of the code invites untrained programmers to use short cuts and not reflect on their changes, which increases the technical debt in the code accumulated over time.

This complexity requires developers to read large portions of the code to understand how it operates and how to extend it. Nevertheless, there is a clean architecture underneath the years of extending and tuning the code.

There is consensus of the necessity of testing in the LAMMPS and LIGGGHTS community. In particular regression testing. However, given the scope of the code, it still suffers from a chicken-egg problem: there are no tests, why start now?

Testing this HPC code is also difficult because it utilizes MPI. Many challenging errors occur only during parallel execution. However, tools such as Unit-Testing Frameworks are all documented and taught with serial programs in mind. At best, multi-threading is tackled.

MPI applications also add another complexity to this application. Multiple implementations of the MPI standard exist. In addition, the hardware used to run the application differ to regular workstation computers.

All of this variability enforces a strong reliance on manual testing. While this is necessary, it should not be the only available testing path. The following chapter will present the test harness which was developed to introduce acceptance and regression testing to LIGGGHTS.

Chapter 4

Acceptance and regression testing of LIGGGHTS

In the course of this work refactoring and developing new features in LIGGGHTS was a routine activity. A large portion of code was modified. Acceptance and regression testing was introduced to LIGGGHTS and CFDEMcoupling to ensure that the changes made did not affect existing functionality. The set of utilities which were used to collect, measure and validate runs and the examples which are used for testing became known as the CFDEMproject Test Harness.

This system employs the simplest form of testing: Given a set of examples these are run by any newly developed version of the code. Not only do these examples have to successfully run, but their dumped or post-processed data must match reference data obtained by a previous version. The testing utility therefore has the following responsibilities:

- Build binaries in specific versions of the code from version control
- Run test cases using these binaries and store the results in a structured way.
- Run post-processing scripts to generate data which can later be used for comparison
- Validate if tests pass (run complete, version consistent with older version)

4.1 Structure of the Test Harness

Figure 4.1 shows the basic structure of the test harness. It is implemented in Python and consists of a backend component and several frontends.

The backend component contains the logic of the test harness and manages its data. Both source code and examples are managed using the git[1] version control system (VCS) [33]. At all times any files in the test harness are linked to a specific version of the source code and of the examples. Compiled binaries are stored in a cache directory on disk. Executed test cases are also placed in a cache folder and extended with metadata.

---

[1] http://git-scm.com/

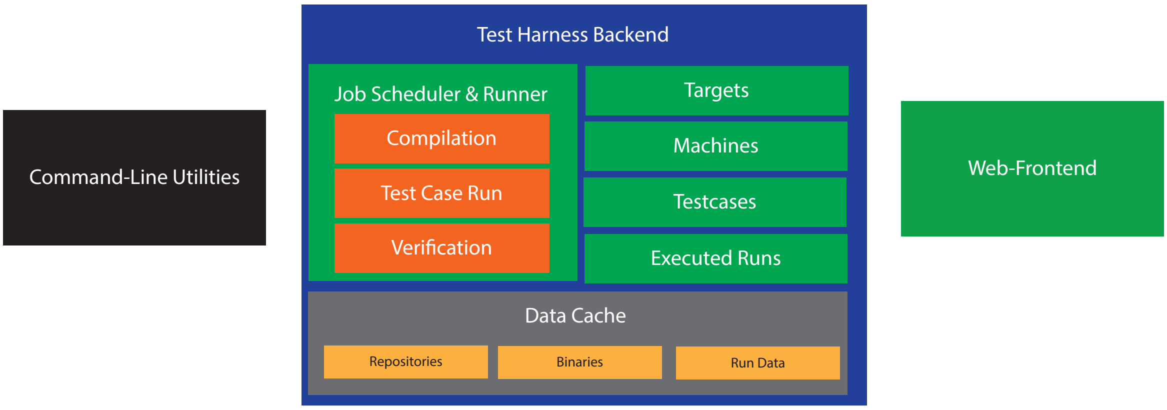


Figure 4.1: Structure of the CFDEMproject test harness

The backend is accessed by a collection of command-line tools to perform certain tasks. These include building binaries, querying the status of test cases and launching test jobs. These form the command-line interface (CLI) frontend.

To further enhance the test harness a web-frontend component was built using the Django Webframework[2]. This is a Python web application which accesses the backend and visualizes the results of the test harness. Through this the results of hundreds and thousands of test case runs can be aggregated and visually presented for performance analysis or manual consistency checks.

4.2 Building binaries

Testing different versions of LIGGGHTS requires managing the creation of binaries and ensuring that test cases are run with the correct executable. Creating a binary consists of multiple steps, which might slightly differ depending on the target platform:

- obtaining a specific version of code from the version control system
- configuration of the build. E.g., activation of additional software modules
- ensuring availability of required dependencies (MPI, VTK library, etc.)
- compilation
- caching binaries with different configurations for later use in test cases

The necessary information to configure such a binary is stored in a *target definition file*. This is a text file in JavaScript Object Notation (JSON) format. The file name specifies the name of the target binary which is created. Listing 4.1 shows the content of such a target definition file.

---

[2] https://www.djangoproject.com/

Listing 4.1: LIGGGHTS-PUBLIC-fedora-2.3.8 target definition file

```
{
  "repository"   :  "LIGGGHTS-PUBLIC",
  "revision"     :  "refs/tags/2.3.8",
  "source_folder":  "./src",
  "make_target"  :  "fedora",
  "generated_executable" : "lmp_fedora",
  "testcases_revision" : "refs/tags/2.3.8",
  "testcases"        : ["Tutorials_public"]
}
```

It instructs the test harness to compile a binary using the source code from the LIGGGHTS-PUBLIC repository using the revision tagged as version 2.3.8. The same revision is used for finding the test cases which should be run. Test cases can be limited to subdirectories. In this case only the public tutorials found in `Tutorials_public` will be used for testing. Note that test case version and source code version must be compatible. Newer versions might change the syntax used in LIGGGHTS input scripts or use features not available in a given version.

The binary is compiled from the `src` folder using the existing Makefile mechanism. It compiles using the fedora make target, which will produce a `lmp_fedora` binary. This binary is them renamed and cached for later use.

Each binary is compiled for a specific target machine. By default, the machine which is used to compile and run test cases is the same which runs the test harness backend. But the test harness allows utilizing other machines as well, including remote clusters with their own Job schedulers. This is achieved by providing a set of shell scripts for each machine which are used to perform tasks such as compilation. For simplicity reasons the remaining text will only assume one target machine. Results obtained and presented in following chapters, however, relied heavily on this capability.

4.3 Running test cases

Test cases in the test harness are not executed directly but through scripts. Running executions and caching the results of test cases is managed by the backend. Test cases are detected in the examples repository specified for each target. Each of these test cases has a unique name and can be scheduled to run using its target binary on a target machine. The available command-line interface allows running individual test cases interactively or generating a list of test cases to run as a batch job. This can be used for automation to run selections of test cases at regular intervals. The following sections describe the setup of such test cases.

4.3.1 Structure of a LIGGGHTS test case

A standardized structure was defined to simplify how a LIGGGHTS test case is run in the test harness environment. Such a test case not only consists of an input script file, which describes the simulation, but also often requires mesh geometries, restart files and post-processing scripts. The following structure has been established:

- **meshes**/ folder: contains all STL mesh files
- **post**/ folder: contains all data dumps before and after post processing
- **post/restart**/ folder: stores restart files necessary to run case
- **scripts**/ folder: containing all post-processing scripts
- **input script file(s)**: a test case can contain one or more input script files. Which of these is used for execution is configured using the `run.config` file
- **run.config** file: configuration of the test case

4.3.2 Configuration of a test case

Running a test case requires information about how to execute it. To store this kind of configuration each test case contains a *run configuration file* called `run.config`. This is a text file which defines a JSON dictionary. It contains a list of run configurations for the same test case. This allows to define multiple parameterized executions using the same test case setup. Listing 4.2 shows such a run configuration file which sets up two runs, one serial and one using MPI.

Listing 4.2: Example of a run configuration file

```
{ "runs" : [
    {
      "name"   : "serial_run",
      "type"   : "serial",
      "script" : "in.serial"
    },
    {
      "name"   : "parallel_run_np4",
      "type"   : "mpi",
      "nprocs" : 4,
      "script" : "in.parallel",
      "variables" : {"XPROC" : 2, "YPROC" : 2, "ZPROC" : 1}
    }
  ]
}
```

Each run configuration is given a unique name. They specify which input script should be used to run the simulation and which execution type (serial, MPI, profiling tools) is used.

Depending on the execution type, additional parameters may be necessary. A serial execution does not require any extra configuration. For MPI executions the number of MPI processes can be specified using the `nprocs` parameter. If an MPI run should be configured with more options, the `mpi/custom` execution type allows specifying MPI options in greater detail with the `mpi_options` parameter.

Additionally, each LIGGGHTS execution can be parameterized using variables. These are passed to LIGGGHTS using the `-var` option, allowing to specify variables which are replaced inside the input script. Listing 4.3 shows the execution command generated from the second configuration.

Listing 4.3: Generated execution call

```
mpirun -np 4 $LIGGGHTS_BINARY -in in.parallel \
       -var XPROC 2 -var YPROC 2 -var ZPROC 1
```

The passed variables are then used inside of the LIGGGHTS input script as parameters. Listing 4.4 demonstrates how the domain decomposition can be parameterized using the `XPROC`, `YPROC` and `ZPROC` variables.

Listing 4.4: Input script line with parameters

```
processors ${XPROC} ${YPROC} ${ZPROC}
```

4.3.3 Dependencies between run configurations

In many cases simulations consist of more than just a single input script. A common pattern seen in DEM simulations is having an initialization script which produces a restart file containing a predefined state of a system. E.g., in a silo discharge simulation the initialization script could fill the silo with particles and produce the initial packing. A second script is then used to run the physically interesting simulation. For this purpose run configurations can specify dependencies. They can depend on the result of another configured run. Listing 4.5 illustrates how dependency runs are specified using their name in the `depends_on` option.

Listing 4.5: Run configuration with a dependant run

```
{
  "runs" : [
    {
      "name"   : "init",
      "type"   : "serial",
      "script" : "in.init"
    },
    {
      "name"    : "run",
      "depends_on" : "init",
      "type"    : "serial",
      "script" : "in.run"
    }
  ]
}
```

A dependent test case run can then only execute if the dependency run was executed first. Since dependencies usually produce data which is used in the following run, such as restart files, the contents of the post directory are copied to the dependent run prior to execution.

4.3.4 States of a test case run

Once configured, each run configuration of a test case can be executed independently. Throughout its lifetime inside the test harness a test case run can go through six different states. Figure 4.2 summarizes the progression from each of these states to each other.

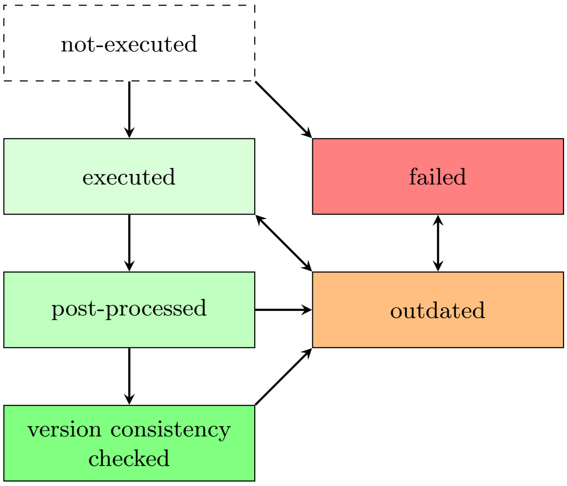


Figure 4.2: States of a test case run

not-executed: Initially a new run will not have been executed. This means that no data is stored in cache which documents its execution.

executed: Once a run is scheduled and run by the test harness, it can succeed and finish without an error, which progresses the state to *executed*.

failed: If something goes wrong, such as a segmentation fault during execution or any other error code is returned while running the test case, its state will be marked as *failed*.

post-processed: Post-processing only occurs if a run has been executed successfully. Once post-processing has completed, the run is marked as *post-processed*.

version-consistency-checked: Once post-processing data is available, additional checks can be performed. This includes checks for version consistency which are explained in Section 4.4. The result of such checks is stored separately. If the checks have been performed, the run state will progress to this state.

outdated: During any of the previous states, the binary might be updated through recompilation. In this case all results of that target will be marked as outdated. While previous results still remain available, the outdated mark should encourage the user to rerun the test cases using the new binary.

4.3.5 Post-Processing of a LIGGGHTS run

LIGGGHTS dumps data which is often post-processed to obtain a set of data files or text files which document the physically significant quantities of the simulation. LIGGGHTS provides a variety of options to output data directly during simulation. The log file itself and the information printed to the console give valuable feedback about the simulation outcome. A simulation is built as a sequence of one or more runs of a defined number of time steps. During such a run dumps can output data periodically.

Thermo output

One option to output data in LIGGGHTS is to use the `thermo` command. This command schedules a series of computations after a specified interval of time steps and outputs them. It is used to output a custom list of global simulation properties. A typical usage can be seen in Listing 4.6.

Listing 4.6: Typical thermo configuration in LIGGGHTS

```
compute         rke all erotate/sphere
thermo_style    custom step atoms ke c_rke
thermo          1000
```

The `thermo_style` command is used to define a list of properties which should be printed during computation. The `step` field stands for the global time step. `atoms` is the total number of atoms/particles in a simulation and `ke` stands for kinetic energy in the system.

There are many more built-in properties which can be accessed this way. Besides the built-in properties additional properties can be calculated using a *compute* command. In this listing a compute is registered with the name `rke` which should be applied to `all` particles in the simulation. The compute selected is `erotate/sphere`, which calculates the rotational kinetic energy in the system. In the `thermo_style` command this additionally computed quantity is accessed through a preceding `c_` and the name of the compute `rke`. The computation then only occurs when the `thermo` command needs it. An example of the data printed to the screen during simulation can be seen in Listing 4.7.

Listing 4.7: Thermo output

```
Step Atoms KinEng c_rke 
       1       12 0.0004686209            0 
    1000       12 0.00050072975           0 
    2000       25 0.0011229892            0 
    3000       38  0.001921502            0 
    4000       51 0.0027794101            0 
    5000       64 0.0037826566            0 
```

While the information dumped by the `thermo` command is part of the log file, the `-thermo` option was added to LIGGGHTS for easier extraction specifically targeted for running in the test harness. It allows dumping thermo output to a specified file.

fix print output

An alternative to dumping data via the `thermo` command is using the `print` fix as shown in Listing 4.8. Fixes are extensions to the LIGGGHTS integration loop which can become active at certain points in time. In this case the `print` fix outputs a line of text every few time steps. The line of text is produced by a formatting string which can be parameterized with any global variables in the system. The output can optionally be printed on the screen and be further saved to a text file. Note the title option can be used to specify the first line of that file to specify the column headers.

Listing 4.8: Usage of print fix to outout time and kinetic energy every 100 time steps to a file

```
fix out1 all print 100 "${time}␣${ke}" screen no &
                  title "t␣ke" file post/data.txt
```

The benefit of using print fixes is that there can be more than one compared to the `thermo` command and it allows to dump data at different time intervals.

Particle data, VTK files

Finally, the largest source of data is periodically storing particle information. A portable format for doing this is the VTK format. It can be produced by using the custom/vtk dump style. VTK is a format widely used for visualizations of large scientific data. Using this format allows utilizing the existing VTK library which provides wrappers to many programming languages, including Python. This allows writing post-processing scripts which access the data using the VTK library.

Unlike `thermo` or `fix print` dump custom/vtk only outputs per-atom/per-particle data. This can include the position, linear and angular velocities, forces, torques and many more, including computed per-atom properties from fixes or computes.

An example of such a dump custom/vtk command is shown in Listing 4.9. It defines a dump which writes data every 100 time steps containing all particles to a VTP file. The command ends with a list of properties which should be stored for each particle.

Listing 4.9: Example of dump custom/vtk

```
dump dmp1 all custom/vtk 100 post/dump_*.vtp id type x y z vx vy vz &
                          fx fy fz omegax omegay omegaz radius
```

Post-processing scripts

Each test case run can be configured to run an arbitrary number of post processing scripts in a predefined order. This is done by adding `post_scripts` to the configuration. This option stores a list of script file names, relative to the test case folder. Scripts must be made executable and contain she-bang (`#!`) lines at the beginning. This allows using any language or interpreter to run the script on Unix operating systems. Currently most post scripts for test cases are written either for shell (Bash), Octave or Python.

4.4 Version Comparison

Running LIGGGHTS produces information through dumping data and post-processing. This information can be used to create a profile of a test case which allows verifying version consistency between different runs. It enables to compute the similarity between two test case runs.

The comparison between two different executions requires to first define which of the two is used as reference. This specifies which execution of the test case contains the desired outcome.

Listing 4.10 shows how this is done in the test harness by specifying a *reference* target in a target definition file of one of the binaries. In this case the code version should be the latest revision from the master branch in version control. Any test cases executed using this target should be compared to the LIGGGHTS-PUBLIC-fedora-2.3.8 reference target. This way the latest master branch will be compared to executions from version 2.3.8.

Listing 4.10: LIGGGHTS-PUBLIC-fedora-master target definition file

```
{
  "repository"   :  "LIGGGHTS-PUBLIC",
  "revision"     :  "refs/remotes/origin/master",
  "source_folder":  "./src",
  "make_target"  :  "fedora",
  "generated_executable" : "lmp_fedora",
  "testcases_revision" : "refs/remotes/origin/master",
  "testcases"        : ["Tutorials_public"],
  "reference"        : "LIGGGHTS-PUBLIC-fedora-2.3.8"
}
```

The next step is to define which data points must be compared to evaluate the consistency between a run and its reference. It is unlikely that comparing the individual particle states, such as position and velocity, between two different executions of the same test case will yield similar results. Numerical simulations are far too easily influenced by changes of code, optimizations, compiler flags, parallelism and execution order of floating-point operations. There are too many variables which can introduce numerical noise to the outcomes. Which is why the comparison of integral and averaged quantities is needed.

These quantities must either be dumped or be produced by one of the post-processing scripts. Once they are available in readable format, the consistency checks can evaluate them.

4.4.1 Comparing plots of data

Any data of the thermo data file is automatically used by the test harness for version comparison. Each quantity is placed in a XY-plot where X is the time step and Y is the individual quantity. Plots of two executions are then compared to each other. To ensure that each test case has at least some data for comparison, any test case in the test harness is required to include the following quantities in its thermo data file:

- current time step
- number of particles in a simulation
- total kinetic energy
- total rotational kinetic energy

These quantities can be used to characterize the system as a whole. Larger deviations usually indicate the simulation is behaving differently. To tolerate a certain amount of uncertainty, which might be caused through numerical noise, the comparison of XY plots uses a confidence interval in each data point.

Every data point of a trajectory is checked to be within an ϵ-neighborhood around its reference value. If all data points are within this neighborhood, the trajectory is consistent with its reference trajectory.

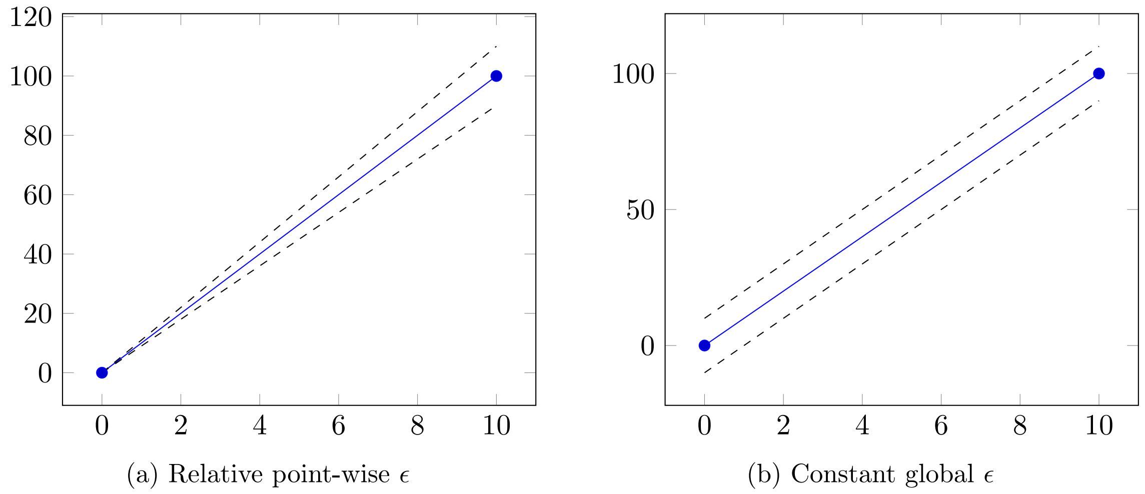
In its current form the comparison algorithm defines two levels of consistency. A deviation of less than 5% is considered to be in perfect agreement with the reference trajectory. Larger deviations up to 10% are labeled as being in good agreement. Beyond 10% the algorithm will label trajectories as inconsistent.[3]

One way to determine the deviation is to calculate the relative offset Δ_k in each point k relative to the reference value $x_{k,ref}$.

$$\Delta_k = \frac{|x_k - x_{k,ref}|}{|x_{k,ref}|} \tag{4.1}$$

---

[3] originally 1% and 5% margins were used, but too many false-positives occurred with that setting



(a) Relative point-wise ϵ (b) Constant global ϵ

Figure 4.3: Acceptable error margin using a relative point-wise ϵ or a global constant ϵ

If each point is not larger or smaller than 10% of its reference point, the new value is consistent with its reference. However, this type of definition has proven to not be practical. The ϵ values in each data point are made proportional to the reference value. This means larger reference values will have a large tolerance, while smaller values, such as values near zero, will have little or no tolerance at all.

A better solution was found by using a heuristic which defines a constant global ϵ value for all data points. It uses the difference between the minimum and maximum value of a trajectory. ϵ is then defined as a certain percentage of this value.

$$\epsilon = 0.1 \frac{\max(x_{k,ref}) - \min(x_{k,ref})}{2} \tag{4.2}$$

Figure 4.3 shows the two ϵ definitions side by side. The benefit of a global ϵ is that it treats trajectories similarly if they are shifted by a constant value. Only the relative change of value remains relevant. It is also not influenced by any values becoming negative or being close to zero. With such a global ϵ the local relative deviation Δ_k of each data point can be computed.

$$\Delta_k = 0.1 \frac{|x_k - x_{k,ref}|}{\epsilon} \tag{4.3}$$

This value is normalized between 0 and 1. Ideally, if the trajectory is consistent, each value should never surpass the 10% margin. The consistency of the entire trajectory is then defined as:

$$C = 1 - \max_k \Delta_k \tag{4.4}$$

If C reaches 1.0, it means the trajectory is fully consistent with its reference. Values above $C > 0.9$ are still tolerable and should capture any numerical effects. Below this value at least one point has deviated a significant amount.

Inconsistency does not automatically mean that the change is not reasonable. It only serves as a hint, a feedback to the developer to have a closer look. In most cases nothing changes except smaller fluctuations. But once the change becomes too large, a human must act and decide if the observed difference is still tolerable. Consistency checks are therefore a tool, which can always be overruled by a user.

4.4.2 Defining custom plots

While data plots are automatically generated based on thermo data contents, some simulations may monitor different quantities which are a result of post-processing. Generating CSV files and making the test runner aware of them allows adding these results to version consistency checking. This is done in two steps:

1. Define data series which load data from a CSV text files
2. Define plots which use these data series

Data series and plots are defined in the run configuration file. They are part of individual runs. Listing 4.11 shows the definition of a data series called `torqueData` which reads in the columns `t` and `torqueZ` from a `torqueZ2.txt` text file. This file is a result of the `plotForce.m` Octave post-processing script.

Listing 4.11: Defining data sources and plots in run configuration

```
...
"post_scripts" : ["plotForce.m"],
"data" : {
  "series" : [
    {
      "name" : "torqueData", "file" : "torqueZ2.txt",
      "columns" : ["t", "torqueZ"]
    }
  ]
  "plots" : [
    {
      "name" : "torqueZ", "title" : "Torque",
      "xdata" : "torqueData.t", "xlabel" : "time␣[s]",
      "ydata" : ["torqueData.torqueZ"], "ylabel" : "torque_z␣[Nm]",
      "legend" : ["torqueZ␣(rotor)"]
    }
  ]
...
```

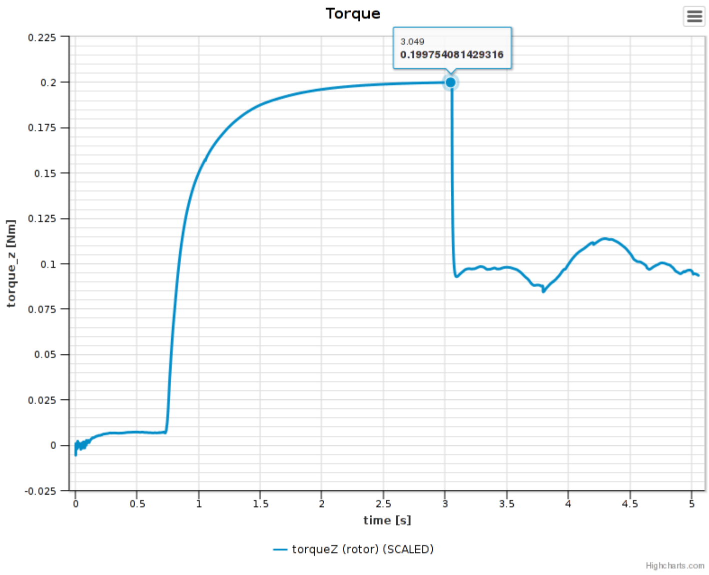


Figure 4.4: Plot generated by web-frontend based on run configuration and test case data.

Using these data columns, a plot can then be set-up. In this case the plot is named `torqueZ` and given the title "Torque". It must define which data columns should be used for each axis. The `xdata` field specifies that `torqueData.t`, should be used for the x-axis. This references the `t` column from the `torqueZ` data series. For the y-axis a list of data columns can be given. Here only `torqueZ` from the `torqueData` series is used.

Finally, for presentation in the web frontend described in Section 4.6, additional properties can be specified to configure the plot. Axes can be labeled using `xlabel` and `ylabel` properties. The data series in a plot can be named using the legend list. Figure 4.4 shows how the web frontend then visualizes this plot.

4.4.3 Comparing reported text and numerical results

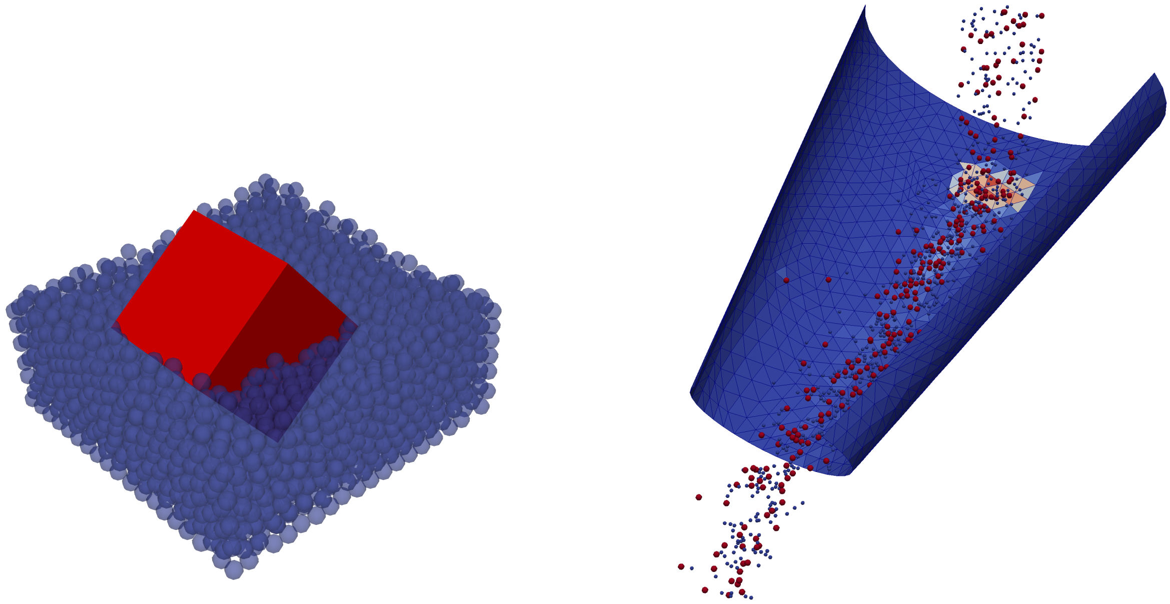
In some test cases the final results is not the trajectory of some quantity, but measured values at the end of the simulation. This information will either be computed directly by LIGGGHTS and dumped to disk or be produced by one of the post-processing scripts. An example of this could be the computed coefficient of restitution determined by a script. One way of making this result accessible for consistency testing is storing it in a separate text file and listing this file as one of the `"textfiles"` used for consistency checking in the data section of the run configuration.

The file is then compared word by word between the two different versions. For consistency the words must match except those parts which can be converted into floating-point numbers. These numeric values must however then be within a predefined threshold from the reference value to be consistent.

4.5 Building a collection of test cases

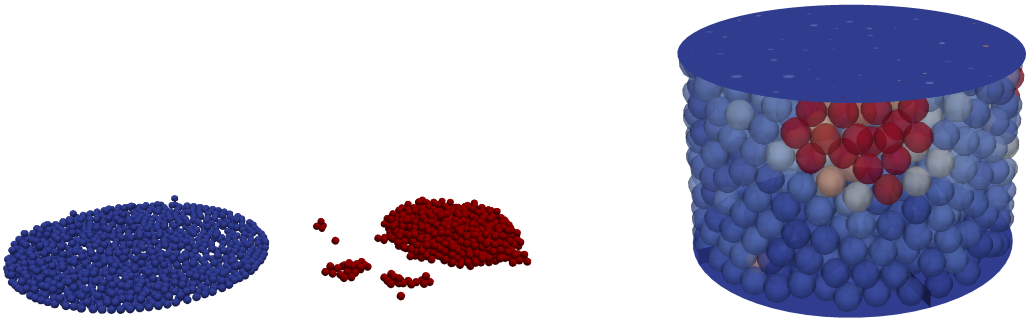
With the test harness in place all that is needed is a collection of test cases which run in a minimal amount of time and test a large portion of the existing functionality in LIGGGHTS. Once these test cases were defined, refactoring larger portions of code could be done without fearing destruction of existing functionality.

The suite of test cases had grown significantly thanks to the efforts of Daniel Queteschiner during the development phase of the test harness. Overall about 200 testcases where defined in the DCS version of the test harness. Figure 4.5 shows four screenshots of test cases currently part of the test harness.



(a) 6 degrees-of-freedom testcase which simulates a mesh cube interacting with a bed of particles

(b) Chute wear test case which simulates the wear on a chute caused by a constant stream of particles



(c) Cohesion test case which simulates a collection of particles dropping on the floor with and without cohesion active.

(d) Servo Wall test case where top wall pushes down, while the lower plate rotates around the center axis.

Figure 4.5: Some test cases from the test harness. Most of these cases were taken from existing examples which are shipped with LIGGGHTS. Original authors: Christoph Kloss, Andreas Aigner and Daniel Queteschiner.

4.6 Web Frontend

Managing the execution of hundreds of tests by hand is a challenging and laborious task. This was a main reason why the test harness was created to automate the most common activities. To fully comprehend and collect the information produced by all of these test case executions, the execution scripts were extended with a web frontend. This is a dynamic website generated based on the execution run and version consistency data. Overall the web frontend has three primary functions:

- Summarize test case run results
- Summarize test case version consistency results
- Allow comparison of arbitrary test case runs

The goal of the first two functions is to give developers a way of quickly obtaining feedback about their changes. Without having to manually go through log files or comparing videos of simulations.

4.6.1 Overview of test cases

Information is presented in an aggregated and comprehensible form. Figure 4.6 shows the overview of all test case executions. A hierarchical structure is enforced. Data is first organized by machine. All machines compile all target binaries and run test cases using these binaries. Each target binary then has its unique set of test cases. E.g., some machines might not support executions with more than 8 cores.

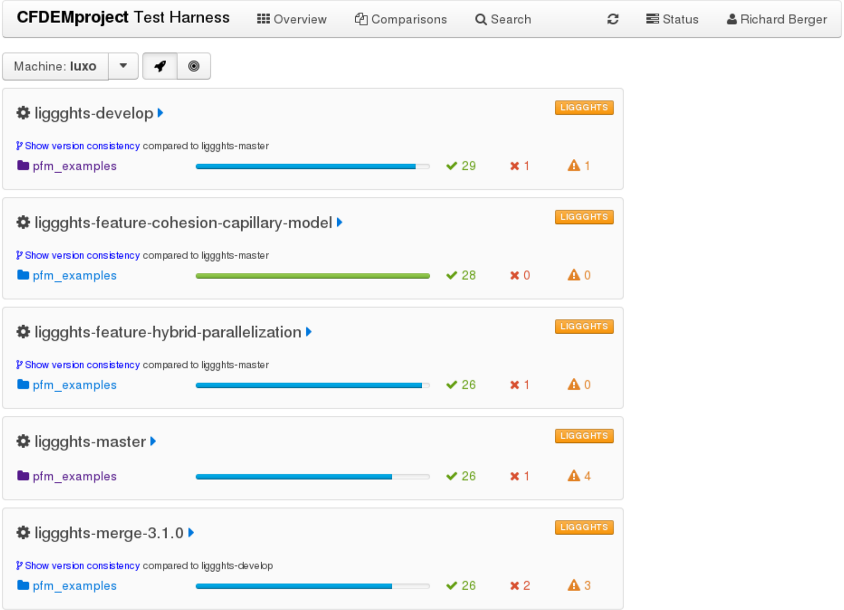
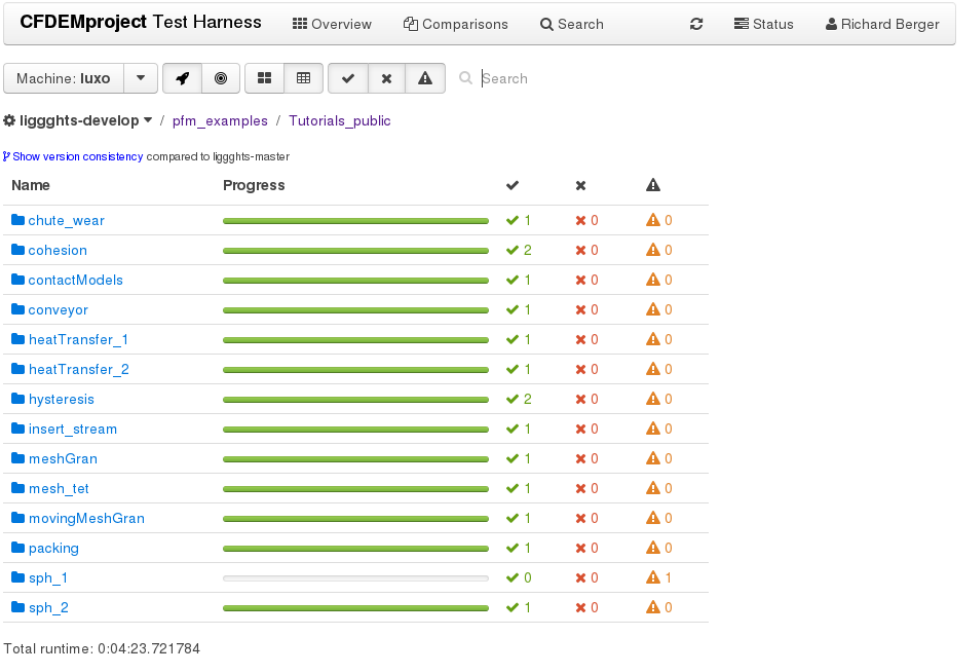

CFDEMproject Test Harness
Overview
Comparisons
Search
Status
Richard Berger
Machine: **luxo**
liggghts-develop
LIGGGHTS
Show version consistency compared to liggghts-master
pfm_examples
✓ 29
✖ 1
⚠ 1
liggghts-feature-cohesion-capillary-model
LIGGGHTS
Show version consistency compared to liggghts-master
pfm_examples
✓ 28
✖ 0
⚠ 0
liggghts-feature-hybrid-parallelization
LIGGGHTS
Show version consistency compared to liggghts-master
pfm_examples
✓ 26
✖ 1
⚠ 0
liggghts-master
LIGGGHTS
pfm_examples
✓ 26
✖ 1
⚠ 4
liggghts-merge-3.1.0
LIGGGHTS
Show version consistency compared to liggghts-develop
pfm_examples
✓ 26
✖ 2
⚠ 3

Figure 4.6: Overview of Test Harness executions

At the highest level, the overview only shows the targets of the currently selected machine. Each target contains a list of test case folders. In this case each target only contains a single folder. Each folder summarizes the total number of executions using a colored bar. Only successful runs contribute the value of the bar. The color encodes the success rate. If less than 25% of all test cases failed, the bar will be colored in red. Beyond this it will remain blue until it reaches a 100% and turns green. Each folder lists the number of successful, failed and not executed tests.

Clicking on one of the example folders navigates into that folder. For each subfolder the success statistics are again aggregated and the total run time is computed. Folders can be navigated through, similar to any other file browser. At the lowest level a test case folder lists its run configurations and their status.


CFDEMproject Test Harness
Overview
Comparisons
Search
Status
Richard Berger
Machine: luxo
Search
liggghts-develop / pfm_examples / Tutorials_public
Show version consistency compared to liggghts-master

Name	Progress	✓	✗	⚠
chute_wear		1	0	0
cohesion		2	0	0
contactModels		1	0	0
conveyor		1	0	0
heatTransfer_1		1	0	0
heatTransfer_2		1	0	0
hysteresis		2	0	0
insert_stream		1	0	0
meshGran		1	0	0
mesh_tet		1	0	0
movingMeshGran		1	0	0
packing		1	0	0
sph_1		0	0	1
sph_2		1	0	0

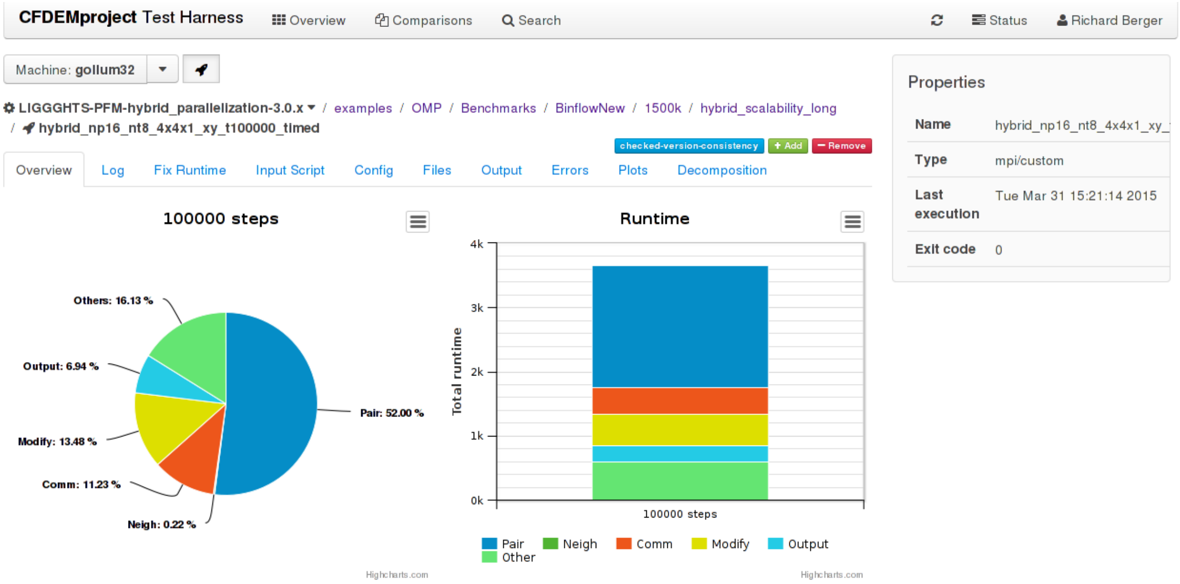
Total runtime: 0:04:23.721784

Figure 4.7: Test cases are organized in a folder hierarchy which is browsable and can be filtered based on run status.

4.6.2 Test case run information

Each run can be viewed individually as shown in Figure 4.8. The detailed view of a run presents run-time statistics which break down how much time was spent in important portions of code. It also lists the input script, the full run configuration, log file output, console output and error output. Finally, it automatically generates plots based on dumped data, such as thermo output or custom specified data series, as has been presented in Figure 4.4.

In addition to viewing a single run in isolation, runs can be selected and compared against each other (see Figure 4.9). This allows spotting differences between two different test case configurations. Each of the detailed run subviews are exchanged with a comparison. Run-time statistics are compared next to each other. Stacked bar charts are placed next to each other in



LIGGGHTS-PFM-hybrid_parallelization-3.0.x / examples / OMP / Benchmarks / BinflowNew / 1500k / hybrid_scalability_long / hybrid_np16_nt8_4x4x1_xy_t100000_timed

Properties

Name: hybrid_np16_nt8_4x4x1_xy_

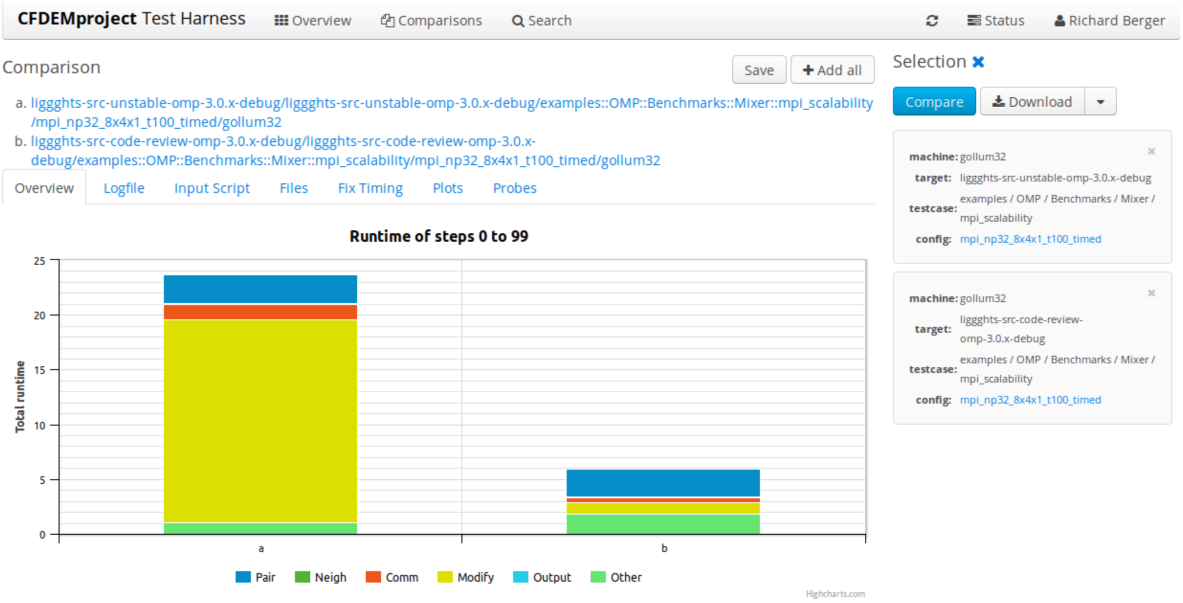
Type: mpi/custom

Last execution: Tue Mar 31 15:21:14 2015

Exit code: 0

Figure 4.8: Detailed information of each test case run time breakup

this view. This can be used to compare the speedup obtained by runs with different processor numbers or different binaries. Both input script and run configuration are compared to each other using a unified diff format. These diffs are however limited to a comparison between two runs. Outputs from log files, screen and error output are also compared using a diff view. Finally plots from all runs are aggregated and presented in a single plot for each shared data series.



a. liggghts-src-unstable-omp-3.0.x-debug/liggghts-src-unstable-omp-3.0.x-debug/examples::OMP::Benchmarks::Mixer::mpi_scalability/mpi_np32_8x4x1_t100_timed/gollum32

b. liggghts-src-code-review-omp-3.0.x-debug/liggghts-src-code-review-omp-3.0.x-debug/examples::OMP::Benchmarks::Mixer::mpi_scalability/mpi_np32_8x4x1_t100_timed/gollum32

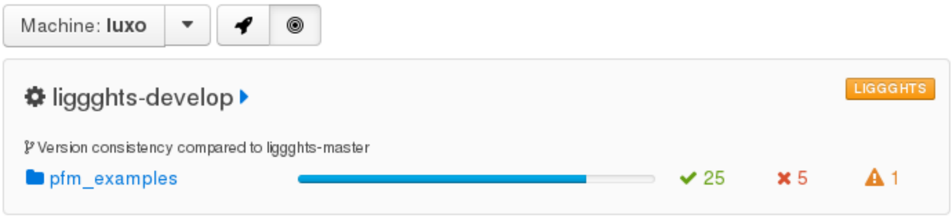
machine: gollum32
target: liggghts-src-unstable-omp-3.0.x-debug
testcase: examples / OMP / Benchmarks / Mixer / mpi_scalability
config: mpi_np32_8x4x1_t100_timed

machine: gollum32
target: liggghts-src-code-review-omp-3.0.x-debug
testcase: examples / OMP / Benchmarks / Mixer / mpi_scalability
config: mpi_np32_8x4x1_t100_timed

Figure 4.9: Comparison of two test case runs. This can be extended to an arbitrary number of runs and helps to identify bottlenecks and spot differences in configuration.

4.6.3 Version Consistency

The comparison capabilities of the web frontend form the basis for visualizing version consistency. Starting from the overview, all test cases and runs can be viewed in a version consistency mode if the target binary has a specified reference binary (see Figure 4.10). All success, failure and not executed counts are then exchanged. For version consistency a run can either be consistent, not-consistent, or not checked. The same color coding is used as for run executions.


Machine: **luxo**

liggghts-develop

LIGGGHTS

Version consistency compared to liggghts-master

pfm_examples ✓ 25 ✖ 5 ⚠ 1

Figure 4.10: Overview of version consistency of all test cases

The navigation among test cases and runs is similar. One main difference is that looking at a single test case run in detail does not show a single execution anymore, but always opens a comparison between the current target binary run and its reference run. Comparing two runs for version consistency adds additional information in the detailed views. For one, it will add the version consistency value to the overview, which is the lowest consistency value during the consistency tests. A more notable difference are comparison plots which include acceptance margins between 5 and 10%. An example of this is shown in Figure 4.11 which illustrates the difference in torque between two different binaries. Trajectories moving outside of the acceptable tolerances can be detected and can lead to further investigation.

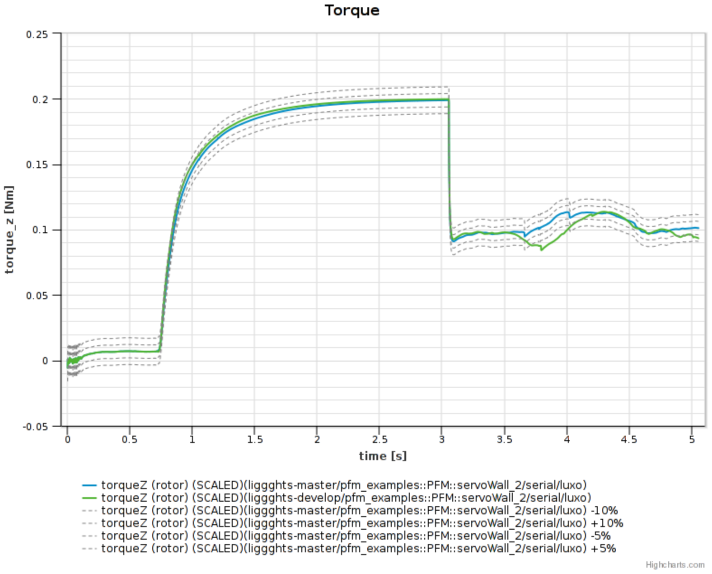


Figure 4.11: The test case specifying this torque plot is reported of being inconsistent because its trajectory has moved beyond the acceptable bounds. These act as a hint and require a human to make the final decision.

4.7 Continuous Integration with Jenkins

The test harness allows to verify whether the functionality present in one version of LIGGGHTS is consistent with a previous version. This capability allows detecting anomalies introduced into the code. With the test harness in place, continuous integration [34, 35] for LIGGGHTS could now be realized.

The idea behind continuous integration is that software should be developed in such a way that the current version is always releaseable. If changes are applied to the code it should be compiled and tested for correctness. A continuous integration (CI) server automates these tasks. The goal is to create a feedback loop which helps developers to detect mistakes and regressions as they happen, instead of stumbling upon them months later.

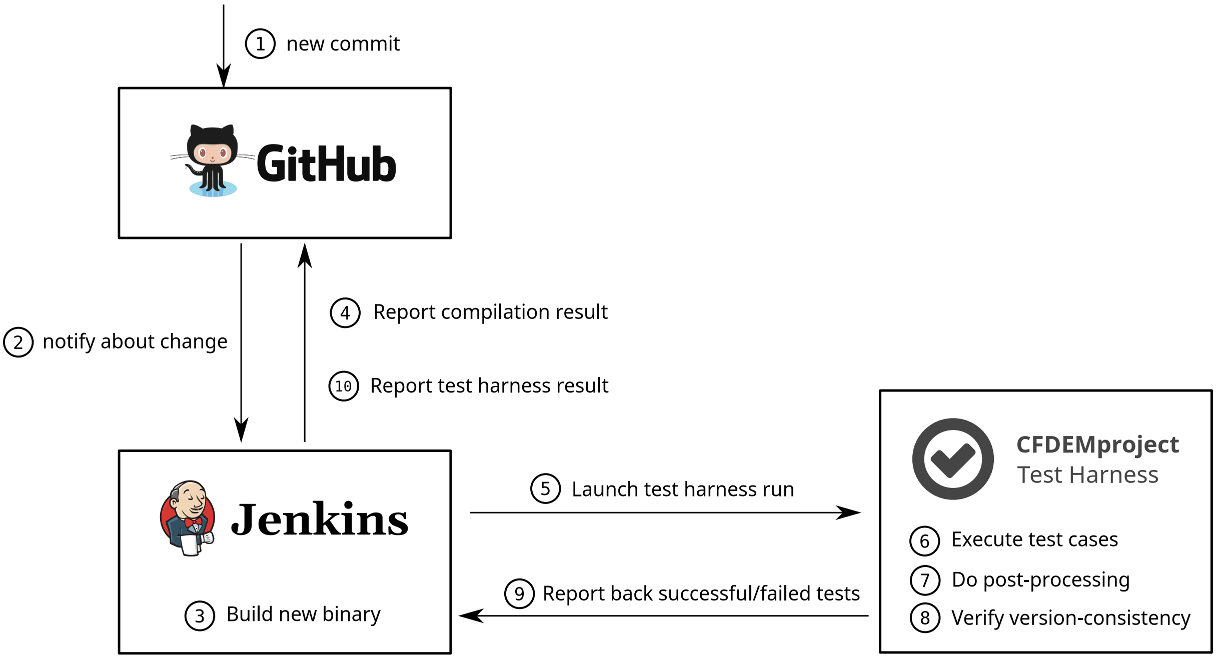

① new commit
GitHub
② notify about change
④ Report compilation result
⑩ Report test harness result
Jenkins
③ Build new binary
⑤ Launch test harness run
⑨ Report back successful/failed tests
CFDEMproject
Test Harness
⑥ Execute test cases
⑦ Do post-processing
⑧ Verify version-consistency

Figure 4.12: Continuous Integration for LIGGGHTS

Figure 4.12 shows a high-level overview of the procedure which was set-up for LIGGGHTS. The LIGGGHTS code has for a long time been hosted in git repositories on GitHub[4] which provides many features commonly used in open-source projects. This includes issue-tracking, wikis, and release management. It also provides an API which allows external services to interact with the git repository.

One of the usages for this API is the addition of a continuous integration server. Jenkins[5] is an open-source implementation of such a CI server. It allows to specify tasks which should be run either based on a time schedule or through external triggers. It is an extensible system with a vibrant plug-in developer community. One of these plug-ins allows to integrate with GitHub and react to changes in its git repositories.

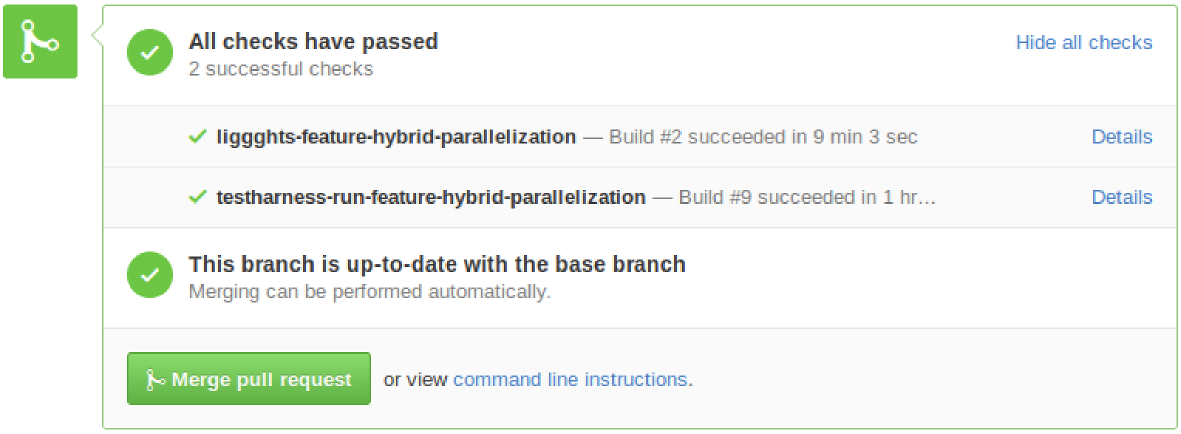
---

[4] http://www.github.com
[5] http://www.jenkins-ci.org

Developers work on branches in the git repository and can mark these branches for continuous integration in Jenkins. The Jenkins server will then listen to changes which occur in the repository via the GitHub API. Any new commit in an observed branch will trigger a sequence of actions.

First the latest code is fetched from GitHub and the binary is built. The outcome of this compilation is reported back to GitHub. If successful, Jenkins launches a test harness run. This will execute all test cases, perform necessary post-processing and verifies version consistency of these cases with the last stable version of LIGGGHTS. The results of the test harness run are reported back to Jenkins using the JUnit XML format. This allows reviewing and generating statistics as if they were regular unit tests. After completion of the test harness run, the results are also reported back to GitHub.

Continuous integration allows verifying if contributions to the code can be trusted. It enforces that the code can be compiled and verifies it does not break existing functionality. This can be integrated into a Pull-Request driven workflow [33], which is a common workflow found in the open-source community. Contributors fork an existing repository, apply their changes in a local branch and create a pull-request to the original project so their changes can be integrated. A CI server can then perform its checks and tests. Maintainers can use this information to decide if a contribution should be merged into the code base. Figure 4.13 shows a screenshot of how GitHub modifies the pull request user interface and adds useful information from the CI server.



All checks have passed

2 successful checks

✓ **liggghts-feature-hybrid-parallelization** — Build #2 succeeded in 9 min 3 sec

✓ **testharness-run-feature-hybrid-parallelization** — Build #9 succeeded in 1 hr…

This branch is up-to-date with the base branch

Merging can be performed automatically.

or view command line instructions.

Figure 4.13: Results of continuous integration run on a pull request on GitHub

Chapter 5

Structure of LIGGGHTS 2.x

One of the main benefits of LIGGGHTS is its extensibility inherited from LAMMPS. LIGGGHTS has used the existing mechanisms to add many features such as granular force kernels, mesh support and heat transfer into the code. The philosophy behind extensions in LAMMPS and LIGGGHTS is to add features by creating new files in the source directory and recompiling. It is therefore a monolithic code with compile-time extensibility built in.

5.1 Basic Concepts

5.1.1 Extension points in LIGGGHTS

LIGGGHTS defines a rich set of extension points to its core functionality. Extensions are written by deriving from a small set of base classes which define the common interface.

Pair Styles: A pair style defines how particle-particle interaction forces are computed. In addition, it also manages the creation of contact histories. Apart from forces, LIGGGHTS has also added heat transfer to the pair style.

AtomVec Styles: AtomVec styles define what additional information is stored for each particle and allocates the needed memory. E.g., a spherical particle will store a radius, density, angular velocity and torque in addition to properties such as position, velocity and force.

Fixes: Fixes allow modifications of the simulation universe at specific points in time during the integration loop. The `Fix` base class defines many hooks where new functionality can be attached by subclassing.

Computes: A compute defines how scalar or vector quantities are calculated. Either a single value or per-atom values are possible. These can then be dumped as output or used to control the execution of the script through control-flow constructs.

Commands: New commands can be added to the input script language by defining a subclass of the `Command` class.

5.1.2 Extension Mechanism

All extensions are orchestrated by preprocessing shell scripts during compilation. Extensions are defined as files with a defined name prefix. E.g., Pair styles headers have a prefix of `pair_` while Compute styles use a prefix of `compute_`. During preprocessing these files are detected and a style header file is generated which includes all these headers.

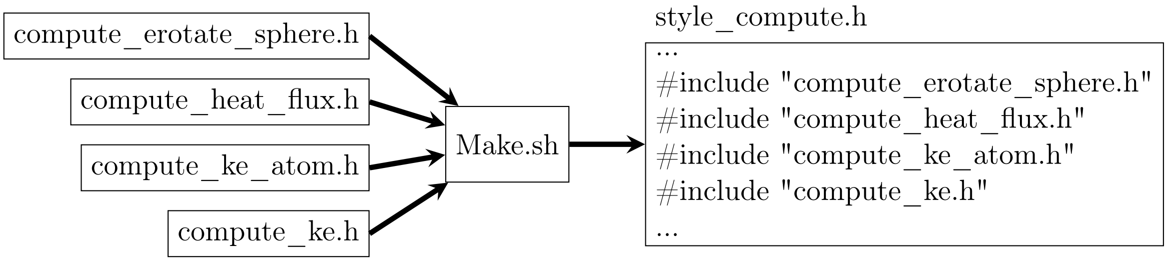


Figure 5.1: Headers with a prefix are collected and included in a generated style header.

Defining a Compute style

Style headers have a predefined structure. In addition to regular include-guards, which ensure each header is included only once, they contain a file-wide **`#ifdef`**-**`#else`**-**`#endif`** construct which controls what part of the header is included. This is used to generate code by defining Macros prior to inclusion of the header.

Listing 5.1 shows how a compute style is defined. Defining `COMPUTE_CLASS` tell the preprocessor to use the `ComputeStyle` Macro instead of the remaining header. Using this macro properties of the given extension can be specified.

Currently the only information passed to this macro is the string used to identify the extension and the class name of the implementing class. Expansions of `ComputeStyle` macro are used to generate arbitrary code based on this information.

If `COMPUTE_CLASS` is not defined all regular declarations are included by the header. Similar preprocessor variables and macros exist for PairStyles and other extension points (`PAIR_CLASS`, `PairStyle`).

Listing 5.1: Basic structure of `compute_ke.h`

```
#ifdef COMPUTE_CLASS
ComputeStyle(ke,ComputeKE)
#else
#ifndef LMP_COMPUTE_KE_H
#define LMP_COMPUTE_KE_H

#include "compute.h"

namespace LAMMPS_NS {
```

```
class ComputeKE : public Compute {
  ...
};
}
#endif
#endif
```

Generated code from style files

The automatically generated style headers are included to generate code using the C++ preprocessor. Usually this involves building up code which instantiates the correct class based on the selected style parameter. In both LAMMPS and LIGGGHTS the most common method of doing this was creating a long IF-cascade. The following code sample illustrates how an IF statement is extended with additional branches using the preprocessor.

Listing 5.2: Code using style file to generate code

```
  if (0) return NULL;
#define COMPUTE_CLASS
#define ComputeStyle(key,Class) \
  else if (strcmp(style,#key) == 0) return new Class(lmp);
#include "style_compute.h"
#undef ComputeStyle
#undef COMPUTE_CLASS
```

The C++ preprocessor expands this code and applies the macro to each included header. This way the following code gets generated:

```
  if (0) return NULL;
  else if (strcmp(style,"heat/flux") == 0) return new ComputeHeatFlux(lmp);
  else if (strcmp(style,"ke/atom") == 0) return new ComputeKEAtom(lmp);
  else if (strcmp(style,"ke") == 0) return new ComputeKE(lmp);
```

5.2 Pair Styles

A core component of the LIGGGHTS DEM simulation engine are its pair styles. These define how particles interact with each other based on a variety of force models. Many models have come from literature, but others have also been developed and validated internally through experiments.

5.2.1 Force Loop

Each pair style implements a method called `compute_force` which contains the main force loop. It is executed between the two Velocity-Verlet integration steps. In this loop the current force acting on each particle is computed based on interactions with other particles.

All particles are traversed and checked whether they interact with any neighboring particles. This assumes that there exists a valid neighbor list which must have been constructed in an earlier step. Listing 5.3 illustrates the basic structure of such a force loop.

Listing 5.3: Granular force kernel loop

```
for(all particles i) {
  for(all neighbors j of i) {
    if(i and j have overlap) {
      // collision case
    } else {
      // no-collision case
    }

    // apply F to i

    if(use newton 3) {
      // apply -F to j
    }
  }
}
```

One central quantity is the overlap between the particle spheres. Some models act only if there is an overlap, others still apply forces even though particles are no longer in contact. In either case models try to minimize computation by limiting themselves to what is needed. Deriving complicated model parameters in a non-collision case for example may not be necessary in every case. Other quantities such as the relative velocity between particles have applicability across different models, which is why re-computation of these in different models should be avoided.

5.2.2 Granular force kernels in LIGGGHTS

In general the equations of motion of two colliding particles i and j can be written as follows:

$$m_i \ddot{\mathbf{x}}_\mathbf{i} = m_i \mathbf{g} + \sum_{i \neq j} \mathbf{F}_{ij}$$

$$m_j \ddot{\mathbf{x}}_\mathbf{j} = m_j \mathbf{g} + \sum_{i \neq j} \mathbf{F}_{ji}$$

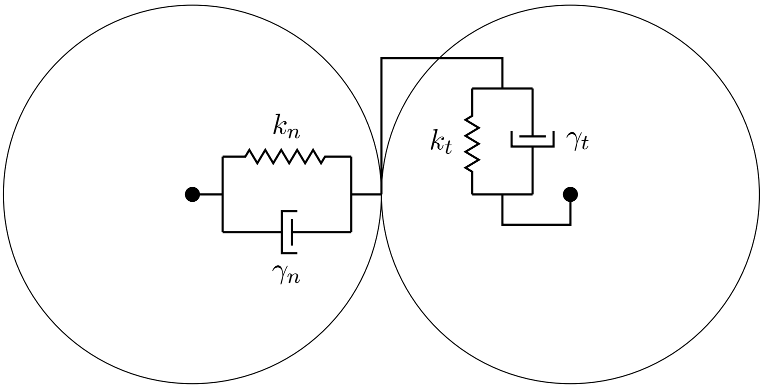


Figure 5.2: spring-dashpot system

Due to Newton's third law the force $\mathbf{F}_{ji}$ is given by:

$$\mathbf{F}_{ji} = -\mathbf{F}_{ij}$$

Contact forces in LIGGGHTS are computed based on the soft-sphere model. In this model particles can overlap to a certain extent. It assumes that particles deform on impact which is the source of stresses, forces and torques.

In DEM these forces are usually modeled using a spring-dashpot model. Figure 5.2 illustrates such a system with springs and dashpots for both normal and tangential force components. These couple the two EOM through the collision force term:

$$\mathbf{F}_{ij} = \underbrace{(k_n \delta \mathbf{n}_{ij} + \gamma_n \mathbf{v}_{n_{ij}})}_{\text{normal force}} + \underbrace{(k_t \delta \mathbf{t}_{ij} + \gamma_t \mathbf{v}_{t,ij})}_{\text{tangential force}}$$

The resulting total collision force then made up of normal and tangential force terms. Each component consists of a spring with a spring coefficient k and dashpot with a damping coefficient γ. In general these coefficients are functions of time, the positions of both collision partners i and j, their respective velocities and properties p_{ijk}.

$$\begin{aligned} k_n &= k_n(t, \mathbf{x}_i, \mathbf{x}_j, \mathbf{v}_i, \mathbf{v}_j, p_{ik}, p_{jk}) & \qquad \gamma_n &= \gamma_n(t, \mathbf{x}_i, \mathbf{x}_j, \mathbf{v}_i, \mathbf{v}_j, p_{ik}, p_{jk}) \\ k_t &= k_t(t, \mathbf{x}_i, \mathbf{x}_j, \mathbf{v}_i, \mathbf{v}_j, p_{ik}, p_{jk}) & \qquad \gamma_t &= \gamma_t(t, \mathbf{x}_i, \mathbf{x}_j, \mathbf{v}_i, \mathbf{v}_j, p_{ik}, p_{jk}) \end{aligned}$$

There is large variety of models [12, 10, 36] to compute these coefficients. E.g., in the original work of Cundall [9], a linear spring model was used with constant coefficients.

Hooke model

One other model using constant coefficients for springs and damping is the Hooke model implemented in LIGGGHTS. Its formula depends on material properties and particle properties such as:

- Effective radius R^*
- Effective mass m^*
- Effective Young's modulus Y^*
- Characteristic Impact Velocity V
- Coefficient of Restitution e

The effective quantities are given by the following formula:

$$\begin{aligned} R^* &= \frac{R_i R_j}{R_i + R_j} \\ m^* &= \frac{m_i m_j}{m_i + m_j} \\ \frac{1}{Y^*} &= \frac{1-\nu_i^2}{Y_i} + \frac{1-\nu_j^2}{Y_j} \end{aligned}$$

where Y_i and Y_j are the Young's modulus and ν_i and ν_j the Poisson ratio of the particle's material.

Using all these parameters, the model computes constant coefficients for a given collision pair of particles i and j. Constant in this context means that there is no dependence on time, position or velocities of the particles.

$$\begin{aligned} k_n &= \frac{16}{15}\sqrt{R^*}Y^* \left(\frac{15 m^* V^2}{16\sqrt{R^*}Y^*}\right)^{\frac{1}{5}} \\ \gamma_n &= \sqrt{\frac{4 m^* k_n}{1 + \left(\frac{\pi}{\ln e}\right)^2}} \\ k_t &= k_n \\ \gamma_t &= \gamma_n \end{aligned}$$

Hertz model

This non-linear model derives the coefficients based on Hertzian theory [37]. The resulting coefficients depend on the overlap distance δ_n of the two particles. In addition, this model also requires the computation of the effective shear modulus G^* and β derived from the coefficient of restitution e.

$$k_n = \frac{4}{3}Y^*\sqrt{R^*\delta_n}$$

$$\gamma_n = -2\sqrt{\frac{5}{6}}\beta\sqrt{2m^*Y^*\sqrt{R^*\delta_n}} \qquad \beta = \frac{\ln(e)}{\sqrt{\ln^2(e)+\pi^2}}$$

$$k_t = 8G^*\sqrt{R^*\delta_n}$$

$$\gamma_t = -2\sqrt{\frac{5}{6}}\beta\sqrt{8G^*\sqrt{R^*\delta_n}m^*} \qquad \frac{1}{G^*} = \frac{2(2-\nu_i)(1+\nu_i)}{Y_i} + \frac{2(2-\nu_j)(1+\nu_j)}{Y_j}$$

Tangential forces and torques without shear history

The tangential force in LIGGGHTS is computed based on a frictional component and velocity damping.

$$\mathbf{F}_\mu = \mu\mathbf{F}_n$$
$$\mathbf{F}_d = \gamma_t\mathbf{v}_{t,rel}$$

The lower force of these two is used to compute the final tangential force F_t and the resulting torques. This ensures that the upper limit of the tangential force follows the Coulomb criterion $F_t \leq \mu F_n$.

$$F_t = \min(F_n, F_d)$$
$$\mathbf{F}_t = F_t\frac{1}{\|\mathbf{v}_{t,rel}\|}\mathbf{v}_{t,rel}$$
$$\mathbf{M} = R_c\mathbf{e}_n \times \mathbf{F}_t$$

Tangential forces with shear history

If shear history is taken into account, tangential forces are computed based on a shear contribution which is integrated over time. This requires each particle to store shear history values across time steps through the use of contact history values.

$$\mathbf{F}_t = k_t\underbrace{\left\|\int_{T_0}^{T}\mathbf{v}_{t,rel}(\tau)d\tau\right\|}_{\delta_t}\mathbf{e}_t + \gamma_t\mathbf{v}_{t,rel}$$

The value of the resulting tangential force is limited by the Coulomb's criterion.

Cohesion

Another force which can influence particle behavior is cohesion, which can lead to particles attracting each other. This attraction can even outlast the particle contact. A basic cohesion model is the simplified Johnson-Kendall-Roberts (JKR) model [38]. It introduces an additional normal force contribution. Once particles are in contact, this force will try to maintain the contact. The contact force is defined as

$$F = kA$$

where k is the cohesion energy density (J/m^3) and A the particle contact area. In case of two spheres this area is a circle with a radius dependent on the particle distance.

Rolling friction

Using spherical particles in DEM simplifies computation, but neglects the effects of the real shapes of particles. One way of modeling non-sphericity is by using a model of rolling friction. The constant directional torque (CDT) model [39] is one of these available models in LIGGGHTS. It adds a torque contribution and is configured by a rolling friction coefficient. The model also depends on the normal spring coefficient and the relative angular velocity.

5.2.3 Properties

The previous sections have given an outline of the types of force laws implemented in LIGGGHTS. Selecting either the Hertzian or Hooke model defines coefficients necessary to compute normal and tangential forces. Other models such as rolling friction also add contributions. These might depend on results of normal and tangential models. This enforces a specific order of computations if certain models are active.

What all models have in common is their need of material properties derived from each particle and inter-particle properties such as the relative velocity computed during collision. Overall there are three types of properties:

Particle properties: Particle properties such as radius, mass and material type are unique to each particle. A collision of two particles can produce unique combinations of these properties. E.g. while particles might be of the same type, they could have different radii in a polydisperse simulation. Other particle properties include values stored in a contact history, e.g. for storing shear contributions.

Particle properties are usually defined during their creation, e.g. through a particle template as illustrated in Listing 5.4.

Listing 5.4: Definition of particle properties by template

```
fix pts1 all particletemplate/sphere 1 atom_type 1 &
             density constant 2500 radius constant 0.0015
fix pts2 all particletemplate/sphere 1 atom_type 2 &
             density constant 2500 radius constant 0.0025
```

Type specific properties: Some properties are not specific to a single particle, but only depend on the type of the particle. These usually involve material properties such as Young's modulus or Poisson ratio. For these types of properties a lookup table can be implemented, because the number of particle types is usually small compared the particle count. Type properties are specified using `property/global` fixes (see Listing 5.5).

Listing 5.5: Definition of type properties via property/global fixes

```
  fix m1 all property/global youngsModulus peratomtype 5.e6 5.e6
  fix m2 all property/global poissonsRatio peratomtype 0.45 0.45
```

Type Pair properties: There are some properties which are dependent on the combination of particle types coming together. E.g., the coefficient of friction can be dependent whether the contact partners are of the same material or not. For this a matrix of values needs to be specified for each type combination.

	A	B
A	0.3	0.15
B	0.15	0.3

Table 5.1: Matrix of per-atom type property

Such a table is specified using a `peratomtypepair` definition of a `property/global` fix.

```
fix m4 all property/global coefficientFriction peratomtypepair 2 &
                                               0.3 0.15 &
                                               0.15 0.3
```

Run-time properties: Finally, some values are only available at run-time. If a collision occurs, the particles will be separated by a certain distance δ_n. By combining their velocities, a relative velocity $\mathbf{v}_{rel}$ can be computed and further dissected into normal $\mathbf{v}_{n,rel}$ and tangential components $\mathbf{v}_{t,rel}$. Run-time properties are usually expensive to compute. They might involve costly operations such as calling the `sqrt` or `pow` function. Such quantities should be computed only once, even if they are used in different places of the force computation. Re-computation of such quantities should be avoided at all cost.

5.2.4 Contact History

Many contact models only compute forces based on information of the current time step. Any intermediate results obtained inside of the granular force kernel are discarded after computation.

Models with integral quantities, however, need a mechanism to accumulate values across time steps for each contact pair. This calls for a dynamic data structure which adjusts to the ever changing number of contact pairs, stays memory and cache efficient, and correctly communicates data between processors during parallel runs.

Further, each model combination might have different needs. While tangential history effects with a Hertzian normal model might only need three history values to track three shear stress components, activating a rolling friction submodel extends the needed history by another three values for each contact pair.

For this reason pair styles also manage a so-called contact history which allows storing arbitrary values for each particle-particle contact. However, in LIGGGHTS 2.x this data structure is nothing more than a single allocated block of memory with two dimensional indexing. The complexities of managing and accessing this contact history are scattered throughout the code base, making it hard for model developers to understand and use this functionality.

5.3 Walls

DEM simulations require an approach to introduce geometries into the simulation domain. Initially LIGGGHTS operates within a orthogonal finite block region. Particles leaving this region will automatically be discarded. To prevent this behavior, walls can be introduced into the domain using fixes. Currently LIGGGHTS supports the following two types:

- primitive walls
- mesh walls

Both wall types are configured using one of the available wall/gran fixes. In LIGGGHTS 2.x subclasses of `FixWallGran` implement different contact models, similar to the ones available for pair styles.

5.3.1 Primitive Walls

The primitive wall type defines either an infinite plane or cylinder. Planes are positioned by specifying an offset from the point of origin along one of the main axis. Cylinders additionally specify a second offset and a radius.

Each primitive wall maintains a separate neighbor list. This reduces the amount of particles needed to be checked to only the ones near the wall. A wall fix must also maintain a contact history if the used contact model needs to store history values for each particle-wall interaction.

5.3.2 Mesh Walls

More complex geometries are defined using meshes. Figure 5.3 shows the different classes which make up the mesh implementation in LIGGGHTS. This is a high-level overview, hiding the details of template driven mesh code derived from the `AbstractMesh` base class. Mesh geometries are loaded from STL files by `FixMeshSurface` or one of its subclasses. The mesh itself is managed by an implementation of the `AbstractMesh` base class, most commonly a triangle mesh implemented by `TriMesh`. Besides the mesh geometry `FixMeshSurface` further instantiates two

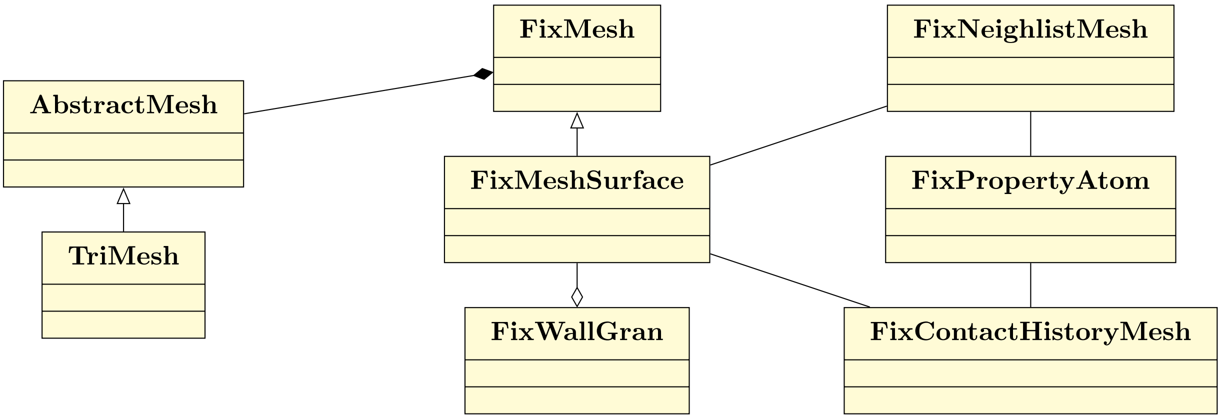


Figure 5.3: Simplified class hierarchy of the mesh implementation

fixes to track additional data. E.g., an instance of `FixNeighlistMesh` is created to periodically create a neighbor list of particles for each mesh triangle. `FixContactHistoryMesh` is instantiated for each mesh to manage contact history data. Since the contact history must be able to accomodate all potential partners, the number of neighboring triangles is synchronized between the two fixes. In the current implementation this is done through a property/atom fix because it automatically handles synchronization across other processors for this particle-bound property. Finally, an instance of `FixWallGran` is responsible for computing the forces acting on particles due to walls based on a contact model.

These meshes can then further be translated, scaled and rotated by fixes of type `mesh/move`. This can either happen once or continuously over time to create a moving mesh.

The MPI parallelization of meshes ensures that both triangles and their associated contact histories are migrated to other processors when necessary. Mesh communication is implemented similar to particle communication. Only data from halo regions is synchronized when necessary.

5.4 Limitations of Extension Points

One of the problems of the existing extension mechanisms is that base classes have grown to enormous proportions over time to host a large variety of derived classes. It enforces polymorphism as only strategy for extensions. However, this type of extensibility is not sufficient for performance sensitive computation kernels. These use inlining and try to avoid virtual calls at all cost.

Many pair styles and wall fixes were added using the existing framework. However, this approach caused LIGGGHTS to reach a critical limit of maintainability. Many styles were created by duplicating large portions of code in order to avoid virtual calls. Because of this code duplication changes in one pair style often led to changes in multiple files at once. This inevitably has caused some parts of the application to diverge.

Figure 5.4 illustrates the granular pair style class hierarchy in LIGGGHTS 2.x. `PairGran` implements the main interface required by each pair style and is derived from the LIGGGHTS

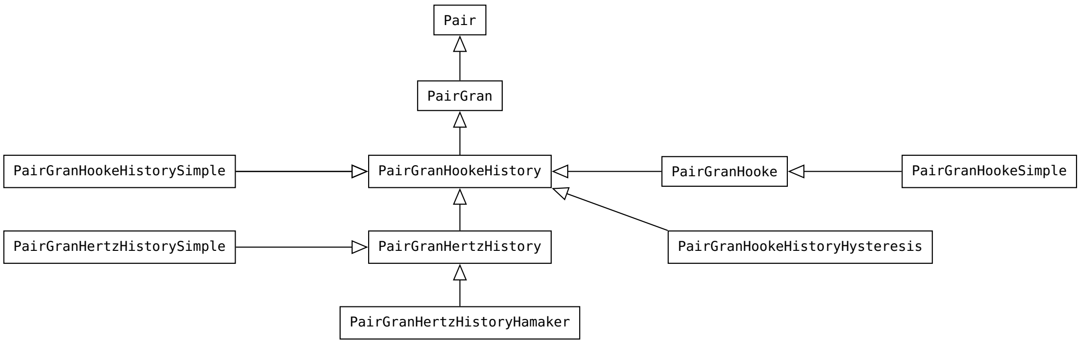

Pair
PairGran
PairGranHookeHistorySimple
PairGranHookeHistory
PairGranHooke
PairGranHookeSimple
PairGranHertzHistorySimple
PairGranHertzHistory
PairGranHookeHistoryHysteresis
PairGranHertzHistoryHamaker

Figure 5.4: Hierarchy of Pair Styles in LIGGGHTS 2.x

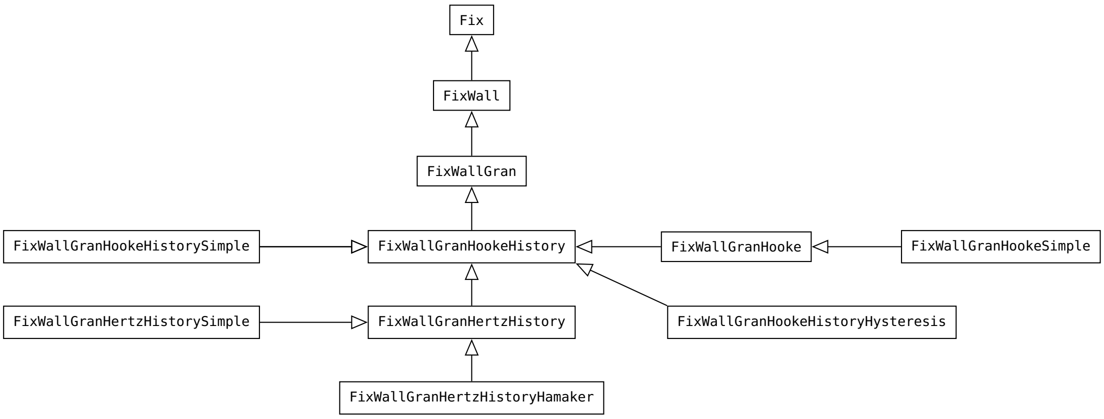

Fix
FixWall
FixWallGran
FixWallGranHookeHistorySimple
FixWallGranHookeHistory
FixWallGranHooke
FixWallGranHookeSimple
FixWallGranHertzHistorySimple
FixWallGranHertzHistory
FixWallGranHookeHistoryHysteresis
FixWallGranHertzHistoryHamaker

Figure 5.5: Structure of granular FixWall Styles in LIGGGHTS 2.x. This list was extended to reflect Pair implementation. In reality some wall implementations never existed even though there was pair style.

base class `Pair`. Force calculations are the responsibility of derived classes, implementing the `compute_force` member function. One of the problems of this design is that although there is a base class `PairGranHookeHistory`, which could implement the i-j-Force loop for all other pair styles, this is not done. Instead code is duplicated to avoid virtual member calls. This leads to multiple copies of the force loop which only differs by a small portion specific for each model.

Since force models are also applied to particle-wall interactions the entire class hierarchy is duplicated for wall fixes as shown in Figure 5.5.

All of these different code variants had to be synchronized. If a rolling friction submodel was added in one pair style, it had to be migrated to all N copies of the force loop of Pair Style implementations. Eventually different versions of the same rolling friction model coexisted in the same code base. Finally, the same changes were necessary for particle-wall interactions.

Chapter 6

New Structure of LIGGGHTS 3.x

This chapter describes the changes that were applied to the granular force kernels in LIGGGHTS, both for particle-particle and particle-wall interactions. These modifications were a prerequisite to work on optimizations such as a new OpenMP parallelization discussed in Chapter 10. Without them code duplication would have made prototyping and maintaining the code base difficult.

Modifying code without changing its meaning is called *refactoring* [24]. It is an essential step each piece of software will eventually go through as requirements change. With testing in place, making such modifications to the existing code base no longer are a fearful activity. Changes which break existing functionality are noticed immediately during testing. The key to success is to run tests as quickly and often as possible.

The original structure of LIGGGHTS 2.x had reached an impasse. Newer models caused code to diverge. Common modifications on all models were time consuming, inefficient and error prone. Working on new kinds of parallelizations and other optimizations in pair styles using this code base was not practical. It would have meant to duplicate pair styles and wall fixes, which in turn were copies of each other.

Both pair and wall computations shared the same physical models, but their implementations lived in separate source files and had diverged over time. One extreme example was the implementation of rolling friction, which was implemented using C++ templates in one instance and **`switch`** statements in a different part of the code.

A better approach was needed. To improve the situation and reduce the amount of code duplication, a new structure for granular force kernels in LIGGGHTS was designed and implemented. This new structure defines a common base for all force kernels and introduces groups of contact models. A pair style or wall fix is then assembled by selecting an implementation of each group. The refactoring of the code eventually led to the release of LIGGGHTS 3.x. It also triggered the development of the test harness to verify version consistency described in Chapter 4.

6.1 Granular Force Kernels

The new structure of granular force kernels is visualized in Figure 6.1. While the base classes such as `PairGran` and `FixWallGran` still exist, they are no longer extended through a web of subclasses to implement force laws. The class hierarchy present in LIGGGHTS 3.x now decouples the force computations from the LAMMPS/LIGGGHTS class hierarchy through the two interfaces `IGranularPairStyle` and `IGranularWall`.

These and all implementation files are now part of a separate LIGGGHTS namespace. Further namespaces are used to group implementations. Both interfaces reduce the required number of methods to the bare minimum necessary to drive a granular computation. This includes passing through settings, initialization and triggering a force computation. Granular force kernels implement this interface and are instantiated through object factories.

The bare essence of the original force kernels was distilled by removing all force computation and model information from the original implementations. While there were multiple variants scattered across the original models, the overall concept has been the same in all of them: Looping over particles or triangles, checking for overlaps with other particles and computing a force when necessary. How this force is computed is of no importance for the driving code. It may be a granular pair style or a wall fix. But for each type, there now is only a single implementation of the necessary boilerplate code.

Listing 6.1 shows pseudo-code for the pair-wise force loop implemented in the `Granular` template inside the `LIGGGHTS::PairStyle` namespace. Its main purpose is to find out which type of interaction should be computed. Either a collision takes place between two neighboring particles or not. Once this distinction is made, any model can be used to compute the resulting forces and torques on the particle pair. The `ContactModel` type of the template specifies a class which operates as a black box. Given sufficient information about what happened with the interaction pair, an object of this class will return forces acting on those particles. Finally, the loop can then decide what should happen with the computed forces.

Listing 6.1: Pair force loop

```
for(all particles i) {
  for(all neighbors j of i) {
    if(collision) {
      // retrieve necessary data & compute collision forces
    } else {
      // retrieve necessary data & compute optional forces
    }
    // apply forces to particle i and j if necessary
  }
}
```

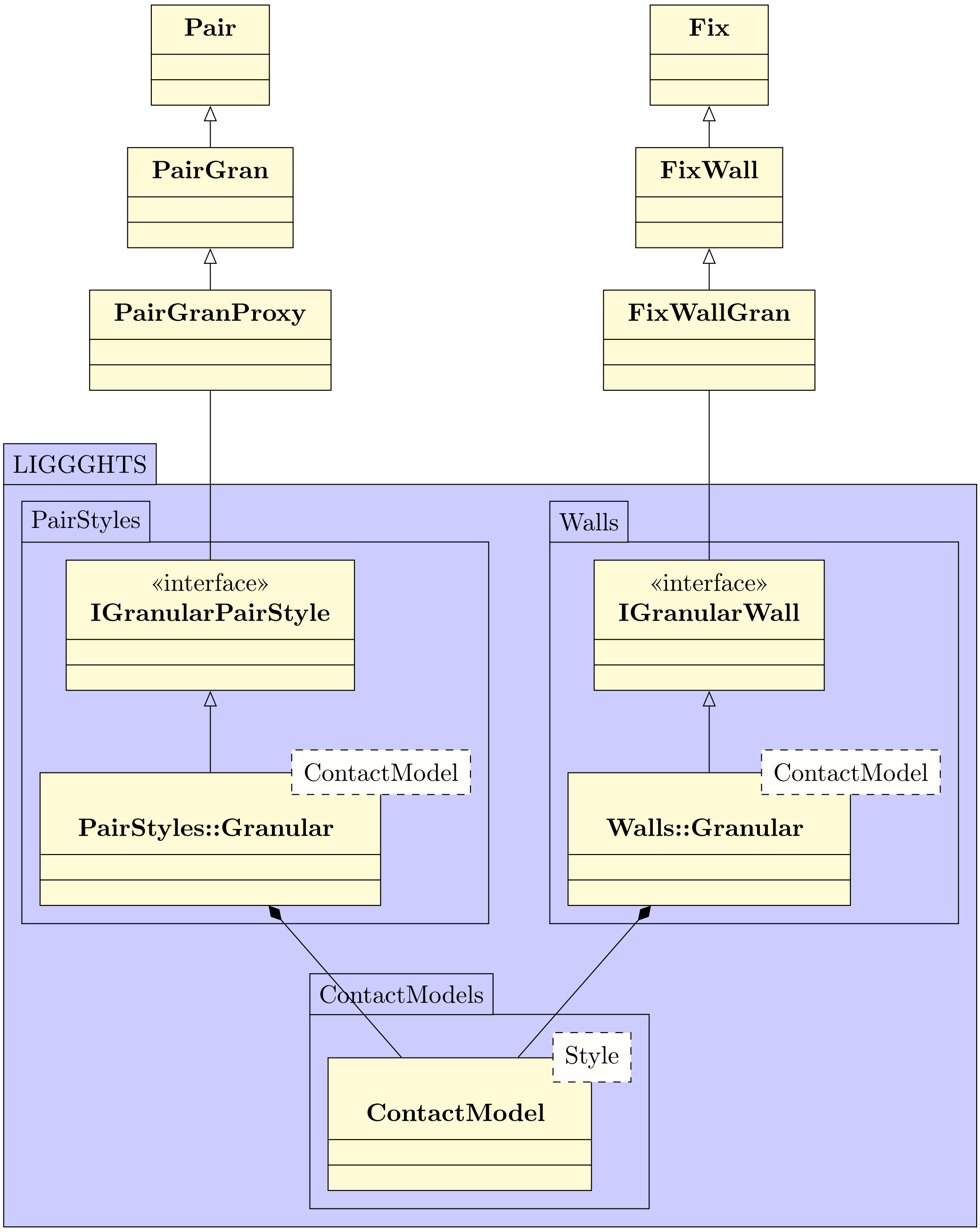


Figure 6.1: New structure of the granular force kernels in LIGGGHTS 3.x. Both pair style and wall kernels are now in a separate namespace and use unified contact models which are instantiated through the C++ template mechanism at compile time.

Particle-wall force loops operate in a similar way. Listing 6.2 is a simplified version of such a particle-wall force loop. Primitive walls and mesh walls share the same overall algorithm. The only difference is that finding all neighbors of mesh walls leads to the traversal of more complicated data structures. But once an interaction pair is identified and its overlap condition evaluated, the same contact model as in the pair styles can be used to determine the forces acting on both particle and wall. Contact models are given the additional information that one of the contact partners is a wall. This can influence force computation[1].

Listing 6.2: Wall force loop

```
for(all neighbors i of wall} {
  if(collision) {
    // retrieve necessary data
    // compute collision forces
  } else {
    // retrieve necessary data
    // compute optional forces
  }
  // apply forces to particle i and wall if necessary
}
```

6.2 Contact Models

Contact models are an abstraction introduced to LIGGGHTS to simplify the development of granular force kernels. It was motivated by the need of replacing the granular pair and wall force computations with alternative implementations for multi-threading. Ideally force computations should be reusable components which can be treated as a black box. If sufficient information is given to this black box, it should return the corresponding forces. Kernels utilizing the force computation can then try to minimize the usage of this computationally expensive component. Internals of the force computation itself must not influence the architecture of the driving code.

At the same time model developers do not have to be cluttered with implementation details of how contact histories are made available across multiple MPI processes or what data structure is chosen to store it. Or how user specified values from the input script get transformed into parameters.

The main challenge is implementing contact models in a way which avoids adding additional overhead and finding practical generalizations [32].

---

[1] Note that the current wall implementation still contains its loops in `FixWallGran`. The instance of `Granular` from the `LIGGGHTS::Walls` namespace currently only does the final translation of the collision event. Primitive walls and mesh walls could have been treated as separate kernels. This would potentially add further optimization opportunities for the compiler. However, this would have caused refactorings in the mesh implementation and its supporting code, which was beyond the scope of this work.

6.2.1 Responsibilities of a contact model

Contact models are implementations of an idealized black box plugged into common force kernels. To truly decouple the two worlds of both driver code and force implementation, the data exchange between them has to be standardized.

Figure 6.2 shows the flow of information in and out of a contact model. Through its lifetime a contact model will go through two stages: Initialization and Computation.

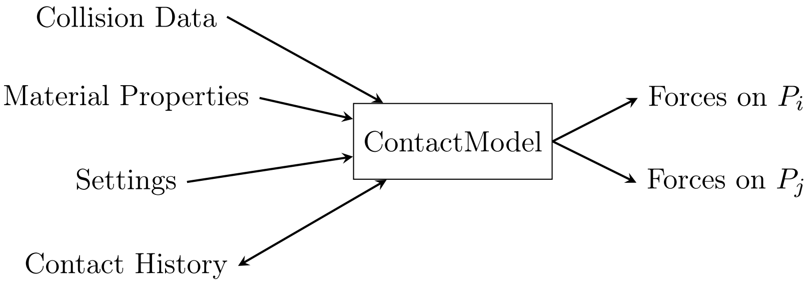


Figure 6.2: Flow of information in and out of contact model

Initialization: During initialization a contact model will be configured through settings, register its need for certain material properties and request additional contact history values. Settings allow tuning the model. May it be through an additional model coefficient or a flag which enables special force terms.

Some force models require users to specify properties for each material type. Retrieval and computation of these properties is a recurring task which is why this has been standardized. Contact models now only specify which properties they need and where they are stored.

Equally challenging is the addition of contact history values. A contact model can now simply request a given amount of history values, without knowing how they are provided.

Computation: Once configured a contact model can be used to compute an arbitrary number of interactions. During computation a contact model is not allowed to change its state. This is because contact models could be used by multiple threads simultaneously. If any computation had side effects on its state, threads would interfere with each other. Instead, any changing information needed must be fed into the model from the outside. This includes information about which contact partners are interacting, their overlap distance, velocities and other properties such as radius, mass and material type. This data will be referred to as *collision information.* Each individual collision event will provide this data to the contact model. If no collision occurs, but particles are still in close proximity to each other, less information is given to the model.

After computation, the forces of both interaction partners are returned. Contact history information is a special case in this regard. It not only enters the contact model as input but also represents one of its outputs.

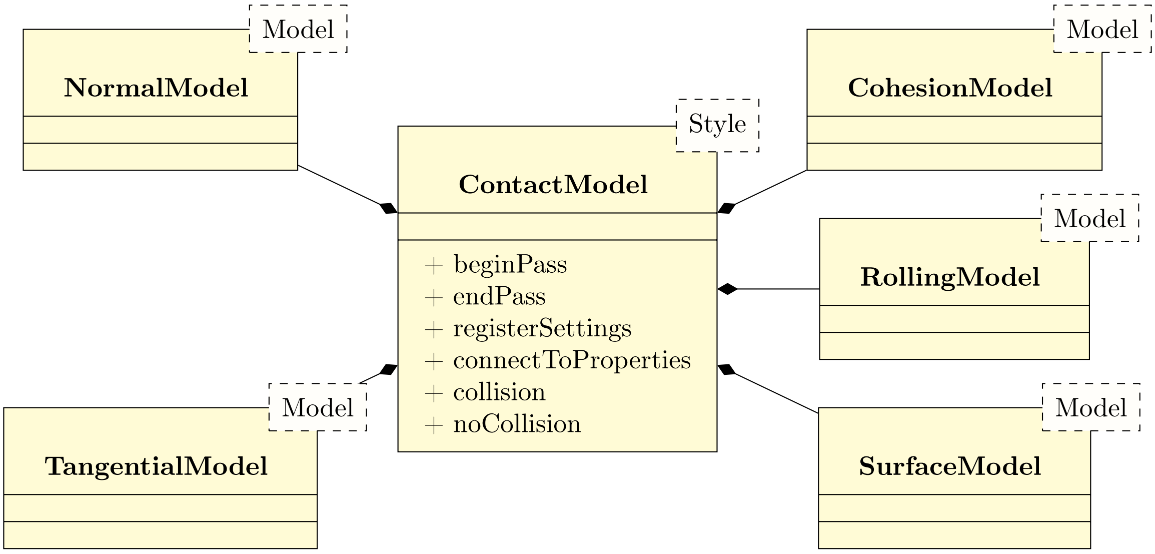


Figure 6.3: A `ContactModel` template is a composition of submodel specializations. Its interface serves a proxy of calling the corresponding submodels in the correct order.

6.2.2 Structure of a contact model

Instead of large computational kernels a contact model splits up the entire computation into several submodels. Each submodel is responsible for one aspect of the force computation. It also enforces a strict execution order between these models. Submodels at a later stage can rely on computed values from previous stages. Currently a contact model consists of a combination of the following submodels:

- normal force model
- tangential force model
- cohesion model
- rolling friction model
- surface model

Unlike a classic object-oriented solution a contact model does not use polymorphism for extension and variability. Instead, it is implemented using the C++ template mechanism. This avoids dynamic function calls in the interior of the computational kernels and allows the compiler to optimize these parts of code more efficiently[2].

A contact model itself is a template which specializes using a `Style` type. This type contains a configuration which selects specializations of all contained submodel templates, such as `NormalModel<T>` and `TangentialModel<T>`. Through this mechanism, all variants of granular force kernels can be generated at compile-time.

---

[2] A polymorphic version was found to be about 15% slower than its templated counter-part.

Figure 6.3 illustrates the interface of a contact model template. Each interface method serves as a proxy for calling the corresponding submodel methods in a predefined order. For model developers this means that they only have to choose which type of model they want to implement and create a new specialization of that class template. The compilation process then takes care of generating contact model variants which include the new submodel.

6.2.3 Force evaluation of a contact model

The force computation itself was simplified and reduced to calling either `collision` or `noCollision` methods for each submodel. A common granular force kernel can make the decision whether particles are in contact. Based on this decision it can then call one of these methods providing enough information about the contact to let the contact model compute and return forces.

`beginPass` and `endPass` mark the beginning and end of the force evaluation during a time step. These two methods allow setting up or clean up resources which are needed across multiple contacts during a pass over all possible contacts.

6.2.4 Settings & Flags

Models use settings to parameterize their internal behavior. During execution these settings and flags do not change. Each submodel is allowed to implement a `registerSettings` method which can then register a set of standardized setting types. Currently this includes scalar doubles, on/off flags and option lists which map a list of strings to an integer value. This gives model developers a standardized approach of defining configuration settings. Listing 6.3 shows an example of how a model specifies a boolean flag which controls tangential damping. By default, this setting is set to **`true`**. Each setting is mapped to a member variable during registration. At run-time parsing the settings will update the member variable when needed.

Listing 6.3: Register `tangential_damping` setting which can be on or off

```
template<>
class NormalModel<HERTZ> : protected Pointers {
  void registerSettings(Settings & settings) {
    settings.registerOnOff("tangential_damping", tangential_damping, true);
  }

private:
  bool tangential_damping;
};
```

6.3 Global Properties

Global properties, such as material parameters, can be accessed and made available inside of the `connectToProperties` method of a contact model. Because these types of properties are defined using the LIGGGHTS input script language, retrieving this information can be challenging. For this reason contact models use an abstraction which simplifies this by accessing scalars, vectors or matrices by name. Once the needed property has been found, it can be mapped and synchronized to class member variables. By doing this, the force computation routines no longer have to contain internals of the LAMMPS/LIGGGHTS framework, but rely on the fact that the needed data is available locally in their member variables.

One key ingredient of both pair styles and wall fixes is that any model implemented relies on a set of parameters. These have to either be specified by the user or must be computed. Some computed quantities require the availability of others which have to be specified by the user. One example for this is the computation of the effective Young's modulus between two particle types. Users must specify both the Young's modulus and Poisson ratio of each material. At run-time a matrix of effective Young's moduli for each material combinations must be computed.

To avoid reimplementation of these common tasks the creation of global properties has been generalized. Such properties can be scalars, vectors or matrices. Properties are managed by an instance of the `PropertyRegistry` class. They are registered by name and by specifying a factory method.

Listing 6.4 shows such a method for the effective Young's modulus. The property uses the property registry to obtain its prerequisites by first registering the Young's modulus and Poisson ratio properties. It then creates a matrix large enough to store all possible material combinations. Using the `VectorProperty` instances of Young's modulus and Poisson ratio the matrix is then populated with the corresponding values.

Listing 6.4: Definition of a global property factory method which computes the effective Young's modulus matrix.

```
namespace MODEL_PARAMS {
  static const char * YOUNGS_MODULUS = "youngsModulus";
  static const char * POISSONS_RATIO = "poissonsRatio";

  MatrixProperty * createYeff(PropertyRegistry & registry) {
    const int max_type = registry.max_type();
    registry.registerProperty(YOUNGS_MODULUS, &createYoungsModulus);
    registry.registerProperty(POISSONS_RATIO, &createPoissonsRatio);

    MatrixProperty * matrix = new MatrixProperty(max_type+1, max_type+1);
    VectorProperty * youngsModulus = registry.getVectorProperty(YOUNGS_MODULUS);
    VectorProperty * poissonRatio = registry.getVectorProperty(POISSONS_RATIO);
    double * Y = youngsModulus->data;
    double * v = poissonRatio->data;

    for(int i=1; i < max_type+1; i++) {
      for(int j=1; j < max_type+1; j++) {
        const double Yi=Y[i];
        const double Yj=Y[j];
        const double vi=v[i];
        const double vj=v[j];
        matrix->data[i][j] = 1./((1.-(vi*vi))/Yi+(1.-(vj*vj))/Yj);
      }
    }
    return matrix;
  }
}
```

Each submodel can specify which properties it needs its own `connectToProperties` method. It is passed a reference to the property registry which manages all instantiated properties. Registration and evaluation of the property do not necessarily occur at once. Instead, each model uses member variables which are then connected to the given property. During force computation the framework makes sure each model sees a current set of property values.

Listing 6.5 shows the registration of the effective Young's modulus property in the Hooke normal model. After registration, the property registered under the name `"Yeff"` is connected to the member variable `Yeff` of the contact model. The framework then ensures that the `Yeff` member variable always points to the current matrix storing the effective Young's moduli.

Listing 6.5: Connecting the contact model member variable Yeff to the global property Yeff which is created using the createYeff factory method.

```
template<>
class NormalModel<HOOKE> : protected Pointers {
  ...
  void connectToProperties(PropertyRegistry & registry) {
    registry.registerProperty("Yeff", &MODEL_PARAMS::createYeff);
    registry.connect("Yeff", Yeff);
    ...
  }

protected:
  double ** Yeff;
};
```

6.3.1 Contact History

Contact models also generalize handling of contact histories. A contact history is an array of values which are stored for each contact pair across time steps. During construction of each submodel an object which implements the `IContactHistorySetup` interface is passed to the constructor. Using this interface each model can request an arbitrary amount of contact history values. Pair styles and walls both implement this interface, but operate differently.

The sum of all contact history values from all submodels determines the total size of a contact pair's contact history. Note that because any combination of submodels is possible, submodels can not assume any order of history values. Instead, during registration of a history value its offset inside the total contact history is returned. By storing and using this offset each model can operate on its local history values and ignore the rest of the contact history. Listing 6.6 shows the construction of the tangential model with shear history. It registers three values storing the amount of shear in each direction. The offset of these history values inside the complete contact history array is stored in the member variable `history_offset`.

Listing 6.6: Constructor of tangential model which adding three contact history values

```
TangentialModel(class LAMMPS * lmp, IContactHistorySetup * hsetup) {
  history_offset = hsetup->add_history_value("shearx", "1");
  hsetup->add_history_value("sheary", "1");
  hsetup->add_history_value("shearz", "1");
}
```

During `collision` and `noCollision` events the contact history of the current contact pair is passed as part of the collision data structure. Each submodel can access its values by using its offset value. This is shown in an excerpt of the `TangentialModel` with shear history in Listing 6.7. This code snippet also demonstrates another important aspect of contact history managment. The lifetime of history values is controlled by submodels using the `touch` flag. This bitmask indicates if a submodel requests that the history is maintained. How long such values are retained depends on each submodel. E.g., values could be stored as long as contact partners are overlapping and discarded afterwards. Other models might keep history values beyond this point while particles are still in each other's neighbor list.

Listing 6.7: Usage of contact history in TangentialModel

```
void TangentialModel<HISTORY>::collision(CollisionData & cdata,
  ForceData & i_forces, ForceData & j_forces) {
  ...
  double * const shear = &cdata.contact_history[history_offset];
  // shear[0] = shearx
  // shear[1] = sheary
  // shear[2] = shearz
  if(cdata.touch) *cdata.touch |= TOUCH_TANGENTIAL_MODEL;
  ...
}

void TangentialModel<HISTORY>::noCollision(ContactData & cdata,
  ForceData&, ForceData&) {
  double * const shear = &cdata.contact_history[history_offset];
  shear[0] = 0.0;
  shear[1] = 0.0;
  shear[2] = 0.0;
  if(cdata.touch) *cdata.touch &= ~TOUCH_TANGENTIAL_MODEL;
}
```

In case of the tangential model with shear history its values are retained while particles are in contact. Once the contact no longer exists the history values are marked for deletion by unsetting the bit. Additionally, the history values are reset to zero.

Other models, such as cohesion models, might continue retaining history values even after contact partners are no longer in contact.

6.4 Compile-time Assembly of Contact Model Variants

The addition of contact models in the LIGGGHTS code base includes changes to the compilation process of LIGGGHTS. So far LIGGGHTS only generated style header files prior to compiling the source code. This mechanism is reused to generate style files for each contact model submodel. E.g. headers with a `normal_model_` prefix are collected to a `style_normal_models.h` header file.

Using these lists of submodels a new step is placed prior to compilation. This new precompile step generates all permutations of contact models with their submodels. With 6 normal models, 2 tangential models, 5 cohesion models, 4 rolling friction models and 2 surface model variants, there are now 480 different combinations. Figure 6.4 illustrates all possible variations of contact models currently available in LIGGGHTS 3.x.

The mechanism behind the precompile step is a script which generates an additional header file called `style_contact_models.h`. In this file all combinations of submodels are placed using a generic `GRAN_MODEL` macro. An example of this is shown in Listing 6.8.

Listing 6.8: Contents of generated `style_contact_models.h` file

```
GRAN_MODEL(HERTZ, TANGENTIAL_HISTORY, COHESION_OFF, ROLLING_OFF, SURFACE_DEFAULT)
GRAN_MODEL(HERTZ, TANGENTIAL_HISTORY, COHESION_OFF, ROLLING_CDT, SURFACE_DEFAULT)
GRAN_MODEL(HERTZ, TANGENTIAL_HISTORY, COHESION_OFF, ROLLING_EPSD, SURFACE_DEFAULT)
GRAN_MODEL(HERTZ, TANGENTIAL_HISTORY, COHESION_OFF, ROLLING_EPSD2, SURFACE_DEFAULT)
...
```

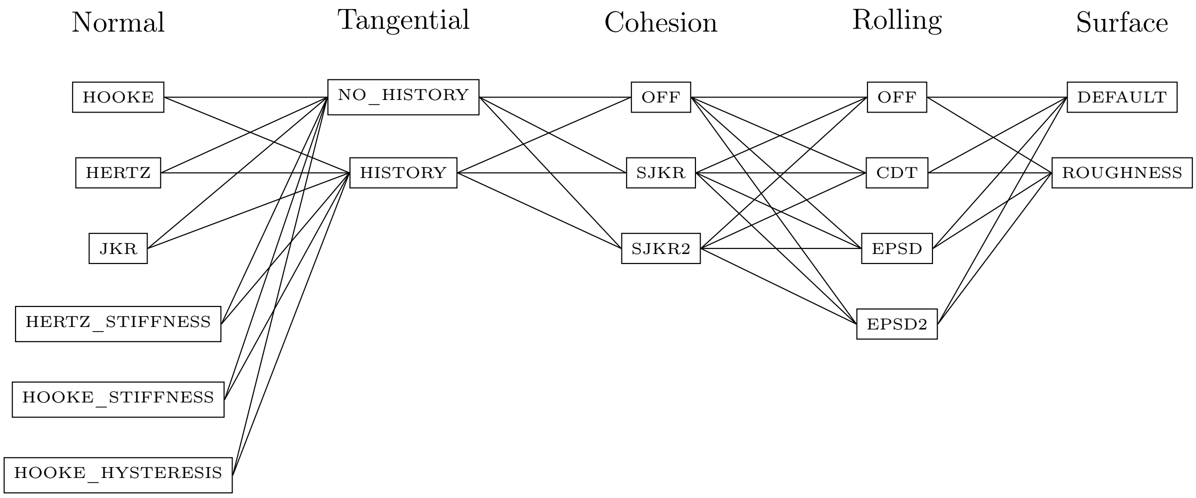


Figure 6.4: Contact model combinations in LIGGGHTS 3.1.0 (PFM version)

During compilation this header is then used to define this macro in such a way that C++ templates get instantiated with each combination and registering constructors to an object factory.

The downside of this approach is that with each new model the number of possible combinations grows and compile-time increases because the compiler has to instantiate more templates. A major benefit however is that it shifts the mechanic and cumbersome task of maintaining multiple combinations of submodels to the compiler. At the same time, because each model is statically linked and can be individually optimized by the compiler, in general better object code can be generated.

A similar argument can be given concerning the code driving the force kernels. Adding an alternative granular force loop is now much easier while maintaining the benefits of static linking and inlining. But at the same time it doubles the amount of compilation because each model is recompiled individually with alternative implementations. Overall it is a trade-off of compilation time, maintainability and efficiency of the code.

Part II

Efficiency

Chapter 7

Measuring performance and efficiency

A main objective of this work was the identification of bottlenecks in the existing code base and finding new algorithms for existing problems. Optimization itself is a last step in a sequence of engineering steps. It requires the minimization of a cost function which quantifies the objective which has to be met. To maximize the performance of the code means reducing the total amount of time needed for solving a given problem. Improving the efficiency of a code means maximizing the utilization of the available computational resources. This chapter describes the background and techniques to measure these two quantities.

7.1 Single Processors

Prior to any optimization a measure of performance has to be defined. A first choice can be the total execution time of a code for performing a certain task. Reducing the execution time is a good indicator that algorithms require less work or utilize the hardware more efficiently. However, this measure alone is only a relative measure for efficiency.

If the original code was very inefficient and was using an unsuitable algorithm, a better algorithm can significantly decrease the total execution time by orders of magnitude while still being inefficient. The execution time alone does not contain information about how well the hardware was utilized. For this reason additional indicators are needed to obtain a more complete picture of how the hardware was used.

Modern hardware has made measuring performance and efficiency of code challenging. In the early days of computing programmers could rely on a simple mental model of how computers operated based on the Von-Neumann architecture [40]. Instructions and data were fetched from memory, executed one by one and results written back to memory. Hardware manufactures and compiler vendors have worked hard to keep this model in place. It is sometimes referred to as a guarantee of sequential consistency. But in reality modern architectures have departed from this type of model long ago [41].

For many years programmers could rely on a steady increase in computing performance by upgrading hardware. A computation on a 400 MHz CPU would become twice as fast on an

800 MHz CPU. Twice as many instructions were processed in the same amount of time. As the speeds of sequential CPUs increased into the GHz range, the gap between CPU and memory access times became larger. Nowadays instructions are processed multiple orders of magnitude faster than memory accesses to RAM. This development gave rise to many attempts to hide the increasing memory latency. This includes pipelining, instruction-level parallelism (ILP) and memory caches [42].

Pipelining allows CPUs to execute multiple instructions at once by decomposing each instruction into multiple phases. At each point in time, a CPU will then not execute a single instruction, but have multiple instructions in flight concurrently. If instructions do not depend on each other, they can be scheduled in a way to optimize utilization of hardware resources. E.g., while one instruction is being decoded, another instruction could be using the floating-point unit to perform a calculation. Pipelining increases the throughput of a CPU and is invisible to programmers. Hardware designs guarantee that whatever comes out of these pipelines is semantically equivalent to what the original instruction sequence intended. Architectures with 15+ stages are common in today's CPUs which are known as super-scalar processors.

To further increase throughput and mitigate the cost of memory accesses CPUs gained larger registers which could be operated on using Single-Instruction Multiple Data (SIMD) instructions. CPUs can load larger chunks of memory into a register, e.g. four double precision floating-point numbers, and execute a single instruction on all four floating point numbers at once. Over time these registers have even become larger in size and offer a broad range of functionality for special applications. These special instructions include Multimedia Extensions (MMX), Streaming Extensions SSE, SSE2, and lately Advanced Vector Extensions AVX and AVX-2. Code which uses SIMD is considered to be *vectorized.*

Concurrently, to bridge the gap between memory and CPU register latency, chip designers started adding faster on-chip memory to cache portions of memory that were used frequently. CPUs nowadays typically contain an instruction cache and a data cache. Multiple levels of caches were introduced to optimize cost and access patterns to memory. The closest and fastest cache are the L1d and L1i caches which contain only a few kilobytes. They are backed up by a larger L2 cache of several hundred kilobytes. Finally, many modern CPUs have an additional L3 cache which is several megabytes in size. The intent of caches is to keep the working set of a program as close as possible to the processing units. Programs therefore need to be written in a cache aware fashion to leverage the benefits.

All of these developments make reasoning about the performance and efficiency of sequential programs more difficult. The efficiency of a CPU pipeline is sensitive to code with branches. Instructions in-flight might be discarded if an earlier instruction completes and makes prefetched branch instructions obsolete. The highest throughput of a CPU can only be achieved if vectorized code is used to fully utilize available computing resources such as highly optimized floating-point units. Finally, performance is depended on how well caches are utilized to avoid stalling CPU instructions waiting for data.

7.2 Multi-threading and Multi-core

As sequential performance increased, hardware designers eventually reached what is known as the power wall. While transistor sizes continued to decrease and frequencies were getting higher (≈4 GHz), chip designers reached physical limitations. Due to shrinking of transistors, leak currents and heat generation became a major challenge. To satisfy the increasing demand for computational power the industry started producing processors which could deal with multiple instruction streams at the same time. Thus, hardware with multi-threading was born.

Multi-threading is enabled by duplicating some of a processor's resources such as registers and instruction pipelines. This allows multiple streams of instructions to be scheduled and executed independently, while other resources such as floating-point units might be shared. A main limitation of multi-threading is that the memory subsystem is a shared resource. Memory controllers and therefore their bandwidth is shared. This makes multi-threading ideal for applications where multiple threads operate on small chunks of shared memory which can be placed in caches. It is, however, not suitable if each thread operates on its own memory region because this reduces the effective memory bandwidth available to a thread. It also introduces the risk of threads disturbing each other by removing portions of memory from the cache although it still might have been useful for another thread. This situation where threads continuously invalidate the cache of each other is commonly known as "trashing the cache".

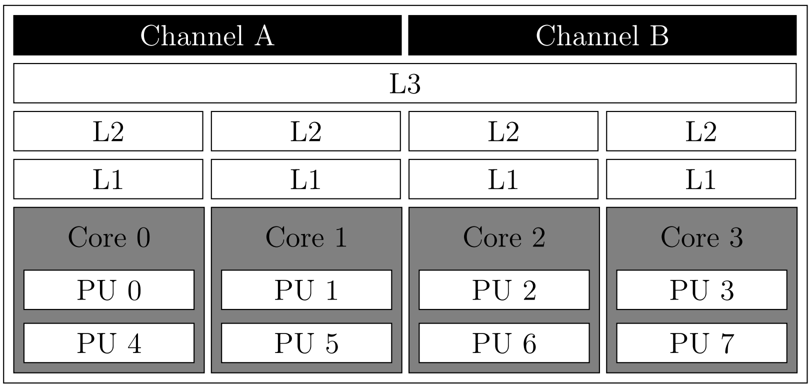


Figure 7.1: Block diagram of Intel Core i7-3770 (Ivy Bridge) processor. This model can use up to two memory channels, a 8MB L3 cache which is shared among all cores and a 256KB L2 cache and 32KB L1 cache per core. Each core has two processing units (PUs). If hyper-threading is enabled, one core can run two threads simultaneously.

Eventually processor vendors started adding multiple full-fetched processors onto a single chip. These have become known as multi-core processors. Figure 7.1 illustrates the topology of a current Intel Core i7-3770 processor. Each core can fully operate independently. Due to the guarantees of sequential consistency, additional complexity in the memory subsystem were necessary. While each processor can work independently on chunks of memory, CPU vendors must ensure that each processor core sees a valid state of memory at all times. If core A updates a memory region which is cached by core B, core B must be notified that it has to update its cached data if it uses it. This problem is solved by processor vendors by enforcing protocols for memory accesses and becomes non trivial as the number of processors on the same chip increases. But in general one can state that any updates across processor core caches is burdened with additional overhead which can reduce throughput.

In memory-bound applications, memory access is the key bottleneck which limits application performance. How applications access memory greatly influences the overall throughput achieved. In early architectures uniform memory access (UMA) was common. Any instruction accessing memory would require a certain uniform amount of time. Today's architectures, however, are very sensitive to data locality.

One of the reasons for this is the introduction of caches. Through caches access times to frequent memory locations were dramatically reduced. In multi-threaded and multi-core architectures memory cohesion protocols further influence access times since data occasionally has to be kept in sync between caches.

Another reason is that due to the added complexity of multi-cores and the memory bandwidth limitations, processor vendors started integration of memory controllers directly into processors. If such a processor contains more than one memory controller, they are usually only accessible by a subset of processor cores. Processor cores accessing memory which is not attached to their memory controller involves a more time consuming interaction with another core's memory subsystem. This leads to the effect that certain memory locations can be accessed more quickly than others depending on which core code runs. Architecture with this type of behavior are said to have non-uniform memory access (NUMA).

This type of behavior is even more dominant if systems use multiple sockets. Additional latency is introduced because each socket only accesses a subset of all memory banks. To access the remaining memory banks associated with another socket processors must communicate through a processor interconnect. This is visualized in Figure 7.2.

In summary, many factors influence how fast and efficient code runs on a given architecture. Measuring total execution time alone only gives a very rudimentary picture. To gain a deeper insight of how an application is performing additional information about its execution needs to be gathered.

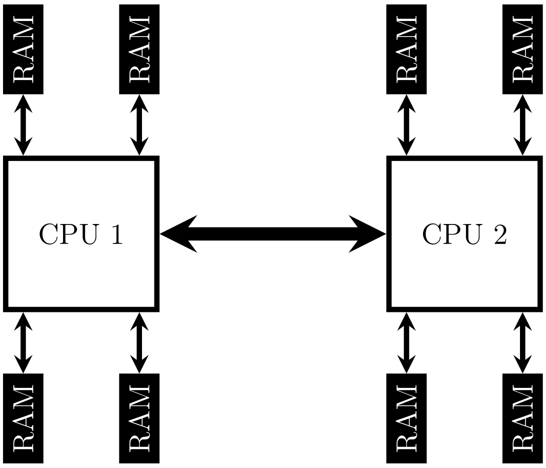


Figure 7.2: Dual Socket Configuration. Each CPU uses its own memory controllers. E.g., a AMD Opteron 6272 has two memory controllers each supporting up to two memory channels. If one such processor is in one socket and needs to access memory connected to another processor in a second socket, communication over the processor interconnect is necessary.

7.3 Profiling applications

Performance information can be gathered by running applications with profiling tools. Profiling is a special form of execution which can be intrusive and non-intrusive to the program under investigation. The following sections introduce different profiling tools used to gather information about serial programs as well as MPI and multi-threaded programs.

7.3.1 Instrumentation

Intrusive forms of profiling involve manipulating the program code prior to execution. The simplest form is manual instrumentation of code, by adding time measurement to certain code sections and storing these performance metrics for later use. Listing 7.1 shows the general form of such a manual instrumentation.

Listing 7.1: Manual instrumentation using C++11 chrono library

```cpp
#include <chrono>
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
  auto start = chrono::system_clock::now();

  // EXECUTE CODE UNDER TEST
  ...

  auto end = chrono::system_clock::now();
  chrono::duration<double> elapsed_seconds = end - start;
  cout << "duration:" << elapsed_seconds.count() << endl;

  return 0;
}
```

If time is measured by accumulating very short time intervals, the resolution of the used timer can become a limiting factor. Regular time functions such as the system clock only have a resolution of a few milliseconds. To still measure accurately, higher resolution timers have to be used. In C++11 the chrono library offers `chrono::high_resolution_clock` which represents the clock with the smallest tick rate provided by a C++ implementation. On Linux and GCC this can offer a timer with a resolution up to a few nanoseconds.

A more general approach is automatic instrumentation of code done by compilers. One example is the `-pg` option in the GCC compiler which adds gprof profiling support to the compiled binary. This adds special code to functions and basic blocks of programs, allowing collecting metrics about how much time was spent in functions and which functions were called. This type of profiling is useful to spot expensive regions of computation and even help detect bugs.

The main downside of instrumentation is that it adds a significant overhead to an existing code. While instrumentation on a function level can be useful, instrumenting each individual instruction quickly becomes unfeasible and will likely distort the performance results due to the high overhead.

7.3.2 Sampling

An alternative to the intrusive and expensive profiling using instrumentation is sampling. This non-intrusive method does not modify the binary code but instead monitors its execution.

This type of profiling periodically gathers information from special CPU registers called performance counters. These give detailed information about a CPU's pipeline and memory subsystem. Based on this information and other registers, such as the program counter, these profilers generate statistics about application performance. Interpretation of these performance counters is crucial to understanding the strengths and weaknesses of a code.

7.4 Case Study: Profiling packed insertion of LIGGGHTS

To illustrate the application of sampling profilers a small serial LIGGGHTS benchmark is investigated using the Linux performance tools `perf`. In this benchmark particles with a diameter of 5mm are inserted at random locations in a cubic domain with an extent of 50cm in each direction. The random number generator in all simulations uses the same seed to be reproducable. All insertions occur during a single time step, after which the benchmark concludes. To perform these insertions we use a fix called insert/pack. The goal of the benchmark is to evaluate the performance of inserting an increasing amount of particles into the domain.

Profiling this benchmark and obtaining statistics at the end is done using the `perf stat` command. By default, this command produces statistics about many important characteristic performance counters such as cycle count, page-faults, number of instructions executed and total execution time. The total list of available hardware and software counters is available through the `perf list` command. The desired counters can be selected using the -e option in `perf` commands. In the example seen in Listing 7.2 we focus on efficiency based on the total number of instructions needed to accomplish the task.

The result of one such benchmark simulation using `perf stat` is summarized in Table 7.1. It shows that for inserting 150,000 particles in a single time step the LIGGGHTS implementation needs 265 billion instructions and finishes within about 30 seconds. Further simulations with different numbers of particle insertions allow us to create Figure 7.3 which illustrates the $O(n^2)$ run-time complexity of the insertion algorithm.

To pinpoint the location of the code which is responsible for this behavior we need more detailed statistics on a function level. Instead of creating global statistics of a program run, Listing 7.2: Collecting run-time statistics from hardware counters using `perf stat`

```
# use default list of hardware counters
perf stat liggghts -in in.insertPack -var NPARTICLES 150000

# only collect number of instructions
perf stat -e instructions liggghts -in in.insertPack -var NPARTICLES 150000
```

31333.885661	task-clock (msec)	0.999 CPUs utilized
400	context-switches	0.013 K/sec
64	cpu-migrations	0.002 K/sec
43,350	page-faults	0.001 M/sec
117,702,157,528	cycles	3.756 GHz
48,162,543,735	stalled-cycles-frontend	40.92% frontend cycles idle
0	stalled-cycles-backend	0.00% backend cycles idle
265,545,421,703	instructions	2.26 insns per cycle
		0.18 stalled cycles per insn
19,412,819,023	branches	619.547 M/sec
8,559,475	branch-misses	0.04% of all branches
31.373257833	seconds time elapsed	

Table 7.1: Performance statistics for 'liggghts -in in.insertPack -var NPARTICLES 150000'

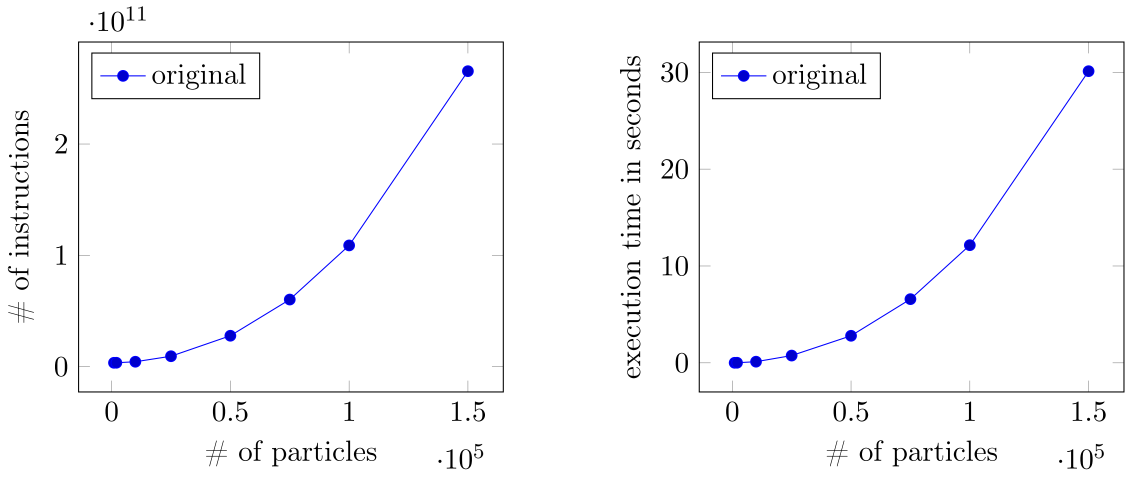
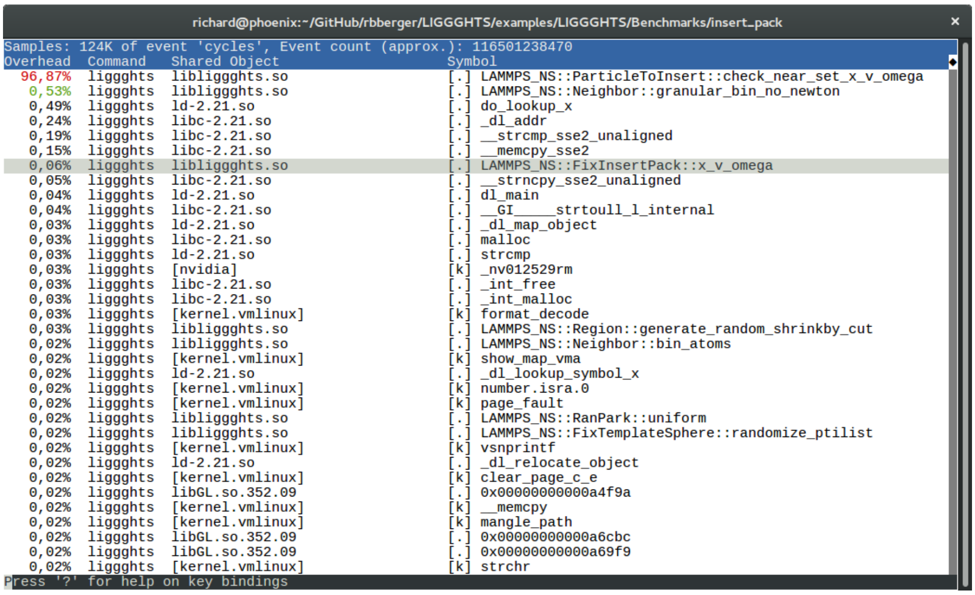


Figure 7.3: Performance of insert/pack with increasing particle number. Execution time is proportional to the number of instructions executed. It shows that the original insertion method has quadratic run-time complexity $O(n^2)$.


richard@phoenix:~/GitHub/rbberger/LIGGGHTS/examples/LIGGGHTS/Benchmarks/insert_pack
×

```
Samples: 124K of event 'cycles', Event count (approx.): 116501238470
Overhead  Command   Shared Object                    Symbol
  96,87%  liggghts  libliggghts.so                   [.] LAMMPS_NS::ParticleToInsert::check_near_set_x_v_omega
   0,53%  liggghts  libliggghts.so                   [.] LAMMPS_NS::Neighbor::granular_bin_no_newton
   0,49%  liggghts  ld-2.21.so                       [.] do_lookup_x
   0,24%  liggghts  libc-2.21.so                     [.] _dl_addr
   0,19%  liggghts  libc-2.21.so                     [.] __strcmp_sse2_unaligned
   0,15%  liggghts  libc-2.21.so                     [.] __memcpy_sse2
   0,06%  liggghts  libliggghts.so                   [.] LAMMPS_NS::FixInsertPack::x_v_omega
   0,05%  liggghts  libc-2.21.so                     [.] __strncpy_sse2_unaligned
   0,04%  liggghts  ld-2.21.so                       [.] dl_main
   0,04%  liggghts  libc-2.21.so                     [.] __GI_____strtoull_l_internal
   0,03%  liggghts  ld-2.21.so                       [.] _dl_map_object
   0,03%  liggghts  libc-2.21.so                     [.] malloc
   0,03%  liggghts  ld-2.21.so                       [.] strcmp
   0,03%  liggghts  [nvidia]                         [k] _nv012529rm
   0,03%  liggghts  libc-2.21.so                     [.] _int_free
   0,03%  liggghts  libc-2.21.so                     [.] _int_malloc
   0,03%  liggghts  [kernel.vmlinux]                 [k] format_decode
   0,03%  liggghts  libliggghts.so                   [.] LAMMPS_NS::Region::generate_random_shrinkby_cut
   0,02%  liggghts  libliggghts.so                   [.] LAMMPS_NS::Neighbor::bin_atoms
   0,02%  liggghts  [kernel.vmlinux]                 [k] show_map_vma
   0,02%  liggghts  ld-2.21.so                       [.] _dl_lookup_symbol_x
   0,02%  liggghts  [kernel.vmlinux]                 [k] number.isra.0
   0,02%  liggghts  [kernel.vmlinux]                 [k] page_fault
   0,02%  liggghts  libliggghts.so                   [.] LAMMPS_NS::RanPark::uniform
   0,02%  liggghts  libliggghts.so                   [.] LAMMPS_NS::FixTemplateSphere::randomize_ptilist
   0,02%  liggghts  [kernel.vmlinux]                 [k] vsnprintf
   0,02%  liggghts  ld-2.21.so                       [.] _dl_relocate_object
   0,02%  liggghts  [kernel.vmlinux]                 [k] clear_page_c_e
   0,02%  liggghts  libGL.so.352.09                  [.] 0x00000000000a4f9a
   0,02%  liggghts  [kernel.vmlinux]                 [k] __memcpy
   0,02%  liggghts  [kernel.vmlinux]                 [k] mangle_path
   0,02%  liggghts  libGL.so.352.09                  [.] 0x00000000000a6cbc
   0,02%  liggghts  libGL.so.352.09                  [.] 0x00000000000a69f9
   0,02%  liggghts  [kernel.vmlinux]                 [k] strchr
Press '?' for help on key bindings
```

Figure 7.4: Profiling with perf record & report shows that 97% of all CPU cycles are spent inside of the `check_near_set_x_v_omega` routine of the `ParticleToInsert` class.

perf also allows capturing events with a granularity of individual instructions and functions. `perf record`, by default, captures how many cycles are spent in each instruction on average.

`perf report` then allows to navigate this recorded data, highlighting which functions contained the most of the interesting quantity. In our case, the number of CPU cycles. Listing 7.3 demonstrates the usage of these utilities. Figure 7.4 shows the text-based user interface of the `perf report` utility. It highlights that during a 150,000 particle insertion, one single method is responsible for 97% of all CPU cycles. Section 8.3 will explain where this quadratic run-time complexity comes from and shows how this method and the insertions in general were optimized to obtain a complexity of $O(n)$.

Listing 7.3: Capturing performance counter data on the instruction level using `perf record` and visualizing it with `perf report`.

```
perf record liggghts -in in.insertPack -var NPARTICLES 150000
...
perf report
```

After optimization and validation of the results, repeating all benchmark simulations with the optimized version produces performance data which prove the linear run-time complexity of the new algorithm. Figure 7.5 contains plots similar to Figure 7.3 and additionaly combines the plots to illustrate the dramatic increase of efficiency. In the given benchmarks the optimized version is in the end nearly two orders of magnitude faster than the original version. This is reflected in the number of instructions needed to accomplish the same task: 4.7 billion instead of 265 billion instructions.

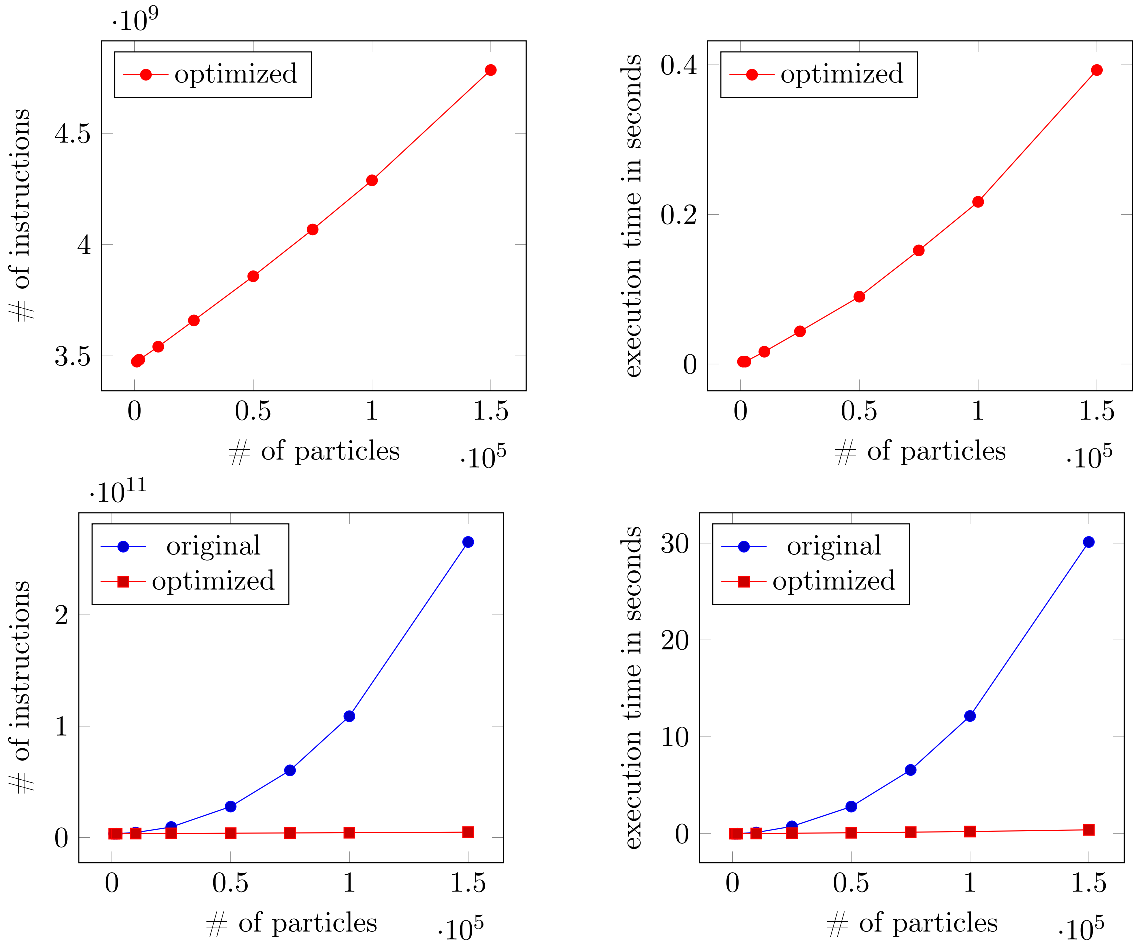


Figure 7.5: Performance of insert/pack with increasing particle number after optimization. It shows that the new insertion algorithm leads to nearly linear performance and is orders of magnitude faster than the original implementation with rising particle counts.

7.5 Case Study: Investigating MPI bottlenecks using HPCtoolkit

The previous case study showed how serial applications can be investigated using off-the-shelf profiling tools like perf. In general these programs attach to their child processes and monitor them by sampling certain events or counters. As the number of processes increases and even threads might influence a program, these profilers are no longer suitable to manage the increasing complexity of multi-process/multi-threaded applications.

Many commercial toolchains contain more advanced profilers for multi-threaded applications. E.g., Microsoft Visual Studio contains profilers specifically targeted for detecting multi-threading issues, such as race conditions. For MPI applications the Intel Software stack allows profiling many-process applications using Intel vTune and gives detailed information if the Intel MPI implementation is used.

On Linux several groups have developed toolkits for HPC profiling. One of them are the Tuning and Analysis Utilities (TAU) [43]. Another toolkit of this kind is HPCtoolkit [44]. It combines profiling and tracing of mixed MPI and threaded applications and uses static code analysis to reconstruct execution information even from optimized binaries. Without such a reconstruction it would be difficult to pinpoint the code origin of instructions.

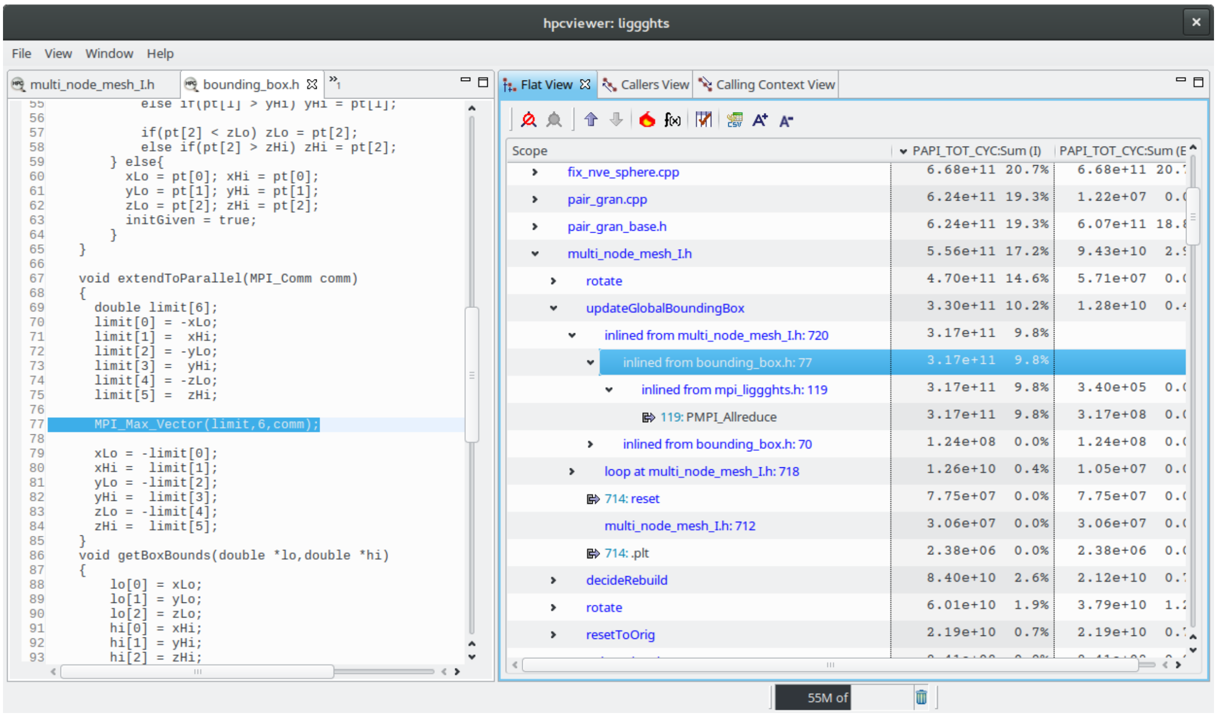
Using HPCtoolkit involves multiple steps which might slightly differ depending on the type of application that is being investigated. In a first step the application is run using a wrapper, similar to a perf execution. For MPI programs each rank launches its own copy of the `hpcrun` wrapper. The options specified to this program are a list of hardware counters or events which should be captured. To strike a balance between performance and accuracy the sampling frequences of each quantity can be specified. A successful run will produce a folder which contains files storing the captured data.

Prior to viewing the data the `hpcstruct` utility is used to analyze the binaries and shared libraries of the application. Optimizations such as inlining and instruction reordering make it difficult to determine which part of the machine code came from which line of code. Through static code analysis the relationship between instructions and lines of code can be reconstructed. For this reason the source code directory is one of the parameters given to the utility. It produces a file with a `.hpcstruct` extension for each binary or library analyzed.

Together with the captured raw data, these analysis results are then merged using the `hpcprof` utility into a so-called experiment database. HPCtoolkit allows combining multiple runs into a single database, thus allowing to compare quantities of different executions.

Experiments can then be viewed using one of the two graphical user interfaces: `hpcviewer` or `hpctraceviewer`. The latter can only be used if recordings were made with the trace option enabled. This is slower and more demanding than sampling-only because the profiler will then also capture information of the current call-stack during a sample.

The value of this kind of profiling and trace tool will be illustrated by an example in the LIGGGHTS code base. It was used to study the MPI scalability of the mesh implementation in



```
            else if(pt[1] > yHi) yHi = pt[1];

            if(pt[2] < zLo) zLo = pt[2];
            else if(pt[2] > zHi) zHi = pt[2];
        } else{
          xLo = pt[0]; xHi = pt[0];
          yLo = pt[1]; yHi = pt[1];
          zLo = pt[2]; zHi = pt[2];
          initGiven = true;
        }
    }

    void extendToParallel(MPI_Comm comm)
    {
      double limit[6];
      limit[0] = -xLo;
      limit[1] =  xHi;
      limit[2] = -yLo;
      limit[3] =  yHi;
      limit[4] = -zLo;
      limit[5] =  zHi;

      MPI_Max_Vector(limit,6,comm);

      xLo = -limit[0];
      xHi =  limit[1];
      yLo = -limit[2];
      yHi =  limit[3];
      zLo = -limit[4];
      zHi =  limit[5];
    }
    void getBoxBounds(double *lo,double *hi)
    {
        lo[0] = xLo;
        lo[1] = yLo;
        lo[2] = zLo;
        hi[0] = xHi;
        hi[1] = yHi;
        hi[2] = zHi;
```

Scope	PAPI_TOT_CYC:Sum (I)		PAPI_TOT_CYC:Sum (E	
fix_nve_sphere.cpp	6.68e+11	20.7%	6.68e+11	20.
pair_gran.cpp	6.24e+11	19.3%	1.22e+07	0.
pair_gran_base.h	6.24e+11	19.3%	6.07e+11	18.
multi_node_mesh_I.h	5.56e+11	17.2%	9.43e+10	2.
rotate	4.70e+11	14.6%	5.71e+07	0.
updateGlobalBoundingBox	3.30e+11	10.2%	1.28e+10	0.
inlined from multi_node_mesh_I.h: 720	3.17e+11	9.8%		
inlined from bounding_box.h: 77	3.17e+11	9.8%		
inlined from mpi_liggghts.h: 119	3.17e+11	9.8%	3.40e+05	0.
119: PMPI_Allreduce	3.17e+11	9.8%	3.17e+08	0.
inlined from bounding_box.h: 70	1.24e+08	0.0%	1.24e+08	0.
loop at multi_node_mesh_I.h: 718	1.26e+10	0.4%	1.05e+07	0.
714: reset	7.75e+07	0.0%	7.75e+07	0.
multi_node_mesh_I.h: 712	3.06e+07	0.0%	3.06e+07	0.
714: .plt	2.38e+06	0.0%	2.38e+06	0.
decideRebuild	8.40e+10	2.6%	2.12e+10	0.
rotate	6.01e+10	1.9%	3.79e+10	1.
resetToOrig	2.19e+10	0.7%	2.19e+10	0.

Figure 7.6: HPC Viewer

LIGGGHTS[1].

A profiling run with tracing enabled allows to visualize large MPI runs such as 32 MPI processes on a single 32-core blade. For clarity and space reasons, this example will however only show a 4-core run on a desktop machine.

The methods responsible for the bad scaling were detected by sampling the total number of cycles executed. A sample period of 340100 was chosen on a system with 3.4GHz. This produces about 1000 samples / second. It was executed with the trace option enabled to also capture the call stack.

The binary and shared library were compiled with a release configuration, which means compilation with -O2 optimizations enabled. Additionally, debugging symbols were added using the -g option for better results during code reconstruction. `hpcstruct` was used to analyze both the `liggghts` binary and the `libliggghts.so` shared library which contains most of the code.

Finally, the results of these steps were merged using `hpcprof`. The resulting experiment database could then be investigated with `hpcviewer` and `hpctraceviewer`.

Figure 7.6 shows a screenshot of `hpcviewer`. It provides similar information as `perf report` on a serial program. But it also allows comparing multiple executions at the same time. An optimization can be evaluated by comparing quantities like spent instructions in functions.

The viewer allows navigating the experiment results from the module level to the function level and down to the instruction level. In our case we investigated the reason for the large amount of time spent in the mesh code. After navigating the call hierarchy, the tool showed that

---

[1] While working on the OpenMP implementation, the bad scaling eventually limited any further progress.

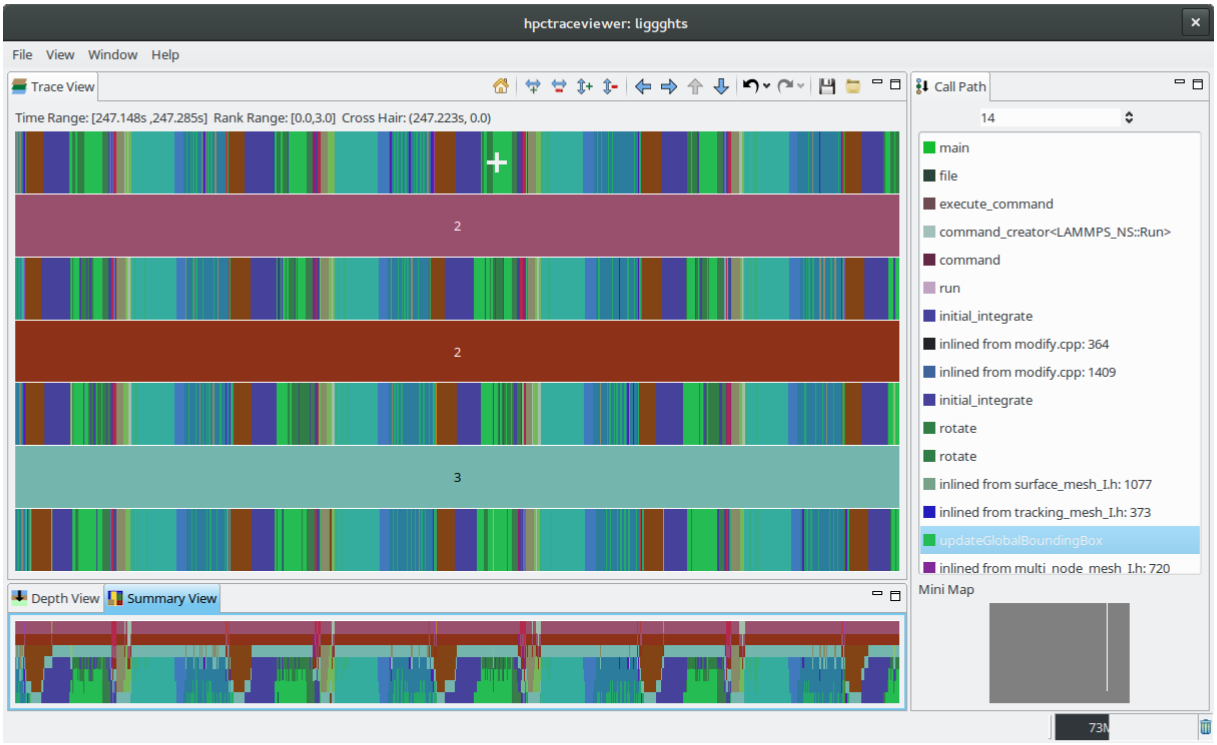

hpctraceviewer: liggghts
File View Window Help
Trace View
Time Range: [247.148s ,247.285s] Rank Range: [0.0,3.0] Cross Hair: (247.223s, 0.0)
2
2
3
Depth View
Summary View
Call Path
14
main
file
execute_command
command_creator<LAMMPS_NS::Run>
command
run
initial_integrate
inlined from modify.cpp: 364
inlined from modify.cpp: 1409
initial_integrate
rotate
rotate
inlined from surface_mesh_I.h: 1077
inlined from tracking_mesh_I.h: 373
updateGlobalBoundingBox
inlined from multi_node_mesh_I.h: 720
Mini Map
73M

Figure 7.7: HPC Trace Viewer showing how many CPU cylces are spent inside of the call of the `MPI_Max_Vector` utility function.

a significant number of cycles was spent in a method called `updateGlobalBoundingBox`.

Viewing the experiment database using `hpctraceviewer` one can obtain a visual representation of what is going on. Figure 7.7 shows a screenshot of the graphical output. It shows a timeline from left to right, illustrating the execution of multiple processes through multiple horizontal bars. By zooming in and adjusting the call stack depth which should be resolved one can distinguish individual time steps of the LIGGGHTS integration loop. Clicking into one of the sequences will determine what call stack was present at the point in time and in this thread or process. How detailed this information is depends on the resolution of the captured samples. By analyzing a single time step and exploring the call stacks over time one can gain insights about of what is going on in the given simulation.

Figure 7.7 shows that during each time step a longer sequence of time is spent in the method `updateGlobalBoundingBox`. Whatever load imbalance might be spent in the simulation will always lead to inefficiencies in this code because it effectively synchronizes all processes at this point multiple times. Even though this problem can already be seen with 4-core runs, it becomes worse with 32-cores or even more once Infiniband communication gets added.

Seeing this behavior one might be inclined to accept that the mesh implementation should be seen as a black box and apparently needs this synchronization. But this would have made any attempt to OpenMP parallelize meshes futile because any benefit would be undermined by constant MPI synchronization. Section 8.1 will outline the steps taken to resolve this situation.

Chapter 8

Core Optimizations

Measuring and profiling LIGGGHTS revealed hotspots in the existing implementation. In many cases the refactorings done to enable multi-threaded code in LIGGGHTS triggered further investigation in other areas of the code which restricted further advances. Some optimizations turned out to be more generally applicable and improved the core of LIGGGHTS itself[1]. The following sections describe three improvement examples to the core of LIGGGHTS which were a by-product of the overall work.

8.1 Optimization of dynamic mesh transformations

Meshes in LIGGGHTS may be translated, scaled and rotated over time. Users specify these operations by using mesh/move fixes in the input script. Transformations to a mesh are always applied in the same order they are specified. E.g., a mesh could be translated while rotating along one axis.

If the computation is spread out across multiple MPI processes, triangles may migrate from one processor to another. Similar to the particle case the parallelization of meshes operates using halo regions and ghosts. Each subprocess is responsible for all triangles living inside its MPI subdomain. Therefore, a process never sees the entire mesh during computation.

An MPI process stores two sets of triangles for each mesh. An initial set of triangles without any run-time transformations and a transformed set of triangles. During each time step the transformations are applied in the order they were specified based on the initial set of triangles.

An alternative implementation would be to transform all triangle nodes during each step based on their last position. However, this is not done because this would introduce numeric artifacts by accumulating numerical errors due to cutoffs.

After each transformation, the bounding box of a mesh may have changed. For this reason it is recomputed and stored for later use. This is done by looping over all triangles of the mesh and gradually growing the bounding box until all elements are contained.

---

[1] Comparing reference MPI run times to the new multi-threaded run times is somehow misleading. Without the improvements due to threaded code the baseline MPI performance would not have increased.

Because an MPI process only sees the part of a mesh it is responsible for, this only represents the local bounding box. In order to obtain the bounding box of the entire mesh all MPI processes have to exchange information and determine a bounding box which spans across all local boxes. This is done using a collective MPI communication directive called `MPI_Allreduce`. This method takes an array of data and applies an operation on each element. The resulting array with the final values is then distributed back to all processors. In this case the global bounding box is determined by applying the `MPI_MAX` operation to the vector $(-x_{min}, x_{max}, -y_{min}, y_{max}, -z_{min}, z_{max})$. The MPI operation will then find the largest value for each element in this vector across all processors.

This simple operation has significant performance implications on the scalability of the entire code. As was shown in Section 7.5 profiling revealed that large portions of time are spent inside of these collective MPI calls. In simulations with multiple moving meshes, after each transformation of every mesh all MPI processes have to synchronize during every time step. This synchronization serializes execution and limits the total parallel speedup as the number of processes increase. Figure 8.1 shows the parallel scalability of fixes of a mesh dominant simulation (see Validation in Chapter 11) before and after optimization. The following will explain the two main methods used to achieve this improvement.

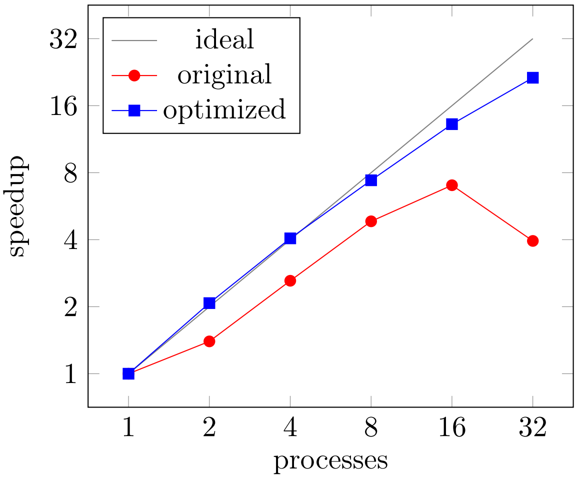


Figure 8.1: MPI strong scalability of mesh computations in Mixer case study. Extensive synchronization and allocations in critical paths were limiting the scalability of the mesh implementation.

8.1.1 Avoiding expensive operations in critical paths

Traces captured using HPCtoolkit exposed that collective MPI communication occasionally lead to significant synchronization overheads. In a 64 process simulation calls to `MPI_Allreduce` frequently led to waiting times of about 500ms. In the analyzed simulation this duration was equivalent to computing 10-50 full time steps. 63 MPI processes were waiting for one individual process to complete the operation.

It turned out to not be caused by an issue concerning the MPI calls themselves, but the surrounding code. The global bounding box computation and other parts of the code often do not use the MPI primitives themselves, but utility functions to simplify common tasks. In case of the global bounding box computation a utility called `MPI_Max_Vector` is called. This method internally uses `MPI_Allreduce`. Listing 8.1 shows its implementation.

Listing 8.1: Original implementation of `MPI_Max_Vector` utility function

```
inline void MPI_Max_Vector(double *vector,int len,MPI_Comm comm) {
  double *vector_all = new double[len];
  MPI_Allreduce(vector,vector_all,len,MPI_DOUBLE,MPI_MAX,comm);
  for(int i = 0; i < len; i++) vector[i] = vector_all[i];
  delete []vector_all;
}
```

What this code does is take an array of data and its length and return the global maximum of each element compared to values from other processors. In order to store the result of the MPI call a receive buffer is allocated with the same length as the original array. After receiving the result, the contents of the receive buffer are then copied back to the original array.

In the global bounding box computation, this array contains 6 double-precision floating-point numbers which require 48 bytes. This small allocation is the reason why MPI collective calls periodically seem to stall in HPCtoolkit traces. One of the processors needs more time than others to allocate the necessary receive buffer. Memory allocation is performed by a system call. There is no garantee by the operating system that successive allocations will take an equal amount of time. In the MPI case, there is also no guarantee that allocations across multiple compute nodes will take the same amount of time. Therefore, every now and then an allocation on one processor might take longer than on others. Many small allocations further keep the memory management of the operating system busy. What was therefore observed was an artifact of a large amount of small allocations over time with some occassional exceptionally long allocations.

This is a situation which could be avoided because in the `MPI_Max_Vector` case the allocation is not necessary. MPI allows to specify in-place operations which do not need a second buffer if the inputs are discarded anyway. These in-place operations will use the source buffer as input and output. The simplified implemention of `MPI_Max_Vector` is shown in Listing 8.2.

Listing 8.2: Improved and simpler implementation of `MPI_Max_Vector` using `MPI_IN_PLACE` as send buffer, which tells MPI to reuse the receive buffer for both sending and receiving.

```
inline void MPI_Max_Vector(double *vector,int len,MPI_Comm comm) {
  MPI_Allreduce(MPI_IN_PLACE,vector,len,MPI_DOUBLE,MPI_MAX,comm);
}
```

Because of this discovery many other utility functions were analyzed and simplified in the same manner. They now avoid unnecessary allocations which potentially stall computation on single processes and make use of optimized MPI functionality such as in-place operations.

8.1.2 Limiting collective communication and unnecessary work to minimum

While the implementation of the synchronization routine was made more efficient, the necessity of synchronization itself has a much larger impact. Synchronization during mesh operations limits the total scalability of the code, because mesh transformations are performed one by one synchronized over all processes.

A review of the code exposed that most of these synchronizations were avoidable. It also showed that the expensive operation of updating the bounding box of a mesh can be avoided in most cases.

The global bounding box of a mesh is only used in LIGGGHTS if the mesh is used for insertion. The bounding box is needed to compute the insertion region. However, in all other cases, such as using a mesh as granular wall, the global bounding box computation is unnecessary.

What this means is that while the transformation of meshes does influence the global bounding box, its computation can be postponed at a later stage and limited to cases where it might be needed. This avoids the traversal of all mesh triangles and synchronization during each time step.

The only requirement is that the implementation must remember if the current bounding box is outdated. When accessed, it can then recompute the global bounding box. This concept is often referred to as *lazy evaluation*. It is often realized by using a `dirty` flag. Instead of calling `updateGlobalBoundingBox` this flag is now set. The member function `getGlobalBoundingBox` seen in Listing 8.3 then checks this flag and launches a recomputation when necessary. Note that because it uses MPI calls in its implementation, any control flow which leads to this code must be executed on all MPI processes.

Listing 8.3: Getter of global boudinging box with lazy evaluation

```
template<int NUM_NODES>
BoundingBox MultiNodeMesh<NUM_NODES>::getGlobalBoundingBox()
{
  if(bbox_.isDirty()) {
    updateGlobalBoundingBox();
    bbox_.setDirty(false);
  }
  return bbox_;
}
```

8.2 Optimization of the mesh neighbor list building

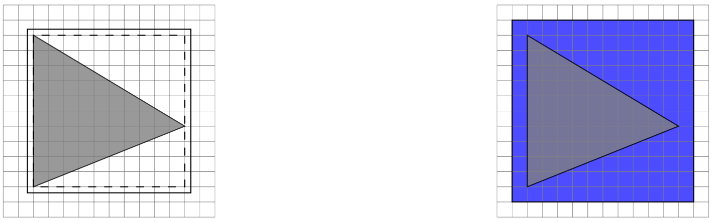
Meshes in LIGGGHTS use neighbor lists to reduce the amount of collision checks during contact detection. Similar to particle-particle neighbor lists, these are created prior to the force computation and remain valid for a few consecutive time steps. The neighbor list has two functions. It not only stores a list of particles in the proximity of a mesh, but also stores the number of particles which are near each triangle. The latter information is a necessary prerequisite for allocating enough storage for a particle-mesh contact history.

Building a neighbor list is an expensive operation because it requires the traversal of all mesh triangles and particles in the system. Traversing all mesh triangles is mandatory, but reducing the amount of particles which have to be visited is possible and can significantly reduce the build times of a list.

8.2.1 Existing implementation

The original mesh neighbor list implementation in LIGGGHTS 2.x reduces the amount of particles by using the available particle binning from the existing system. For each triangle the bounding box is determined. To capture all possible neighbors this bounding box is expanded by a similar cutoff distance as used for Verlet lists (see Section 2.3.2). The final bounding box is then used to limit the particle bins which are searched. This provides a bin range along the x, y and z axis. Figure 8.2 shows the extension of a mesh triangle's bounding box and the resulting range of bins for a 2D case.

Each bin in this range is then visited and the particles within that bin are checked for the neighbor criteria. Particles within a neighbor cutoff distance are added to the list and counted as neighbor of that specific triangle.

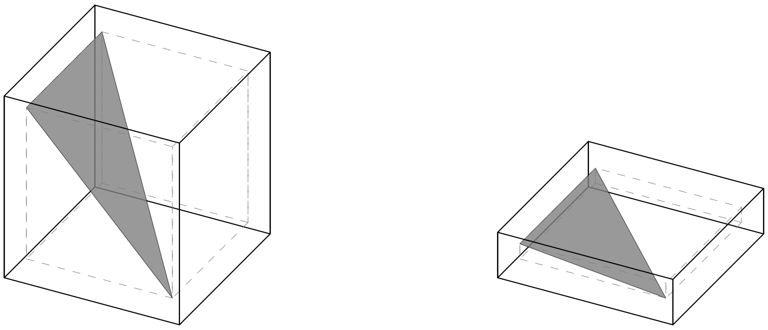


(a) Extension of triangle bounding box (b) Bins inside of extended bounding box

Figure 8.2: Visualization of particle bins surrounding a mesh triangle in 2D

8.2.2 Improved contact detection algorithm

While the binned contact detection algorithm used in the neighbor list builds reduces the amount of traversal significantly, many of the checked bins are not needed. In many cases mesh triangles are much larger than particle bins. In this case the bounding box of a triangle contains a large number of bins which are well beyond the neighbor cutoff. This is illustrated in Figure 8.2b for

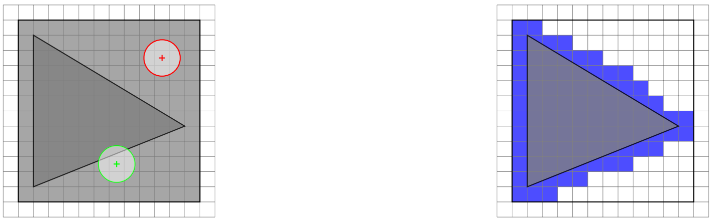


(a) large bounding box (b) small bounding box

Figure 8.3: Bounding box and extended bounding box of a mesh triangle. Based on the orientation this leads to a larger volume being checked.

a 2D case. Depending on the orientation of a triangle in 3D the bounding box will contain more or less particle bins as illustrated in Figure 8.3.

To further improve the contact detection algorithm for meshes the number of particle bins, which need to be checked, was reduced to a minimum. A second preprocessing step was added during the neighbor list building. Instead of accepting a range of particle bins in x, y and z direction the algorithms were adopted to work on arbitrary lists of bins. After determining the range of particle bins based on the bounding box, the algorithm then simulates a collision check with each of these bins. During this check a virtual particle is placed in the center of the bin and given a radius which spans across the diagonal of the bin plus the additional cutoffs and skin radius. By detecting a contact with this large virtual particle the entire bin can be either included or excluded from the bin list. This procedure reduces the amount of bins visited. In the 2D case illustrated in Figure 8.4b the original 144 bins are reduced by 55 bins which is almost a 40% decrease. In 3 dimensions the savings are even larger. Given that each bin is associated with a list of particles which is traversed and checked this method saves a significant amount of computational time.



(a) Bin contact detection (b) Optimized triangle bins

Figure 8.4: Optimization of mesh neighbor list build

The cost of the additional overhead introduced by the second phase of collision checks can be reduced by caching the resulting particle bin lists for static cases. If both the simulation domain boundaries and the mesh are static, the bin lists do not have to be recomputed but can be reused from a cache.

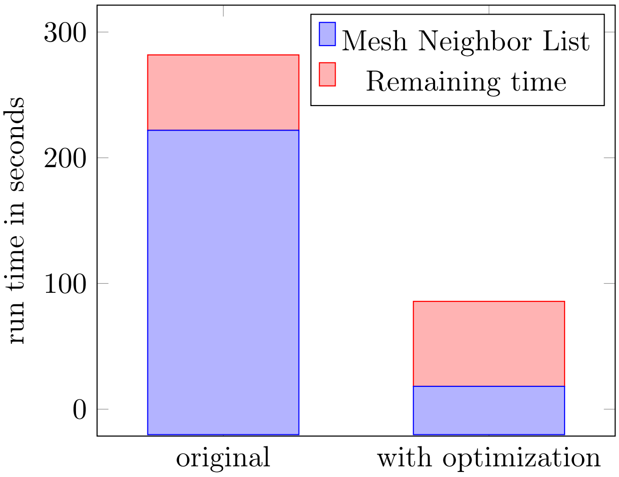


Figure 8.5: Run times of transfer chute simulation with static meshes. Between LIGGGHTS 2.x and LIGGGHTS 3.x a speedup of 3x can be observed, mostly due to an 12x improvement in mesh neighbor list build times.

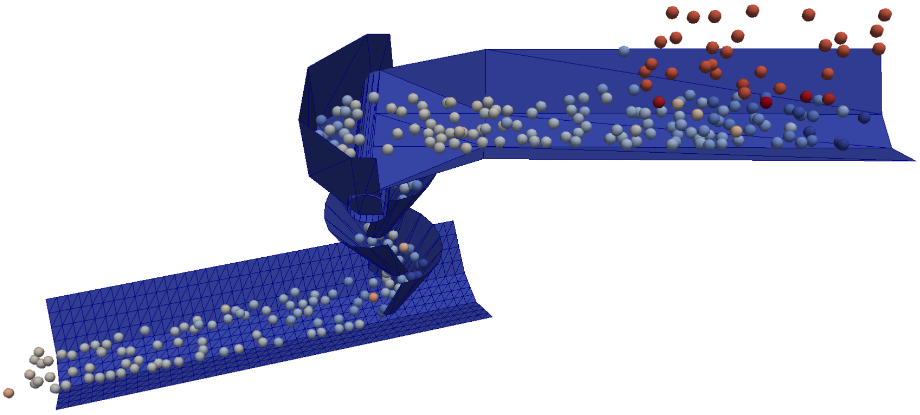


Figure 8.6: Transfer Chute Simulation

Figure 8.5 shows the impact of these changes on the serial run time of a transfer chute simulation seen in Figure 8.6. In this setup particles are inserted at the top of the domain and travel across two conveyor belts. These belts are vertically separated from each other and also change the direction of particles by a few degrees along the horizontal axis. Over time there are never more than 800-1000 particles in the simulation domain.

Profiling this case revealed that mesh neighbor list building was a major bottleneck. In LIGGGHTS 2.x 80% of the entire simulation time was spent inside of mesh neighbor list code. With the above optimizations of precomputing a smaller set of bins and caching them for static meshes LIGGGHTS 3.x can operate 12 times faster inside the mesh code. Overall this leads to an effective serial speedup of 3x.

8.3 Case Study: Optimizing the insertion of particles

Section 7.4 outlined how bottlenecks in existing code can be identified and efficiency is measured. It showed that the existing insertion algorithm in LIGGGHTS was exhibiting $O(n^2)$ run-time complexity. Through profiling the method causing this behavior was identified. 97% of the entire execution time is spent inside the `check_near_set_x_v_omega` in the `ParticleToInsert` class.

8.3.1 Original implementation

While this behavior was observed with insert/pack fix, `ParticleToInsert` is used by all insertion methods in LIGGGHTS. Independent of how particles are inserted, LIGGGHTS will determine an insertion volume. During each insertion time step a new memory buffer `xnear` is allocated to hold position (x, y, z) and radius of particles. This buffer must be able to accommodate all particles which either are already in the insertion region or could be inserted during the current time step. Inserting 3 million particles therefore requires an allocation of about 90MB every time step.

After filling this buffer with potential collision partners, particles are inserted one by one. The location of a new particle is randomized and can occur anywhere inside the insertion volume. To increase the chance of finding a free insertion location in dense random packings each insertion is attempted multiple times.

Finding a valid insertion location is done by generating a new location and checking if there is any overlap with particles in the `xnear` buffer. If there is no overlap, the particle is marked for insertion at this location and added to the `xnear` buffer. Particles are allocated and inserted once all insertion positions of the current time step were determined or the maximum number of attempts have failed. Finally, the `xnear` buffer is freed.

Why this algorithm exhibits an $O(n^2)$ run-time complexity can be explained by the increasing number of particles needed to be checked during each insertions. Additionally, the continuous reallocation of large memory buffers further adds to the poor performance of this algorithm.

8.3.2 Improved implementation

A more efficient algorithm can be implemented by borrowing techniques which have already been established in DEM. Instead of requiring each particle to check for overlaps with all particles in a region, reducing the number of particles to check through binning could be an option.

However, there are two problems for reusing the existing particle binning already present in LIGGGHTS. For one, particles are not inserted immediately, only their position is generated and checked. Only after all positions have been determined, the global particles arrays are modified and reallocated if necessary. This avoids costly allocations. Binning of particles only occurs to particles which are part of the global arrays. Second, the algorithms which provide this functionality are not implemented in a reusable form.

Listing 8.4: Interface of RegionNeighborList

```
class RegionNeighborList {
public:
    RegionNeighborList();
    bool hasOverlap(double * x, double radius) const;
    void insert(double * x, double radius);
    size_t count() const;
    void reset();
    bool setBoundingBox(LAMMPS_NS::BoundingBox & bb, double maxrad);
};
```

Therefore, the binning algorithm was extracted and generalized into a reusable component: `RegionNeighList`. The interface of this class is shown in Listing 8.4. It hides the implementation details of how overlaps are checked. It only allows adding new particles to the list and determine if there is any overlap with already added particles.

Internally it uses the same efficient binning algorithm used in LAMMPS/LIGGGHTS. But it was written from scratch and using the C++ STL where it made sense. Insertion could now be significantly simplified by using this simple and reusable component:

1. LIGGGHTS uses an instance of `RegionNeighborList` to generate a neighbor list not for the entire simulation domain, but only the insertion region. This is done by setting a bounding box which defines the area of interest.
2. During each time step, this neighbor list is emptied and refilled with all existing particles which are still inside the insertion region. Note that this does not automatically cause reallocation of memory. Internally data structures will resize and if necessary grow to accommodate. But allocations are kept to a minimum.
3. While checking for particle overlaps, only particles in the proximity of the potential insertion position are checked. This is thanks to particle binning and only checking neighboring particle bins around the insertion area.
4. Once an insertion position is found, the newly inserted particle is added to the neighbor list.

The overall speedup of using this improved algorithm has been described in Section 7.4.

Chapter 9

Parallelization of LIGGGHTS

DEM simulations are expensive computations compared to many other methods in computational fluid dynamics. Instead of working on averaged field quantities the trajectory of each individual particle is tracked. This requires the use of small time step sizes because every collision must be resolved. A simulation of few seconds real time with several ten thousand particles takes hours if not days to complete on a single processor. Lab-scale and industrial-scale applications, which contain millions of particles, take weeks or months to complete. These types of problems can therefore only be investigated if additional computational resources are used.

Through parallelization of the code the simulation workload can be decomposed and distributed onto multiple processors. This chapter describes which forms of parallelism exist in general and how they can be utilized by application programmers. It also explains the limits of parallel scalability, the importance of load-balancing and how LIGGGHTS has previously tackled these challenges through its MPI parallelization. Finally, a new form of parallelization is introduced which combines two different parallelization strategies and allows better utilization of existing and future hardware.

9.1 Forms of Parallelism

Today developers are forced to exploit parallelism in their application to continue to scale on newer hardware. From a computer architecture point of view there are three types of parallelism available [42]:

Instruction Level Parallelism (ILP): This is a form of parallelism which is automatically exploited by current CPU architectures. A stream of instructions does not have to be executed sequentially by a processor if there are no dependencies between them. Superscalar instruction scheduling allows processors to rearrange instructions and execute them in a way which maximizes hardware utilization. E.g., a floating-point operation can be performed at the same time as an instruction which manipulates integers. Pipelining and advanced instruction scheduling techniques like branch prediction allow processors to execute serial programs much faster. Yet this form of parallelism can not be directly controlled by developers. It is influenced indirectly by structuring computations in a way which allows compilers to generate streams of instructions to have favorable performance characteristics. In general the optimizations made by compilers are more efficient than manual tuning of instructions. It is therefore recommended to write well structured and portable code and give compilers enough information to help them during their optimizations.

Data Level Parallelism (DLP): Another form of parallelism is the usage of simultaneous operations across large sets of data. Wide registers allow manipulating large chunks of memory and perform computations with this data using vector instructions. This form of parallelism is predominantly found in GPUs which manipulate large chunks of data using the same operations and are less optimized for changes in control flow. CPUs have also gained considerable capabilities in this area using Single-Instruction Multiple Data (SIMD) instructions. To make use of this form of parallelism developers need to ensure that their code can be vectorized by a compiler and make use of the special vector operations. In certain scenarios additional hints must be given to a compiler by making the intent more explicit [45] [46].

Thread Level Parallelism (TLP): The last form of parallelism available to developers is using multiple concurrent instruction streams to solve a given computational problem. These instruction streams can either be executed on one processor using multiple cores, multithreading or multiple processors in a cluster. The increase of cores in modern processors allows processing more and more tasks concurrently on a single chip. Whether each task is identical or different from other tasks depends on the application.

Writing scalable and efficient software requires exploiting all of these types of low-level parallelism. While the first two forms are enabled through optimizing instructions on a given target platform, either manually or by a compiler, thread-level parallelism allows developers to write concurrent applications at a higher level using software. Parallel applications can span multiple processors either by using multiple sockets on a single compute node or using multiple nodes in a cluster.

For programming multi-processor systems a different classification can be made which distinguishes based on the locality of processors and their memory: Distributed Memory programming and Shared Memory Programming.

9.1.1 Distributed Memory Programming

For decades parallelization of a computation meant to distribute the workload onto multiple nodes on a compute cluster or massively parallel processor (MPP). Each node is given a subset of the entire problem which is then processed independently as a separate process on its own local memory.

During computation these processes explicitly synchronize with others and exchange information via an interconnect such as Ethernet or Infiniband. This requires that information is packaged for transport and the transmission of the data is orchestrated. Such systems are often visualized as network of nodes exchanging messages, as seen in Figure 9.1.

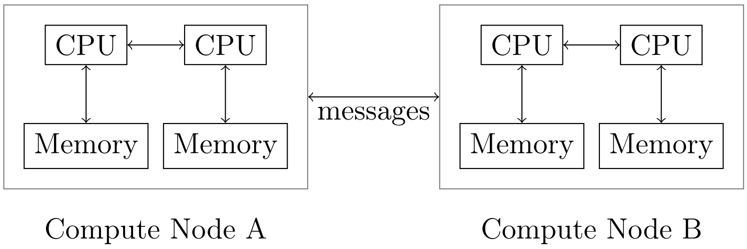


Figure 9.1: In distributed memory programming each process operates on its own local memory and communicates with other processes via messages. This requires each process to duplicate necessary data structures and allocate sufficient memory for sending and receiving data.

Over time this approach has been standardized into the Message Passing Interface (MPI) [47]. This is a library interface defining how compute nodes can communicate individually with each other or in groups. In addition, MPI functions also take care of low-level data conversions in heterogeneous compute clusters. E.g., transforming data from a little-endian machine to big-endian and vice-versa. It provides bindings for common languages in HPC such as C, C++ and Fortran. The available functions are divided into point-to-point communication and collective communication routines.

Point-to-point operations define the communication between individual nodes using send and receive buffers. There are two types of point-to-point operations. Blocking functions start a transmission and only return once all data has been sent or received. The second type are non-blocking operations which return immediately after calling them and allow processes to continue working while data is being transmitted. At a later point in time the status of the transmission can be evaluated and a process can choose to wait for its completion.

In contrast, collective operations operate at a higher level and provide efficient implementations to common problems such as gathering and scattering information from and to groups of processes. These operations include one-to-all, all-to-one and all-to-all functions. Each implementation of the MPI standard can optimize these routines to improve performance on certain types of machines and interconnect fabrics, e.g., clusters using Infiniband. Popular implementations currently are OpenMPI, MVAPICH2, and Intel MPI.

9.1.2 Shared Memory Programming

With the rise of multi-core processors and graphics processing units (GPUs) a different programming model has gained momentum. Shared memory programming allows multiple threads of execution to work concurrently in the same memory space (see Figure 9.2). Unlike message passing which requires explicit communication, shared memory programs communicate implicitly. This is both a strength and weakness of this approach. It avoids the overhead of allocating communication buffers and transferring data over an interconnect. But given the shared nature of data it also becomes the responsibility of the programmer to ensure value consistency if data is manipulated by multiple threads.

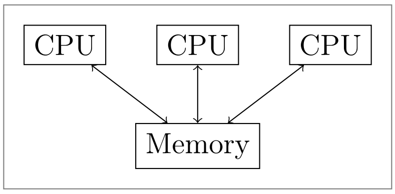


Figure 9.2: In shared-memory programming multiple processing cores operate inside the same memory space. Communication between threads of execution occur implicitly and require more diligent usage of shared data.

There are a variety of models available to developers to write shared memory programs. Most models are adaptations of the so-called fork-join model which is visualized in Figure 9.3. While a program begins as a single process with one shared memory space, it may fork and create concurrently running threads of execution at any point in time. Processors with multiple cores allow running threads not only concurrently, through the time-slicing scheduler of the operating system, but truly in parallel. After completion of their tasks, threads may then join back into the main thread of execution.

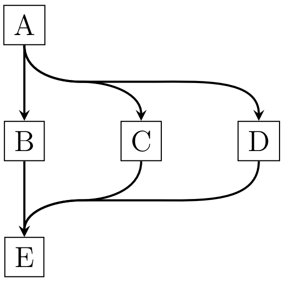


Figure 9.3: Fork-Join model: a main process forks itself and creates multiple concurrently running threads which can access the same memory address space. After completion, threads join back into the main process.

The recently published C++11 standard includes a standard library for concurrent programming [48] [49]. It allows spawning threads and provides synchronization primitives such as locks and mutexes. Prior to this a common approach for writing threaded programs on UNIX systems was by using the POSIX thread library (pthreads) [50]. This C library allows directly managing threads and implements the fork-join model including necessary synchronization primitives.

However working with threads at this level is cumbersome and error-prone. Given the cost of spawning threads and stopping them, it is not efficient to continuously launch new threads in an HPC code. Higher level approaches such as OpenMP or Intel Threading Building Blocks use thread-pools which keep unused threads dormant and reactivate them once there is work to process.

OpenMP is an industry standard [51] [52] for shared-memory programming which has gained significant momentum in the last couple of years especially in the area of accelerator cards such as the Intel Xeon Phi Co-processor. Instead of manually managing threads it only requires programmers to annotate their existing code with special compiler statements called *pragma statements*. At compile time supporting compilers will then generate parallelized code using this standardized threading model and its rules.

Intel Threading Building Blocks [53] is a very successful C++ library which uses standard C++ and does not need any compiler modifications. It provides a higher-level task-based model which lets developers specify parallelism in their code through functors and includes an automatic scheduler which load-balances the workload through work-stealing [54].

Another approach gaining more and more attention is Cilk Plus [55]. Similar to OpenMP it adds a small set of keywords to the language and a support library to enable a powerful multi-threading programming model. It also requires compiler support and uses a work-stealing scheduler to automatically load balance parallel tasks. Unlike other approaches Cilk Plus lets developers mark opportunities of parallelization. If those opportunities are used depends on the hardware and is decided by the scheduler.

For specialized hardware such as GPUs the CUDA programming language [56] and OpenCL APIs [57] have become the standard approach. These provide C or C-like interfaces to program architectures with many threads. To standardize the implementations between CPU and GPU codes there are also approaches which try to unify the programming models. OpenCL can be used for both CPU and GPU executions but requires entirely different compute kernels. OpenACC is a standard to allow offloading work to both accelerator cards and GPUs [58] [59]. It follows in the footsteps of OpenMP and uses compiler directives to markup existing code for parallelization.

This shows that even though utilizing multi-core and multi-threading architectures has become a necessity - from a developer's point of view - the industry has not yet agreed on a standard model of programming shared memory architectures. Some approaches are so new, that their compiler support are at best experimental.

9.1.3 Mixed-mode Programming

With the proliferation of shared-memory architectures in mainstream computing the environment in high-performance computing also started to change. Instead of building clusters with powerful individual processors, multi-core processors also became common in cluster computing. Nowadays each cluster node is even equipped with one or more GPUs or accelerator cards. Utilizing these clusters therefore requires a mixed-mode programming model. Both message passing and shared-memory programming must be used together to fully utilize the available computing resources. The most common approaches found in literature are variants of MPI+X, such as MPI+OpenMP or MPI+CUDA [58, 60, 61, 62, 63]. MPI version 3 also introduced shared-memory programming directly into the MPI standard, allowing to utilize shared-memory regions within groups of processes on the same compute node [64].

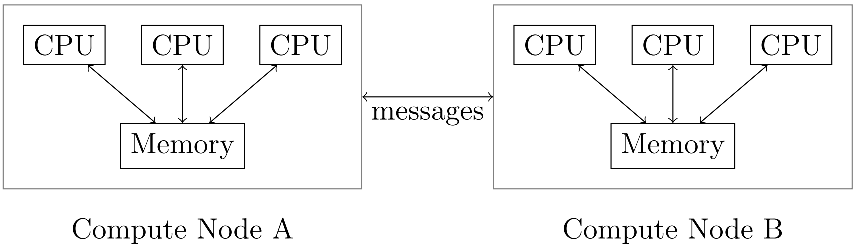


Figure 9.4: Mixed-mode programming or hybrid programming combines both distributed and shared-memory programming approaches. While nodes communicate through messages, each node consists of one or more shared-memory subnodes.

9.2 Limits of Parallelization

The overall goal of running computations in parallel is to speed up their execution and complete within less time. For DEM this means being able to do more simulations in the same time or longer simulations as before. A naive approach to parallelization would be assume that doubling the amount of computational resources allows cutting the computation time in half. In other words we expect a speedup of 2x, which means the simulation is two times faster than before. Formally the speedup S of a parallel implementation is defined as

$$S = \frac{T_S}{T_P}$$

where T_S is the serial run time, and T_P is the parallel run time. Closely related to this is parallel efficiency E:

$$E = \frac{T_S}{PT_P}$$

where P is the number of processors. Parallel efficiency is a normalized value between 0 (= not efficient) and 1 (=fully efficient). Ideally any parallel application should strive towards $S = P$ and $E = 1$. In this case the parallel code would exhibit linear scaling and fulfill the expectation of improving performance at the same rate as adding resources.

However, achieving this ideal limit in general is very difficult. In 1967 Gene Amdahl made an observation about parallel programs in which he stated [65]:

> ...the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

This insight has since been referred to as *Amdahl's law* and formula were developed to capture its essence. Parallel speedup by a factor of P is limited to the fraction of code f which benefits from parallelization. The total parallel run time T_P is then the sum of the remaining serial run time and the run time of the parallelized code.

$$T_P = f\frac{T_S}{P} + (1-f)T_S = T_S\left(\frac{f}{P} + (1-f)\right)$$

If the number of processors increases the speedup of the parallelized code then becomes

$$S = \frac{T_S}{T_P} = \frac{1}{\frac{f}{P} + (1-f)} \underset{P\to\infty}{\approx} \frac{1}{1-f}$$

This shows that for larger processor counts the speedup of the application becomes limited by the serial portion of the code. Even if an application only contains 1% of serial code, the speedup can not exceed 100x.

There is however a less pessimistic view on this matter usually referred to as *Gustafson's observation*. In 1988 John Gustafson revisited Amdahl's law and made the argument that scalability should not be measured with a fixed problem size [66]. While the serial parts of an application inherently limit the maximum speedup obtained, the utilization of more parallelization allows solving larger problem sets. If the amount of work being processed in the parallel parts of the code grows larger than in the serial portions, then according to Amdahl's law the serial fraction of the code becomes less. This in turn increases the parallel speedup.

Since both observations of Amdahl and Gustafson are correct measuring scalability is ambiguous. It depends on whether the speedup is computed with a fixed problem size in mind or with a forward looking increasing problem size. In literature the evaluation of speedup of a parallel code with an increasing number of processor cores and a fixed problem size is referred to a *strong scaling*[55]. If the problem size grows at the same rate as the number of processors, the term used is *weak scaling*. Achieving strong scalability, which means approximately reaching linear speedup, remains the harder problem.

9.3 MPI Parallelization of LIGGGHTS

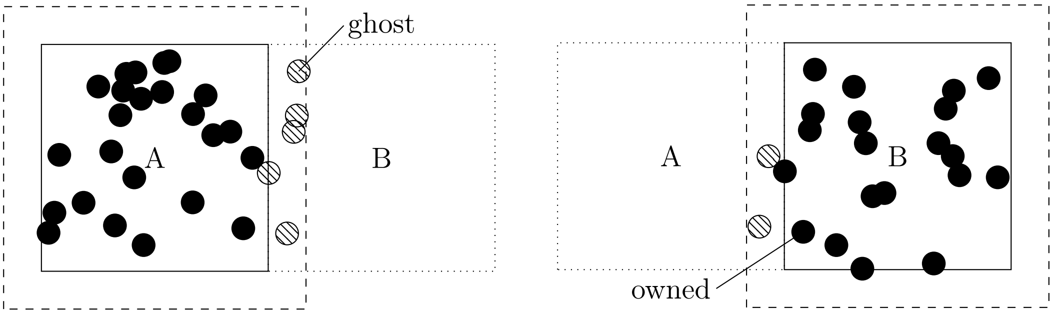
LIGGGHTS has inherited a well optimized and scalable MPI parallelization from the molecular dynamics code LAMMPS. As described in Section 2.5, LIGGGHTS uses domain decomposition to distribute the workload among multiple MPI processes. During each time step neighboring processors exchange information through the usage of halo regions. After synchronization, each processor then operates independently inside its respective subdomain.

9.3.1 Static Domain Decomposition

The original domain decomposition present in LAMMPS and LIGGGHTS allows users to specify the number of domain cuts along the x, y and z dimension of the simulation domain. With this information a Cartesian grid of subdomains is created and then mapped to MPI processes. Once defined this grid of processes and subdomains can not change.

Communication cost of domain decomposition

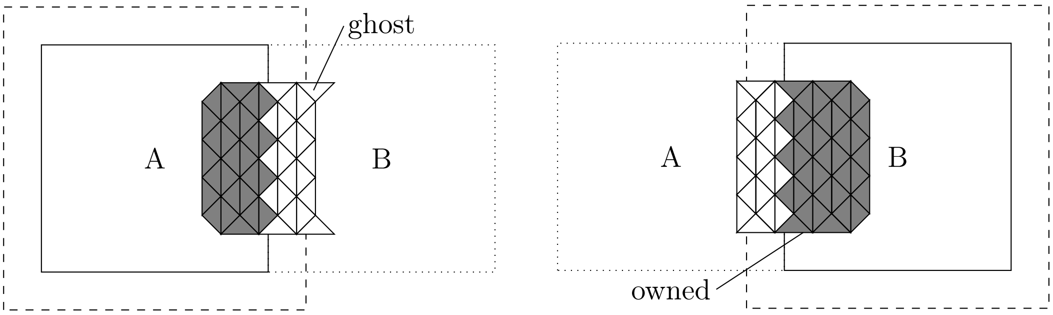
Figure 9.5 illustrates how domain decomposition influences the visible amount of particles of two adjacent processes. Each process is responsible for all computations inside its subdomain. It therefore *owns* all particles in its region. Since these elements might interact with other particles in neighboring processes, each process maintains copies of remote data from halo regions, which extend into neighboring subdomains. Their extent is defined by a cutoff that is proportional to the largest particle diameter. Particles inside a halo region are called *ghost particles*, and their information - such as position and velocity - is synchronized from the process that owns them.



(a) Processor A (b) Processor B

Figure 9.5: Particles distributed between two adjacent processes. Particles within the halo region of a process are duplicated as ghost particles. This way collisions are detected on both processors.

Besides particle data LIGGGHTS also contains an MPI parallelization for moving and non-moving mesh geometries. In its current implementation mesh elements such as triangles are treated similar to particles. Triangles inside a halo region of a neighboring processor are synchronized between processes. These duplicated elements are used for collision detection and referred to as *ghost triangles.* Figure 9.6 shows how a mesh is seen by two adjacent processes if the geometry spans both subdomains. Note that each triangle can only be *owned* by one process, which in turn is responsible for aggregating force contributions on that mesh element[1].



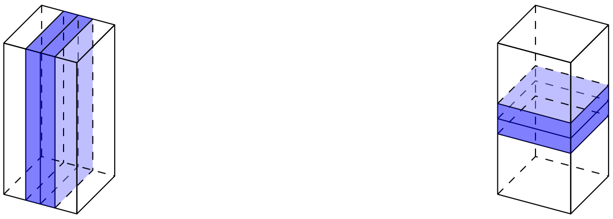
(a) Processor A (b) Processor B

Figure 9.6: Mesh triangles distributed between two adjacent processes with ghost triangles being duplicated within the halo region of a process. This way collisions are detected on both processors.

---

[1] forces on mesh elements are used to evaluate stress and wear on a given geometry.

The volume of all halo regions in a given simulation is proportional to the maximum amount of communication needed in a given decomposition. Depending on how a simulation domain is decomposed the communication volume of halo regions changes. Figure 9.7 shows the halo regions of two different decompositions of a 1x1x2 domain. In Figure 9.7a a cut along the y axis introduces a cut plane parallel to the xz-plane. Cutting the same domain along the z axis introduces an interface parallel to the xy-plane as seen in Figure 9.7b. Because the halo cutoff is uniform along all directions, the volume of the xz halo region is twice as large as the xy region. This means the theoretical maximum amount of communication in the first decomposition can be up to twice as large as in the second case.



(a) xz halo region (b) xy halo region

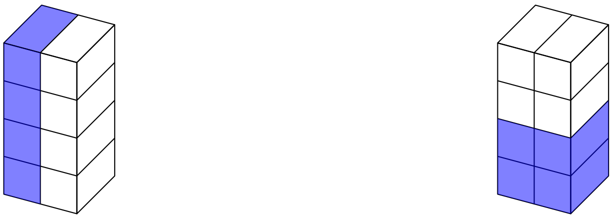
Figure 9.7: Halo regions of two different decompositions of a domain with an extent of 1x1x2.

The communication cost is also influenced by how moving mesh geometries are distributed among subdomains and if they are subdivided into multiple parts through cuts. Each subdivision of a mesh along a cut plane introduces a halo region with ghost triangles. These have a much higher communication cost than particles alone. The information exchanged between processors for a single mesh triangles is at least three times larger than of a single particle. Supplementary properties might be communicated in addition to vertices. Therefore, subdividing meshes should be avoided to minimize communication. Note that similar communication cost is associated to static meshes if the domain boundaries move and intersect with them over time (see Section 9.3.3).

Finally, the cost of communication over halo regions also depends on how MPI processes are mapped to hardware. Subdomains which share large communication volumes should be placed physically close to each other and utilize higher bandwidth and low latency connections. Figure 9.8 shows two possible mappings of 8 MPI processes. In both cases a decomposition of 1x2x4 is used[2]. Mapping subdomains of this decomposition to processes and in turn to hardware processor cores can lead to different communication patterns. In this example we assume to map to an 8-core machine, consisting of two sockets each equipped with a 4-core processor. The processes on one socket are colored in blue, while the remaining white processes are placed on the second socket. In Figure 9.8a processes are divided into two columns. While this mapping reduces communication of processors on the same socket, it also requires all four processes on each socket to communicate with their counterpart on the other socket over a slower interconnect. The mapping shown in Figure 9.8b increases the amount of communication between processors

---

[2] a decomposition of AxBxC means that there are A parts along the x-axis, B parts along the y-axis and C parts along the z-axis.



(a) (b)

Figure 9.8: Two process mappings of a 1x2x4 decomposed domain.

on the same socket. But concurrently it limits inter-socket communication to two processes on each socket, which is the smallest possible communication volume for the slowest communication link in the given system.

Limitations and Challenges

While static domain decomposition allows using multiple processes at once, there are limitations to this geometric subdivision:

1. Subdomains must have a minimum extent which must be larger than twice the cutoff used for halo regions in each direction. Otherwise, halo regions of non-adjacent processes would overlap.
2. As the number of processes increases, the number of cuts in each dimension must increase. Even if cuts along a dimension are not favorable, the size constraint of subdomains eventually requires adding cuts in additional dimensions to further increase the number of possible subdomains.
3. Domain decomposition using a global grid does not allow refinement. Any cuts introduced into the domain span across the entire domain and introduce additional subdomains.
4. Subdivision assumes an equal workload among all subdomains which is usually not the case in DEM simulations. Subdivision in empty areas of a simulation will not add any benefit.

In addition to these limitations users face an increasing challenge as the number of available processors rises. Figure 9.9 shows the possible static domain decompositions for 2, 4 and 8 processes. What this sample of decompositions shows is an increasing variety of decompositions becoming possible. Table 9.1 further lists the number of possible decompositions up to 1024 processes[3]. Choosing a suitable decomposition becomes an optimization problem. Which decomposition maps best to the available hardware? How should MPI processes be mapped to subdomains to minimize communication and latency? Which decomposition distributes the expected workload best among all compute resources?

---

[3] This list was produced using a trivial Python script which determines all possible decompositions for a given number of processors.

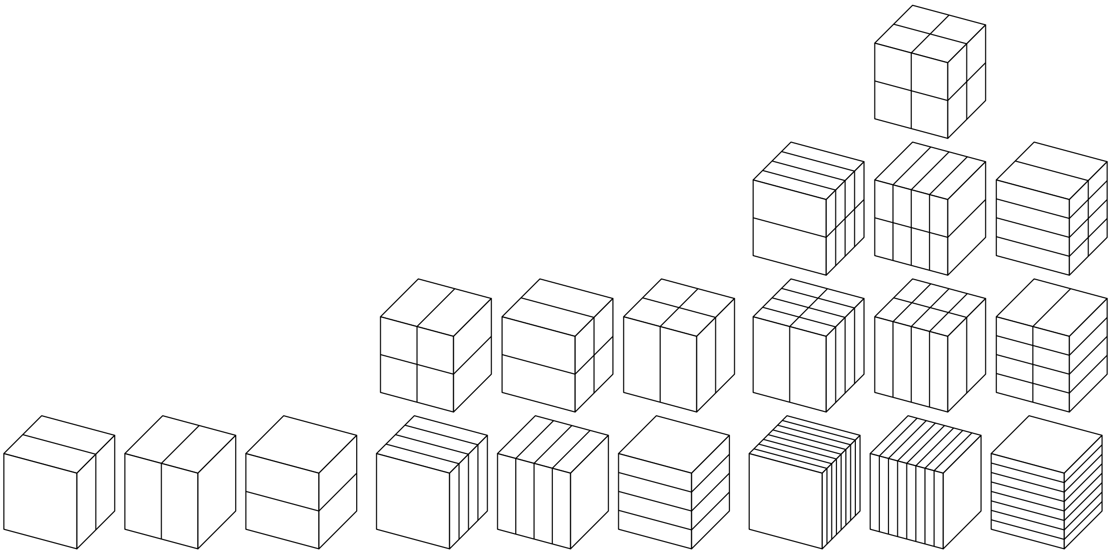

(a) $P = 2$ (b) $P = 4$ (c) $P = 8$

Figure 9.9: Possible domain decompositions with 2, 4, and 8 processes

P	# of decompositions
2	3
4	6
8	10
16	15
32	21
64	28
128	36
256	45
512	55
1024	66

Table 9.1: Number of possible decompositions with a given number of processes

There are no general answers to these questions. But in many cases the decomposition may not only be a poor choice, it can produce simulations which are as slow as a serial execution or worse. The penalty of a wrong decomposition is therefore very high.

Only measurements allow giving definitive answers for each type of simulation on a given architecture. In many cases such exploratory simulations are not feasible because realistic simulations are too time consuming and complicated to set up. For this reason experience in creating a qualitative estimate of the expected workload and knowledge about the available hardware resources are necessary to guide the selection process.

The minimization of the communication volume can be used as a first start. If LIGGGHTS is allowed to automatically choose the number of cuts along more than one direction it will automatically choose a decomposition with the lowest communication volume. However, this assumes uniform communication cost across all halo regions.

Mapping of processes to hardware to optimize the communication pattern remains a main challenge and requires manual tuning. It requires locating the ranks of MPI processes in a domain decomposition and assigning these ranks to available resources.

9.3.2 Load-balancing in DEM

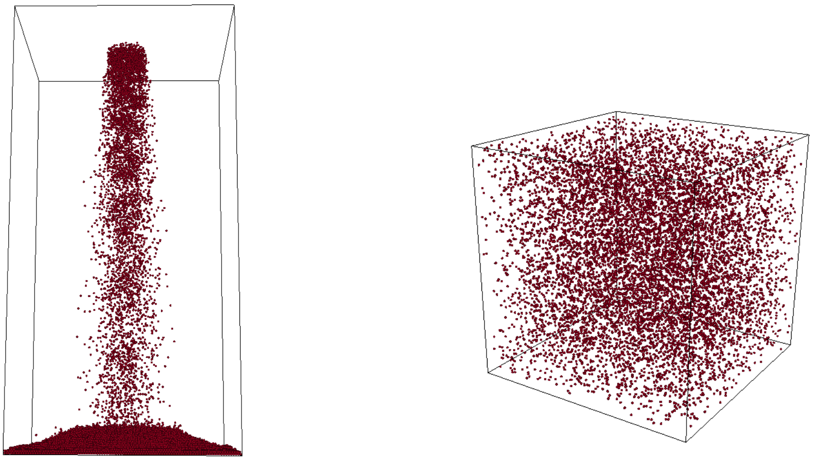
Reducing the amount of communication between processors is essential for achieving high performance in parallel computations. Equally important is the need to keeping all processors busy with work. If the computational load is not evenly distributed among processors, some processors will stall the entire computation at the next synchronization point because of too much work. While these processors then perform a majority of work, others will be idling and waiting. This is both true for working with MPI processes and multi-threaded code.

To avoid such inefficiencies, the workload must be balanced in both cases. This requires a deep insight into what contributes to the main workload of each processor. In DEM this has been identified as being mainly caused by particle-particle interactions, followed by particle-wall interactions. In industrial cases these can make up to 80% of the total simulation time. Other critical algorithms include neighbor list building and integration of EOMs.

The total workload of these steps can not always be precomputed, but is dependent on the progression of the simulation itself. As particles move through a given geometry and through the domain, the particle density and number of force computations in each subdomain will change.

The setup itself has a large influence on how the workload distributes within a simulation. Figure 9.10 shows two extreme cases. In Figure 9.10a particles are pouring onto the floor. Particles near the bottom have a larger number of neighbors, while particles which are still in flight travel side-by-side and nearly never collide. Therefore, computations near the floor will be more expensive. The other case seen in Figure 9.10b shows a simulation with an evenly distributed system of particles traveling in random directions and velocities. Without gravity acting on particles, this case will have a fairly equal amount of work assigned to each particle.

Real-world simulations are often a mixture of these two extreme types. They can also change



(a) Stream of particles filling domain from top to bottom

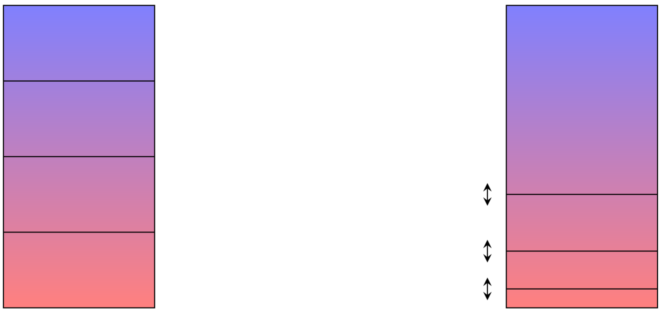
(b) Evenly distributed particles moving in random directions

Figure 9.10: Two extreme examples with different workloads. In the first example the workload is focused near the bottom, while in the second case it is evenly distributed throughout the domain.

from one extreme to another over time. Therefore, simulation engines need capabilities to dynamically adjust to changing work distributions. Load-balancing mechanisms may improve throughput, but must work in concert with the remaining system to minimize communication overhead.

9.3.3 Dynamic MPI Domain Decomposition

Load-imbalance in DEM simulations motivated the original dynamic load balancing strategy in LIGGGHTS. This strategy - which is illustrated in Figure 9.11 - automatically rearranges subdomain boundaries and tries to balance the number of particles among MPI processes.



(a) static decomposition

(b) dynamic decomposition

Figure 9.11: Dynamic domain decomposition adjusts domain boundaries along one or more axis. In this example the particle density is the highest on the bottom of the domain. Evenly distributing along the z-axis would keep the top processor idle. Adjusting domain boundaries allow to evenly distribute the number of particles among processors.

Starting from the static Cartesian grid of subdomains, the dynamic domain decomposition periodically adjusts the location of subdomain cuts along one or more directions. This is done for each direction individually. First, the number of particles in each slice of a direction is aggregated. Ideally, the number of particles is evenly distributed among all slices in that direction. The imbalance in the distribution is measured by the maximum number of particles in a single slice divided by the total number of particles. If this value exceeds a threshold, load balancing along that direction commences.

Slices are resized by adjusting their boundaries using a recursive multi-sectioning algorithm. Based on the previous split locations and the aggregated sum of particles up to these positions, the algorithm interpolates new split locations so each slice could contain the desired number of particles. Using these new splits, the actual number of particles in the new slices is then computed over all processors and the quality of the new decomposition is evaluated. This procedure is repeated until either the desired distribution of particles is reached by a fair margin or the maximum number of attempts has been made.

Finally, all processors are made aware of the new decomposition, and particle data is migrated. During the next communication phase inside the integration loop, this triggers an update of particle neighbor lists and other helper data structures. The following summarizes the sequence of important operations during and after MPI communication:

1. (Optional) Periodically adjust domain boundaries along one or more axes. If necessary, migrate data.
2. Exchange particles and contact history data between MPI processes
3. Apply spatial sorting of local particle data to each MPI process to increase data locality and improve cache utilization
4. Update particle neighbor lists
5. Exchange mesh triangles and contact history data between MPI processes
6. Update mesh triangle-particle neighbor lists

While this dynamic decomposition alone improves load balancing of MPI-only simulations significantly, it is limited by its focus on balancing the number of particles rather than the actual workload among processors. It is also restricted to only moving existing cut planes. Meshes are simply decomposed on the basis of the subdomains obtained by balancing particles. This can shift computational effort to individual processors, which leads to additional undesirable load imbalance. It can also cause costly communication due to migration of mesh data over moving domain boundaries.

9.4 MPI/OpenMP Hybrid Parallelization of LIGGGHTS

While LIGGGHTS is already parallelized using MPI the limitations of this existing approach led to the exploration of other options to further increase throughput. The MPI implementation dates back to a time when processing was distributed among multiple cluster nodes, each having a serial processor. In the last few decades processors have however evolved into massively parallel circuits themselves making shared-memory programming increasingly important.

In the following the efforts of bringing multi-threading optimizations to LIGGGHTS are described which optimize key parts of its integration loop. The motivation behind adding this capability was to allow exploration of the shared-memory model. The goal was to improve the throughput by using more efficient methods in a shared memory space and reducing the communication overhead introduced by MPI. For this a second layer of parallelization was added using OpenMP on top of the existing dynamic MPI domain decomposition. Because shared-memory programming alone does not allow scaling across multiple compute nodes, both parallelization strategies must coexist in the same code base. This led to the development of an *MPI/OpenMP hybrid parallelization.*

Multi-threaded code introduces two new groups of challenges to DEM. On the one hand computing capabilities have to be added which can run concurrently. Threaded code must ensure correctness by maintaining thread-safety on shared code paths. However, while correctness can be enforced through simple means, such as critical sections and locks, the true challenge is to find efficient implementations which add a benefit compared to their MPI counter parts.

On the other hand distributing *particles* evenly among threads does not imply an even distribution of the *workload*. Therefore, dynamic load balancing at the OpenMP level had to be implemented for both particle-particle and particle-wall interactions in various forms.

9.4.1 Related Work

The original MPI code of LAMMPS used a static domain decomposition [13] [67] [68] which partitions space such that the area of communication between MPI ranks is minimized. A similar approach was taken by Kačianauskas et al. [67], but they observed an increase in computational effort for simulations of polydisperse material compared to monodisperse material. Gopalakrishnan and Tafti [68] reported an almost ideal speed increase in the DEM portion of a CFD-DEM fluidized bed simulation on up to 64 processors, and a parallel efficiency of 81% on 256 processors.

Static domain decomposition works well for homogeneous condensed matter simulations, but in DEM the typically inhomogeneous and changing distribution of particles across subdomains results in performance-limiting load imbalances. This motivated the development of a load-balancing scheme in LIGGGHTS through which domain boundaries are dynamically adjusted at run-time. This has since been backported into LAMMPS [69] and was described in Section 9.3.3.

The idea of moving domain boundaries is not new. It has previously been implemented by Srinivasan et al. [70] in the context of molecular dynamics. They generate equally loaded rectangular regions by moving domain boundaries in increments of a load-discretizing (LD) grid that is coarser than the computation domain (CD) grid. Load balancing occurs on multiple levels by adjusting cuts in the x, y and z directions. The load balancing is based on particle density or number of pairs in each subdomain. In their simulations, a reduction in computation time by as much as 50% was achieved.

More sophisticated load-balancing techniques were employed by Plimpton et al. [71], who applied three different decompositions in a joint finite-element (FE) and smoothed particle hydrodynamics (SPH) code. They employed static FE decomposition of mesh elements, while SPH-decomposition and contact-decomposition of contact-nodes and SPH-particles were performed dynamically using Recursive Coordinate Bisection (RCB) [72]. This geometric algorithm was chosen because it produced well-shaped subdomains and exhibited linear scaling to the problem size.

RCB and many other dynamic load-balancing algorithms were evaluated by Hendrickson and Devine [73]. They listed geometric methods like RCB as being well suited to geometric problems such as particle simulations. RCB in particular is considered to be one of the fastest and easiest to implement in this group of algorithms. It also has the valuable property of being incremental, meaning that small changes in a domain only lead to small changes in the decomposition. This property minimizes expensive communication between processors.

Recently, LAMMPS has also gained the ability to use RCB directly for its MPI decomposition [74]. However, this implementation only balances the number of particles in each subdomain, not the actual workload. It is also incompatible with the current MPI parallelization of meshes in LIGGGHTS.

A different parallelization strategy, known as particle subset method, was employed by Kafui et al. [22] in their CFD-DEM code. They applied a “mincut” graph-partitioning algorithm to a graph of particles and their contacts that generates partitions of particles with a minimal number of contacts with particles in another partition. Based on these partitions, for each MPI process working on a single partition, particles are assigned and halo regions defined. Partitions are recomputed at regular intervals for dynamic balancing.

The desire for better load balancing and better utilization of available compute resources motivated many groups to experiment with the particle subset method using shared-memory parallelizations in their codes. Since fall 2011, LAMMPS has included an add-on package called USER-OMP [75], which provides multi-threaded and thread-safe variants of selected modules and subroutines, in particular force kernels, neighbor-list builds and a few other selected modules. In 2013, we started our exploration of using these OpenMP modifications for LIGGGHTS [76].

Concurrently, Amritkar et al. [77] developed an OpenMP parallelization for MFIX DEM code which uses the particle subset method. They argued that for the N-body particulate phase of their CFD-DEM simulation, a parallelization over the number of particles is more suitable. To support their claim, they presented measurements of a fluidized bed simulation (uniform particle distribution) and a rotary kiln heat transfer simulation (non-uniform distribution). Despite higher overheads for fetching non-local data, the better load balancing makes the OpenMP parallelization 50-90% faster than MPI-only. To achieve optimal speed increases, they ensured that data was stored locally by following the first-touch policy, and that thread/process affinity was set using placement tools.

Finally, Liu et al. [61] further expanded on existing MPI and OpenMP work and described a hybrid MPI/OpenMP parallelization of their MFIX-DEM solver. They emphasized that data locality and thread placement policies play a critical role in scaling OpenMP to large core counts. They also mentioned the necessity of distinguishing between private and global data in multi-threaded code, which avoids race conditions by each thread using its own copy of data and reductions. Due to reduced MPI communication, their hybrid becomes faster. Scaling was presented using a coupled CFD-DEM run of a 3D fluidized bed simulation. They reported speed increases of 185x on 256 cores (72% efficiency) of their hybrid parallelization compared to 138x using a standalone MPI computation with 5.12 million particles.

Previous works, such as Henty et al. [78], also developed MPI, OpenMP and hybrid versions of a DEM benchmark code and evaluated their performance. They explained the difficulties introduced by multi-threading, such as updating a global force array from multiple threads. Due to overheads introduced to their OpenMP implementation, however, their hybrid models were not more efficient on Symmetric Multi-Processor (SMP) nodes.

Rabenseifner et al. [62] provided an overview of parallel programming models on hybrid platforms and described the difficulties of mapping different approaches to hardware. They listed potential gains of using a hybrid approach, including the benefit of improving load balance and reducing memory consumption.

The potential benefits and pitfalls of hybrid parallelization using MPI and OpenMP was also mentioned by Smith et al.[79]. In their work, they showcased results of a Game of Life implementation and a mixed-mode Quantum Monte Carlo code. They concluded that a mixed strategy might not always be the most effective, but in situations where MPI exhibits poor scaling due to load imbalance an improvement might be achieved.

Many of the individual building blocks of the LIGGGHTS MPI/OpenMP hybrid parallelization are well known in the literature. However, this work illustrates the benefit of combining multiple load-balancing techniques using two levels of parallelization. It showcases how partitioning can be used to avoid data races and to implement a lock-free version of important DEM routines. By doing some additional work in advance, it avoids look-up tables in its force loops and reduces safeguard complexity inside the loop to constant time. It also describes the additional challenges introduced through usage of mesh geometries and contact histories in contact models.

9.4.2 Outline of the hybrid parallelization

The overall concept underlying this MPI/OpenMP hybrid parallelization is illustrated in Figure 9.12. The existing MPI domain decomposition is still used to generate subdomains. Load-balancing through dynamic domain decomposition allows adjusting boundaries dynamically along one or more axes. This enables the creation of a first (coarse) distribution of the global workload and utilization of multiple compute nodes. Process pinning can be used to avoid costly communication between nodes and sockets. The main difference is that not all processor cores are used by MPI.

After domain decomposition, each MPI process further distributes the workload of particle-particle and particle-wall interactions by splitting them into partitions. These are recomputed periodically to adjust to changing workloads. Partitions are processed by a team of threads, and each partition is assigned to a single thread. By making core algorithms aware of partitions, force computations can be implemented without the need of critical sections or custom-placed locks. The following sections describe the individual building blocks of this implementation.

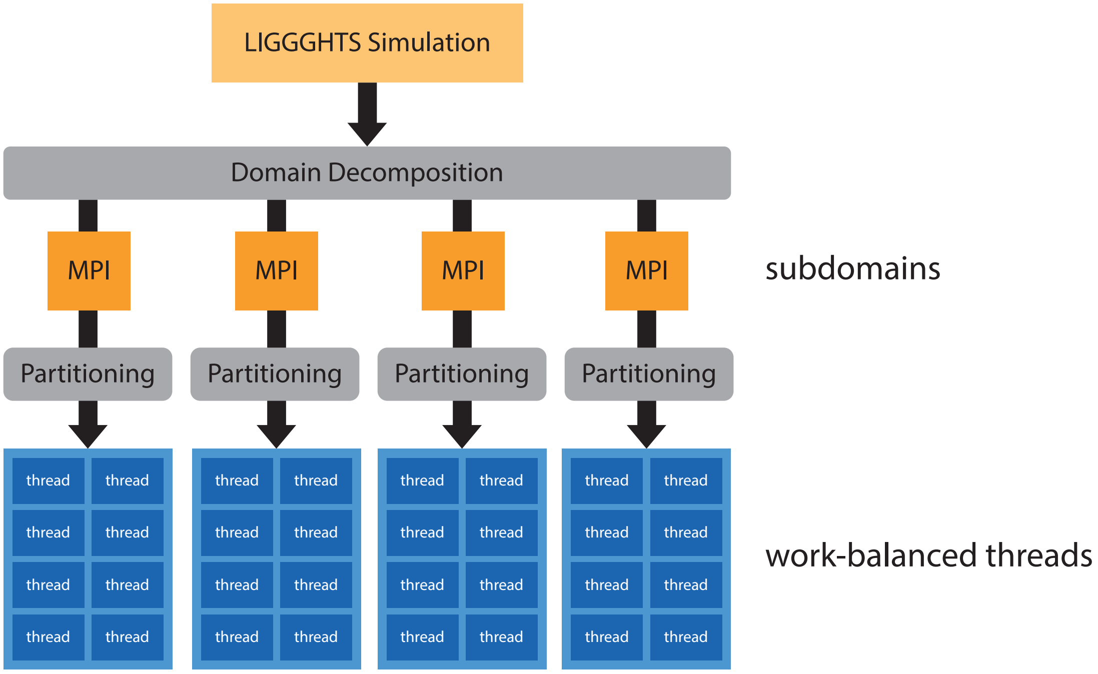


Figure 9.12: Overview of the MPI/OpenMP hybrid parallelization. The workload of a simulation is first distributed along one or more axes using a simple MPI decomposition. MPI load balancing, which adjusts boundaries over time, can still be used. Each MPI subdomain then further divides its subdomain into partitions of equal workload. All work-intensive algorithms launch multiple threads that work only on particles in a partition assigned to them.

Chapter 10

OpenMP parallelization of LIGGGHTS

Adding OpenMP multi-threading to LIGGGHTS required wide-spread changes to the entire code base. Refactoring of granular force kernels, which was described in Chapter 6, was an important milestone in this process. What followed was the addition of multi-threaded versions of all major portions of computation. Figure 10.1 shows a simplified view of the entire integration loop currently present in LIGGGHTS. Multiple squares indicate parts of the computation which contain code paths benefiting from additional parallelization.

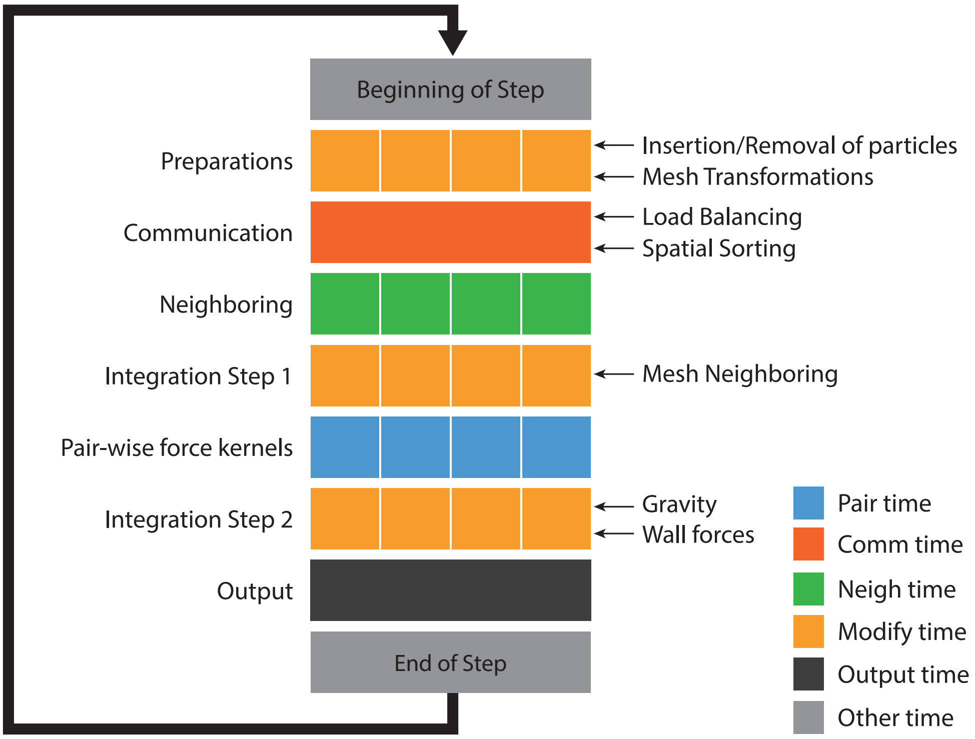


Figure 10.1: LIGGGHTS integration loop (simplified). At the beginning of each step, particles may be inserted and meshes transformed. Particles are exchanged between MPI processes, and load balancing may occur. After communication, neighbor lists are built. Forces are computed between integration steps 1 and 2. The parts of the loop that can be parallelized are shown as 4 separate squares. The color codes indicate how these sections contribute to the total timing breakdown.

Many essential algorithms had already been ported to OpenMP in the USER-OMP package from LAMMPS. This includes applying gravity on particles, neighbor list building and Velocity-Verlet integration. However, threaded versions of granular particle-particle and particle-wall compute kernels were still missing. LIGGGHTS also contains a different contact history implementation and additional complexity due to meshes. Also, since DEM simulations are more likely to exhibit unbalanced workloads, dynamic load-imbalancing had to be implemented at the OpenMP level.

The following sections describe the OpenMP implementation of LIGGGHTS in detail. Section 10.1 first introduces the basic building blocks of OpenMP, followed by a short discussion of the common challenges in multi-threaded programs in Section 10.2.

Section 10.3 then explains the challenge of data races in multi-threaded variants of the granular particle-particle kernels. Possible strategies to cope with these challenges are described. Section 10.3.3 introduces a partitioning approach which avoids data races by duplicating work when necessary. Because static work assignments to threads do not lead to a balanced computation, dynamic load balancing using a geometric method is presented in Section 10.3.4. It is followed by Section 10.4 which explains the necessary modifications in neighbor list building.

The parallelization of particle-wall computations requires a different set of changes. Section 10.5 describes the operation of the OpenMP variant of walls. Avoiding conflicts and load-imbalance in mesh neighbor list building is the topic of Section 10.5.2. Once these neighbor lists are built the particle-wall workload is balanced using a simple method outlined in Section 10.5.3.

Finally, Section 10.6 summarizes how these changes are packaged into the global integration loop to enable the hybrid MPI/OpenMP parallelization of LIGGGHTS.

10.1 Shared-memory programming using OpenMP

OpenMP defines a set of compiler directives and supporting run-time library. It allows writing portable multi-threaded code with minimal changes to existing code and provides language bindings to support both Fortran and C/C++ code bases. This is why in the past many scientific codes have adopted this method to add threading to their code base.

The benefit of using OpenMP is that it not only adds a uniform model for threading, but also offers a way to offload work to accelerators and help compilers with vectorization. Working with accelerators is beyond the scope of this work, but choosing OpenMP enables this future work. At the same time other groups are working on vectorization of LAMMPS [63, 80] which is why no effort was made to duplicate this work.

As of writing (2015), the OpenMP standard has recently been released as version 4.5. Being a standard it only defines the available functionality. The implementation however has to be made available by compiler vendors.

10.1.1 Compiler Support

OpenMP support has been part of the GCC compiler since 2006 initially supporting the 2.5 standard in version 4.2. As the standard continued to evolve, GCC followed with about one year of delay. E.g., the OpenMP 3.0 standard was finalized in 2008 and support was released by 2009 in GCC version 4.4. The later added 3.1 standard from 2011 became part of the version 4.7 release in 2012. In Summer of 2013 the OpenMP standard 4.0 was finalized. By 2014 GCC had added support in version 4.9.

Intel has supported OpenMP since its 12.x series of compilers, which were initially released in 2011. Through various updates the compiler gained many capabilities which predated the standard solutions, e.g., pragma statements which are useful for vectorization of code or work offloading onto accelerator cards. These vendor-specific additions have been standardized in OpenMP 4 which is now fully supported in the 16.x compiler series released in 2015.

Concurrently an experimental OpenMP branch was created for the Clang compiler in late 2013. It added full OpenMP 3.1 support and is set to become part of the mainline compiler soon. It uses an open-sourced OpenMP run-time from Intel and also includes OpenMP 4 features such as offloading and SIMD pragma statements.

This shows that in the last 3 years the momentum behind OpenMP has built up. Which is why this approach was chosen to investigate shared-memory programming in LIGGGHTS. However, it also required staying at the cutting-edge of currently available compilers to make use of the newest standard additions.

10.1.2 OpenMP Directives

In its simplest form writing OpenMP-enabled code only requires adding compiler directives in front of C++ statements and blocks. Listing 10.1 shows how a block of code can be executed by multiple threads by using the `#pragma omp parallel` directive. To compile this code with GCC the `-fopenmp` flag must be added.

Directives manipulate individual statements of code. If a sequence of statements should be executed in parallel, they must be grouped by a C++ scope which defines a block of code.

Listing 10.1: Simple example of OpenMP parallel region

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char ** argv) {
  #pragma omp parallel nthreads(4)
  {
    int nthreads = omp_get_num_threads();
    int tid = omp_get_thread_num();

    #pragma omp critical
    printf("Thread␣%d␣out␣of␣%d\n", tid, nthreads);
  }
  return 0;
}
```

In this code four threads are spawned and individually execute the following code block. OpenMP contains an API to retrieve information about the runtime system. The function `omp_get_num_threads` returns the number of actively running threads in the current parallel region. `omp_get_thread_num` returns the index of the current thread. Ideally any computation occurring in a parallel block should be independent of other thread work. However, often shared resources must be accessed. This program shows the usage of the `critical` directive on a single statement to ensure that only one thread is writing to console using `printf`.

10.1.3 Work-sharing

OpenMP allows to easily parallelize loop structures. Listing 10.2 shows the usage of the `for` directive inside of a parallel region. This will instruct the compiler to decompose the iteration space of the loop at runtime and execute subranges of the original loop on different threads.

Listing 10.2: OpenMP Parallelization of a for-loop

```
#pragma omp parallel
{
  #pragma omp for
  for(int i = 0; i < nall; ++i) {
    // each thread will get an equal range of the loop to work on
  }
}
```

Because this is such a common pattern, both parallel and for directives can be combined in a more compact form as seen in Listing 10.3.

Listing 10.3: Compact form of OpenMP parallel region and work-sharing construct

```
#pragma omp parallel for
for(int i = 0; i < nall; ++i) {
  // each thread will get an equal range of the loop to work on
}
```

How a for loop is decomposed and distributed among threads is controlled by its scheduling algorithm. By default, OpenMP uses a **`static`** work schedule which decomposes the original iteration into equal sized chunks. Other schedules such as `dynamic` and `guided` introduce additional overhead by using a work queue. Chunks of work are added to this work queue and retrieved by threads on an on-demand basis. In the `guided` schedule the chunk size gradually decreases to allow better load-balancing at the end.

10.1.4 Reductions

A common task in loops is the accumulation of values to a variable. It is inefficient to operate on a shared variable. Instead, OpenMP offers the reduction clause which allows each thread to operate on a local copy of a variable. After completion the results of all threads are *reduced* using a specified operator to single value and saved in the original variable. Listing 10.4 shows OpenMP variant of the gravity force computation. It not only adds a gravity force to each particle but also accumulates the potential energy due to gravity. At the end the variable `egrav` will contain the total potential energy due to gravity in the system.

Listing 10.4: OpenMP reduction in parallelization of gravity force computation

```
egrav = 0.0;
#pragma omp parallel for reduction(+:egrav)
for (int i = 0; i < nlocal; ++i) {
  if (mask[i] & groupbit) {
    const double massone = mass[type[i]];
    const double fx = massone*xacc_thr;
    const double fy = massone*yacc_thr;
    const double fz = massone*zacc_thr;
    f[i][0] += fx;
    f[i][1] += fy;
    f[i][2] += fz;
    egrav -= fx*x[i][0] + fy*x[i][1] + fz*x[i][2];
  }
}
```

10.2 Challenges in multi-threaded programs

Manipulating the same memory space using multiple threads introduces a new set of challenges for application developers. On the one hand the validity of the executed program is no longer garanteed. Threaded programs can invalidate assumptions made in the serial code. On the other hand threading can cause performance degradation if it is not used properly.

10.2.1 Data Races

One of the most common problems in multi-threaded code are data races. Multiple threads can corrupt their data if they manipulate it at the same time. For a data race to occur two or more threads must work on the same memory region. If at least one of them is writing this memory location, there is a potential data race. Without any precautions a thread accessing the memory location may read one of the following values:

- The value prior to the write operation
- The value after the write operation
- An intermediate value if the write operation was still in progress while accessing. This essentially leads to reading binary garbage.

Data races should be avoided whenever possible. This often requires changing an existing algorithm to avoid the conflicting operations. If such changes are not possible, the remaining option is to enforce synchronization between threads in these critical code paths. This is achieved through synchronization primitives such as critical sections, locks or atomic operations.

10.2.2 False Sharing

Even if algorithms avoid data races and are correct, they can run into a problem which dramatically reduces run-time performance. To illustrate this phenomenon a short example is given in Listing 10.5.

Listing 10.5: False sharing example code

```
vector<double> sum(nthreads, 0.0);

#pragma omp parallel for shared(sum)
{
  const int tid = omp_get_thread_id();
  // do computation
  sum[tid] += ...;
}
```

In this example an array with `nthreads` entries is allocated and initialized with zero. The parallel region is then executed by each of these threads. Each thread only manipulates their entry inside of the shared `sum` vector. Although there is no data race, all threads are continuously interrupting each other. The reason for this can be found in the memory subsystem. `std::vector` allocates a continuous block of memory, so that its entries are stored sequentially.

Accessing one entry means manipulating 8 bytes of memory. However, the memory subsystem does not fetch memory in bytes, but in cache lines. Today a cache line is usually 64 bytes long. This means that 8 doubles fit into a single cache line. Once that cache line is loaded from RAM, it can be accessed more efficently from the cache. But why does it become slower?

Each thread or each core has additional levels of cache. This means each thread keeps its own copy of the currently accessed cache line. Manipulating a single value inside of this cache line automatically invalidates the cached value of all other processors. This means that after each write operation of one thread, all others have to update their cache-line. Either through an intelligent memory system, but in the worst case by accessing the updated value from RAM.

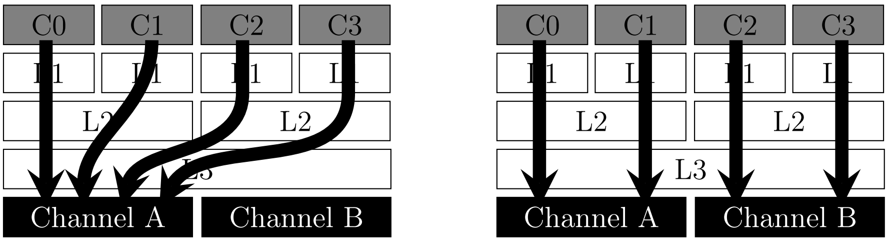
This continuous invalidation of each other's cache lines is called false-sharing. It is caused by the cache-coherent non-uniform memory access (cc-NUMA) present in most of today's processors and should be avoided by not letting multiple processor access the same cache lines. In order to achieve this one must ensure alignment of memory allocations along cache line boundaries and keep in mind the structure of data.

10.2.3 Data Locality

The memory subsystem is one of the most important parts of today's computers and also one of its major bottlenecks [21]. While computational throughput has increased significantly over the last decades, the latency and bandwidth of memory was not able to scale in the same manner [42]. For memory-bound applications this is an increasing problem.

Shared memory programs allow multiple threads to operate within the same virtual memory space. Memory is organized into pages of a typical size of 4096 bytes (4KB). The operating system takes care of mapping virtual memory pages to physical ones. Programmers can not directly influence where pages are placed, but the operating system often makes sure that pages are placed in the proximity of the thread which first uses an allocated page. This is commonly known as the first-touch policy which is the default memory placement strategy under Linux and other operating systems [81]. Memory pages are not placed immediately after allocation, but during the first access to it.

Because of this if memory is allocated by one thread and first accessed by another, the memory page will be allocated using a memory controller closest to the second thread. In order to fully utilize the memory bandwidth of a given CPU, all memory channels must be used. This can only occur if each thread operates on pages allocated close to their core. Otherwise, memory contention over a single memory channel will limit the scalability of the threaded code. Figure 10.2 illustrates the memory contention caused by missing data locality.



(a) global memory access (b) thread-local memory access

Figure 10.2: Memory contention can limit scalability of threaded code. If all threads access memory which is located closer to a single core, the effective available bandwidth is reduced.

Because of data locality and caching reasons, HPC codes should prevent processes and threads to migrate between CPU cores. This is achieved through utilizing both MPI process-binding for and OpenMP thread-binding.

10.2.4 Non-deterministic results

The execution of code using multiple threads can influence the outcome of a simulation result based on the implementation of the parallel regions of code. If thread synchronization is used, the outcome of computation is dependent on the order of execution of the involved threads. This is an undesirable property in numeric simulations. While results may differ due to parallelization compared to a serial run, having simulation results which differ after each execution with an identical setup would make interpreting results even more difficult. This is why the parallelizations presented in the following sections have not only been chosen for thread-safety and load-balancing reasons, but also to ensure deterministic results after each execution.

10.3 Multi-threading of particle-particle computations

The redesign of the granular force kernels - described in Chapter 6 - was driven by the development of their multi-threaded versions. In this new structure writing a threaded version was reduced to writing a replacement for the granular force loop implemented in `PairStyles::Granular`. Consequently, the parallelized version is implemented in `PairStyles::GranularOMP` which is a subclass of the `IGranularPairStyle` interface. Through this all contact models are parallelized using the same multi-threaded force loop.

A major challenge in writing a multi-threaded version of a granular force loop is dealing with the existing code structure and identifying shared resources. Any multi-threaded code must differentiate between local data manipulated by single threads and global data accessible by multiple threads. Once shared code portions have been identifed they must be made thread-safe by avoiding or protecting against data races.

In addition, a challenge arises from the fact that distributing *particles* evenly among threads does not imply an even distribution of the *workload*. Therefore, dynamic load balancing at the OpenMP level becomes necessary.

10.3.1 Data races in granular pair styles

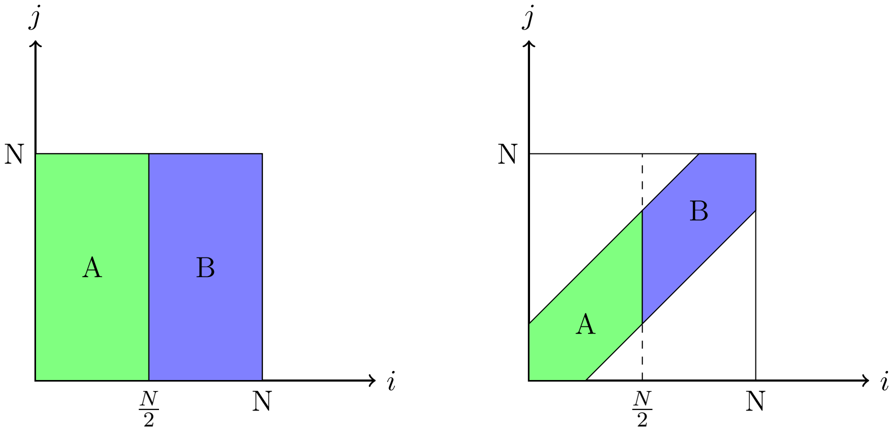
In its simplest form a granular force kernel consists of two nested loops which iterate over all particles and each of their neighbors. Forces and torques acting on a particle are computed using a contact model. These forces and torques are then added to global force and torque array entries of the particle.

A trivial parallelization of this problem is to statically distribute particles among threads. Each thread will then operate on a subset of particles in the simulation universe. With OpenMP such a parallelization can easily be achieved through a work sharing construct as seen in Listing 10.6.

Listing 10.6: Trivial OpenMP parallelizatin of granular force loop

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < nlocal; ++i) {
  for (all neighbors j of particle i) {
    // compute force between i & j
    f[i] = ...;
  }
}
```

This implementation produces a static decomposition of loop iterations giving each thread a chunk of equal size. Figure 10.3 illustrates how the iteration space (i, j) traversed by the granular force kernel is distributed among two threads. Without neighbor lists the computation would be of the order $O(n^2)$. Through the limitation to neighbors and due to spatial sorting what remains



(a) Full iteration space (b) Optimized iteration space

Figure 10.3: Distribution of the iteration space of a granular force loop among two threads A and B. Through the use of neighbor lists and spatial sorting the number of iterations in j are reduced which forms a band across the $i - j$ diagonal

is a symmetric diagonal band in the iteration space which causes a runtime complexity of $O(n)$.

It is trivial to parallelize portions of code using OpenMP when each computation is independent of all other computations. However, as mentioned in Section 2.3.3, granular force loops are further optimized by using Newton's third law and thereby reducing the total force computations by a factor of two. Neighbor lists created by LIGGGHTS only contain a minimum amount of neighbors, assuming that during force computation forces are applied to both contact partners, with opposing signs. Utilizing Newton's law in the force loop leads to an implementation shown in Listing 10.7.

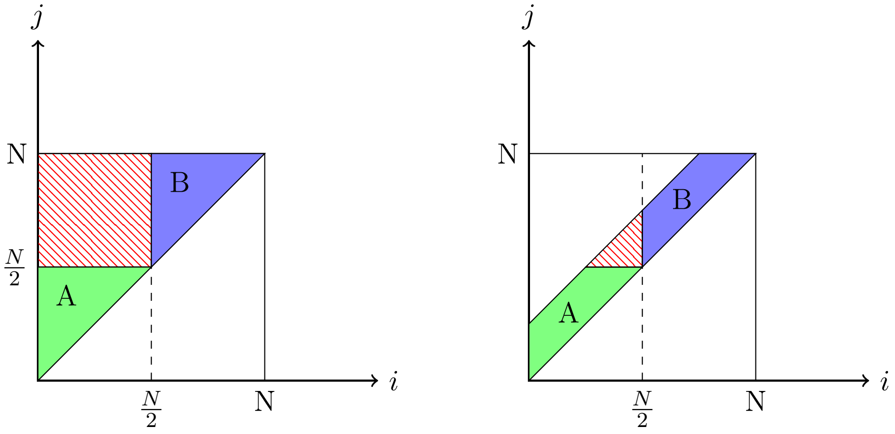
Listing 10.7: OpenMP parallelization of granular force loop using Newton's third law

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < nlocal; ++i) {
  for (all neighbors j of particle i) {
    // compute force between i & j
    f[i] = ...;

    if( /* use newton 3 */  ) {
      f[j] = ...;
    }
  }
}
```

This implementation makes particle-particle force computations inherently dependent on each other. Each computed force of a contact pair is applied to each contact partner, but with opposing directions. However, this adds a potential data race to the computation. Figure 10.4 visualizes the reduction of force computations and highlights the potential zone of conflict for two threads.

Updating the force of a neighboring particle j may interfere with a different thread which is also writing to that particle's force array at the same time. Without proper safe guards, updating the resulting force on a particle by multiple threads can lead to lost or even false data, since such operations are not atomic. Atomic updates of this kind would have to be requested explicitly, and incur a significant performance penalty.



(a) Half iteration space due to Newton's third law

(b) Optimized iteration space

Figure 10.4: Usage of Newton's third law reduces the amount of work by cutting the iteration space in half. Only the upper triangle of the iteration space is evaluated. Neighbor lists and spatial sorting again limit the iteration space to a band above the $i-j$ diagonal. However, in either case the space $j > \frac{N}{2}$ processed by thread A would write to data regions of thread B and is a source of conflict.

10.3.2 Methods to handle data races

There are several existing techniques to tackle the data races present in granular force kernels. Other groups [77, 61, 78] have reported on either working with local copies of data or using synchronization mechanisms, such as reductions, locks, critical sections and atomic updates. These will now briefly be introduced.

Critical Sections

A simple solution is avoiding the data race by marking which region of code should only be executed by one thread at a time. This enforces that certain memory regions can not be updated simultaneously by multiple threads. In OpenMP this is done by surrounding the affected code in braces and adding a `omp critical` pragma in front of the block. This marks a critical section of code.

Each critical section can lead to a serialization of the affected code region. Once a thread reaches this point, it will have to wait for its turn. This leads to idle CPU cylces, because threads can not continue without passing the critical section. This reduces the amount of parallelized code and increases the serial portion, which after Amdahl's law reduces the amount of expectable

```
#pragma omp critical
{
  f[i] += ...;

  if( /* use newton 3 */  ) {
    f[j] += ...;
  }
}
```

speedup. It also neglects the fact that threads accessing particles in the range of another thread is not the norm, but a less likely event. With critical sections all write operations are safeguarded for the worst case, even though it might not have been necessary.

Atomics

An alternative to critical sections are atomic operations. Instead of marking regions of code as mutually-exclusive, atomic operations ensure that arithmetic operations are always fully executed without interruption.

```
#pragma omp atomic
f[i] += ...;

if( /* use newton 3 */  ) {
  #pragma omp atomic
  f[j] += ...;
}
```

Atomics need hardware support for these special types of instructions. There is a limited set of operations which are available as atomics and they introduce their own share of overhead inside of the processor. Given the cc-NUMA architecture of most modern CPUs, updating the same cache line even atomically still leads to synchronization between CPU core caches. Similar to critical sections atomics still add a penalty to all operations, even though in some cases the additional synchronization overhead is not needed.

Array reduction

One way therefore is to avoid conflicting write operations completely by duplicating the necessary data structures. While each thread can read from shared global data, its write operations can be mapped to a thread-local data storage. After each thread completes with its force computation, the thread arrays are then reduced to one global force array. This is the strategy implemented in the original USER-OMP implementation [75].

The main downside of this approach is its extensive use of memory and memory bandwidth. Each thread needs to allocate enough storage to hold computation results for each particle. This means that if a processor is working on one million particles all threads need to allocate a force array (and similar arrays such as torque) to hold the results of each of these particles. It also adds the burden of reinitializing these large force arrays prior to each time step. Contention for memory bandwidth and the growing overhead of the reduction operation with increasing amounts of threads limit the parallel efficiency of this approach.

Custom Placed Locks

Since critical sections are too costly and serialize code at a very coarse level, a more fine grained approach was explored. Critical sections are a high level construct in parallel computation. They are built from lower level building blocks such as locks. A lock is an object which can only be acquired by a single entity. As long as a lock is held, other threads trying to acquire it will either wait (idle) or try again later. Once the owner of the lock releases it, others can acquire it. Locks are implemented using special instructions in processors such as atomic operations. In a critical section a lock is acquired and released after the last critical instruction. By using more than one lock and naming them, more sophisticated strategies can be implemented.

Henty et al. [78] generate a look-up table which includes only particles that are updated by more than one thread. This table is then consulted during each update of the force array to protect critical accumulations with atomic locks. Since the look-up table is valid for many force computations, and most force computations do not require protection, this method minimizes the locking overhead.

Listing 10.8 shows a different implementation using two types of locks. Each thread operates using an `update_lock` to operate within its own memory region, while accesses to neighboring particles are controlled by a `other_lock`. Note that while `update_lock` is constant for each thread, the `other_lock` is a variable which depends on which thread *owns* the neighboring particle. By assigning ownership of particles to threads operating on a subrange, it can easily be decided whether there is a write conflict or not. If the owning thread is not equal to the processing thread, `other_lock` must be equal to the `update_lock` from the owning thread. Otherwise, the neighboring particle is inside of the thread's subrange, which means there is no conflict. It therefore only has to use `update_lock` as its `other_lock`. This indirection makes the common case, which is not having any conflict fast, while rare conflict cases lead to two threads synchronizing over a lock.

Listing 10.8: Using custom locks to

```
omp_set_lock(update_lock);
f[i] = ...;
omp_unset_lock(update_lock);

if( /* use newton 3 */  ) {
  int responsible_tid = ...;
  if (responsible_tid != tid) {
    // other core, acquire lock and update
    other_lock = ...;
  } else {
    other_lock = update_lock;
  }

  omp_set_lock(other_lock);
  f[j] = ...;
  omp_unset_lock(other_lock);
}
```

10.3.3 Avoiding data races through partitioning

Assigning ownership of particles to threads and therefore limiting synchronization in conflict cases was taken one step further. A lock-free implementation is possible, which allows avoiding large portions of duplicated force computations, but at the same time avoid synchronization.

By introducing particle partitions to an MPI subdomain, working sets can be created which can be processed almost independently by multiple threads. Each thread works only on particles that are assigned to its partition. Since collisions of particles in the same partition do not introduce any write conflicts, no synchronization mechanisms such as locks or critical sections are needed. This enables use of Newton's third law within a partition and avoids duplicated force calculations.

To store the thread assignment the existing default atom vector style `AtomVecSphere` used by LIGGGHTS is replaced with a derived class `AtomVecSphereOMP` which allocates an additional integer array `thread`. Contact partners which are not in the same partition can then easily be determined by comparing their partition assignments. For such cases, two different strategies were implemented to resolve write conflicts:

Postponing force updates: Similar to array reduction, one way of avoiding write conflicts is giving threads their own separate memory region to work in. Instead of allocating arrays for forces and torques for all particles for each thread, each thread can only store a list of force updates. A force update contains the memory locations of both force and torque entries which need to be updated and as well as force and torque contributions.

During computation each thread uses this list to store force updates to particles which are not in the same partition. After all threads have finished processing their respective work assignments, these update lists are then traversed serially and merged into the global force arrays to account for the missing contributions. Because of the low number of conflicts, compared to the number of regular collision pairs, this solution is a good compromise. It is similar to array reduction, but avoids allocation of large force arrays for all particles by each thread.

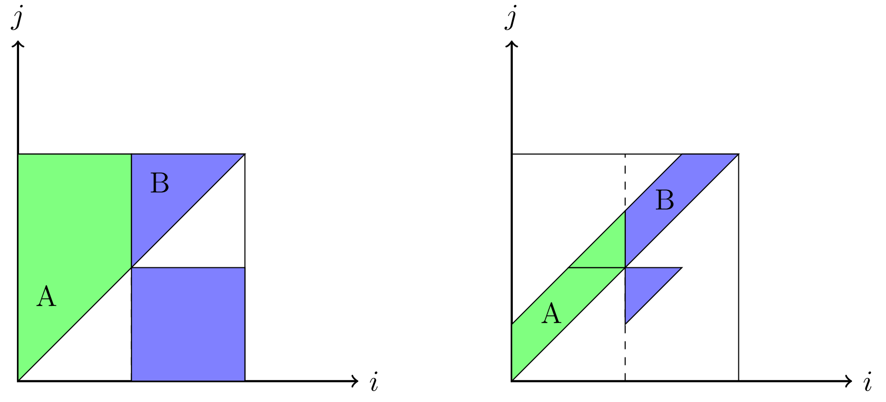
Listing 10.9: Implementation of the force loop which postpones conflicting updates

```
bool same_thread = atom->in_thread_region(tid, j);

f[i] = ...;

if(same_thread) {
  // same thread, use partial newton
  f[j] = ...;
} else {
  updateList->push_back(ForceUpdate(&f[j], &torque[j], forces_j));
}
```

Work-Duplication for conflict cases: Another solution is not to let threads apply forces to particles which are not their responsibility and to avoid writing to another thread's memory. Rather, if the two particles live in different partitions, the threads of both contact partners compute the force of the contact. Figure 10.5 illustrates the modifications necessary in the iteration space of the force loop. Neighbor lists which only consider the upper triangle of the force matrix must be extended accordingly to include these additional force computations when necessary. This method therefore effectively duplicates work in conflict cases, but it avoids costly thread synchronization.



(a) Extended half iteration space (b) Extended optimized iteration space

Figure 10.5: Work duplication in conflict cases requires modification of the particle neighbor lists. Each thread must recompute force contributions if a neighboring particle belongs to another thread. For interactions with neighbors in the same partition Newton's third law can be used. Figure 10.5a shows which forces are computed by two threads if the entire half iteration space is traversed. The remaining iterations due to neighbor lists and spatial sorting are illustrated in Figure 10.5b.

Both work duplication and postponing force updates lead to deterministic results. This means that the resulting forces stored as double floating-point numbers are not dependent on the thread scheduling. They may still differ from those in serial runs, but this is due to differences in the ordering of operations between the serial and threaded executions.

No significant performance difference has been observed between the two implementations. All results presented in Chapter 11 were collected using the strategy which duplicates force computations when necessary. Listing 10.10 shows a pseudo-code implementation of the resulting parallelized particle-particle force loop. Although it closely resembles granular force kernels, which are well known in the published literature, this multi-threaded version of the code is free of any explicit thread synchronization due to partitioning and uses adapted neighbor lists which duplicate work in conflict cases.

Listing 10.10: Pseudo-code of the parallelized particle-particle force loop. Threads work only on particles in their partition. The force F_{ij} is applied to particle i and $-F_{ij}$ to particle j if both particles are in the same partition. Particle i will not be in the neighbor list of particle j in that case. Otherwise, i and j are in each other's neighbor list, and F_{ij} and F_{ji} are computed by separate threads.

```
#pragma omp parallel
for(each particle i in partition of current thread) {
  for(each neighbor particle j) {
    // neighbors j are particles in the same partition
    // whose contact F_ji force has not been
    // computed yet as -F_ij AND
    // particles which live in other partition

    // compute F_ij

    // apply force F_ij to particle i

    // check if particle j is in same partition as particle i
    if(thread[j] == thread[i]) {
      // apply force -F_ij to particle j
    } else {
      // do nothing.
      // newton’s third law inactive,
      // will be recomputed as F_ji by thread owning partition of j
    }
  }
}
```

The overhead of duplicating work and load distribution depends on the choice of partitions used for this strategy. Finding partitions which minimize overlapping contact pairs can reduce the workload. However, the effort of finding *optimal* partitions has to be kept in check to avoid losing the benefits of better load-balancing.

10.3.4 Dynamic Partitioning and Load-Balancing using the Zoltan Library

A general approach to distributing the particle-particle collision workload across threads was implemented using existing partitioning algorithms available in the Zoltan library [82] [83]. This library - developed by Sandia National Laboratories - contains many traditional graph-based and geometric decomposition algorithms. Use of the library requires implementing callback functions, giving access to necessary data such as particle locations. All algorithms return a set of lists which can then be used to generate a mapping of particles to threads.

By default, we use Recursive Coordinate Bisection (RCB) for partitioning [72]. This algorithm gradually introduces cuts in space, thus dividing partitions recursively into equal halves, as illustrated in Figure 3. Since not all particle contacts generate equal amounts of work, each particle is given a weight proportional to the number of contact partners. This allows Zoltan RCB to balance particle partitions based on their weights. Zoltan returns a list of particle IDs and their corresponding partition IDs, which each particle then stores. Each partition ID corresponds to the thread that is responsible for that partition.

By periodically updating partitions and using the number of contact partners as feedback, this method keeps the total number of particle-particle contacts evenly distributed across threads.

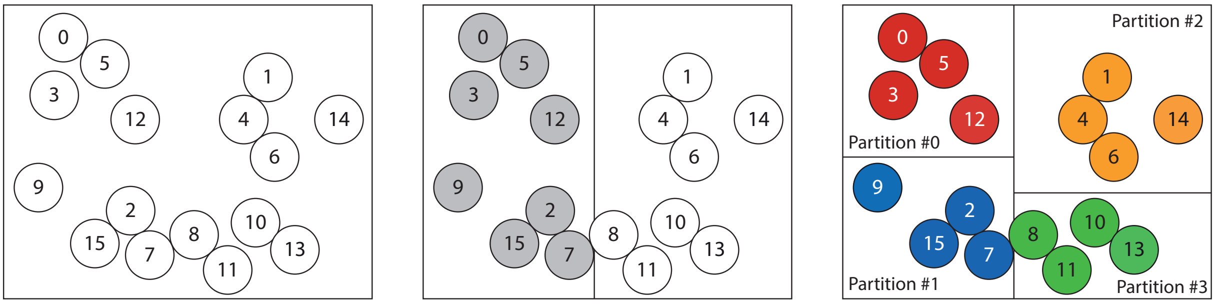
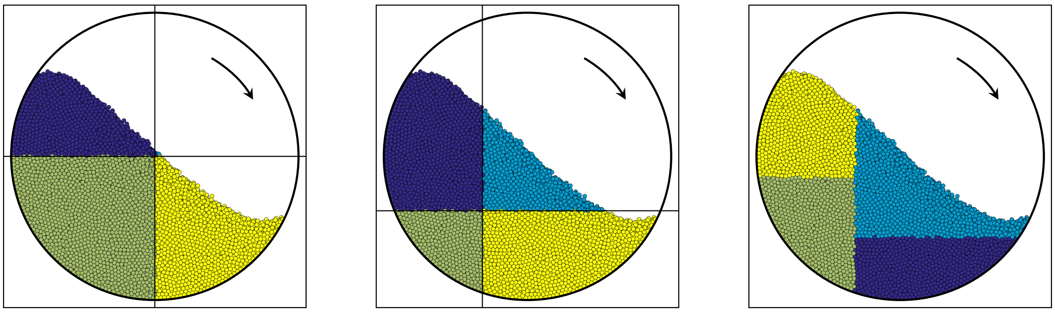


Figure 10.6: Illustration of the Recursive Coordinate Bisection (RCB) algorithm. It gradually partitions space by inserting cut planes, balancing the number of nodes based on their weights.

An example application of RCB load-balancing can be seen in Figure 10.7. In this rotary kiln simulation particles will distribute unevenly over time. A regular grid of MPI processes contains one idle processor. Existing MPI load-balancing will improve the situation, however the added flexibility of RCB allows finding an even better distribution.

Partitioned Spatial Sorting

After assigning each particle to a partition, particle data should be rearranged. Keeping particle data sorted by partitions allows threads to work on contiguous chunks of memory, which improves cache performance. However, partition-based sorting alone does not have a lasting effect. The spatial sorting algorithm available in LIGGGHTS would destroy any artificial particle ordering. It rearranges all local particle data such that it optimizes the access pattern based on their position.



(a) MPI Decomposition (b) Load-Balanced MPI (c) OpenMP with RCB

Figure 10.7: Rotary kiln simulation distributed among 4 processes or 4 threads.

For this reason, the global spatial sorting of particle data had to be replaced. Sorting particles by partitions improves data locality to some extent, but this alone does not achieve the same quality as explicit spatial sorting. Depending on the number of particles assigned to each thread, particle data might still be scattered inside a partition. Neighboring particles need not necessarily be placed next to each other in memory. This can lead to sub-optimal performance and should be avoided.

To resolve this issue, partition-based sorting and spatial sorting were therefore combined into what we call *partitioned spatial sorting* (PSS). The major difference in this algorithm is that it takes partition boundaries into consideration. The procedure is visualized in Figure 10.8 and rearranges data which is taken from the scenario shown in Figure 10.6. All sorting operations are performed on integer arrays which contain only particle IDs. After partition-based sorting of these particle indices, spatial sorting is applied to each partition separately. This generates a permutation vector for each partition, and these are then merged to form a global permutation vector. Since the original spatial sorting algorithm [20] also produces such a global permutation vector, data migration then occurs using the same data migration routines.

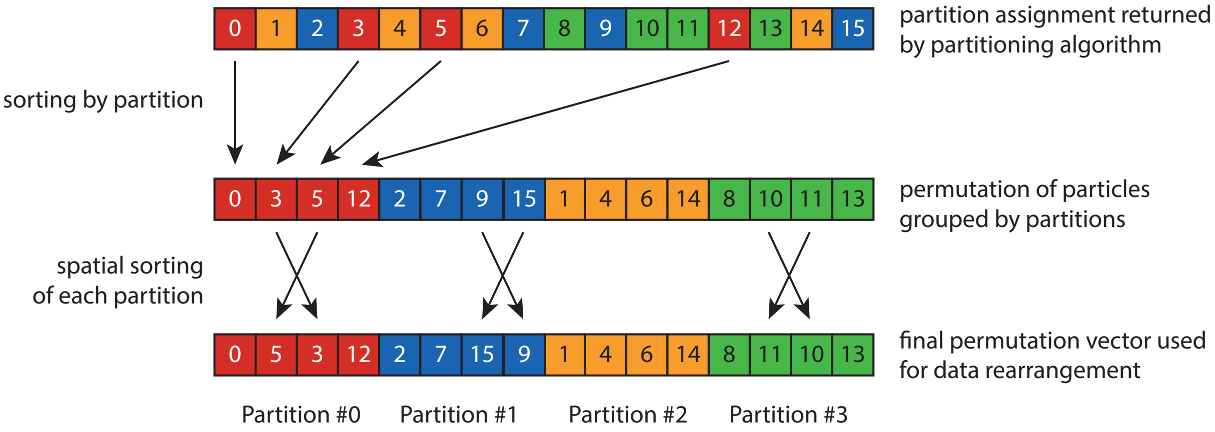

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
partition assignment returned by partitioning algorithm
sorting by partition
0 3 5 12 2 7 9 15 1 4 6 14 8 10 11 13
permutation of particles grouped by partitions
spatial sorting of each partition
0 5 3 12 2 7 15 9 1 4 6 14 8 11 10 13
final permutation vector used for data rearrangement
Partition #0
Partition #1
Partition #2
Partition #3

Figure 10.8: Distributing all particles across partitions of equal size is done by Zoltan, but particles must then be sorted by partition to optimize performance per thread. However, the generated particle order is still not optimal. Partitioned Spatial Sorting; each partition is spatially sorted separately, creating a permutation vector which aligns neighbor data in a cache-efficient way.

Avoiding continuous repartitioning

A weakness of the partitioning approach is the requirement that these partitions be valid at all times. Partitions are defined as ranges inside of global particle arrays. Any change in the domain boundaries, or insertion or deletion of particles will trigger re-neighboring of particles and must therefore prompt a partitioning update.

Particles which enter a new subdomain must be assigned to a partition. At the same time, particles leaving a subdomain will create gaps in global particle arrays, which are then filled by migrating elements from the end of the global array. This causes particles of the last partition to suddenly become part of a different partition. Since partition-aware algorithms assume a certain data order, these changes need to be cleaned up by sorting data and reestablishing partition-based particle ordering.

Triggering a full repartitioning during all of these events would be too time consuming. To avoid the computational cost of continuously repartitioning all particles, full repartitioning only happens at a certain frequency. In the meantime, new particles are assigned to existing partitions. Afterwards, the particle arrays are kept in order by using our adapted version of spatial sorting. Note that, while this strategy is significantly faster than full repartitioning, the continuous sorting of particle data adds an overhead to this partitioning approach.

10.4 Multi-threaded neighbor list building in LIGGGHTS

Apart from computing forces themselves one of the most costly operations occurring during a LIGGGHTS simulation are neighbor list builds. In an MPI parallelized simulation neighbor lists are by definition smaller and their computation is distributed among multiple processors. In a multi-threaded execution there is, however, only one large neighbor list. To achieve similar speedups as with MPI in this part of the code, building neighbor lists has to utilize multiple threads.

As mentioned in Section 2.3 these algorithms have been heavily optimized in the past. Because of the memory intensive operations the caching effects become noticable in these algorithms. Spatial sorting improves the data locality for these operations.

During neighbor list building particles are first assigned to bins. Using these bins the neighborhood of a particle is traversed and potential neighbors, which are within the Verlet cutoff, are stored in a global data structure. While distributing the detection of neighbors among threads is trivial, storing the results in a shared and compact global data structure is not.

The main operation during such a build - besides neighbor detection - is the allocation of memory to store neighbor information. Two types of data have to be allocated:

Particle Neighbor list: Each particle must store a list of detected neighbors, which is a sequence of integer numbers specifying their index inside the global particle array. The size of this list is unknown prior to the build. But a valid assumption can be made on how many neighbors at most each particle can have.

Contact History with each neighbor: Each potential neighbor requires the allocation of a contact history vector. The length of this vector is defined by the selected contact model and consists of double-precision floating-point numbers[1]. After allocation of the contact history vector, previous data has to be copied from the old contact history into the new structure.

Memory allocation is the most expensive operation in the neighbor list algorithm, which is why the serial implementation minimizes it by allocating large blocks of memory at once. These memory pages are gradually filled up by neighbor data. This reduces the burden of allocation overhead, but introduces a critical operation inside of the neighbor list build loop. The neighbor list and contact history themselves are vectors of pointers to each allocated data block.

The allocation of blocks of memory is managed by a utility class called `MyPage<T>`. It hides the usage of memory pages and returns consecutive chunks of allocated memory. Allocation is then done in two steps. First the requested amount of memory is allocated and a pointer returned. In a second step the calling code, which requested the chunk of data, can specify how much of the memory was actually used. This allows the allocator to reuse unused portions of memory during the next allocation.

---

[1] While LAMMPS also contains a history implementation it is limited to three shear values. In LIGGGHTS contact histories can contain an arbitrary amount of data.

Two instances of this utility class are used during neighbor list building. One instance of `MyPage<int>` manages the allocation of integer sequences to store the particle neighbor lists. A second instance of `MyPage<double>` is used to retrieve consecutive chunks of double floating-point numbers to store contact history data.

Figure 10.9 shows how such allocators are used during a neighbor list build. First sufficient memory is requested to hold information of all potential neighbors. After detecting the actual number of neighbors this information is given back to the allocator and the unused memory portions can be reused for the next particle.

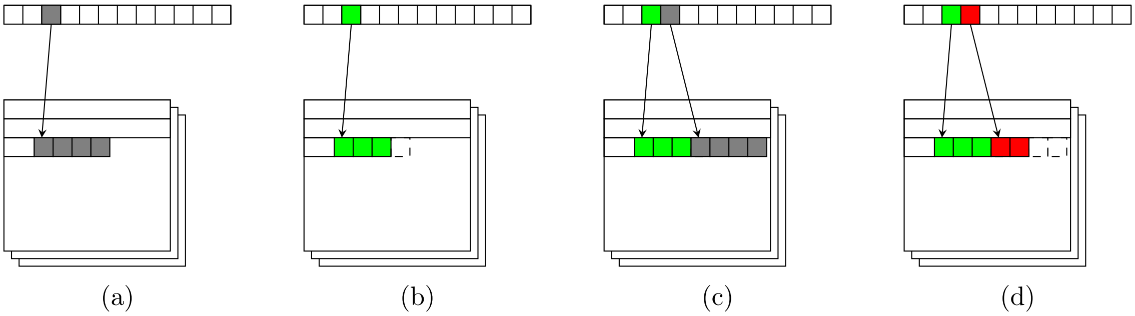


Figure 10.9: Memory allocation during neighbor list building using memory paging. (a) each particle gets enough memory space to store the maximum amount of neighbors. (b) the actual neighbors are stored and used space reported back to the allocator. (c) the allocation of the next neighbor list then starts at the beginning of the previous list. (d) this is repeated until all neighbor lists are generated.

If this allocation scheme would be kept in a multi-threaded version of the code, each thread would need exclusive access to these two page-based allocators. Only one thread can be allowed to retrieve new chunks of memory at a time. This would lead to threads synchronizing after each neighbor detection.

To avoid this situation the OpenMP version of the neighbor list build duplicates these allocators for each thread. Threads can then operate independently and allocate their own memory pages without interfering with others. The global neighbor list becomes a list of pointers to data living in the pages owned by each thread, as illustrated in Figure 10.10.

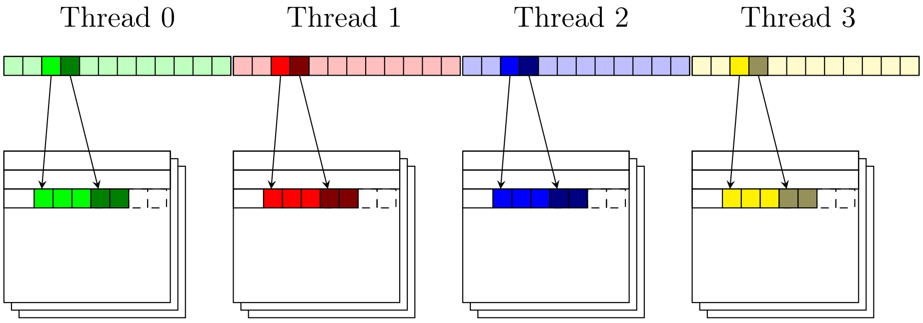


Figure 10.10: Multi-threaded generation of neighbor list using a page data structure for each thread. Each thread works on a sub-sequence of the particle list using its own memory pages to allocate neighbor lists and contact history information.

10.5 Multi-threading in particle-wall interactions

Particle-wall interactions are the second most expensive computation in LIGGGHTS which can benefit from multi-threading. The ratio between the number of mesh triangles and particles determines whether particle-particle or particle-wall interactions dominate the total computation time. Because of the code refactorings described in Chapter 6 both interactions now share the same force kernels. In general they also share many characteristics such as the usage of neighbor lists and contact histories. However, their implementation differs greatly.

As mentioned in Chapter 5, many classes make up the mesh implementation. Different parts must come together to enable the computation. Adding multi-threading to this code base required modifying many of these components to achieve a benefit. It also triggered optimizations described in Section 8.1 and 8.2.

In general the tasks performed by the collection of objects in the mesh implementation can be summarized as follows:

1. Transform and synchronize mesh triangles and associated data at the beginning of each time step
2. Generate a neighbor list of particles for each mesh triangle. This occurs right after particle-particle neighbor lists are created
3. Bring each mesh contact history into a reset state. This means that prior to the force computation the entire contact history of a mesh is marked for deletion.
4. Perform force computation and mark which contact history values should be retained
5. Clean up contact history by removing data from no longer existing contact pairs which are still marked for deletion.

Writing a multi-threaded version of these tasks required adding multi-threaded variants of all the responsible subcomponents. Some of these changes were trivial, such as the parallelization of mesh transformations, which operate on independent sets of data. Other parts, such as the mesh contact history, had to inherit the property of thread-local allocations, as described in Section 10.4. The remaining sections will describe how mesh components must individually operate in a load-balanced way, while accessing global data structures in a thread-safe manner.

10.5.1 Mesh wall computations with OpenMP

Parallelizing mesh computations using OpenMP shares many common features from particle-particle interactions. They are implemented by `FixWallGranOMP` which subclasses `FixWallGran`. Internally it uses an instance of `Walls::Granular` which contains an instance of the selected contact model.

While each individual unit of work can be performed in parallel by applying common contact models, write conflicts limit the exploitable parallelism. The wall implementation also requires the generation of a neighbor list which not only reduces the computation overhead but also influences the allocations necessary for its contact history.

Unlike the particle-particle case, which introduces write conflicts due to Newton's third law, write conflicts in mesh walls are caused by the fact that particles may collide with more than one triangle at any given time. Since forces are stored on a per-particle basis, processing multiple triangles in parallel can lead to force contributions for the same particle. These accesses must either be protected by locks or critical sections, or completely avoided in the first place. Similar to the approach taken for particle-particle contacts, the introduction of particle partitions allows avoiding conflicts. Each partition of particles is only processed by a single thread. While all threads process all meshes, all triangles and their neighbors, force contributions are only computed if a given particle is part of a thread's partition. Listing 10.11 shows the simplified version of the threaded granular wall force loop.

Listing 10.11: Pseudo-code of parallelized particle-wall force loop. All threads visit all meshes and their triangles, but only compute forces for particles which are in their partition.

```
#pragma omp parallel
for(each mesh wall)
  for(each triangle in mesh wall)
    for(each particle neighbor of triangle)
      if(particle in partition of current thread) {
        // compute force of triangle-particle collision
      }
```

Using particle partitions for meshes introduces a new load balancing problem and influences the manipulation of contact histories. Partitions created using Zoltan RCB and PSS described in Section 10.3.4 should not be reused for particle-wall contacts. This is because the workload of wall computations differs from the particle-particle case. Partitions obtained by RCB distribute the workload based on particle positions and the number of neighboring particles. Using the same partitions for particle-wall computations is not desirable because it does not consider the number of wall contacts, which is proportional to the wall workload. If on the other hand wall contacts were considered during RCB, the resulting partitions would produce a less efficient spatial sorting of particles, and the workload during particle-particle computations becomes less balanced. Partitions used by particle-particle and particle-wall computations therefore need to be decoupled.

The generation of particle partitions occurs during neighbor list building. Section 10.5.2 explains that this operation itself is parallelized and needs load-balancing. How the resulting neighbor lists are used to generate load-balanced partitions of particles is described in Section 10.5.3.

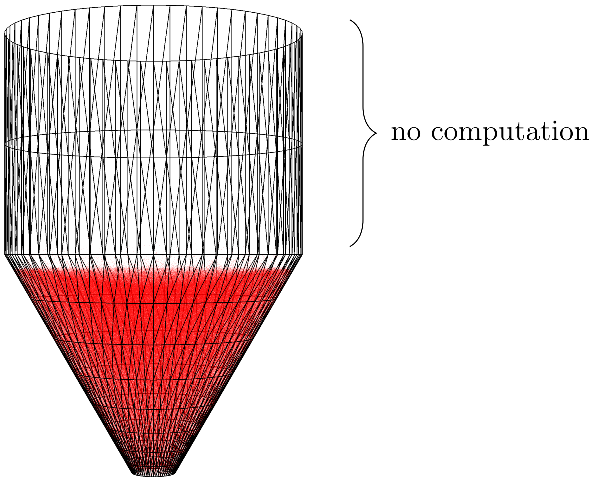


Figure 10.11: Silo simulation where only the lower parts of the mesh interact with particles. This leads to an uneven workload among mesh triangles during neighbor list building and force computation.

10.5.2 Mesh Neighbor List building with OpenMP

The total workload of the particle-wall interactions depends on the number of meshes, triangles per mesh and proximity of particles to triangles. This is evidently the case in simulations such as a silo discharge seen in Figure 10.11. This workload can be reduced through using neighbor lists to avoid checking unnecessary particle-triangle combinations. In Section 8.2 the serial implementation and optimization of the existing algorithm have been described. Each triangle of a mesh is inspected by traversing a limited list of particle bins. Particles which are in the neighborhood of a triangle are stored in the neighbor list. Checking each triangle can be performed in parallel by letting threads operate on a subset of mesh triangles.

Adaptations to avoid write conflicts

The serial implementation of mesh neighbor list building algorithm not only determines which particles are near each triangle, it also computes a total number of triangles interacting with each particle. This information is needed by the mesh contact history to allocate enough memory for each particle contact. Due to its dynamic nature and the migration of particle data across processor boundaries the number of neighbors is an integral quantity managed by an additional fix of type property/atom. This class takes care of the necessary communication overhead. During neighbor list building the number of neighbors is reset to zero and increased while looking for particle neighbors of mesh triangles.

If triangles are processed in parallel, multiple triangles might have to be added as neighbors of the same particle. Therefore, accessing this neighbor counter becomes a potential conflict in the parallel case. To avoid this conflict the neighbor list build is split up into two phases:

1. Generate triangle neighbor lists
2. Generate list of particle neighbors based on triangle neighbor lists

First each thread operates on a list of triangles and produces a list of particle indices of its neighbors. While doing this, the neighbor count of a particle is not modified. After all threads complete their work assignment the generated lists are used to determine the number of neighbors for each particle by traversing them and incrementing the neighbor counters.

Load-Balancing of neighbor build

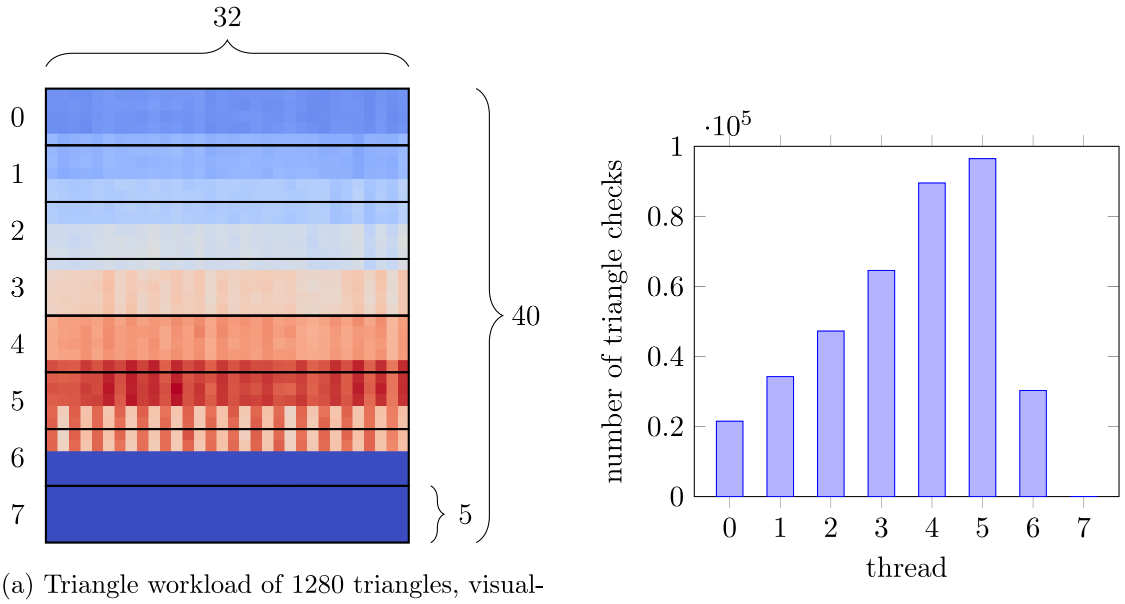
Building neighbor lists of different triangles in parallel easily leads to an unbalanced workload. Not all triangles need to perform the same amount of neighbor checks. A triangle in a dilute region of the simulation domain will have to perform fewer checks compared to a triangle in a dense region of particles. Load balancing of these triangle neighbor checks is therefore critical in order for all threads to be used efficiently.

Figure 10.12 illustrates the workload which has to be processed in a silo discharge simulation with 1.5 million particles on a mesh with 1280 triangles (see Figure 10.11). The relative workload of each triangle is visualized in Figure 10.12a. It shows the list of 1280 triangles as a 32x40 matrix, filled from left to right, top to bottom. Each triangle is represented by a square colored using a heat-map color scheme. Triangles with blue coloring do not have to traverse any particles, while red triangles need to check the most. About 400,000 particles are processed in this example during neighbor list build. Individual mesh triangles can have up to 700 neighboring particles in this geometry.

If this sequence of triangles is processed in parallel by multiple threads using a static schedule, each thread visits a consecutive chunk of triangles. Figure 10.12a shows how 8 threads will process this workload. Each thread operates on 160 triangles, which is a 32x5 block in this visualization. The total amount of neighbor checks performed by each thread are summarized in Figure 10.12b. It demonstrates that the workload is unevenly distributed. While one thread only performs 5 neighbor checks, which are not even recognizable in this chart, another thread processes about 100,000 particles.

What is therefore done is to load-balance the neighbor list building itself by scheduling which triangles are processed by which threads. OpenMP work-sharing constructs could be used for this in combination with a dynamic scheduler or the OpenMP tasking model. However, to avoid non-determinism and overheads introduced by these OpenMP features, a simple and deterministic solution was implemented.

At the beginning of the simulation an initial work schedule of triangles is created consisting of a list of ascending triangle indices. This list is then periodically sorted by the number of neighbors of the corresponding triangle in descending order. The sorting itself is done using an in-place QuickSort algorithm to avoid additional allocations. By doing this the work schedule contains a sequence of triangles which is sorted from the highest to the lowest workload.



(a) Triangle workload of 1280 triangles, visualized as 32x40 matrix filled from left to right, top to bottom

(b) Workload distribution among 8 threads with static schedule (160 particles per thread)

Figure 10.12: Neighbor list workload of a silo discharge simulation with 1.5 million particles on a mesh with 1280 triangles. About 400,000 particles are checked during mesh neighbor list building. (blue = no checks, red = max. number of checks)

While building neighbor lists, this sorted work schedule is used to implement a static round-robin scheme. Strided access to the triangle array is used for this implementation. A visual interpretation of this procedure is to reinterpret the sorted one-dimensional sequence as a 2D matrix as has been done in Figure 10.13a for the silo discharge example. Threads then process columns of this matrix using the strided access pattern, which balances the workload.

The implementation uses this visual interpretation to compute meaningful matrix dimensions and a stride length. The shape of this matrix is chosen to be approximately square, which is why the initial side length is computed as:

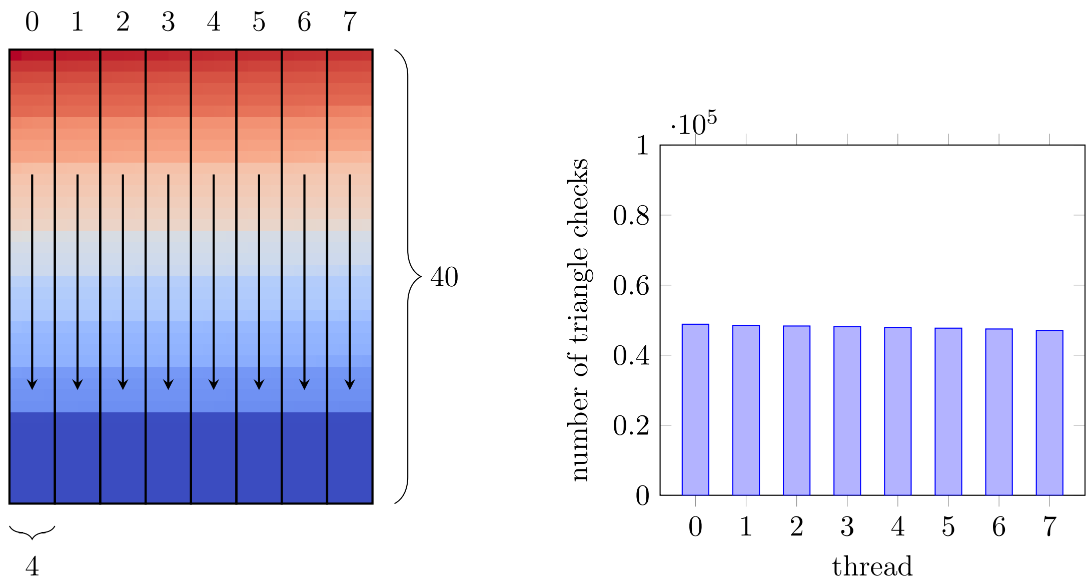
$$n_{side} = \lfloor \sqrt{N_{triangles}} \rfloor + 1$$

This side length is then divided into $n_{threads}$ equal parts which determines the number of elements n_{width} processed in a row by each thread.

$$n_{width} = \left\lfloor \frac{n_{side}}{n_{threads}} \right\rfloor$$

Multiplying n_{width} with the number of threads returns the final width of the matrix and the stride length n_{stride} used for processing the sequence. Note that due to integer division and rounding this number will be smaller or equal to the initial width n_{side}.

$$n_{stride} = n_{width} \cdot n_{threads} \leq n_{side}$$



(a) Sorted workload processed by threads using strided access

(b) Workload distribution after sorting and load-balancing through strided access

Figure 10.13: Sorted and load-balanced neighbor list workload of the same silo discharge simulation as in Figure 10.12

The prodecure is demonstrated using 8 threads for the silo discharge example in Figure 10.13. The workload sorted by triangle neighbors can be seen in Figure 10.13a. In this case 1280 triangle results in a choice of $n_{side} = 35$, which in turn leads to a column width n_{width} of 4 and a stride length n_{stride} of 32. It the resulting 32x40 matrix each thread only processes a column of width 4. Because of the initial sorting, this access pattern then distributes the workload among threads evenly, as can be seen in the accumulated triangle checks in Figure 10.13b.

The achieved distribution is by no means optimal, but sufficiently fair and computationally cheap. By periodically resorting the triangle schedule based on neighbor counts the dynamic schedule allows to react to changes in the simulation domain.

Load-balancing and multi-threading incur overheads which not always pay off. It is only used if the number of triangles exceeds a heuristic threshold. E.g., using threading is enabled if the number of triangles exceeds a number proportional to the number of threads in the system.

10.5.3 Load-balanced particle partitions for particle-wall contacts

With triangle-particle neighbor lists in place partitions of particles can be created which enable a lock-free parallelization of the wall computation.

The chosen approach generates partitions right after building the neighbor list of a mesh. At this point in time the expected workload caused by processing each particle can be approximated by its number of neighboring triangles. This information has been generated at this point because it is needed for the contact history.

To create a load-balanced work schedule of particles, a list of particle indices is created containing only particles which have at least one triangle contact partner. In general this list is much shorter than the total number of particles in the system.

Given this reduced set of particle indices these are then sorted by the number of contact partners in a descending order. By assigning each element of the resulting list to a thread using a round-robin scheme the workload is distributed among threads in a coarse way, similar to the load balancing described in the previous Section 10.5.2.

In addition, the particle indices are finally sorted using stable sort into consecutive sequences by their thread assignment. These consecutive sequences can be used to quickly access all particles affected by one thread, e.g. to manipulate the contact history. The thread assignment itself is further stored in an array for each local particle. This allows easy constant time look-up if a particle is within a thread partition. While these helper structures increase the memory footprint of the application, the added benefit of better load balancing outweighs the cost of additional memory usage.

Having a short sequence of all particles processed by a thread also allows to quickly move on if a mesh does not have any contacts at all. In the original serial implementation this was done by traversing all empty neighbor lists of all triangles in each mesh. Detecting lack of work on a mesh was simplified by checking if a thread's partition is empty.

10.6 Summary of the MPI/OpenMP Hybrid Parallelization

The MPI/OpenMP hybrid parallelization allows each MPI process to spawn multiple OpenMP threads on top of the existing MPI domain decomposition. This is controlled either by the LIGGGHTS input script or through the OpenMP environment variable `OMP_NUM_THREADS`. Since for better performance threads should not migrate between cores, thread-pinning was implemented and the development was based on the GCC 4.8-4.9 with OpenMP 3.1. By relying on the fact that this OpenMP implementation internally uses the pthread library, thread-pinning could be implemented using pthread methods inside of OpenMP parallel regions. In future versions, this could be replaced with the more generic "Places" feature introduced in OpenMP 4.

The previous sections have described the strategies used to enable a scalable OpenMP variant of LIGGGHTS. Through these and the other optimizations mentioned in Chapter 8 the hybrid operation of both MPI and OpenMP was made possible.

The lock-free strategy developed for this hybrid parallelization uses a combination of well-known algorithms to obviate the need for synchronization mechanisms such as locks. This is achieved by additional work for data partitioning and rearrangement. These steps are done during the communication and load-balancing phase of LIGGGHTS and extend the version presented in Section 9.3.3:

1. (Optional) Periodically adjust domain boundaries along one or more axes. If necessary, migrate data.
2. Exchange particles and contact history data between MPI processes
3. **Use RCB to partition local particle data into $N_{threads}$ partitions**
4. **Sort local particle data on each MPI process based on partitions and apply spatial sorting to increase data locality and improve cache utilization**
5. Update particle neighbor lists
6. Exchange mesh triangles and contact history data between MPI processes
7. Update mesh triangle-particle neighbor lists
8. **Distribute mesh triangles based on particle neighbors between threads**
9. **Distribute particles based on triangle neighbors between threads**

Steps 3 and 4 constitute dynamic load balancing of particle-particle interactions across threads. Section 10.3.4 described how particle data is partitioned and load-balanced by the RCB algorithm. It also outlined necessary changes to the existing spatial sorting algorithm and the enforced partition-aligned particle memory layout. Load balancing of particle-wall interactions occurs in Steps 8 to 9, which was explained in Section 10.5.

Chapter 11

Validation of the Hybrid Parallelization

The performance characteristics of the hybrid parallelization were investigated by several test cases. This chapter presents results obtained for these cases which were gathered through extensive automated testing of different decompositions and configurations using the test harness. The parameter space for MPI decompositions and MPI load-balancing settings was explored. This was necessary because choosing an unfavorable MPI decomposition can easily lead to run times that are 2-4x slower than optimal. In each case, the result with the best run time was chosen for a given number of cores. All of these tests were performed on AMD cluster blades, each having two sockets equipped with 16-core AMD Opteron 6272 CPUs. The topology of these processors is visualized in Figure 11.1. Runs up to 32 cores were executed on a single blade, while 64 core runs used two and 128 core runs four blades connected through Infiniband QDR (40Gb/s).

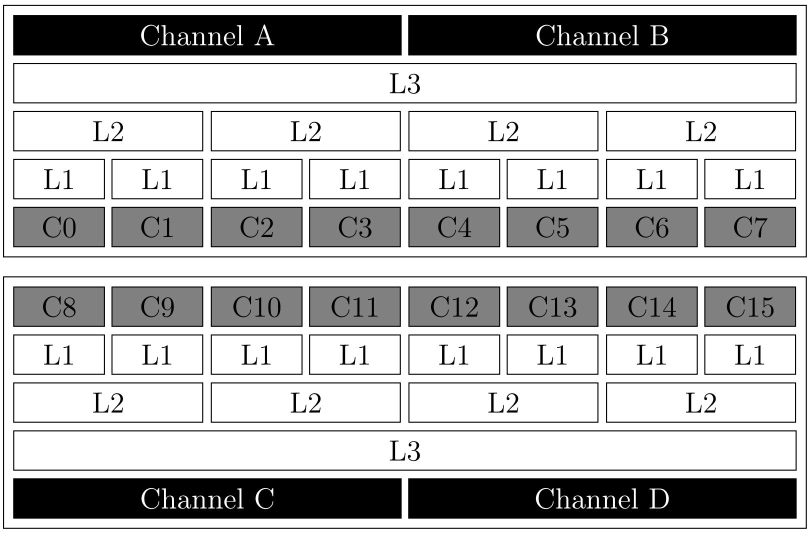


Figure 11.1: Block diagram of an AMD Opteron 6272 processor. This model has 16 cores. It contains two memory controllers, each of which can utilize two memory channels. Each memory controller is backed by a L3 cache (8 MB). These caches are shared among 8 cores. Teams of two cores share a floating-point unit and L2 cache (2 MB). Each core has a 16KB L1 data cache.

Throughout the following text, the scalability and timing of code sections of the LIGGGHTS code are used to characterize the performance of the code. These statistics are produced by

LIGGGHTS itself and include:

Pair time (T_{pair}): Time spent inside of granular particle-particle force computation kernels.

Comm time (T_{comm}): Time spent in the explicit MPI communication portion of the integration loop.

Neigh time (T_{neigh}): Time spent building neighbor lists for particles.

Modify time (T_{modify}): Time spent in parts of the code which manipulate the state of particles, meshes, and other computations such as gravity, integration, and particle-wall interactions.

Output time (T_{output}): Time spent dumping data to disk. These results are neglected in the following sections.

Other time (T_{other}): All remaining time between the end of a time step and the beginning of a new one.

Note that the times measured by LAMMPS/LIGGGHTS are averaged over all MPI processes. This is done because code regions other than MPI communication measured in T_{comm} are not synchronized by barriers, but truly run in parallel. Each MPI process measures the time in each code section independently, and the resulting final time of each code section is then averaged over all processors at the end of the simulation. This *hides* the load imbalance from timings of code sections such as T_{pair} and T_{modify}, but the imbalance becomes visible in T_{other}. Since all processors have to synchronize after each time step, faster processors must wait until all processors have completed. This waiting or synchronization time between time steps is accumulated in T_{other}. It is the result of the total loop time minus the total average time spent in all code sections:

$$T_{other} = T_{total} - (T_{pair} + T_{comm} + T_{neigh} + T_{modify} + T_{output})$$

If all processors spend an equal amount of time in computation, the waiting time in T_{other} will be minimal and only constitute a minimal percentage of the total run time. As T_{other} increases, more processors will be idle and load imbalance will grow. Load imbalance was therefore defined as follows:

$$\text{load imbalance in } \% = \frac{T_{other}}{T_{total}} \cdot 100$$

Note that for executions of the hybrid parallelization, T_{other} also contains the overhead introduced by partitioning. Using this definition, the load imbalances of MPI and the hybrid parallelization can be compared. It also allows to determine if the added overhead of partitioning pays off.

11.1 Box-in-a-box

This first test case illustrates the load-imbalance issue of MPI decompositions. It uses a cubic domain (50x50x50 cm) with planar walls. A static cube mesh (30x30x30 cm) is placed inside at the center of the domain. 50,000 particles with a diameter of 5 mm are then inserted at the top of the domain. These particles hit the cube and pour over the top edges. Figure 11.2 shows a screenshot of this simulation at halftime. Table 11.1 summarizes the parameters of this setup.

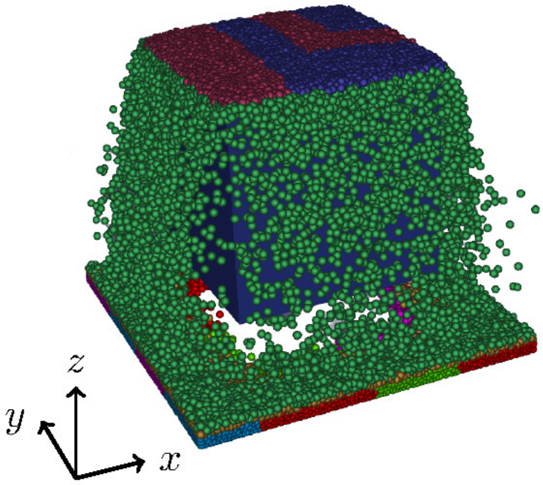


Figure 11.2: Box-in-a-box example; particles are dropped onto a cube, causing them to flow around it. The domain center does not require any computations and causes load imbalance.

contact model	Hooke
Young's modulus	5×10^6 N/m^2
Poisson's ratio	0.45
coefficient of restitution	0.3
coefficient of friction	0.05
characteristic impact velocity	2 m/s
time step	10^{-5} s
particle density	2500 kg/m^3
particle diameter	5 mm
number of particles	50,000
duration	50,000 steps

Table 11.1: Simulation parameters of box-in-a-box example

The reasoning behind this test case is that many collisions occur at the top face of the cube, making it one of the hot zones during computation. Since geometric domain decomposition generates a grid of processors, finding a good decomposition of this problem is hard. Many decompositions lead to MPI processes being forced to work inside the cube mesh, which results in no work being assigned to these processors. Work is also needed along the outer border of

cores	name	decomposition	threads	MPI load-balancing	run time
1	serial	1x1x1	1	-	28m 55s
32	MPI	4x4x2	1	z	3m 23s
32	MPI+OpenMP	2x2x1	8	-	2m 54s
64	MPI	4x2x8	1	xyz	3m 11s
64	MPI+OpenMP	4x2x1	8	xy	1m 52s
128	MPI	16x8x1	1	xy	8m 51s
128	MPI+OpenMP	4x4x1	8	xy	1m 13s

Table 11.2: Configurations used to simulate box-in-a-box example

the domain and at the bottom, where particles hit the floor and move back towards the center of the domain.

The test case was simulated on up to four 32-core blades with a time step of 10 μs. In serial mode, a simulation of 50,000 time steps took about 29 minutes to complete. Figure 6 shows the run times of MPI-only, OpenMP-only and the hybrid MPI/OpenMP implementations utilizing all 32, 64 and 128 cores. Table 11.2 lists which decompositions were used. On 32 cores, an OpenMP-only simulation needed about 4 minutes 18 seconds for completion. The fastest MPI-only simulation on 64 cores needed 3 minutes and 11 seconds. Beyond 64 cores, the MPI-only performance deteriorated. The hybrid parallelization, on the other hand, continued to scale and finished within 1 minute and 13 seconds on 128 cores.

All run times in Figure 11.3 are divided into categories, showing the average amount of time spent in each phase of the integration loop. Note again that, while the total run times shown are absolute times, the timings of the individual categories are results averaged over all MPI processes (except “Other time”).

Most remarkable in Figure 11.3 is the poor performance of MPI-only at 128 cores. To emphasize that this is not caused by poor choice of decomposition, run times of other MPI decompositions are shown in Figure 11.4. A similarly large variation of MPI-only run times was observed for 64 cores, but in this case there was a decomposition that outperformed the 32-core simulation.

Figure 11.3 and Figure 11.4 illustrate that significant time is spent synchronizing and waiting. While T_{pair} decreases linearly between 32, 64 and 128 cores, the long timespan spent outside computation in T_{other} suggests significant load imbalance. Detailed timings further showed that T_{pair} is not uniform across MPI processes. Some processes spend twice the amount of time in pair-wise force computations than others. On average, the time spent in pair-wise force computations was shorter in MPI runs than in both OpenMP-only and the hybrid. However, because of the imbalance, many MPI processes were idle at the end of each time step.

Detailed analysis revealed what happened during 128-core runs: The MPI dynamic domain decomposition constantly tried to shift domain boundaries because of the (intentionally) unfavorable geometric setup of this case. Each process was supposed to be assigned about 400 particles to compute. However, since this was not possible due to geometric constraints, the MPI perfor-

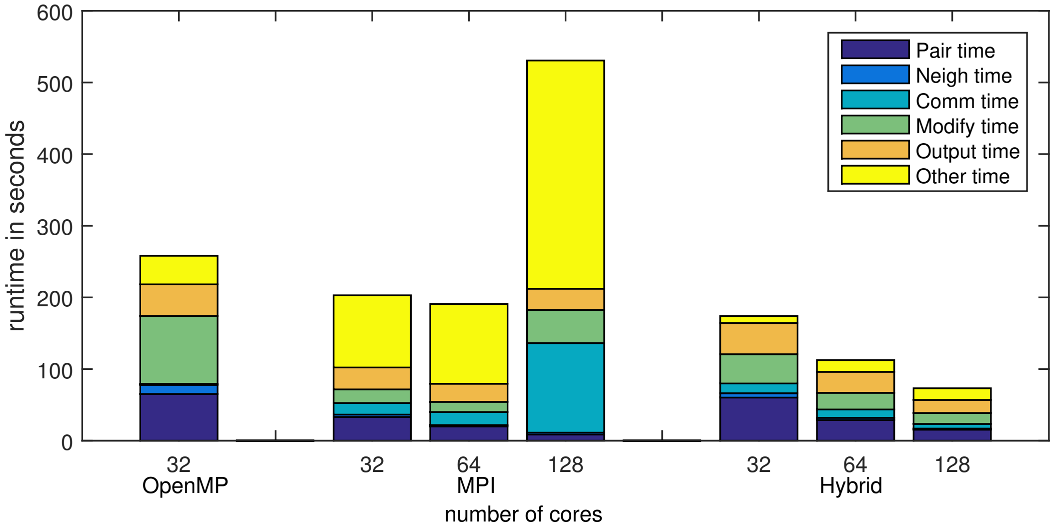


Figure 11.3: OpenMP, MPI, and MPI/OpenMP hybrid runs of Box-in-a-box test case on 32, 64 and 128 cores. The OpenMP-only run suffers from limited memory bandwidth in memory-bound algorithms inside the Modify section of the code. MPI-only has short average run times for each section, but long periods outside computation, which suggests a high load imbalance. Hybrid timings are on average slightly longer, but due to better balancing, processes have shorter waiting times. At 128 cores, MPI domain decomposition no longer scales in this test case.

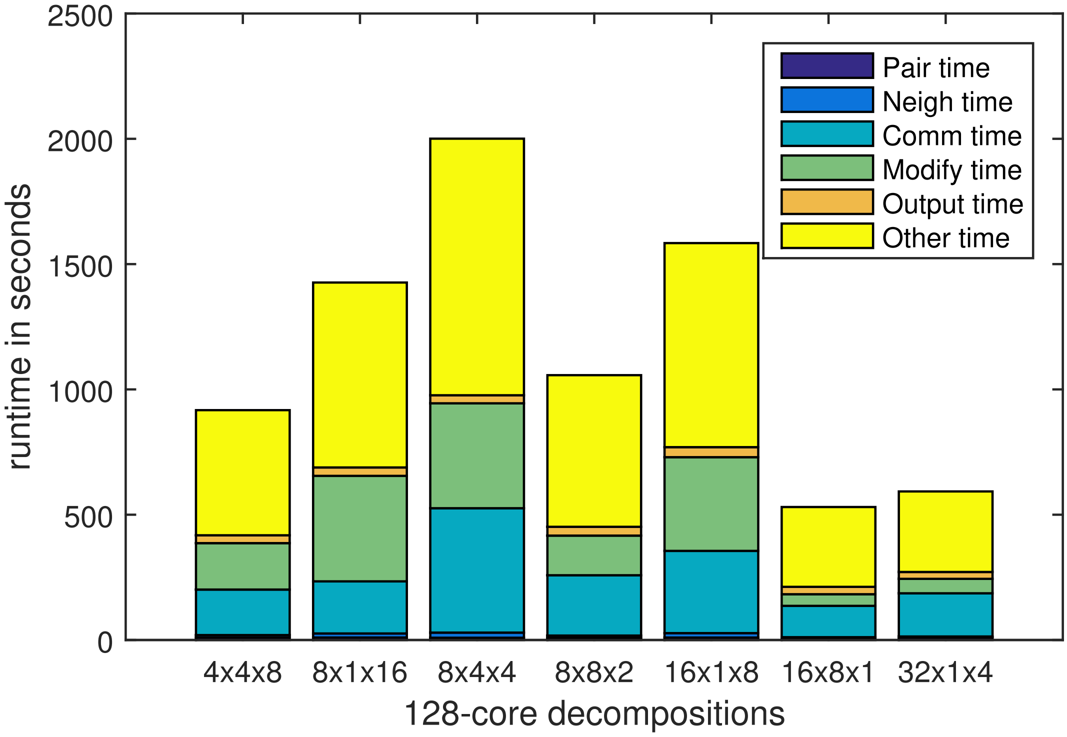
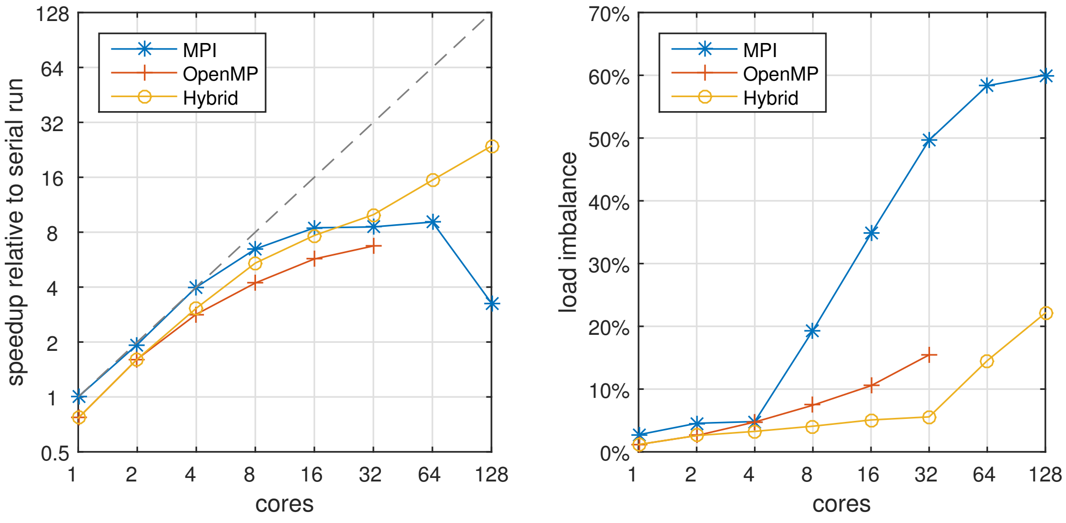


Figure 11.4: Run times of MPI-only simulations using different 128-core decompositions. All decompositions adjusted domain boundaries dynamically at runtime.

mance of LIGGGHTS no longer scaled. Off-node MPI traffic due to data migration and increased synchronization times added up and became more dominant with increasing number of blades.

The hybrid implementation performed better at distributing the workload in this situation. It allowed simpler MPI domain decompositions to be used. This selection was guided by looking at the architecture of the compute nodes used. Each blade had two sockets, each of which contained two teams of 8 cores operating on their own L3 cache. By assigning 4 subdomains to each 32-core blade and binding each MPI process to one of these 8-core teams, the amount of communication between sockets and through memory on each blade was reduced. The hybrid parallelization therefore operated on 4, 8, or 16 teams of 8 cores. Each MPI process then utilized the full 8 cores of each team by spawning 8 threads and applying thread-pinning.



(a) Strong Scalability (b) Load Imbalance

Figure 11.5: Strong scalability and load-imbalance of the Box-in-a-box test case with 50,000 particles. The hybrid parallelization benefits from additional memory bandwidth compared to OpenMP-only. It outperforms MPI when the compute node is fully utilized. The main reason is the increasing amount of load-imbalance in MPI for growing core counts

Figure 11.5a shows the scalability of all three implementations for this test case. All speed increases were measured relative to a serial run, which is equivalent to an MPI-only simulation with a 1x1x1 decomposition. While MPI-only scaled well for lower core counts, finding good decompositions with 8 or more processes becomes increasingly difficult. Because of the large empty region in the center of the domain, the dynamic MPI decomposition algorithm struggled to keep all subdomains busy with work. Figure 11.5b illustrates how load-imbalance becomes more of a problem as the core count increases. While this is less of an issue for the hybrid parallelization, growing imbalance in Figure 11.5b was observed. This is due to the increasing amount of work needed for partitioning with higher core counts.

OpenMP-only runs did not reach the same speedups as their MPI counterparts. One reason is the added overhead introduced by the OpenMP implementation, another the memory bandwidth limitations caused by multiple threads fighting over global memory.

By default, Linux - the operating system in all of these tests - uses a memory policy called first touch. This means that memory is mapped to physical memory not during allocation, but during first access of the first word. Since any global array of data is allocated and initialized by the main thread, it is mapped to the physical memory closest to that processor. This leads to the following bottleneck: memory allocated by threads on one socket cannot be accessed directly by threads on another socket. Any access must pass through the interconnection between CPUs. This means that all memory accesses of threads on a second socket will pass through the processor owning the memory [21].

MPI avoids this problem because each process works on its own smaller memory regions. These automatically map to the closest physical memory and thereby maximize memory bandwidth usage. The hybrid implementation allows us to combine the two approaches and improves performance by using multiple MPI processes working on different memory regions.

In summary, both MPI and OpenMP implementations have very similar speedup characteristics between 1 to 8 cores, with a clear advantage for MPI. However, as the core count increases further the MPI performance becomes limited by load imbalance. The hybrid implementation outperforms the MPI at 32 cores. Due to automatic load balancing and simpler MPI decompositions, the hybrid continues to scale by a factor of 1.5 beyond 32 cores.

11.2 Silo discharge

This test case is a benchmark of a real-world example. 1.5 million particles with a diameter of 1.4 mm are poured into a silo of 40 cm height. The top half of the silo has a diameter of 25 cm, while the lower conical half narrows to a 4 cm diameter. Figure 11.6 shows a cross-section of this silo at the beginning of the simulation. Table 11.3 contains the parameters of this setup.

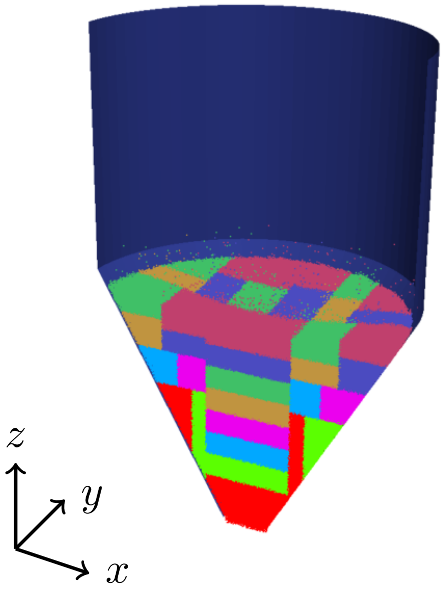


Figure 11.6: Silo discharge; 1.5 million particles filled into a silo are released.

contact model	Hertz with tangential history
Young's modulus	2.5×10^7 N/m^2
Poisson's ratio	0.25
coefficient of restitution	0.5
coefficient of friction (particle-particle)	0.2
coefficient of friction (particle-wall)	0.175
time step	5×10^{-7} s
particle density	1000 kg/m^3
particle diameter	1.4 mm
number of particles	1,500,000
duration	100,000 steps

Table 11.3: Simulation parameters of silo discharge example

After some settling time, the orifice at the bottom is opened, letting all particles escape. This test case has previously been used to illustrate the benefit of MPI load balancing along the z direction. The workload moves from the top portion of the domain downwards. Since the lower part of the silo is conical, the workload is focused on the center region of the lower domain.

Because of the large number of particles, particle-particle interactions accounted for the majority of simulation time. A time step of 0.5 μs was used.

Serial runs of 100,000 time steps took 2 days and 16 hours on one 32-core cluster blade. By fully utilizing all 32 cores of a single blade, MPI reduced the total run time to 3 hours 41 minutes. The hybrid completed in 2 hours 56 minutes. Similarly, when two blades or 64 cores were fully utilized, MPI finished within 2 hours 6 minutes, while the hybrid run only took 1 hour and 41 minutes. In both cases, the speed increase achieved by the hybrid parallelization was about 20%. Increasing the number of cores further broadened the gap between the two implementations. On 128 cores, MPI completed in 1 hour and 49 minutes. The hybrid used a simpler decomposition and finished within 59 minutes, which constitutes a 44% improvement. Table 11.4 summarizes these results and includes further information, such as the decompositions used.

cores	name	decomposition	threads	MPI load-balancing	run time
1	serial	1x1x1	1	-	2d 16h
32	MPI	4x4x2	1	xyz	3h 41m
32	MPI+OpenMP	2x2x1	8	xy	2h 56m
64	MPI	4x2x8	1	xyz	2h 06m
64	MPI+OpenMP	2x2x2	8	z	1h 41m
128	MPI	4x4x8	1	xyz	1h 49m
128	MPI+OpenMP	4x4x1	8	xy	59m

Table 11.4: Configurations used to simulate silo discharge example

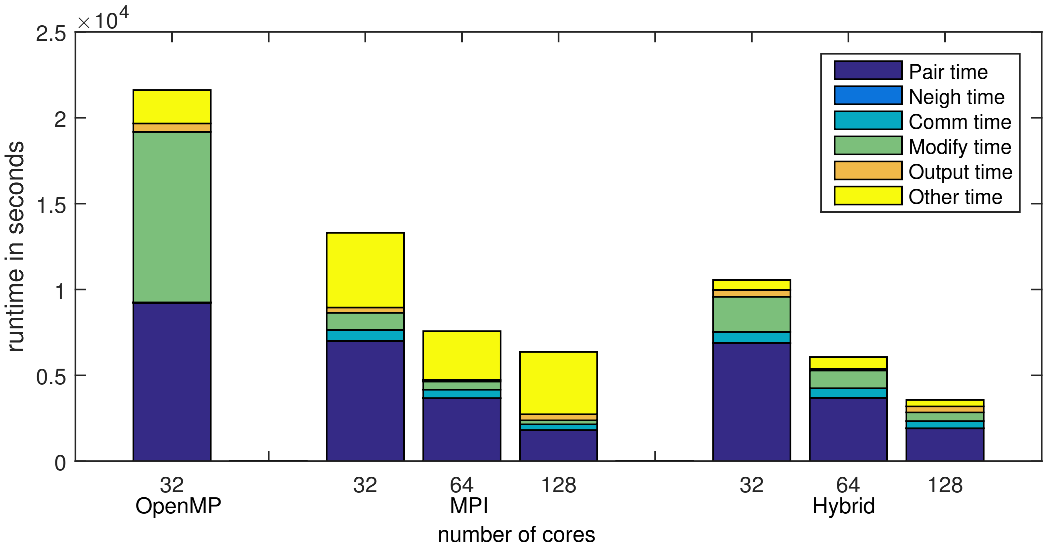
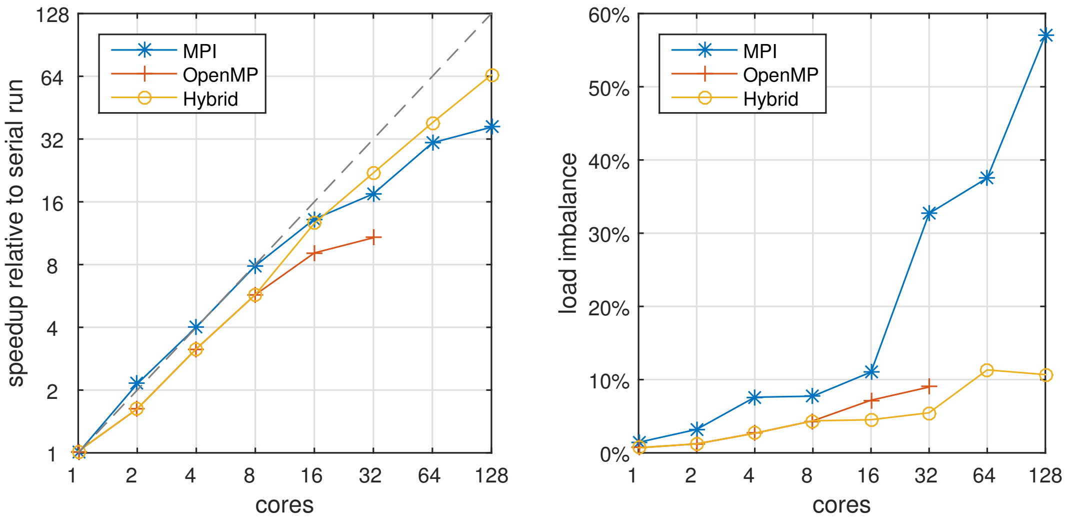


Figure 11.7: Silo discharge results of running MPI, OpenMP and hybrid implementations on 32, 64, and 128 cores. OpenMP-only suffers from limited bandwidth in memory-bound algorithms (Modify timing). Load imbalance of the MPI-only runs, visible as long waiting times, becomes a dominant factor. The hybrid parallelization shows continuous scaling and significantly less load imbalance.

Figure 11.7 shows a detailed comparison of the OpenMP, MPI and hybrid run times. Unlike in the previous case, MPI-only runs continued to scale beyond 32 cores, but longer timespans were, again, spent outside computation. OpenMP and especially the hybrid runs exhibited much less load imbalance. The OpenMP run was added here to show how severe the memory bandwidth limitation is, particularly in larger test cases. Memory-bound algorithms such as integration, which are part of the Modify code section, slow down the simulation significantly.

The scalability of this test case is shown in Figure 11.8a. Again, OpenMP and the hybrid did not reach the speed increases of MPI with core counts up to 8. OpenMP saturated at that point, while the hybrid performed similarly at 16 cores and outperformed MPI at 32 cores. Both MPI and the hybrid then gained a factor of 2x by adding a second 32-core blade. Similar to Figure 11.5b of the first example, Figure 11.8b shows how MPI struggles with load imbalance at higher core counts. In this case, load imbalance was caused by the conical geometry of the silo, which does not map well to the Cartesian grid of subdomains introduced by the MPI decomposition.



(a) Strong Scalability (b) Load Imbalance

Figure 11.8: Strong scalability and load imbalance of the Silo discharge example with 1.5 million particles; OpenMP and the hybrid yield the same results in the 1-8 core cases. Starting with 16 cores, the hybrid also uses MPI processes, resulting in speed increases that are equal to, or better than, those of MPI-only. Increasing load imbalance limits MPI-only scalability

11.3 Mixing process

The final test case simulates a common process found in industry. Granular materials are filled into a mixer of a given geometry. Rotating blades then stir the composition. LIGGGHTS allows simulation of these processes as it supports moving STL meshes. In this example, the mixer consists of multiple blades rotating at a constant speed. The cylindrical mixer used has a length of 4.7 m and a diameter of 1 m. This huge geometry is filled with 770,000 particles with diameters between 9 and 11 mm. Multiple blades rotate around the x-axis inside and stir the particle mixture. The entire test case consists of 12 meshes with a total of 35,354 triangles. Additionally, there are now static and rotating meshes. Figure 11.9 illustrates this setup and the simulation parameters are listed in Table 11.5.

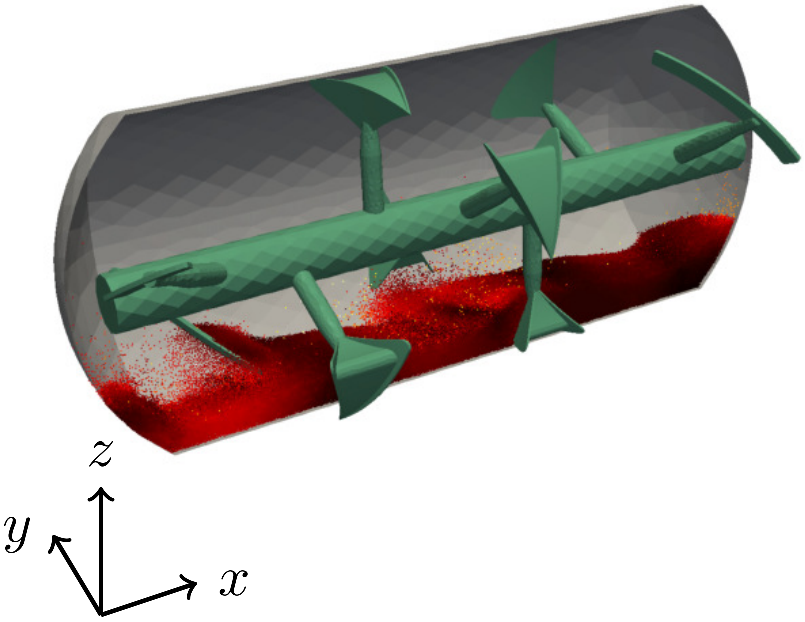


Figure 11.9: Mixing process; 770,000 particles are stirred by a rotating mesh geometry.

contact model	Hertz with tangential history
Young's Modulus	5×10^6 N/m^2
poissonsRatio	0.45
coefficient of restitution	0.9
coefficient of friction	0.05
time step	10^{-5} s
particle density	2500 kg/m^3
particle diameters	9, 10, and 11 mm
particle distribution	uniform
number of particles	772,326
duration	50,000 steps
rotation speed	6 RPM

Table 11.5: Simulation parameters of mixing process example

With a time step of 10 μs, this benchmark took about 15 hours 38 minutes to complete the simulation of 50,000 time steps in a serial run. The MPI implementation completed this benchmark after 47 minutes on 32 cores. With another computing blade added and running on 64 cores, it completed within 40 minutes. With 128 cores, the MPI implementation reduced simulation time to 35 minutes. The hybrid parallelized run, however, needed 52 minutes to complete on 32 cores, and 34 minutes on 64 cores. 20 minutes were needed with 128 cores, which is 42% faster than MPI-only using the same number of cores. The configurations used for these simulations can be found in Table 11.6. Figure 11.10 illustrates these results in detail. It also includes an OpenMP run with 32 cores, which needs twice as much time, again showcasing that OpenMP alone cannot compete.

cores	name	decomposition	threads	MPI load-balancing	run time
1	serial	1x1x1	1	-	15h 38m
32	MPI	8x4x1	1	xy	47m
32	MPI+OpenMP	8x1x1	4	x	52m
64	MPI	8x8x1	1	xy	40m
64	MPI+OpenMP	8x1x1	8	x	34m
128	MPI	16x8x1	1	xy	35m
128	MPI+OpenMP	16x1x1	8	x	20m

Table 11.6: Configurations used to simulate mixing process example

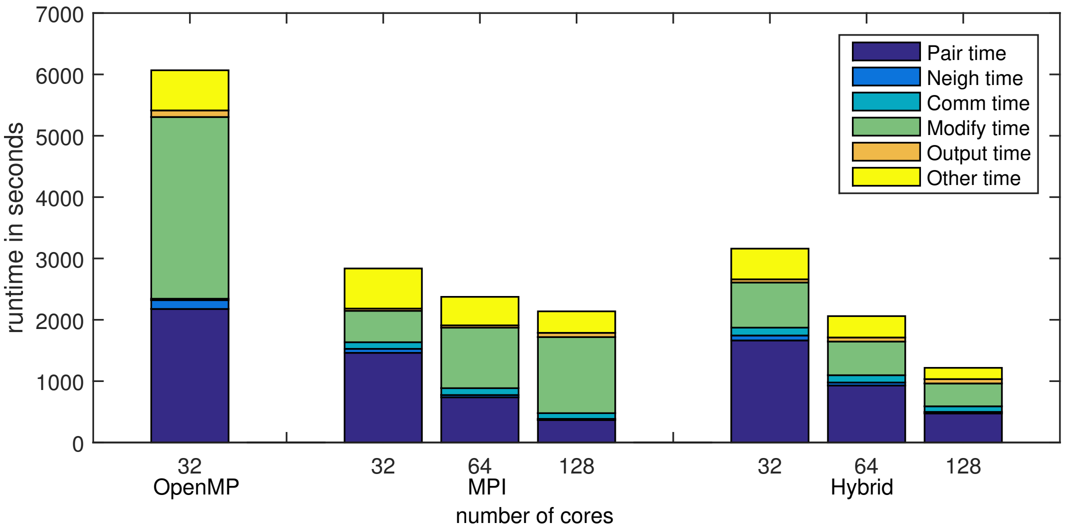
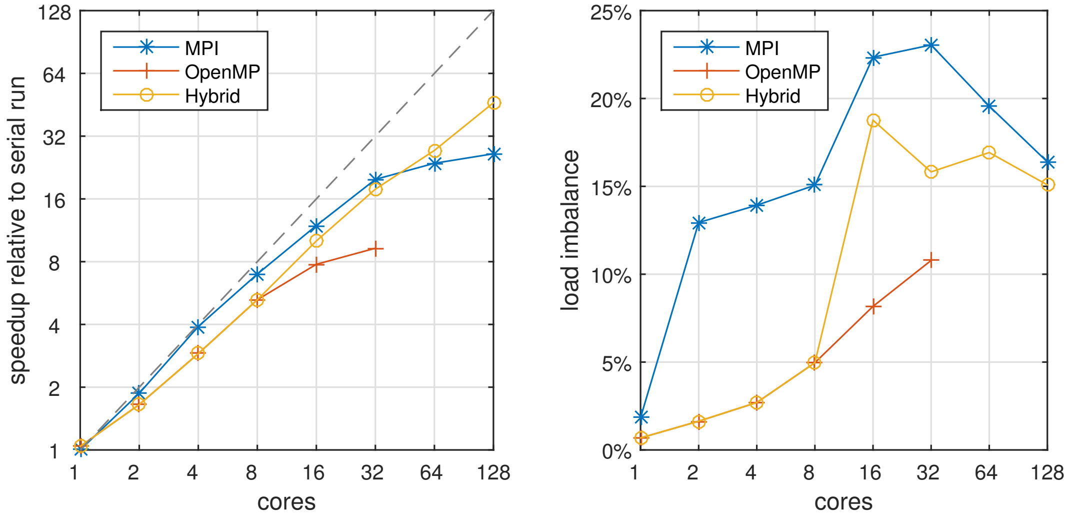


Figure 11.10: Mixing process: results of running MPI, OpenMP and hybrid implementations on 32, 64, and 128 cores. Complex MPI decompositions lead to increased MPI traffic inside the mesh implementation. Mesh communication is a major part of Modify timing. The hybrid might be slightly slower at first, but exhibits continuous scaling, because it can use simpler MPI decompositions.

Unlike in the previous two examples, load imbalance was not the dominant factor in this simulation. Figure 11.11a shows that, overall, the scalability is similar to that in the previous examples, but the hybrid was only able to outperform MPI after adding more cluster blades. Figure 11.11b shows that the load imbalance was between 15%-25%. By our definition, both codes have similar load imbalance at 128 cores, but due to increased MPI traffic caused by the mesh implementation, the time spent in the Modify code section in an MPI-only run was about 3 times longer.



(a) Strong Scalability (b) Load Imbalance

Figure 11.11: Strong scalability and load imbalance of the mixing process test case with 770,000 particles and 35,000 mesh triangles; OpenMP and the hybrid yield the same results in the 1-8 core runs. While the hybrid parallelization shows continuous scaling, the MPI-only runs suffer from increased communication overhead. Load imbalance is not as relevant in this case.

The major difference between this test case and the previous ones is that mesh computations and wall collisions now make up a significant amount of total compute time. These meshes consist of several thousand triangles. Mesh rotation increases MPI traffic between compute nodes due to triangle data migration.

MPI decomposition of meshes also complicates load balancing because it introduces an additional dimension to the imbalance: mesh imbalance. Geometric decomposition of meshes is very sensitive to where the cuts are placed in space. If an unsuitable load imbalance metric is used, one processor can be tasked with processing a significantly greater number of triangles than others. Additional computational costs, such as rotating meshes during each time step, make balancing even more important.

Because of the mesh rotation around the x-axis, choosing a domain decomposition in this example requires a tradeoff between minimizing load imbalance across MPI processes and minimizing MPI communication due to migrating mesh triangles. These two extremes are illustrated by the 64-core examples in Figure 11.12, which shows the amount of time spent in mesh communication compared to the time spent being idle (Other time).

The hybrid parallelization was used to combine the best of both worlds: full utilization

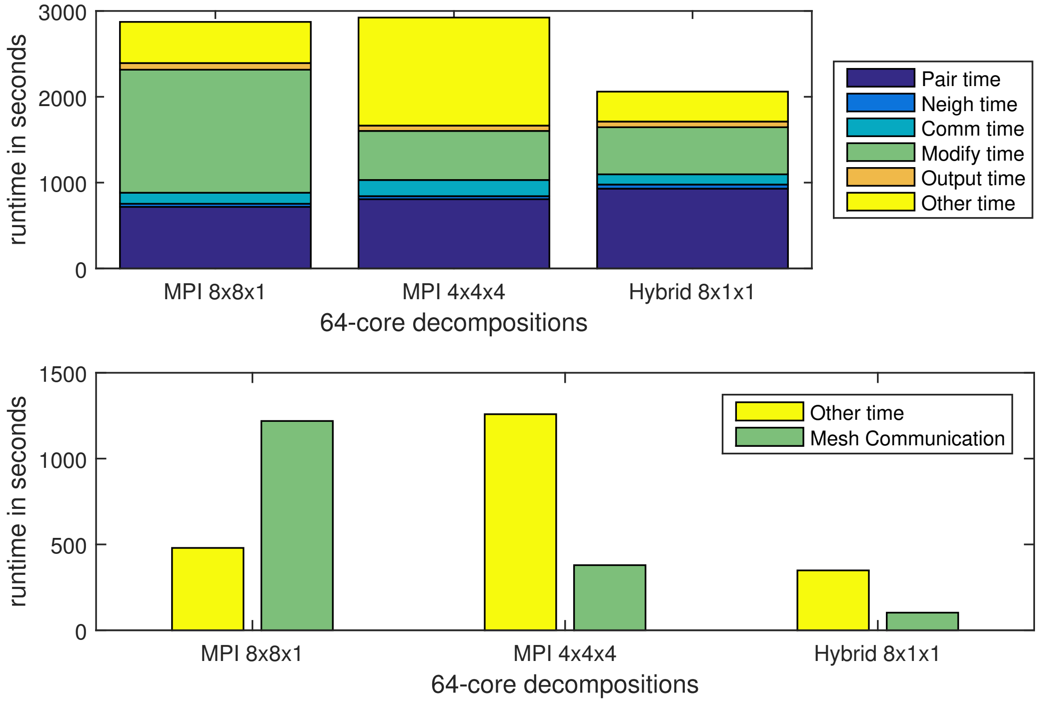


Figure 11.12: Influence of different domain decompositions on mesh communication time (part of Modify time) and time spent outside computation (Other time). While the 8x8x1 decomposition reduced load imbalance, almost 50% of the total run time was spent in mesh communication due to triangles migrating between subdomains in the y direction. A 4x4x4 decomposition improved mesh communication, but led to more load imbalance. The hybrid parallelization used a simple decomposition along the x-axis, reduced the amount of mesh migration, fully utilized all cores (8 threads per MPI process) and minimized load imbalance.

of all compute resources, minimization of load imbalance due to dynamic load balancing, and minimization of mesh communication by choosing domain cuts which reduce triangle migration.

The overall result was that load imbalance at the particle level became less relevant in this test case. For the hybrid, the time spent outside computation was dominated by continuous partition updates.

Well-balanced MPI runs are hard to beat as long as the MPI communication can be kept to a minimum. However, in this type of simulation there is a point beyond which more MPI subdomains lead to an increased amount of MPI traffic due to the mesh implementation. By keeping the number of MPI subdomains low, the hybrid eventually remains scalable, unlike MPI-only simulations, which stall or deteriorate because of communication overheads. The increase in MPI communication can be illustrated by the number of ghost particles and triangles in each simulation. Figure 11.13 shows that the hybrid parallelization uses fewer ghosts while still allowing full utilization of all computational resources.

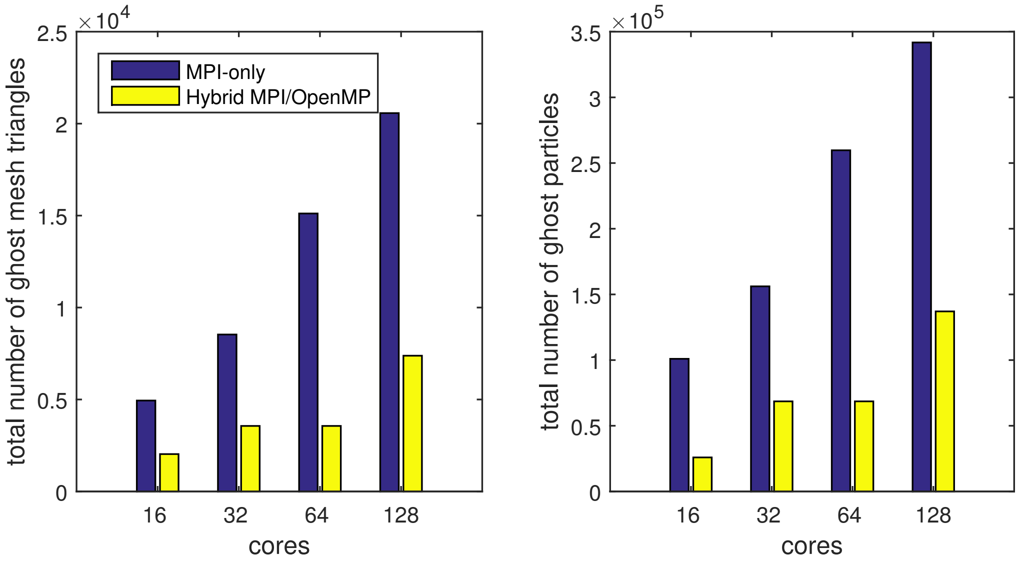


Figure 11.13: Increase in MPI communication observed by visualizing the total number of ghosts in the simulation. Due to simpler decompositions, the hybrid parallelization runs introduced fewer cuts and therefore fewer halo regions. Note that between 32- and 64-core runs the domain decomposition used by the hybrid did not change, only the number of threads increased.

Chapter 12

Conclusion

This work has described the efforts of improving the efficiency of the LIGGGHTS simulation engine and ensuring the quality by refactorings supported by automated testing and validation. As a code which was initially developed by mathematicians, physicists and material scientists the primary concerns were the applicability of the developed physical models. By creating a test harness with a large set of tests the efforts of the past were captured.

From this point on iterative refactorings of the existing code base could take place without failing to see what functionality was damaged. Major changes to the code base were made to simplify the common task of creating new force models both for particle-particle and particle-wall contacts. This paved the way for further advances such multi-threaded OpenMP versions of all granular force kernels. The testing infrastructure also allowed to gradually improve the performance of core algorithms such as wall contact detection and particle insertion. The overall savings by these changes depends on each case.

LIGGGHTS has proven to be a very scalable MPI code thanks to its rich heritage from LAMMPS and the continued collaboration between the two open-source projects. Implementing the hybrid MPI/OpenMP parallelization showed that, although implementing threaded code is simpler to understand, one must go to great lengths to achieve scalable performance, and it is difficult to match a well-optimized MPI implementation.

While the merits of better load balancing are unquestionable, the main challenge lies in reaching a point where reaping these benefits becomes possible.

For low core counts, the current implementation of the MPI/OpenMP hybrid parallelization is less performant than existing code. The MPI implementation is very well suited to these cases, as long as dynamic MPI load balancing is used correctly. However, as the number of cores and the number of compute nodes increase, load imbalance and rising MPI traffic become a greater problem. In our test cases, the hybrid approach achieved speed increases of up to 44% compared to all-MPI parallelization with the same number of processor cores. This is due to improved load balancing with fewer domains in the domain decomposition of the MPI parallelization. The hybrid parallelization allows users to pick a coarse MPI decomposition which maps to the hardware used and to employ OpenMP to utilize all cores available.

The MPI performance graphs presented in this work were the result of extensive testing and variation. Picking an adequate MPI decomposition and load-balancing axes requires much insight and practice. Mistakes are quickly punished by extreme run times as illustrated in the Box-in-a-Box test case. The dynamic load-balancing of our hybrid parallelization provides a remedy by allowing simpler MPI decompositions to be used and offers consistent scaling without major pitfalls.

While many of the individual building blocks of this hybrid parallelization are well known in the literature, the presented work illustrates the benefit of combining multiple load-balancing techniques using two levels of parallelization. It showcases how partitioning can be used to avoid data races and to implement a lock-free version of important DEM routines. By doing some additional work in advance, look-up tables can be avoided inside of force loops and safeguard complexity inside the loop to constant time.

12.1 Impact & Outlook

Overall this work has improved the state of the current LIGGGHTS code base and set important steps for future growth. It simplified the creation of new force models which allows scientists to focus on the physics without needing to understand all the details behind the implementation. Due to core improvements the overall performance of the code allows more and longer simulations to be performed. Finally, the quality of the code base is continuously monitored by an automated system of tests which indicate any regressions.

It also paved the way for further advances using OpenMP code to utilize accelerators such as Intel Xeon Phi processors with even more cores than common in today's architectures. Similar modifications will be necessary to take advantage of GPU computing, once the double-precision floating-point performance improves. GPUs have not been explored in this work due to the control flows present in the current granular force kernels. The existing global data structures present in LIGGGHTS must also become more flexible and become aware of memory locality. Memory alignment throughout the code base would allow better utilization of SIMD instructions.

This work also showed the limitations of multi-threading in such a complicated setting. Many parts of the total system had to be replaced by multi-threaded versions to enable measurable improvements. Moving to more than one compute node still limits the overall scalability because this still has to be done over MPI and the used spatial decomposition. On a single compute node the limitation of global data structure is limiting the available memory bandwidth. Using MPI for each memory channel allows circumventing this problem, but one could also consider adding memory locality awareness to global data structures themselves.

The future of the LIGGGHTS code and its coupling to OpenFOAM CFDEMcoupling will continue to require close collaborations between computer science and basic research. Close relations to the original LAMMPS project will also foster new capabilities to the code. This is the true beauty of open source software.

Bibliography

[1] E. Ortega-Rivas, *Unit Operations of Particulate Solids: Theory and Practice.* CRC Press, 2011.

[2] Y. M. Chen, S. Rangachari, and R. Jackson, "Theoretical and experimental investigation of fluid and particle flow in a vertical standpipe," *Industrial & Engineering Chemistry Fundamentals*, vol. 23, no. 3, pp. 354–370, 1984.

[3] A. Rosato, K. J. Strandburg, F. Prinz, and R. H. Swendsen, "Why the brazil nuts are on top: Size segregation of particulate matter by shaking," *Phys. Rev. Lett.*, vol. 58, pp. 1038–1040, Mar 1987.

[4] H. A. Janssen, "Versuche über Getreidedruck in Silozellen," *Zeitschrift des Vereines deutscher Ingenieure*, vol. 39, no. 35, pp. 1045–1049, 1895.

[5] J. Anderson, *Computational Fluid Dynamics.* McGraw-Hill Education, 1 ed., Feb. 1995.

[6] J. T. Jenkins and S. B. Savage, "A theory for the rapid flow of identical, smooth, nearly elastic, spherical particles," *Journal of Fluid Mechanics*, vol. 130, pp. 187–202, 5 1983.

[7] C. K. K. Lun, S. B. Savage, D. J. Jeffrey, and N. Chepurniy, "Kinetic theories for granular flow: inelastic particles in couette flow and slightly inelastic particles in a general flowfield," *Journal of Fluid Mechanics*, vol. 140, pp. 223–256, 3 1984.

[8] P. C. Johnson and R. Jackson, "Frictional–collisional constitutive relations for granular materials, with application to plane shearing," *Journal of Fluid Mechanics*, vol. 176, pp. 67–93, 3 1987.

[9] P. A. Cundall and O. D. L. Strack, "A discrete numerical model for granular assemblies," *Géotechnique*, vol. 29, pp. 47–65(18), 1979.

[10] A. D. Renzo and F. P. D. Maio, "Comparison of contact-force models for the simulation of collisions in dem-based granular flow codes," *Chemical Engineering Science*, vol. 59, no. 3, pp. 525 – 541, 2004.

[11] J. E. Jones, "On the Determination of Molecular Fields. II. From the Equation of State of a Gas," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 106, no. 738, pp. 463–477, 1924.

[12] T. Pöschel and T. Schwager, *Computational Granular Dynamics: Models and Algorithms.* Springer, 2005.

[13] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[14] C. Kloss, C. Goniva, A. Hager, S. Amberger, and S. Pirker, “Models, algorithms and validation for opensource DEM and CFD-DEM,” *Progress in Computational Fluid Dynamics, An International Journal*, vol. 12, no. 2, pp. 140–152, 2012.

[15] R. M. Mukherjee and R. Houlihan, “Massively Parallel Granular Media Modeling of Robot-Terrain Interactions,” in *1st Biennial International Conference on Dynamics for Design*, vol. 6 of *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pp. 71–78, ASME, 2012.

[16] M. S. Bentley, G. Kargl, N. I. Kömle, and W. Macher, “The dependence of mantle thermal conductivity on particle shape and size,” in *European Planetary Science Congress 2012*, p. 566, Sept. 2012.

[17] R. Berger, C. Kloss, A. Kohlmeyer, and S. Pirker, “Hybrid parallelization of the LIGGGHTS open-source DEM code,” *Powder Technology*, vol. 278, pp. 234 – 247, 2015.

[18] L. Verlet, “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules,” *Phys. Rev.*, vol. 159, pp. 98–103, Jul 1967.

[19] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson, “A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters,” *The Journal of Chemical Physics*, vol. 76, no. 1, pp. 637–649, 1982.

[20] Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng, “Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method,” *Computer Physics Communications*, vol. 161, no. 1-2, pp. 27–35, 2004.

[21] U. Drepper, “What every programmer should know about memory.” http://people.redhat.com/drepper/cpumemory.pdf, 2007. (Online; accessed 9-February-2015).

[22] D. Kafui, S. Johnson, C. Thornton, and J. Seville, “Parallelization of a Lagrangian–Eulerian DEM/CFD code for application to fluidized beds,” *Powder Technology*, vol. 207, no. 1–3, pp. 270–278, 2011.

[23] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1 ed., 2008.

[24] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley, 1999.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[26] Beck, *Test Driven Development: By Example.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[27] J. Langr, *Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better.* Pragmatic Bookshelf, 2013.

[28] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic.* Aug. 2008.

[29] M. M. Lehman and L. A. Belady, eds., *Program Evolution: Processes of Software Change.* San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[30] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[31] W. Cunningham, "The WyCash portfolio management system.," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[32] M. Reddy, *API Design for C++.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2011.

[33] S. Chacon, *Pro Git.* Berkely, CA, USA: Apress, 1st ed., 2009.

[34] G. Booch, *Object-oriented Analysis and Design with Applications (2Nd Ed.).* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.

[35] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series).* Addison-Wesley Professional, 2007.

[36] N. V. Brilliantov, F. Spahn, J.-M. Hertzsch, and T. Pöschel, "Model for collisions in granular gases," *Phys. Rev. E*, vol. 53, pp. 5382–5392, May 1996.

[37] H. Hertz, "Über die Berührung fester elastischer Körper," *Journal für die reine und angewandte Mathematik*, vol. 1882, no. 92, pp. 156–171, 1882.

[38] K. L. Johnson, K. Kendall, and A. D. Roberts, "Surface Energy and the Contact of Elastic Solids," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 324, no. 1558, pp. 301–313, 1971.

[39] J. Ai, J.-F. Chen, J. M. Rotter, and J. Y. Ooi, "Assessment of rolling resistance models in discrete element simulations," *Powder Technology*, vol. 206, no. 3, pp. 269 – 282, 2011.

[40] P. Pacheco, *An Introduction to Parallel Programming.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2011.

[41] 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.

[42] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[43] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, pp. 287–311, May 2006.

[44] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency and Computation: Practice & Experience - Scalable Tools for High-End Computing*, vol. 22, pp. 685–701, Apr. 2010.

[45] D. Naishlos, "Autovectorization in GCC," in *Proceedings of the 2004 GCC Developers Summit*, pp. 105–118, 2004.

[46] "A Guide to Vectorization with Intel® C++ Compilers." https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf, 2012. (Online; accessed 24-November-2015).

[47] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.0," tech. rep., Knoxville, TN, USA, September 2012.

[48] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Feb. 2012.

[49] B. Stroustrup, *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 4th ed., May 2013.

[50] D. R. Butenhof, *Programming with POSIX Threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[51] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[52] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[53] J. Reinders, *Intel Threading Building Blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., first ed., 2007.

[54] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[55] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2012.

[56] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st ed., 2010.

[57] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, pp. 66–73, May 2010.

[58] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[59] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving Portability and Performance Through OpenACC," in *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, (Piscataway, NJ, USA), pp. 19–26, IEEE Press, 2014.

[60] S. Yakubov, B. Cankurt, M. Abdel-Maksoud, and T. Rung, "Hybrid MPI/OpenMP parallelization of an Euler–Lagrange approach to cavitation modelling," *Computers & Fluids*, vol. 80, pp. 365–371, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[61] H. Liu, D. K. Tafti, and T. Li, "Hybrid parallelism in MFIX CFD-DEM using OpenMP," *Powder Technology*, vol. 259, pp. 22–29, 2014.

[62] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, (Washington, DC, USA), pp. 427–436, IEEE Computer Society, 2009.

[63] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[64] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[65] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[66] J. L. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM*, vol. 31, pp. 532–533, May 1988.

[67] R. Kačianauskas, A. Maknickas, A. Kačeniauskas, D. Markauskas, and R. Balevičius, "Parallel discrete element simulation of poly-dispersed granular material," *Advances in Engineering Software*, vol. 41, no. 1, pp. 52–63, 2010. Civil-Comp Special Issue.

[68] P. Gopalakrishnan and D. Tafti, "Development of parallel DEM for the open source code MFIX," *Powder Technology*, vol. 235, pp. 33–41, 2013.

[69] S. Plimpton, "LAMMPS Manual, Fix Balance Command." http://lammps.sandia.gov/doc/fix_balance.html. (Online; accessed 9-February-2015).

[70] S. Srinivasan, I. Ashok, H. Jônsson, G. Kalonji, and J. Zahorjan, "Dynamic-domain-decomposition parallel molecular dynamics," *Computer Physics Communications*, vol. 102, no. 1–3, pp. 44–58, 1997.

[71] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner, "Parallel Transient Dynamics Simulations: Algorithms for Contact Detection and Smoothed Particle Hydrodynamics," *Journal of Parallel and Distributed Computing*, vol. 50, no. 1–2, pp. 104–122, 1998.

[72] M. J. Berger and S. H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Transactions on Computers*, vol. 36, pp. 570–580, May 1987.

[73] B. Hendrickson and K. Devine, "Dynamic load balancing in computational mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 184, no. 2–4, pp. 485–500, 2000.

[74] S. Plimpton, "LAMMPS Manual, Balance command." http://lammps.sandia.gov/doc/balance.html. (Online; accessed 9-February-2015).

[75] A. Kohlmeyer, "LAMMPS Manual, USER-OMP package." http://lammps.sandia.gov/doc/accelerate_omp.html. (Online, accessed 9-February-2015).

[76] R. Berger, A. Kohlmeyer, and C. Kloss, "Toward Parallelization of LIGGGHTS Granulular Force Kernels with OpenMP," in *6th International Conference on Discrete Element Methods*, pp. 181–185, Colorado School of Mines, 2013.

[77] A. Amritkar, S. Deb, and D. Tafti, "Efficient Parallel CFD-DEM Simulations Using OpenMP," *Journal of Computational Physics*, vol. 256, pp. 501–519, Jan. 2014.

[78] D. S. Henty, "Performance of Hybrid Message-passing and Shared-memory Parallelism for Discrete Element Modeling," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, (Washington, DC, USA), IEEE Computer Society, 2000.

[79] L. Smith and M. Bull, "Development of Mixed Mode MPI / OpenMP Applications," *Scientific Programming*, vol. 9, pp. 83–98, aug 2001.

[80] B. M. W. Michael Brown, Anupama Kurpad, *LAMMPS Manual: USER-INTEL package.*

[81] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," in *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, (New York, NY, USA), pp. 377–384, ACM, 2008.

[82] E. G. Boman, U. V. Çatalyürek, C. Chevalier, and K. D. Devine, "The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering and Coloring," *Scientific Programming*, vol. 20, pp. 129–150, Apr. 2012.

[83] K. Devine, E. Boman, R. Heapby, B. Hendrickson, and C. Vaughan, "Zoltan Data Management Service for Parallel Dynamic Applications," *Computing in Science and Engineering*, vol. 4, pp. 90–97, Mar. 2002.

Acronyms

API	Application Programming Interface
AVX	Advanced Vector Extensions
CDT	Constant Directional Torque
CFD	Computational Fluid Dynamics
CFDEM	CFD coupled with DEM
CI	Continuous Integration
CLI	Command-Line Interface
CPU	Central Processing Unit
CSV	Comma-Separated Values
CUDA	Compute Unified Device Architecture
DEM	Discrete Element Method
DLP	Data-Level Parallelism
EOM	Equation of Motion
FE	Finite Element
FPU	Floating-Point Unit
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
JKR	Johnson Kendall Roberts model
JSON	JavaScript Object Notation
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator
LIGGGHTS	LAMMPS Improved for General Granular and Granular Heat Transfer Simulations
MD	Molecular Dynamics
MMX	Multimedia Extensions
MPI	Message Passing Interface
MPP	Massively Parallel Processor
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
OpenCL	Open Compute Language
OpenACC	Open Accelerators
PFM	Deparment of Particulate Flow Modelling
PSS	Partitioned Spatial Sorting
QDR	Quad Data Rate
RAM	Random Access Memory
RCB	Recursive Coordinate Bisectioning
RGB	Red-Green-Blue color space
SIMD	Single Instruction Multiple Data
SJKR	Simplified JKR model
SMP	Symmetric Multiprocessing
SPH	Smoothed Particle Hydrodynamics
SSE	Streaming SIMD Extensions
STL	STereoLithography file format
STL	Standard Template Library
TBB	Intel Threading Building Blocks
TLP	Thread-Level Parallelism
UMA	Uniform Memory Access
VCS	Version Control System
VTK	Visualization Toolkit
XML	Extensible Markup Language

Listings

3.1 MIN and MAX macros found in LIGGGHTS 2.x 23
3.2 Expansion of th MIN macro creating a fault expression 23
4.1 LIGGGHTS-PUBLIC-fedora-2.3.8 target definition file 29
4.2 Example of a run configuration file 30
4.3 Generated execution call 31
4.4 Input script line with parameters 31
4.5 Run configuration with a dependant run 31
4.6 Typical thermo configuration in LIGGGHTS 33
4.7 Thermo output 34
4.8 Usage of print fix to outout time and kinetic energy every 100 time steps to a file 34
4.9 Example of dump custom/vtk 35
4.10 LIGGGHTS-PUBLIC-fedora-master target definition file 35
4.11 Defining data sources and plots in run configuration 38
5.1 Basic structure of `compute_ke.h` 48
5.2 Code using style file to generate code 49
5.3 Granular force kernel loop 50
5.4 Definition of particle properties by template 54
5.5 Definition of type properties via property/global fixes 55
6.1 Pair force loop 60
6.2 Wall force loop 62
6.3 Register `tangential_damping` setting which can be on or off 65
6.4 Definition of a global property factory method which computes the effective Young's modulus matrix. 66
6.5 Connecting the contact model member variable Yeff to the global property Yeff which is created using the createYeff factory method. 67
6.6 Constructor of tangential model which adding three contact history values 68
6.7 Usage of contact history in TangentialModel 68
6.8 Contents of generated `style_contact_models.h` file 69
7.1 Manual instrumentation using C++11 chrono library 77
7.2 Collecting run-time statistics from hardware counters using `perf stat` 79
7.3 Capturing performance counter data on the instruction level using `perf record` and visualizing it with `perf report`. 80
8.1 Original implementation of `MPI_Max_Vector` utility function 87
8.2 Improved and simpler implementation of `MPI_Max_Vector` using `MPI_IN_PLACE` as send buffer, which tells MPI to reuse the receive buffer for both sending and receiving. 87
8.3 Getter of global boudinging box with lazy evaluation 88
8.4 Interface of RegionNeighborList 93
10.1 Simple example of OpenMP parallel region 115
10.2 OpenMP Parallelization of a for-loop 115

10.3 Compact form of OpenMP parallel region and work-sharing construct 116
10.4 OpenMP reduction in parallelization of gravity force computation 116
10.5 False sharing example code 117
10.6 Trivial OpenMP parallelizatin of granular force loop 120
10.7 OpenMP parallelization of granular force loop using Newton's third law 121
10.8 Using custom locks to 124
10.9 Implementation of the force loop which postpones conflicting updates 126
10.10 Pseudo-code of the parallelized particle-particle force loop. Threads work only on particles in their partition. The force F_{ij} is applied to particle i and $-F_{ij}$ to particle j if both particles are in the same partition. Particle i will not be in the neighbor list of particle j in that case. Otherwise, i and j are in each other's neighbor list, and F_{ij} and F_{ji} are computed by separate threads. 127
10.11 Pseudo-code of parallelized particle-wall force loop. All threads visit all meshes and their triangles, but only compute forces for particles which are in their partition. 134

List of Figures

1.1 Methods for describing granular flow 2

2.1 Structure of a basic DEM integration loop 8
2.2 Comparison of Euler and Velocity-Verlet integration 10
2.3 Integration loop with Velocity-Verlet integration scheme 11
2.4 Velocity-Verlet integration with neighbor lists 11
2.5 2D Binning of the domain (100 × 100) with 10 × 10 bins. 12
2.6 Cutoff of Verlet List 12
2.7 Iteration space (i, j) visited by particle neighbor list. 13
2.8 Adding spatial sorting to the integration loop 14
2.9 Contents of neighbor list after spatial sorting 15
2.10 Communication between parallel running integration loops 16

3.1 Levels of software testing 25

4.1 Structure of the CFDEMproject test harness 28
4.2 States of a test case run 32
4.3 Acceptable error margin using a relative point-wise ϵ or a global constant ϵ . . . 37
4.4 Plot generated by web-frontend based on run configuration and test case data. . . 39
4.5 Some test cases from the test harness. 40
4.6 Overview of Test Harness executions 41
4.7 Folder hierarchy of test cases 42
4.8 Detailed information of each test case run time breakup 43
4.9 Comparison of two test case runs. 43
4.10 Overview of version consistency of all test cases 44
4.11 Version consistency of torque plot 44
4.12 Continuous Integration for LIGGGHTS 45
4.13 Results of continuous integration run on a pull request on GitHub 46

5.1 Headers with a prefix are collected and included in a generated style header. . . . 48
5.2 spring-dashpot system 51
5.3 Simplified class hierarchy of the mesh implementation 57
5.4 Hierarchy of Pair Styles in LIGGGHTS 2.x 58
5.5 Structure of granular FixWall Styles in LIGGGHTS 2.x 58

6.1 New structure of the granular force kernels in LIGGGHTS 3.x 61
6.2 Flow of information in and out of contact model 63
6.3 A `ContactModel` template is a composition of submodel specializations 64
6.4 Contact model combinations in LIGGGHTS 3.1.0 (PFM version) 70

7.1 Block diagram of Intel Core i7-3770 (Ivy Bridge) processor 74

7.2 Dual Socket Configuration . . . 76
7.3 Performance of insert/pack with increasing particle number . . . 79
7.4 Screenshot of perf report utility . . . 80
7.5 Performance of insert/pack with increasing particle number after optimization . . 81
7.6 HPC Viewer . . . 83
7.7 HPC Trace Viewer . . . 84

8.1 MPI strong scalability of mesh computations in Mixer case study . . . 86
8.2 Visualization of particle bins surrounding a mesh triangle in 2D . . . 89
8.3 Bounding box and extended bounding box of a mesh triangle . . . 90
8.4 Optimization of mesh neighbor list build . . . 90
8.5 Run times of transfer chute simulation with static meshes . . . 91
8.6 Transfer Chute Simulation . . . 91

9.1 Distributed Memory Programming . . . 96
9.2 Shared-Memory Programming . . . 97
9.3 Fork-Join model . . . 97
9.4 Mixed-mode programming or hybrid programming . . . 99
9.5 Particles distributed between two adjacent processes . . . 101
9.6 Mesh triangles distributed between two adjacent processes . . . 101
9.7 Halo regions of two different decompositions of a domain with an extent of 1x1x2. 102
9.8 Two process mappings of a 1x2x4 decomposed domain. . . . 103
9.9 Possible domain decompositions with 2, 4, and 8 processes . . . 104
9.10 Two extreme examples with different workloads . . . 106
9.11 Dynamic domain decomposition . . . 106
9.12 Overview of the MPI/OpenMP hybrid parallelization . . . 111

10.1 LIGGGHTS integration loop (simplified) . . . 112
10.2 Memory contention . . . 119
10.3 Distribution of the iteration space of a granular force loop among two threads . . 121
10.4 Distributed iteration space after applying Newton's third law . . . 122
10.5 Distributed iteration space with work duplication in conflict cases . . . 126
10.6 Illustration of the Recursive Coordinate Bisection (RCB) algorithm . . . 128
10.7 Rotary kiln simulation distributed among 4 processes or 4 threads. . . . 129
10.8 Partitioned Spatial Sorting . . . 129
10.9 Memory allocation during neighbor list building using memory paging . . . 132
10.10 Multi-threaded generation of neighbor list . . . 132
10.11 Mesh of silo discharge simulation . . . 135
10.12 Neighbor list workload of a silo discharge simulation . . . 137
10.13 Sorted and load-balanced neighbor list workload . . . 138

11.1 Block diagram of an AMD Opteron 6272 processor . . . 141
11.2 Box-in-a-box example . . . 143
11.3 OpenMP, MPI, and hybrid runs of Box-in-a-box test case on 32, 64 and 128 cores. 145
11.4 Run times of MPI-only simulations using different 128-core decompositions . . . 145
11.5 Strong scalability and load-imbalance of the Box-in-a-box test case . . . 146
11.6 Silo discharge; 1.5 million particles filled into a silo are released. . . . 148
11.7 Silo discharge results of running on 32, 64, and 128 cores . . . 149
11.8 Strong scalability and load imbalance of the Silo discharge example . . . 150
11.9 Mixing process test case . . . 151
11.10 Results of running mixing process test case on 32, 64, and 128 cores . . . 152

11.11 Strong scalability and load imbalance of the mixing process test case 153
11.12 Influence of domain decompositions on load imbalance and mesh communication 154
11.13 Increase in MPI communication observed by visualizing the total number of ghosts 155

Curriculum Vitae

Name	Dipl.-Ing. Richard Berger
Email	`richard.berger@outlook.com`
Date of Birth	February 24th, 1986
Citizenship	Austria
Parents	Ing. Christian Berger Yolanda Berger

Professional Experience

2013 - 2016	Researcher at the Department of Particulate Flow Modelling at Johannes Kepler University in Linz, Austria
2008 - 2013	Researcher at the Christian Doppler Laboratory for Automated Software Engineering at Johannes Kepler University in Linz, Austria

Education

2013 - 2016	PhD-Student of Mechatronics (Doktoratstudium) at Johannes Kepler University in Linz, Austria
2012	Diploma in Mechatronics (Dipl.-Ing.) at Johannes Kepler University in Linz, Austria
2005 - 2012	Student of Mechatronics (Diplomstudium) at Johannes Kepler University in Linz, Austria
2005	Military Service in Linz, Austria
2004	Matura with distinction at Schulverein Kollegium Aloisianum in Linz, Austria
1999 - 2004	Schulverein Kollegium Aloisianum in Linz, Austria
1997 - 1998	Jubail Academy, International School (American Division), Al Jubail, Saudi Arabia
1996 - 1997	Akademisches Gymnasium Linz, Austria
1992 - 1996	Volkschule Leonding, Austria

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.