Software Design of a Massively Parallel Rigid Body Framework

Software design einer Bibliothek für massiv parallele Starrkörperdynamik

Der Technischen Fakultät der Universität Erlangen-Nürnberg zur Erlangung des Grades:

Doktor-Ingenieur

Vorgelegt von:

Klaus Iglberger, M.Sc.

Erlangen, 2010
Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 16. April 2010
Tag der Promotion: 16. September 2010
Dekan: Prof. Dr. R. German
Berichterstatter: Prof. Dr. U. Rüde
                     Prof. Dr. M. Resch
Abstract

Granular media are of utmost importance in many natural phenomena, industrial processes, and our everyday life. Still, despite their significance, essential characteristics of granular flow, sedimentation, segregation, or fluidization processes are insufficiently understood. Therefore in the last decades, researchers from various sciences have used numerical simulation to gain deeper insight into the underlying physical processes. Enormous progress has been made in simulating granular media, yet physically realistic scenarios involving billions of interacting particles remain a challenge. The prime reason for this is the complexity of the physical processes: Huge numbers of particles, numerous physical effects, and interdisciplinary aspects contribute to the enormous difficulty to simulate these systems. A second reason is the complexity of the according software development process: The design of appropriate software that can handle the physical complexity, yet satisfies even the highest demands on software quality, is far from trivial.

The requirements on a software focused on the simulation of rigid bodies are high: Reliability, correctness, modularity, extensibility, efficiency, portability, and maintainability must be essential characteristics of this software in order to be able to successfully cope with these complexities. Additionally, due to the size of the target applications, parallelization concepts for massively parallel architectures are crucial aspects for physically relevant scenarios.

My contribution to the scientific community with this dissertation is the pe rigid (multi-) body physics engine. It offers a thoroughly and systematically designed software infrastructure for all kinds of rigid body simulations in both physically accurate scenarios as well as real-time environments. The exceptional design of the pe allows for extremely flexible, yet efficient customizations of all individual phases of a rigid body simulation. In this thesis, I will introduce various aspects of the elaborate software design of the pe framework. A special emphasis will be on its parallelization concepts that are focused on large-scale simulations on massively parallel machines. In this context, I will also present a massively parallel rigid body dynamics algorithm and demonstrate its capabilities by several large-scale granular media simulations on up to 131,072 processor cores with up to 2 billion interacting rigid bodies.
Zusammenfassung


Die Anforderungen an eine Software für die Simulation von starren Körperrn sind sehr hoch: Zuverlässigkeit, Korrektheit, Modularität, Erweiterbarkeit, Effizienz, Portierbarkeit und Wartbarkeit müssen grundlegende Qualitätsmerkmale dieser Software sein, um erfolgreich die Komplexitäten von granularen Medien bewältigen zu können. Zusätzlich benötigt man aufgrund der enormen Größe der Zielanwendungen geeignete Parallelisierungskonzepte, um physikalisch realistische Szenarien simulieren zu können.

Acknowledgments

First of all I want to express my gratitude to my PhD supervisor Prof. Dr. Ulrich Rüde, who made this work possible in the first place. Also thank you very much for the ongoing support of my work and the faith that it would be a considerable contribution.

A lot of thanks to my fiancée and soon to be wife Stefanie Müller, who accompanied me through the entire PhD, enjoyed good times (such as conference trips to various countries), but also gave me support in bad times. Without your lasting support these five years would have been much harder.

Thank you very much to all my colleagues: Frank Deserno, Christoph Freundl, Stefan Donath, Markus Stürmer, Harald Köstler, Tobias Gradl, and Daniel Ritter. Special thanks to Jan Götz for his involuntary beta-testing of the pe framework, Christian Feichtinger, my room mate, for very helpful discussions, Nils Thürey for starting the initial flame for rigid body simulations, and Britta Berneker for the good teamwork in the SiwiR lecture series. Also many thanks to Tobias Preclik and Kristina Pickl for joining and pursuing the pe development with their PhD works.

A special thank you to all the students, who, by choosing a bachelor or master thesis in the context of the pe, considerably contributed to my PhD work: Tobias Preclik, Kristina Pickl, Florian Schornbaum, Fabian Schönfeld, Severin Strobl, Martin Ketzer, Stefan Geisselsöder, Thomas Heller, Dominik Guess, Michael Sünkel, and Tamas Torba. Without their help, their zeal, their ideas, and their suggestions this would not have been possible.

Additionally I want to say thank you to my entire family. Thank you for this safe harbor in my life that I could rely on every time I needed.

Last but not least, I want to thank the authors of my favored books, Herb Sutter, Matthew Wilson, Scott Meyers, and Steven Dewhurst, for the entertaining, yet very instructive way to teach me C++. Without their work, my passion for C++ software design would not have been roused.
Contents

I Introduction

1 Motivation
  1.1 Granular Media Simulations ................................. 5
  1.2 Particle-Laden Flows ..................................... 8

2 The pe Rigid (Multi-)Body Physics Engine ............. 11
  2.1 Goals ....................................................... 11
  2.2 Features .................................................. 13
    2.2.1 Simulation Features ................................ 13
    2.2.2 Simulation Algorithms ............................... 13
    2.2.3 Implementation Features ............................. 14
    2.2.4 Visualization .......................................... 14
  2.3 Simulation Examples ..................................... 14

II Software Design of a Massively Parallel Rigid Body Framework 19

3 Compile Time Constraints ................................. 23
  3.1 Motivation ................................................ 23
  3.2 Static Assertion .......................................... 24
  3.3 Comprehensible Error Messages .......................... 28
  3.4 The pe Compile Time Constraints ......................... 29
  3.5 The MUST_BE_BASE_OF constraints ..................... 32
  3.6 The MUST_BESAME_TYPE constraints .................... 33
  3.7 Conclusion ................................................ 34

4 Smart Scopes ................................................. 35
  4.1 The Basic Concept ........................................ 35
  4.2 The pe Exclusive Section Scope ......................... 38
  4.3 The pe Global Section Scope .............................. 42
  4.4 Conclusion ................................................ 44

5 The pe Logging Functionality ............................ 45
  5.1 Motivation ................................................ 45
  5.2 A First Impression of the pe Logging Tool ............ 46
  5.3 The Logger Class ......................................... 50
## Contents

5.4 The LogSection Class ........................................ 52
5.5 The Logging Sections ........................................ 55
5.6 Limits of the Logging Tool .................................... 57
5.7 Conclusion .................................................. 60

6 The Sandwich Pattern .......................................... 61
6.1 Motivation .................................................. 61
6.2 The Sandwich Pattern ......................................... 63
6.3 The Hierarchical Sandwich Pattern .......................... 69
6.4 Application of the Sandwich Pattern in the pe ............... 70
6.5 Conclusion .................................................. 75

7 Resource Management .......................................... 77
7.1 Motivation .................................................. 77
7.2 Resource Abstraction ......................................... 80
7.3 The BodyManager Class ...................................... 83
7.4 The World Class ............................................ 87
7.5 Conclusion .................................................. 89

8 The Math Library of the pe ..................................... 91
8.1 Motivation .................................................. 91
8.2 Expression Templates ......................................... 93
8.3 Smart Expression Templates ................................. 101
  8.3.1 Expression Dependent Evaluation ..................... 102
  8.3.2 Detection of Aliasing Effects ......................... 106
  8.3.3 Integration of BLAS Functionality .................... 108
8.4 Performance Results ......................................... 113
  8.4.1 Complex Expression $A \cdot (a + b)$ ....................... 114
  8.4.2 Complex Expression $A \cdot (a + b + c)$ ............... 115
  8.4.3 Complex Expression $(A \cdot B) \cdot (a + b)$ .......... 115
  8.4.4 Complex Expression $(A \cdot B) + C$ .................... 116
  8.4.5 Complex Expression $(A + B) \cdot (C - D)$ .......... 117
8.5 Conclusion .................................................. 117

III Massively Parallel Rigid Body Dynamics ....................... 119

9 Rigid Body Dynamics in Distributed Memory Environments ........ 123
  9.1 Domain Partitioning ....................................... 123
  9.2 Parallel Simulation of Volumetric Bodies .................. 129
  9.3 MPI communication ....................................... 134
  9.4 Conclusion .............................................. 135

10 The Parallel Fast Frictional Dynamics Algorithm ............... 137
  10.1 The Serial Algorithm .................................... 139
  10.2 The Parallel Algorithm .................................. 141
  10.3 Scenario-based Evaluation ................................ 145
10.4 Performance Results ........................................... 147
  10.4.1 Serial Performance ........................................... 147
  10.4.2 Strong Scaling Results ...................................... 148
  10.4.3 Weak Scaling Results ....................................... 150
10.5 Validation ...................................................... 154
10.6 Conclusion ..................................................... 155

11 Large-Scale Granular Media Simulations 157
  11.1 The Cube Scenario ............................................. 157
  11.2 The Staircase Scenario ...................................... 158
  11.3 The Silo Scenario ............................................. 159

IV Conclusions and Future Work 165

12 Conclusion and Future Work 167
  12.1 Conclusion ................................................... 167
  12.2 Suggestions for Future Work .................................. 168

V Appendix 171

A Test Machines ....................................................... 173
  A.1 HLRB II – SGI Altix 4700 ..................................... 173
  A.2 Jugene – IBM BlueGene/P .................................... 174

B The SelectType class template .................................... 175

C The MathTrait class template ..................................... 177

D Performance Results of the pe Math Library ............... 181

List of Figures ......................................................... 187
List of Tables ........................................................ 189
Listings .................................................................. 191
List of Algorithms ................................................... 195
Supervised Student Theses ........................................... 197
Bibliography ............................................................ 199
Part I

Introduction
In the last decades, the numerical simulation has established itself as the third pillar of science next to the two classical pillars of theory and experiment. Nowadays, the numerical simulation is basically used in all engineering and natural scientific fields to supplement the research in more and more complex problems, especially when the classical forms of science can hardly be applied anymore. Whereas this is particularly true for disciplines where experiments are not possible at all (as for instance astronomy) and the numerical simulation is the only way to run (virtual) experiments, this is also true for the simulation of rigid bodies. Although classical mechanics, introduced by Isaac Newton in the 17th century, teaches us the basics of the interaction between rigid bodies, interactions between millions and billions of rigid bodies are still subject to research. For instance, although granular media are the second most processed material after water, their characteristics and behavior are still only rudimentally understood.

This part will set the stage for this thesis. It will set this work and its achievements in the according context and it will serve as a motivation for the rest of the thesis. Chapter 1 will start by illustrating the requirements for massively parallel rigid body simulations by introducing several applications involving millions of rigid bodies. With that, it will also demonstrate the need for interdisciplinary, multi-physics research by showing coupled simulations between a fluid dynamics solver and a rigid body simulation. The subsequent Chapter 2 will introduce the C++ framework developed in the context of this thesis for the simulation of massively parallel rigid body scenarios. Next to giving an overview of similar simulation frameworks, this chapter will summarize the goals and features of the framework, before later chapters will focus on its design.
1 Motivation

Whereas the simulation of a small number of rigid bodies and their interaction is reasonably well understood and already incorporated in several commercial simulation packages, the simulation of a huge number of rigid bodies (millions to even billions) is still a very active field of research. One reason for that is that certain occurring physical effects are only rudimentally understood. In order to close the gaps in the physical theories, the numerical simulation is used to complement experiments when experiments are not able to get all the necessary answers.

The following two sections will demonstrate several applications, where the numerical simulation can very effectively be used to gain more insight into the underlying physics. Section 1.1 will highlight the simulation of granular media, currently one of the most important fields of research, and Section 1.2 will focus on the interdisciplinary topic of particle-laden flows.

1.1 Granular Media Simulations

Granular materials are present in various natural phenomena, industrial applications, and even in our everyday life. In contrast to other materials, granular media exhibit a different behavior under different conditions. For instance, large amounts of corn or sugar flowing through a pipe or through the outlet of a silo behave very much like fluids. Yet while standing on top of the medium at rest it resembles more a plastically deformable solid. Granular media can also occur in form of a gas, as, for instance, rings of dust around the large planets of our solar system. Granular media also exhibit the special property to easily make the transition from one state to another: while standing on the beach we sink into the fluid-like sand only for a few centimeters before the sand resembles a solid material.
In industry, the characteristics and the behavior of granular materials are of utmost importance, yet still only rudimentally understood. For instance, the state-of-the-art construction of silos still fails at times due to the unpredictable behavior of granular media; the optimization of ball mills is still a difficult and ongoing research problem [31]; and the transition of granular particles from solid to liquid due to fluidization is a still not perfectly understood effect.

Since experiments are often too expensive, impossible (as for instance in the case of our solar dust clouds), or simply do not give enough details to understand the underlying effects, the numerical simulation is an invaluable complement to traditional research. However, to be able to assume this role, the simulation requires the ability to handle huge numbers of granular particles in order to simulate realistic scenarios.

The simulation of granular media is still one of the most active research fields. Several numerical simulation approaches, mainly based on the representation of granular particles as point masses, have been used for the simulation of granular media. Among them are molecular dynamics [48, 91], Monte Carlo simulations, cellular automata [106, 91], or smoothed particle hydrodynamics [86, 82, 41]. Whereas the simplification to represent granular particles as point masses considerably decreases the complexity of the simulation and facilitates the parallelization of these simulations in order to be able to handle millions and billions of particles, these simulations neglect several important aspects of the processes, such as for instance the true shape of the granular particles and with that their orientation and rotational velocity. Therefore particle-based simulation approaches are not able to incorporate all important physical effects.

Two different simulation approaches are the Discrete Element Method (DEM) [27, 43] and Rigid Body Dynamics (RBD) [21, 85, 19] [6]. Unlike particle-based simulation approaches, the true shape of the granular particles is taken into account, i.e., all particles are fully resolved. This allows the simulation of arbitrarily shaped particles. In addition, contacts between objects are fully resolved, i.e., elastic and inelastic collisions as well as frictional contacts are incorporated. In the DEM, the discontinuities of rigid body collisions are transformed into a stiff force field. Although with this approach the collision resolution calculations are completely local (in contrast to, for instance, global LCP problems; see Chapter 10) and it can therefore be much more easily parallelized for distributed memory...
machines, the regularization approach requires very small time steps to keep interpenetra-
tions between rigid bodies small and to achieve numerical stability. Therefore this approach
results in very high computational costs and very long runtimes even on supercomputers. In
contrast to the DEM, RBD considers all objects as perfectly rigid and undeformable. The
forces are calculated as a result of motion constraints of contacting bodies to prevent inter-
penetration. Due to this approach, rigid body dynamics supports the simulation of very stiff
materials. However, this results in non-unique contact forces for specific rigid body configura-
tions. Additionally, the resolution of collisions is usually much more costly than in the DEM.

The following figures show three examples of granular media simulations. Figure 1.2 shows
a closed box containing 972,000 spherical particles. Figure 1.3 shows a similar scenario, an
opening box initially containing 344,960 spherical particles (see also Section 11.1). Lastly,
Figure 1.4 gives an impression of a silo scenario with 27,270 randomly generated non-spherical
particles (see also Section 11.3). Note that all three simulations have been performed by the
massively parallel rigid body framework introduced in the next chapter.

Figure 1.2: Medium-sized simulation of a granular media scenario consisting of 972,000 spherical
particles. The patches indicate the domains of the 108 involved MPI processes.

Figure 1.3: Small-sized simulation of an opening cube containing 344,960 spherical particles. The
radially arranged patches indicate the domains of the 64 involved MPI processes.
1 Motivation

Figure 1.4: Granular media simulation of 27,270 randomly generated non-spherical particles on 256 MPI processes. Each granular particle is shaped from two to five sphere primitives.

1.2 Particle-Laden Flows

Particle-laden flows occur in many physical and industrial systems such as fluidization, sedimentation, and segregation processes [95]. Despite the high significance of these processes, the detailed modeling of the associated transport phenomena is still ongoing research. Various numerical methods used to simulate particulate flows have been suggested, for instance Stokesian Dynamics [30], Euler–Lagrangian methods [56], distributed Lagrange Multiplier methods [90] and Discrete Element methods [27, 43], among others. Approaches based on the Lattice Boltzmann method have been presented by Ladd [77, 78], Aidun et al. [12] and Qi [93].

Most of these methods do not attempt a fully resolved simulation of the fluid-structure interaction between solid particles and fluid. Instead they use approximations to reduce the computational complexity. For instance, some of these methods can only be applied in special flow regimes (for example potential flow or Stokes flow) or they approximate the particles as dimensionless point masses, which neglects their true shape. However, many physical effects depend on the shape and size of the particles.

Results for a direct numerical simulation of particulate flows with fully resolved particles involving a larger number of particles were presented by Yin and Koch [108] with 8,000 particles and Jin at al [74] with 21,336 spherical particles. To simulate physically realistic scenarios, the number of particles has to be considerably increased. Since the particles have to be resolved in the fluid simulation, this inevitably also results in a finer resolved flow domain with significantly higher computational demands. Therefore realistic simulations of sedimentation, segregation, or fluidization scenarios require, for both the fluid simulation as well as the rigid body simulation, massively parallel simulations.

Whereas fluid solvers are usually already parallelized for distributed memory architectures, most rigid body frameworks are not suited for these kinds of simulations. Although some rigid body simulation methods, such as DEM, can be efficiently parallelized, the coupling to parallel fluid solvers is in general still not supported well. For instance, for an efficient coupling between the frameworks, the domain partitioning of both simulation systems should match in order to minimize communication overhead: Rigid bodies should be known on all processes in which they are contained in the according subdomains. Therefore, next to the
1.2 Particle-Laden Flows

requirement for massively parallel stand-alone simulations suited for instance for granular media (see Section 1.1), there is also the requirement to perfectly support coupling to massively parallel fluid flow solvers.

The following images give an impression of coupled fluid-rigid-body simulations. Figure 1.5 illustrates a fluidization process involving 5,491 spherical particles and $1.92 \cdot 10^7$ (200 × 200 × 480) fluid cells. The simulation was run on 512 processor cores for 252,000 time steps (total runtime: 10 h 25 min). Figure 1.6 shows a segregation simulation of 242,200 spherical particles simulated on 8,192 processor cores in a domain of $1.18 \cdot 10^6$ (384 × 384 × 800) fluid cells (total runtime: 12 h).

Figure 1.5: Coupled LBM/rigid body fluidization scenario (Courtesy of Jan Götz)

Figure 1.6: Segregation simulation of 242,200 spheres simulated on 8,192 cores in a closed box. Density values of 0.8 kg/dm$^3$ and 1.2 kg/dm$^3$ are used for the objects in water with a density of 1 kg/dm$^3$ and a gravitation field. Light particles are rising to the top of the box, while heavy particles fall to the bottom (Courtesy of Jan Götz).
Motivation

These large-scale simulations are a result of the coupling between the massively parallel Lattice Boltzmann fluid simulation framework WALBERLA [40] and the pe rigid body physics engine (see Chapter 2). Via the combination of these two massively parallel simulation frameworks, it is possible for the first time to simulate large-scale scenarios with millions of immersed, fully resolved and arbitrarily shaped rigid bodies incorporating frictional contact dynamics. In [45] we have demonstrated that by this combination the simulation of 37 million fully resolved particles on 8 192 processor cores in a domain of $6.55 \cdot 10^{10}$ fluid cells is possible. Another weak scaling experiment on the Jugene supercomputer (see Appendix A.2) has proven that this simulation can even be run on 294912 processor cores with up to $264\,331\,905$ spherical particles and $1.5 \cdot 10^{11}$ fluid cells.
2  The \textit{pe} Rigid (Multi-)Body Physics Engine

The complexity of the target applications (see Chapter 1) is enormous: the simulation of millions or even billions of interacting rigid bodies, various physical effects, and the coupling to massively parallel fluid dynamics simulations are each by themselves difficult problems on the modeling side. In addition, the software development process poses problems such as reliability, correctness, modularity, extensibility, efficiency, portability, maintainability, to name only a few. The combination of all of these aspects in one framework is therefore extremely intricate and can only be handled by a thorough and systematic software design.

In this chapter I introduce the \textit{pe} rigid (multi-)body physics engine. This engine is designed as a basic framework for all kinds of rigid body simulations ranging from virtual reality scenarios with real-time requirements to large-scale simulations with billions of interacting rigid bodies. Throughout the development of the \textit{pe}, close attention was given to the software design itself to be able to satisfy even the most challenging demands on software quality and to be able to handle the complexity of the target applications. Section 2.1 will summarize the goals of the \textit{pe}. Subsequently, Section 2.2 will enumerate the current features of this software framework, before Section 2.3 will demonstrate its simulation capabilities by means of several simulation scenarios.

2.1 Goals

Most physics engine frameworks focus on virtual reality environments and primarily on the real time requirements of computer games. Several examples of game related physics engines are the Open Dynamics Engine [63], the Havok engine [58], the Bullet engine [57], and NVidia PhysX [62]. Another framework is the OpenTissue framework [64] that primarily aims at physically based animation, but also provides the means to run physically accurate simulations.
The initial attempt to use any of these frameworks in combination with a massively parallel fluid simulation [70] failed for several reasons. Either the engines lacked certain important features, had a bad and unintuitive user interface, no documentation, or they were too much focused on the performance aspect. A common problem with all engines is that they are not prepared for massively parallel simulation, i.e., they do not incorporate the necessary data structures and concepts to be suited for distributed memory simulations.

The lack of a suited rigid body framework motivated the development of the *pe* rigid (multi-)body framework (*pe* being an abbreviation for physics engine). The primary goal of this development was to create a framework suited for all kinds of rigid body methods (DEM, RBD, ...) that provides the software infrastructure for both virtual reality environments as well as physically accurate simulations. Special attention was given to parallelization concepts, which cover thread parallelization in shared memory environments and especially the parallelization in distributed memory environments.

The MPI parallelization and the capability for massively parallel simulations with several billion interacting rigid bodies is a unique feature of the *pe* [68]. A first example of a large-scale rigid body simulation is demonstrated in Figure 2.7. It shows the simulation of 500,000 spheres and boxes falling into a well built of 3,000 fixed boxes. The simulation domain is partitioned into 91 subdomains, where each subdomain is managed by a single processor core. Please note that this simulation only rudimentarily demonstrates the full capacity of the *pe* parallelization and was chosen such that individual objects are still distinguishable in the visualization. The currently largest performed rigid body simulation with *pe* consisted of 2 billion interacting rigid bodies on 131,072 processor cores on the Jugene supercomputer at the Jülich Supercomputing Centre [59] (see Section 10.4). This feature offers completely new possibilities for various research areas, especially if they currently use particle-based simulation approaches to deal with a large number of particles.

Another important goal of the *pe* is to offer an exceptional C++ framework for both scientists and users. Scientists interested in specific aspects of rigid body physics get a very flexible, modular, and extensible framework that can be adjusted to various requirements and simulation scenarios. Due to the modular concept of the *pe*, it is possible to replace every single algorithm of the *pe* (for collision detection, batch generation, collision response, time integration, e.g.) by own implementations. Users solely interested in setting up a particular simulation scenario, both for virtual reality purposes or realistic scenarios, experience a very intuitive and easy to use interface.

In the following listing, the development goals of the *pe* are summarized:

- Software infrastructure for various rigid body simulations (DEM, RBD, ...)
- Support for physically accurate rigid (multi-)body simulations
- Support for rigid body simulations in virtual reality environments
- Support for many parallelization concepts
  - Shared memory parallelization (Threads, OpenMP, ...)

In the following listing, the development goals of the *pe* are summarized:
2.2 Features

2.2.1 Simulation Features

- Large-scale rigid body simulations (currently largest simulation: 2 billion rigid bodies (see [69] and Part III)
- Several algorithms for physically accurate rigid (multi-)body simulations [6]
- Algorithms suited for rigid (multi-)body dynamics in virtual reality environments [11]
- Several predefined primitive geometries: sphere, box, capsule, cylinder, plane [4, 10]
- Compound geometries
- Arbitrary geometries via triangle meshes
- Force generators, such as springs and gravity
- Joints between rigid bodies, such as fixed joints, hinges, and ball joints [5]

2.2.2 Simulation Algorithms

- Accurate, impulse-velocity-based collision response solvers based on linear complementarity problem (LCP) formulations:
  - Lemke algorithm [32]
  - Projected Gauss-Seidel algorithm (PGS) [6]
  - Conjugate Projected Gradient algorithm (CPG) [6] [94]
- Fast collision response algorithms with linear complexity:
  - Fast frictional dynamics solver (FFD) [11] [76]
- Coarse collision detection algorithms
  - Exhaustive Search [39]
2. The pe Rigid (Multi-)Body Physics Engine

- Sweep-And-Prune [21, 36, 38, 39]
- Hierarchical Hash Grids [8] [36, 38, 39]

- Fine collision detection
  - Default collision detection library
  - Convenient integration of third party collision detection libraries

2.2.3 Implementation Features

- Extremely efficient math library for 3-dimensional vectors and matrices [3]
- Sophisticated Expression Template math library for LCPs (see Chapter 8)
- Thread-based parallelization [1]
- GPGPU accelerated simulations [9]
- Massively parallel MPI-based parallelization (see [68] and Part III)
- Thread pool extension for the Boost thread library
- Compile time constraints (see Chapter 3)
- Logging tool for non-parallel, thread-parallel, and massively parallel environments (see Chapter 5)
- Runtime measurement library
- Fully documented source code, including several tutorials and examples of pe simulations.

2.2.4 Visualization

- Raytracing visualization via POV-Ray [92]
- Real-time visualization via Irrlicht [72]
- Visualization via OpenDX [89]
- Real-time editor/debugger [7]

2.3 Simulation Examples

The following pictures give an impression of the simulation capabilities of the pe. The simulation examples range from classical examples of rigid body simulations, as for instance Newton’s cradle (Figure 2.1), to complex simulations that demonstrate the use of compound geometries (see Figure 2.2, 2.5, and 2.6) and show simulations with a higher number of rigid bodies (Figure 2.3, 2.4, and 2.6). Especially noteworthy is the simulation of 500 000 falling bodies demonstrated in Figure 2.7.
2.3 Simulation Examples

Figure 2.1: A very common example of rigid body dynamics is the classical Newton’s cradle scenario: One of five equally sized spheres hits four spheres in a row, which results in the fifth sphere starting to move with exactly the same momentum and kinetic energy as the first sphere, which in turn stops moving. Note that the pe is also able to accurately simulate variations of this scenario, as for instance two spheres hitting the remaining three spheres, or a single larger sphere with twice the mass of the other spheres moving against three smaller spheres.

Figure 2.2: A more sophisticated example is the wrecking ball example. The wrecking ball consists of a sphere and several interlocked chain links, which are compound geometries built of eight individual capsule primitives.

Figure 2.3: The sphere generator is a special scenario for the demonstration of a higher number of random rigid bodies piling in a narrow cavity. The “generator” to the upper left shoots random rigid bodies into the cavity, which quickly start to interact with lots of other rigid bodies.
Figure 2.4: The well example demonstrates the simulation of a high number of random, interacting rigid bodies. This simulation handles 5,000 so-called tri-star unions (each consisting of three capsule primitives).

Figure 2.5: Similar to the wrecking ball example, the chain scenario uses union compound geometries to create chain links built of capsules. However, in this case, the simulation consists of several hundred interlinked chain links.

Figure 2.6: Another impressive example is the castle scenario that shows the destruction of an ancient castle by a huge swinging ram. The castle itself is built of several hundred boxes, the ram is a single capsule geometry and the chain consists of several hundred interlinked chain links, each consisting of four individual capsules.
2.3 Simulation Examples

Figure 2.7: Simulation of 500,000 spheres and boxes falling into a well built of 3,000 fixed boxes. The patches indicate the domains of the 91 MPI processes. Due to the hexagonal setup of the domains, each MPI process has a maximum number of six neighboring processes.
Part II

Software Design of a Massively Parallel Rigid Body Framework
The design of a rigid (multi-)body simulation framework is an extremely challenging task. The basic reasons for this is the complexity of rigid body simulations involving thousands (in case of serial simulations) to billions (in case of massively parallel simulations) interacting rigid bodies and the enormous demands made on such a framework: reliability, correctness, efficiency, robustness, modularity, portability, extensibility, and the intuitive usability (to name the most important) are crucial aspects to be considered during the software design process.

In this part I present six selected implementation highlights of the software developing process of the \textit{pe}. Unfortunately it is not possible to present the \textit{pe} in its entirety due to its complexity and size (which is approximately two hundred thousand lines of C++ code). However, these six selected topics should give an impression of the elaborate software design of the \textit{pe}, which is sometimes achieved by simple yet effective solutions and that is always striving to satisfy all of the requirements mentioned above by an exceptional and elegant C++ implementation [83].

The first topic explained in Chapter 3 is about compile time constraints, which are focused on improving the reliability and correctness of the \textit{pe}. Compile time constraints can both be applied to detect errors at compile time as well as to enforce certain design decisions. Although the basic idea behind compile time constraints is not new (see for instance [28] and [105]), the design of the compile time constraints in the \textit{pe} is unique.

Chapter 4 is devoted to the \textit{pe} smart scopes. The \textit{pe} defines several special scopes/sections focusing on either error detection (for instance in MPI-parallel simulations via the \texttt{pe\_EXCLUSIVE\_SECTION}; see Section 4.2), complexity reduction (for example realized in the \texttt{pe\_GLOBAL\_SECTION}; see Section 4.3), enforcement of programming concepts, or convenience. I will explain the basic idea of these smart scopes and illustrate the purpose and implementation of two particular examples in the context of MPI-parallel rigid body simulations.

Topic number three is the logging functionality of the \textit{pe}. Due to the complexity of the \textit{pe} framework, the logging of errors, warnings, or debug information is one of the most fundamental tasks. Therefore the requirements on the reliability, correctness, and efficiency of the logging tool are accordingly high. Chapter 5 will give a detailed overview of the design of the logging functionality of the \textit{pe}. This tool offers a reliable and efficient information logging for non-parallel, thread-parallel, and massively MPI-parallel environments.

The fourth topic treated in Chapter 6 is the “Sandwich Pattern”. This C++ design pattern was developed in context of the crucial efficiency, modularity, and extensibility requirements of the \textit{pe} and is representative of the fundamental software design of the \textit{pe} framework. Via this design pattern it is possible to replace at compile time certain static and dynamic characteristics of a data type while keeping certain other characteristics unchanged. Because of the use of this design pattern, the \textit{pe} offers a remarkable flexibility to integrate various algorithms, while still providing maximum efficiency without runtime overhead due to abstraction.
Topic number five will deal with the aspect of resource management within the pe framework. Since generally the management of resources is one of the most important tasks in every software framework to guarantee the correctness and reliability of the software, the components dealing with resources require special attention. In Chapter 7 I will give insight into the sophisticated resource management components of the pe framework. Additionally, I will demonstrate the impact of the design of the user interface on the correctness of the resource management.

The last topic treated in this part is the math library of the pe. It is well known that performance optimized linear algebra operations are an essential ingredient for basically all numerical packages. Therefore several standard frameworks are focused on the performance optimized implementation of these operations, as for instance several packages implementing the BLAS standard. However, next to performance, also reliability, usability, and maintainability are important aspects of such a framework, which in many cases only play a subordinate role in their development.

In Chapter 8 I will introduce the smart expression template programming technique used in the math library of the pe. This technique offers an expression-dependent evaluation and optimization of mathematical vector and matrix operations, the automatic detection of aliasing effects, and combines the usability and maintainability of standard C++ programming with the efficiency of an optimized BLAS library. I will demonstrate the efficiency of this approach by comparing the pe math library with the Boost uBLAS library.
3 Compile Time Constraints

Due to the complexity of the pe physics engine, the detection of errors is of crucial importance for the correctness and reliability of the software framework. Therefore the development and application of powerful error detection mechanisms is of utmost significance. In this chapter I will introduce one of the special error detection mechanisms used throughout the pe: the compile time constraints. The pe compile time constraints are an invaluable tool to detect compile time errors and in case of an error to abort the compilation process and report the error by emitting a comprehensible error message. Additionally, they offer the option to enforce certain design decisions.

3.1 Motivation

Most errors of a C++ program occur during the runtime of the program: Invalid access indices, violated pre- and postconditions of a function, failed memory allocations, files that cannot be opened, to name only a few. These kinds of errors are usually handled appropriately by either throwing an exception or aborting the program by an assertion [15, 17]. However, some errors can already or only be detected at compile time. Using exceptions or assertions for these kind of errors would delay the error report until the program is run for the first time. Consider for example Listing 3.1:

```
const std::size_t maxThreads = 4;
// ...
assert( maxThreads > 0 ); // Is the number of threads larger than zero?
```

Imagine that the maximum number of threads is configured via a constant integral value. However, at some point we have to make sure that this value is larger than zero (because zero threads would not be able to perform any work). Due to the use of an assertion, we are only able to detect the error at runtime, although the value is a compile time constant
expression and could therefore be reported at compile time\(^1\). A different kind of error is illustrated in Listing 3.2:

```
Listing 3.2: Second example of a compile time error

typedef int real; // Is this really a floating point data type?
```

In this case we want to create a convenient switch between single-precision floating point values and double-precision floating point values by providing a single typedef defining the new floating point type `real`. This new type is now used throughout our program, which allows us to switch between `float` and `double` values by changing the type in the type definition. However, inadvertently the intended floating point type was set to an integral data type, which will probably make our results useless.

In order to detect errors of these kinds and to report them at compile time, it is desirable to add code that either aborts compilation and reports the error as descriptive as possible, or in case no error is detected cleanly compiles and creates no additional overhead at runtime. A solution for this problem are static assertions and compile time constraints, as for example proposed and implemented in the Boost library [28] and admitted to the technical report 1 (TR1) of the C++ standard [98, 23, 24, 25].

In this chapter I introduce the unique design of the compile time constraints of the *pe* framework. They are a combination of the static assertion from the Boost library [28] and several ideas from Andrei Alexandrescu [14] and Matthew Wilson [105] to create comprehensible error messages. Section 3.2 will demonstrate several ways how to abort compilation in case of an error depending on a compile time constant expression and will introduce the Boost static assertion. Section 3.3 will handle the topic of comprehensible error messages, before Section 3.4 will describe the *pe* compile time constraints in detail. Section 3.5 and Section 3.6 will highlight two interesting compile time constraints.

### 3.2 Static Assertion

The basic idea of static asserts is to provide compile time validation of an expression. Clearly only compile time constant expressions can be evaluated at compile time:

```
Listing 3.3: Proper use of static assertions

// Compile time evaluation of a compile time static expression
STATIC_ASSERT( sizeof(double) >= sizeof(float) );

// Compilation error: Attempt to evaluate a runtime expression at
// compile time
double& Vector::operator[]( size_t index )
{
    STATIC_ASSERT( index < size_ ); // Bound check of the vector access index
    // ...
}
```

\(^1\)Even worse, we are only able to detect this error in case we did not specify NDEBUG during the compilation!
3.2 Static Assertion

The first of these two examples illustrates the use of a static assertion for a compile time constant expression, i.e., an expression whose result can be calculated at compile time. In this case the static assertion will not abort the compilation process, since the size of a double precision floating point value should always be larger or equal to a single precision floating point value. The second of the examples attempts to apply a static assertion to a runtime expression. Obviously, it is not possible to tell at compile time whether or not the index will be within bounds.

In case a compile time constant expression evaluates to true, the static assertion should not abort the compilation process and in the optimal case should also not create any additional overhead by adding code that has to be evaluated at runtime. In case an expression evaluates to false, the static assertion should abort the compilation process immediately with an error message that should indicate as accurately as possible what went wrong.

Unfortunately, static asserts are not part of the C++ programming language (otherwise this would be something to be explained in every C++ primer). As a result, it is up to the programmer to simulate static assertions as good as possible using the language features available. Obviously it is necessary to use the result of the compile time constant expression to generate either perfectly valid code in case the expression evaluates to true, or to generate invalid code that the compiler rejects by emitting an error message.

There are several ways to create invalid code based on the result of an expression\(^2\). Several of these ideas work for both C and C++, whereas one of these ideas is restricted to C++ only. The most commonly used approach is to declare an array of either size 1 in case the expression is true, or 0 in case the expression evaluates to false:

**Listing 3.4:** First approach for static assertions

```
#define STATIC_ASSERT( expression ) \
int array[(expression) ? (1) : (0)]

// ...

STATIC_ASSERT( sizeof(int) < sizeof(short) ); // Compile time error
```

Depending on the result of the expression, the size of the array will be either 1 or 0. Since an array size of zero is invalid in both C and C++, the compiler will issue an error in case the result of the given expression is false. Note that the definition of the static assert in this way will limit its usability since it introduces the variable array into the current scope and also potentially adds code overhead. Additionally, although arrays of size 0 are invalid some compilers merely issue a warning instead of a compilation error. The following scheme resolves both of these issues:

**Listing 3.5:** Improvement of the first approach for static assertions

```
#define STATIC_ASSERT( expression ) \
do { typedef int array[(expression) ? (1) : (-1)]; } while(0)
```

\(^2\)The approaches illustrated in this chapter are taken from [105].
The array is wrapped in the scope of a do-while-loop, that is at maximum executed once. However, due to the single type definition inside the loop, the compiler can easily remove it from the executable. Additionally, instead of creating an array of size 0 in case of a \texttt{false} expression, an array of size -1 is created, which causes all compilers to abort the compilation process.

A second approach to generate invalid code in case of a \texttt{false} expression is based on switch statements. In this case, the generation of invalid code is based on the requirement that each case clause must have a different value \cite{105}:

\begin{verbatim}
Listing 3.6: Second approach for static assertions
#define STATIC_ASSERT ( expression ) \nswitch ( 0 ) { case 0: case ( expression ); }
\end{verbatim}

A third approach relies on the fact that bitfields must have a size of at least one bit \cite{105}:

\begin{verbatim}
Listing 3.7: Third approach for static assertions
#define STATIC_ASSERT ( expression ) \nstruct x { unsigned int v : ( expression ); }
\end{verbatim}

Another idea is based on the compile time instantiation of a template specialization. Therefore this approach is only suited for C++ codes:

\begin{verbatim}
Listing 3.8: Fourth approach for static assertions
template < bool >
struct STATIC_ASSERT_FAILED;
template<>
struct STATIC_ASSERT_FAILED<true>
{
enum { value = 1 };}
#define STATIC_ASSERT ( expression ) \n{ STATIC_ASSERT_FAILED<expression> assert_failed; }
\end{verbatim}

The idea here is to first declare a template class with a single boolean template argument and to specialize this class for a value of \texttt{true}. In case the expression evaluates to \texttt{true}, the specialization is selected and the compiler creates an object of type \texttt{STATIC_ASSERT_FAILED<true>} within a new scope. In case the expression evaluates to \texttt{false}, the compiler tries to instantiate \texttt{STATIC_ASSERT_FAILED} for \texttt{false}, but only has the declaration available. Therefore the compilation process is aborted with the error message that no definition for \texttt{STATIC_ASSERT_FAILED<false>} can be found.

The static assertion in the Boost library is based on the last version of \texttt{STATIC_ASSERT}. However, the Boost library improves the idea in one very important aspect: Most compilers will complain about the unused and unreferenced object \texttt{assert_failed} \footnote{At least if all warnings are enabled.}. Therefore the Boost library transforms the definition of a temporary object into a type definition:
### 3.2 Static Assertion

#### Listing 3.9: The Boost static assertion

```cpp
namespace boost {

#define BOOST_JOIN ( X, Y ) BOOST_DO_JOIN ( X, Y )
#define BOOST_DO_JOIN ( X, Y ) BOOST_DO_JOIN2(X,Y)
#define BOOST_DO_JOIN2 ( X, Y ) X##Y

template < int > struct BOOST_STATIC_ASSERT_TEST {}; 

// Undefined base template

template < bool > struct STATIC_ASSERT_FAILURE;

// Specialization for true

template<> struct STATIC_ASSERT_FAILURE <true> { enum { value = 1 }; }; 

// Macro definition

#define BOOST_STATIC_ASSERT ( expression ) 
  typedef 
  BOOST_STATIC_ASSERT_TEST< 
  sizeof ( STATIC_ASSERT_FAILURE < ( bool )( expression ) > ) > 
  BOOST_JOIN ( BOOST_STATIC_ASSERT_TYPEDEF , __LINE__ )

} // namespace boost
```

The basic idea of selecting either the defined template specialization for `true` expressions and to abort the compilation in case the undefined template is required for `false` expressions is still the same as before. However, since instead of an object in this approach a type definition is created, the definition of this type alone would not suffice:

#### Listing 3.10: Instantiation of the `STATIC_ASSERT_FAILURE` template

```cpp
// No compiler error
typedef STATIC_ASSERT_FAILURE<true> BOOST_STATIC_ASSERT_TYPEDEF_1;

// Also no compiler error!
typedef STATIC_ASSERT_FAILURE<false> BOOST_STATIC_ASSERT_TYPEDEF_2;
```

For the type definition alone, the template does not have to be instantiated. The compiler accepts both of these type definitions, even though the second type definition is based on the undefined base template. Therefore even for a `false` expression this would not result in a compilation error. In order to force the instantiation of the `STATIC_ASSERT_FAILURE` template, two ideas can be used. The first idea is to explicitly access the `value` member enumeration of the `STATIC_ASSERT_FAILURE` template in order to instantiate the `BOOST_STATIC_ASSERT_TEST` template. Now the compiler has to instantiate the template in order to check the compile time constant value of the member enumeration to be able to create the according type definition. The second idea is to evaluate the size of the `STATIC_ASSERT_FAILURE` type. This also forces the compiler to instantiate the type. However, in order to be able to define a new type in a type definition, the `BOOST_STATIC_ASSERT_TEST` still has to be wrapped around the construct. Note that the argument to the `BOOST_STATIC_ASSERT` is explicitly cast to `bool` using old-style casts. The reason for this is that currently too many compilers have problems with `static_cast` when used inside integral constant expressions.
The second part of the \texttt{BOOST_STATIC_ASSERT} type definition is the \texttt{BOOST_JOIN} macro. This piece of macro magic joins the two arguments together, even when one of the arguments is itself a macro (see section 16.3.1 of the C++ standard [98]). The key is that macro expansion of macro arguments does not occur in \texttt{BOOST_DO_JOIN2} but does in \texttt{BOOST_DO_JOIN} [28].

### 3.3 Comprehensible Error Messages

Although these techniques allow for the detection of compile time errors and the abortion of the compilation process, one major problem remains: Comprehensible error messages. Consider the static assertion as introduced in Listing 3.5 by creating an array of negative length in case the compile time static expression evaluates to \texttt{false} in the following example:

\begin{alltt}
\begin{verbatim}
# define STATIC_ASSERT( expression )
   do { int array[(expression)?(1):(-1)]; } while(0)
// ...
STATIC_ASSERT( sizeof(double) < 1 );
\end{verbatim}
\end{alltt}

Listing 3.11: First example of bad error messages

The example is constructed such that the expression will always evaluate to \texttt{false}. What we would hope for is a compilation error telling us that the size of a double value will never be smaller than one byte. However, imagine the poor programmer confronted with the following error message (G++ 4.2.1):

\textbf{Error: Size of array "array" is negative}

Here the compiler is perfectly right in complaining about an invalid, negative array size. However, the error message is in no way corresponding to the actual error, i.e., that our assumption that a \texttt{double} has less than one byte will always be wrong.

Note that depending on the compiler, we can improve the situation slightly. If the compiler (as in the case of the GCC compiler) includes the name of the array in the error message, we can use this to give the programmer the information that this particular error corresponds to a failed compile time assertion:

\begin{alltt}
\begin{verbatim}
# define STATIC_ASSERT( expression )
   do { int STATIC_ASSERTION_FAILED[(expression)?(1):(-1)]; } while(0)
\end{verbatim}
\end{alltt}

Listing 3.12: Second example of bad error messages

Now the compiler will give us the following error message:

\textbf{Error: Size of array "STATIC_ASSERTION_FAILED" is negative}

This improves the situation somewhat: Although this error message still does not contain any information about the actual error, it points to the fact that a static assertion has failed. Using the \texttt{BOOST_STATIC_ASSERT}, the error messages read for instance as following:
Error: Invalid use of "sizeof" for incomplete type
"boost::STATIC_ASSERTION_FAILURE<false>"

In this case, the error message contains the name of the template instance the compiler
failed to instantiate. Again, the name is not corresponding to the actual error, but a pro-
grammer gets the idea that a compile time assertion failed due to an invalid condition.
Although the kind of the error is still a mystery, the programmer gets a hint of what to
expect if he looks for the source of the error.

As already stated, compile time constraints are not part of the C or C++ programming
languages. Therefore, although these error messages are far from being precisely aiming at
the actual error, this is the best that we can get by using the available language features.
An important realization is that we have to name our variables/classes accordingly and have
to hope that the compiler will print the name in the corresponding error message to give us
a hint of the source of the error.

3.4 The pe Compile Time Constraints

The pe compile time constraints are an extension of the ideas of BOOST_STATIC_ASSERT. As
illustrated in the last section, the only drawback of this macro is the comprehensibility of
the resulting error messages that always point out the instantiation failure of the undefined
STATIC_ASSERTION_FAILURE base template. Therefore the consequent continuation of this scheme is
to create several specific FAILURE classes that point to an individual error as closely as possible.
The following example demonstrates this for the pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE
compile time constraint:

Listing 3.13: Example for a pe compile time constraint

```
// This class is used as a wrapper for the instantiation of the specific
// constraint class templates. It serves the purpose to force the
// instantiation of either the defined specialization or the undefined
// basic template during the compilation. In case the compile time
// condition is met, the type pe::CONSTRAINT_TEST<> is defined.
template< int > struct CONSTRAINT_TEST {};

// Helper template for the compile time constraint enforcement. Based on
// the compile time constant expression used for the template instantiation,
// either the undefined basic template or the specialization is selected. If
// the undefined basic template is selected, a compilation error is created.
template< bool > struct CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_FAILED;
template<> struct CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_FAILED<true> {
    enum { value = 1 };
};

// Creates a compilation error in case T is not a floating point data type
#define pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE(T) \
    typedef pe::CONSTRAINT_TEST< \
        pe::CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_FAILED< \
            boost::is_floating_point<T>::value >::value > \
        BOOST_JOIN( CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE_TYPEDEF, __LINE__ )
```
3 Compile Time Constraints

All \(\text{pe}\) constraints work according to this pattern: The \(\text{pe}::\text{CONSTRAINT\_TEST}\) template is used as a wrapper around the instantiation of the accordingly named constraint class. These classes are always named in capital letters in order to stand out in the resulting error messages and such that the error is described as accurately as possible. Additionally, as a convention, all compile time constraints use the prefix \text{CONSTRAINT\_} to indicate the purpose of these classes.

In contrast to the Boost static assertion, the \(\text{pe}\) employs the strategy to use the member enumeration value of the corresponding constraint class to instantiate the \text{CONSTRAINT\_TEST} and does not use the \text{sizeof} operator. Also in contrast to Boost, the compile time test is already incorporated in the constraint. In this example, we can use one of the Boost type traits, \text{boost::is\_floating\_point} [28]. In case the given data type is a (possibly cv-qualified) floating point data type, the value member enumeration of the type trait evaluates to \text{true}, otherwise to \text{false}.

The instantiation of the constraint class is wrapped in a type definition in an accordingly named macro. By convention, the macro is named after the constraint class and has an additional prefix \text{pe\_} to indicate its origin and to minimize name clashes.

This compile time constraint is for instance used in the \text{Quaternion} class of the math library\(^4\) of the \(\text{pe}\):

```
Listing 3.14: Application of a \(\text{pe}\) compile time constraint

1 template<typename Type>
2 class Quaternion {
3     // ...
4     Type values_[4];
5     // ...
6     pe_CONSTRAINT_MUST_BE_FLOATING_POINT_TYPE( Type );
7     // ...
8 };
```

The \text{Quaternion} class is used for representing the orientation of rigid bodies in a rigid body simulation. The determinant of the \text{Quaternion} will always be 1. Therefore the four values of the \text{Quaternion} have to be in the range \([0..1]\) during the entire simulation and independent of the current orientation of the objects. In consequence, the values of a \text{Quaternion} can only be floating point values. This design is enforced by the \text{pe\_CONSTRAINT\_MUST\_BE\_FLOATING\_POINT\_TYPE} compile time constraint. Due to the compile time constraint, it is not possible to instantiate the \text{Quaternion} with any data type except built-in (fundamental) floating point data types (i.e., \text{float}, \text{double}, and \text{long double}).

By the time of this writing, the \(\text{pe}\) altogether features 44 compile time constraints. The following list gives a complete overview of the available constraints. As you will note, in most cases every constraint is available in a positive and a negative formulation, as for instance the \text{pe\_CONSTRAINT\_MUST\_BE\_POD} and \text{pe\_CONSTRAINT\_MUST\_NOT\_BE\_POD}. This results in several constraints with similar meanings, but allows for a very flexible selection of constraints according to the current programming context. For instance, \text{CONSTRAINT\_MUST\_BE\_BASE\_OF} and

\(^4\)For a detailed insight into the \(\text{pe}\) math library, see Chapter 8.
3.4 The \textit{pe} Compile Time Constraints

\texttt{CONSTRAINT\_MUST\_BE\_DERIVED\_FROM} can be interchanged by switching the arguments, but depending on the context the constraint is used one or the other may be preferable.

- \texttt{pe\_CONSTRAINT\_MUST\_BE\_ARITHMETIC\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_ARITHMETIC\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_BASE\_OF(B, D)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_BASE\_OF(B, D)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_STRICLY\_BASE\_OF(B, D)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_STRICLY\_BASE\_OF(B, D)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_BOOLEAN\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_BOOLEAN\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_BUILTIN\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_BUILTIN\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_CLASS\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_CLASS\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_POINTER\_MUST\_BE\_COMPARABLE(P1, P2)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_CONST(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_CONST(T)}
- \texttt{pe\_CONSTRAINT\_POINTER\_MUST\_BE\_CONVERTIBLE(FROM, TO)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_DERIVED\_FROM(D, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_DERIVED\_FROM(D, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_STRICLY\_DERIVED\_FROM(D, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_STRICLY\_DERIVED\_FROM(D, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_FLOATING\_POINT\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_FLOATING\_POINT\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_INTEGRAL\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_INTEGRAL\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_NUMERIC\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_NUMERIC\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_POD(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_POD(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_POINTER\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_POINTER\_TYPE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_HAVE\_SAME\_SIZE(T1, T2)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_HAVE\_SAME\_SIZE(T1, T2)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_SAME\_TYPE(A, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_SAME\_TYPE(A, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_STRICLY\_SAME\_TYPE(A, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_STRICLY\_SAME\_TYPE(A, B)}
- \texttt{pe\_CONSTRAINT\_MUST\_HAVE\_SIZE(T, S)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_HAVE\_SIZE(T, S)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_SUBSCRIPTABLE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_SUBSCRIPTABLE\_AS\_DECAYABLE\_POINTER(T)}
- \texttt{pe\_CONSTRAINT\_SOFT\_TYPE\_RESTRICTION(T, TYPELIST)}
- \texttt{pe\_CONSTRAINT\_TYPE\_RESTRICTION(T, TYPELIST)}
- \texttt{pe\_CONSTRAINT\_MUST\_BE\_VOLATILE(T)}
- \texttt{pe\_CONSTRAINT\_MUST\_NOT\_BE\_VOLATILE(T)}
3 Compile Time Constraints

3.5 The MUST_BE_BASE_OF constraints

Let us highlight some of the more interesting constraints. The CONSTRAINT_MUST_BE_BASE_OF constraint enforces that a certain class must be a public or non-public direct or indirect base class of a second class:

```
class A {}
class B : public A {}
class C : public B {}
class D : private A {}
class E;

int main()
{
    pe_CONSTRAINT_MUST_BE_BASE_OF ( A, A ); // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF ( A, B ); // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF ( A, C ); // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF ( A, D ); // No compilation error
    pe_CONSTRAINT_MUST_BE_BASE_OF ( A, E ); // Compilation error!
}
```

Only the fifth constraint will abort the compilation to report that E is not related to A. Note that this constraint allows for the test whether A is a base of A, i.e., A can be seen as directly related to A. In contrast to this constraint, the CONSTRAINT_MUST_BE_STRICTLY_BASE constraint creates a compilation error in case the two given types are equal:

```
class A {}
class B : public A {}
class C : public B {}
class D : private A {}
class E;

int main()
{
    pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF ( A, A ); // Compilation error!
    pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF ( A, B ); // No compilation error
    pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF ( A, C ); // No compilation error
    pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF ( A, D ); // No compilation error
    pe_CONSTRAINT_MUST_BE_STRICTLY_BASE_OF ( A, E ); // Compilation error!
}
```

Internally this is solved by the concatenation of several is_base_of type traits from the Boost library:

```
// Definition of the soft MUST_BE_BASE_OF constraint macro
#define pe_CONSTRAINT_MUST_BE_BASE_OF(B,D) \
    typedef \
    CONSTRAINT_TEST< \
    CONSTRAINT_MUST_BE_BASE_OF_FAILED< is_base_of<B,D>::value >::value > \
    BOOST_JOIN( CONSTRAINT_MUST_BE_BASE_OF_TYPEDEF, __LINE__ )
```

32
3.6 The MUST_BESAME_TYPE constraints

Two other interesting constraints are focused on evaluating at compile time if two given data types are the same: The pe_CONSTRAINT_MUST_BE_SAME_TYPE compile time constraint and the pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE constraint. The difference between these two constraints is that the former ignores any cv-qualifiers on both types, whereas the latter one takes them into account:

Listing 3.18: The MUST_BESAME_TYPE constraints

```cpp
pe_CONSTRAINT_MUST_BE_SAME_TYPE( int, int ); // No error
pe_CONSTRAINT_MUST_BE_SAME_TYPE( const int, int ); // No error
pe_CONSTRAINT_MUST_BE_SAME_TYPE( int, volatile int ); // No error
pe_CONSTRAINT_MUST_BE_SAME_TYPE( float, double ); // Error!

pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( int, int ); // No error
pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( const int, int ); // Error!
pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( int, volatile int ); // Error!
pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE( float, double ); // Error!
```

In order to test whether two types are the same, the Boost library offers the is_same type trait. However, this type trait always takes the cv-qualifiers into account and would therefore only be suited for the second constraint. Therefore, the pe implements two additional type traits according to the naming convention of the pe:

Listing 3.19: The IsSame type traits

```cpp
template< typename A, typename B >
struct IsSame
{
  private:
    class No {};  
    class Yes { No no[2]; }; 

    static A& createA();
    static B& createB();

    static Yes testA( const volatile B& );
    static No testA(...);
    static Yes testB( const volatile A& );
    static No testB(...);

  public:
    enum { value = ( sizeof( testA( createA() ) ) == sizeof( Yes ) ) &&
    ( sizeof( testB( createB() ) ) == sizeof( Yes ) )
    };
};
```
The `IsSame` type trait is defined according to ideas of Andrei Alexandrescu [14] and Herb Sutter [52] to test the relationship of two data types. The `IsStrictlySame` type trait relies on the strict type matching of template instantiations to test whether two types are exactly the same.

Using these two type traits, the two constraints can be implemented as follows:

### Listing 3.20: Implementation of the MUST_BE_SAME_SIZE constraints

```cpp
#define pe_CONSTRAINT_MUST_BE_SAME_TYPE(A,B) \
    CONSTRAINT_TEST< \
        CONSTRAINT_MUST_BE_SAME_TYPE_FAILED< IsSame<A,B>::value >::value > \
    BOOST_JOIN( CONSTRAINT_MUST_BE_SAME_TYPE_TYPEDEF, __LINE__ )
```

```cpp
#define pe_CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE(A,B) \
    CONSTRAINT_TEST< \
        CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE_FAILED< \
        IsStrictlySame<A,B>::value >::value > \
    BOOST_JOIN( CONSTRAINT_MUST_BE_STRICTLY_SAME_TYPE_TYPEDEF, __LINE__ )
```

### 3.7 Conclusion

In this chapter I have introduced the compile time constraints of the `pe`. Although the basic idea of compile time constraints is not new, the design of these constraints in the `pe` is unique. By the combination of ideas from the Boost library, Andrei Alexandrescu, and Matthew Wilson and by pursuing this road of development, a very powerful error detection mechanism was created within the context of the `pe`, which helps to ensure the correctness and reliability of this complex software framework.
This chapter is devoted to the pe smart scopes. These extended C++ scopes are either used to detect errors (as for instance deadlock situations in MPI-parallel environments), reduce the complexity of certain tasks, enforce programming concepts, or simply for convenience. In Section 4.1 I will explain the basic idea behind the pe smart scopes. The following two sections will show two examples of two of the most commonly applied smart scopes. Section 4.2 will give insight into the implementation details of the pe_EXCLUSIVE_SECTION, which is used in MPI-parallel simulations to introduce a scope that is exclusively executed by a single process. Section 4.3 will introduce the pe_GLOBAL_SECTION used for the setup of globally known, process spanning rigid bodies.

4.1 The Basic Concept

The problems the pe smart scopes address are manifold. Some of them are designed to detect errors. Others are designed to reduce the complexity of standard tasks and therefore prevent errors from happening. Yet others are enforcing a certain programming concept of the pe. Or they are merely convenient. A first example of a pe smart scope, the pe_EXCLUSIVE_SECTION, is illustrated in Listing 4.1. Via this tool it is possible to create an execution scope for a specific process in a SPMD (Single Processor, Multiple Data) environment, within which it is possible to detect possible deadlock situations:

Listing 4.1: First example of the use of a pe smart scope

```
// Exclusive section: This section is only executed by process 0
// and skipped by all other processes.
pe_EXCLUSIVE_SECTION ( 0 ) {  
  // ...  
}
```

Another example is the pe_GLOBAL_SECTION smart scope for the setup of globally known rigid bodies in a MPI-parallel environment (see Listing 4.3). In the pe context, it enforces the
concept of global rigid bodies: Whereas usually rigid bodies have to be created on exactly one process (see Section 9.2), some very large, process spanning rigid bodies, as for instance infinite planes, should be known on all processes. In order to create these global bodies, the pe\_GLOBAL\_SECTION smart scope has to be used:

```
Listing 4.2: Second example of the use of a pe smart scope
1 // ...
2 // Global parallel region
3 // The pe\_GLOBAL\_SECTION macro starts a global section for the setup of
4 // global rigid bodies. This smart scope relaxes the restriction that
5 // rigid bodies may only be created on the process their center of mass
6 // is contained in.
7 pe\_GLOBAL\_SECTION
8 {
9 // Setup of a global sphere
10 createSphere( 2, Vec3( 10.0 , 20.0 , -40.0 ), 5.0 , oak );
11
12 // Setup of a global union
13 pe\_CREATE\_UNION( union3 , 3 ) {
14 // ...
15 }
16 }
17
18 // ...
```

These two examples clearly show the advantages of smart scopes in general. They ...

- ... offer an easy, clear, and intuitive user interface.
- ... stand out very visible in the source code and clearly mark the beginning of a new scope.
- ... carry semantical information in that they are explicitly used by a programmer and convey the message that all code contained in their scope is subject to special treatment and/or effects.

Although the purposes of the different smart scopes are manifold, the ideas behind their implementation are very similar. All scopes are based on the perfect symmetry of construction and destruction of stack objects, whose most popular implementation is the RAII idiom [97, 96, 53, 34], in combination with a macro and an if-statement. Listing 4.3 demonstrates the general design of a smart scope:

```
Listing 4.3: Implementation concept of the smart scopes
1 #define pe\_NAME\_OF\_THE\_SMART\_SCOPE( ARG1, ARG2, ... ) \n2 if( pe::ScopeObject scopeObject = pe::ScopeObject( (ARG1), (ARG2), ... ) )
```

All smart scopes are implemented as a macro to hide the details of the implementation. According to this design decision, the names of all smart scopes are in capital letters preceded by a lower case pe_ prefix (in this case pe\_NAME\_OF\_THE\_SMART\_SCOPE). A positive side effect is that due to the capital letters smart scopes usually stand out clearly in the surrounding source code. Depending on the requirements on the smart scope, the macro may take any
numbers of arguments (even zero), in the example indicated by ARG1 and ARG2. During the preprocessor phase, the macro is replaced by a simple if-statement, whose conditional expression contains the core of the smart scopes: Within the scope of the if-statement, a temporary object is created on the stack and initialized as required depending on the given arguments. The sole implementational requirements on this object is that it possesses an accessible copy constructor and can be converted to a data type that can be used to evaluate the conditional expression. During construction of this temporary object the special conditions and effects expected from the according smart scope are initiated. The temporary object persists until the end of the scope of the if-statement. After that it is destroyed and its destructor is called, which finalizes the work of the smart scope and resets all conditions and/or effects.

The choice to use an if-statement is based on several considerations. As the name suggests, smart scopes are focused on creating a new scope, within which the special property of the smart scope is in effect. There are only a couple of alternative language features that allow to create a new scope. However, several of them cannot be used as intended within the scope of a function, as for instance namespace, class and function scopes. Valid alternatives are however the for-loop, the while-loop, and the switch-statement. In comparison to the if-statement, however, all three alternatives have severe disadvantages. The switch-statement for instance requires at least one case-label within its scope. This would prevent an intuitive use of a smart scope and would leak implementation details. On the other hand, due to their nature, the two loop constructs are much more complicated to use for the purpose of smart scopes than the if-statement. Among these four choices, the if-statement is by far the best alternative, even if the smart scope does not require a conditional initiation. But even the if-statement is not without a disadvantage. Due to the nature of the if-statement the following example will result in unexpected behavior:

Listing 4.4: Unexpected behavior of a pe_EXCLUSIVE_SECTION (I)

```c
1 // Runtime condition (might be changed)
2 int condition = 0;
3
4 // ...
5
6 // If-else branches, both containing an exclusive section (?)
7 if( condition == 0 )
8     pe_EXCLUSIVE_SECTION( 0 ) {
9         // ...
10    }
11 else
12     pe_EXCLUSIVE_SECTION( 1 ) {
13         // ...
14    }
```

The intention of this code is to execute an exclusive section for process 0 in case the variable condition is equal to zero, and an exclusive section for process 1 otherwise. However, although the indentation of the code suggests this behavior, due to the if-statement hidden inside the pe_EXCLUSIVE_SECTION macro the true flow of execution is the following:
4 Smart Scopes

Listing 4.5: Unexpected behavior of a pe_EXCLUSIVE_SECTION (II)

```c
if( condition == 0 )
  pe_EXCLUSIVE_SECTION( 0 ) { 
  // ...
}
else
  pe_EXCLUSIVE_SECTION( 1 ) { 
  // ...
}
```

By adjusting the indentation, it becomes obvious that the else-branch is not related to the initial if-statement, but to the hidden one in the first pe_EXCLUSIVE_SECTION macro. Therefore the flow of execution is not as the programmer of this code intended it to be. This error, however, can be easily prevented by added brackets to the initial if-condition and the related else-branch:

Listing 4.6: Unexpected behavior of a pe_EXCLUSIVE_SECTION (III)

```c
if( condition == 0 ) {
  pe_EXCLUSIVE_SECTION( 0 ) {
  // ...
}
else {
  pe_EXCLUSIVE_SECTION( 1 ) {
  // ...
}
```

Due to these brackets the flow of execution is safe and works as expected. In summary it can be stated that (ab-)using an if-statement is the best choice for creating smart scopes and its disadvantage, which results from the general problem to use macros [34, 52], can easily be overcome by a clean programming style.

The next section will give more insight into the implementation of two smart scopes. Section 4.2 will explain in detail the implementation of the pe_EXCLUSIVE_SECTION smart scope, whereas Section 4.3 will explain the pe_GLOBAL_SECTION smart scope.

4.2 The pe Exclusive Section Scope

The pe_EXCLUSIVE_SECTION provides the feature to start a code section exclusively executed by a single process in a MPI parallel environment. Listing 4.7 shows an example of the application of the pe_EXCLUSIVE_SECTION:

Listing 4.7: Application of the pe_EXCLUSIVE_SECTION

```c
int main( int argc , char** argv )
{
  // Initialization of the MPI system
  // The MPI system must be initialized before any pe functionality is used.
  // It is recommended to make MPI_Init() the very first call of the main
  // function.
  MPI_Init( &argc , &argv );
```

38
4.2 The pe Exclusive Section Scope

```c
// Parallel region
// This code is executed by all processes of the parallel simulation.
// ...

// Exclusive section
// This section is only executed by process 0 and skipped by all other
// processes. This can for example be used to perform special setups or
// checks in a parallel simulation environment.
pe_EXCLUSIVE_SECTION( 0 ) {  
    // ...
}

// Second parallel region
// This code is again executed by all processes of the parallel
// simulation.
// ...

// Finalizing the MPI system
// The MPI system must be finalized after the last pe functionality has
// been used. It is recommended to make MPI_Finalize() the very last call
// of the main function.
MPI_Finalize();
```

Irrespective of the total number of MPI processes, the section introduced by `pe_EXCLUSIVE_SECTION` is exclusively executed by process 0. All other processes skip this code section entirely. Programmers familiar with MPI recognize this concept as basic principle to distinguish between the different processes in the SPMD environment of MPI. Usually, differentiations between processes are made by the use of a simple `if`-statement:

**Listing 4.8:** Default implementation of behavioral differentiation

```c
// Starting a code region that is only executed by process 0
if( rank == 0 ) {
    // ... Code that is exclusively executed by process 0
}
```

The problem with this code is the missing possibility of code inside the `if`-statement to check whether or not it is executed on a single process or on all processes. For instance, it would be possible to call functions that expect to be executed in parallel on all processes, as for instance the `simulationStep` function, inside the scope of the `if`-statement and with that provoke a deadlock situation:

**Listing 4.9:** Default implementation of behavioral differentiation

```c
// Starting a code region that is only executed by process 0
if( rank == 0 ) {
    // Starting a single simulation step exclusively on process 0.
    // This results in a deadlock since internally the function expects
    // to be called on all involved MPI processes.
    theWorld() -> simulationStep( 0.01 );
}
```
4 Smart Scopes

Using a `pe_EXCLUSIVE_SECTION` instead, the `simulationStep` function is able to detect that it is exclusively executed on a single MPI process and can respond accordingly:

```
Listing 4.10: Recommended implementation of behavioral differentiation

// Starting a code region that is only executed by process 0
pe_EXCLUSIVE_SECTION( 0 )
{
    // Starting a single simulation step exclusively on process 0.
    // The function can detect the erroneous call inside an exclusive section
    // and will respond with a std::runtime_error exception
    theWorld()->simulationStep( 0.01 );
}

// ...”
```

Due to the obvious advantages of the `pe_EXCLUSIVE_SECTION` smart scope in comparison to a simple `if`-statement and in order to avoid inexplicable deadlock situations, throughout the `pe` engine the `pe_EXCLUSIVE_SECTION` is used instead of a plain `if`-statement.

Listing 4.11 shows the implementation of the `pe_EXCLUSIVE_SECTION`. As already indicated by the capital letters, it is implemented as a macro pseudo function:

```
Listing 4.11: Implementation of the `pe_EXCLUSIVE_SECTION` macro

#define pe_EXCLUSIVE_SECTION( RANK ) \
    if( pe::ExclusiveSection exclusiveSection = ( RANK ) )
```

Internally, the `pe_EXCLUSIVE_SECTION` turns out to be also based on an `if`-statement. However, instead of a simple comparison, the temporary object `exclusiveSection` of type `ExclusiveSection` is created and initialized with the given rank\(^1\). This object’s lifetime is bound to the scope of the `if`-statement, i.e., the moment the scope of the `if`-statement ends, the object is destroyed. Obviously, the implementation of the `pe_EXCLUSIVE_SECTION` relies on the perfect symmetry of construction and destruction of the `ExclusiveSection` class:

```
Listing 4.12: Implementation of the `ExclusiveSection` class

class ExclusiveSection
{
    public:
        ExclusiveSection( int rank )
        {
            if( active_ )
                throw std::runtime_error( "Nested exclusive sections detected" );

            if( GlobalSection::isActive() )
                throw std::logic_error( "Invalid exclusive inside global section" );

            if( rank < 0 || rank >= MPISettings::size() )
                throw std::invalid_argument( "Invalid rank for exclusive section" );

            active_ = ( MPISettings::rank() == rank );
        }
```

\(^1\)Note the initialization via the copy constructor. This turns out to be the only possible way to create a temporary object of class type inside the `if`-statement.
4.2 The `pe` Exclusive Section Scope

The `ExclusiveSection` class merely provides four functions: A constructor, a destructor, a conversion operator to `bool`, and the static `isActive` function. Additionally, the class has a single static `bool` variable that acts as the activity flag of the exclusive section.

The constructor starts with a couple of tests. The first test checks whether or not the activity flag is already active. This can only happen in case an exclusive section is already active, i.e., in case a `pe_EXCLUSIVE_SECTION` is nested inside another `pe_EXCLUSIVE_SECTION`. In case a nested exclusive section is detected, a `std::runtime_error` exception is thrown to indicate the error. The second `if`-condition checks if an exclusive section is used inside a `pe_GLOBAL_SECTION` (see Section 4.3). Since it is logically wrong to execute code exclusively on one process inside a global section, a `std::logic_error` is thrown in case this error is detected. The third test controls whether the given rank is in the range of valid ranks. After these three tests, the activity flag of the exclusive section is set in case the rank of the process corresponds to the given rank.

The task of the destructor is to reset the activity flag to `false`. Since the destructor is automatically called once the scope of the surrounding `if`-statement ends, the activity flag is automatically reset after every `pe_EXCLUSIVE_SECTION`.

The conversion operator to `bool` merely returns the current status of the activity flag. This operator is called directly after the constructor of the `ExclusiveSection` class in order to evaluate the surrounding `if`-statement. In case the activity flag is `true`, i.e., in case the given rank corresponds to the rank of the process, the code inside the `if`-statement is executed. In case the operator returns `false` (i.e., in case the given rank does not match the rank of the process), the `if`-condition evaluates to `false`, leading to a skipping of the code inside the `if`-scope.
4 Smart Scopes

The `isActive` function provides the possibility to check whether a piece of code is executed inside an exclusive section. This function can for instance be used inside the `simulationStep` function to guarantee that no deadlock situation can occur:

Listing 4.13: Application of the `isActive` function inside the `simulationStep` function

```cpp
void World::simulationStep( real timestep )
{
    // ...
    // Checking if the function is called inside an exclusive section
    if( MPISettings::size() > 1 && ExclusiveSection::isActive() )
        throw std::runtime_error( "Invalid call inside exclusive section" );
    // ...
}
```

4.3 The `pe` Global Section Scope

The `pe_GLOBAL_SECTION` provides a scope to create global rigid bodies. Global rigid bodies are fixed bodies that are known on all MPI processes. This smart scope can for instance be used for the setup of very large, fixed, process-spanning bodies that might act as simulation boundaries. Listing 4.14 shows an example of the use of a `pe_GLOBAL_SECTION`:

Listing 4.14: Implementation of the `pe_GLOBAL_SECTION` macro

```cpp
int main( int argc , char ** argv )
{
    // Initialization of the MPI system
    // The MPI system must be initialized before any `pe` functionality is
    // used. It is recommended to make `MPI_Init()` the very first call of
    // the main function.
    MPI_Init( &argc , & argv );
    // Acquiring a handle to the simulation world
    WorldID world = theWorld();
    // Setup of the process connections
    connect( /* ... */ );
    // ...
    // Default parallel region
    // This code is executed by all processes of the parallel simulation.
    // Rigid bodies created in this region may only be created if their
    // center of mass lies within the domain of the local process. In this
    // example, the global position of the iron sphere 1 with a radius of
    // 1.2 is checked by the `World::contains()` function prior to its
    // creation. Therefore the sphere is created on exactly one process.
    const Vec3 gpos( 2.0 , 4.0 , -3.0 );
    if( world->contains( gpos )
    {
        createSphere( 1 , gpos , 1.2 , iron );
    }
```

42
4.3 The pe Global Section Scope

The implementation of the pe_GLOBAL_SECTION smart scope is similar to the implementation of the pe_EXCLUSIVE_SECTION:

Listing 4.15: Implementation of the pe_GLOBAL_SECTION macro

```cpp
#define pe_GLOBAL_SECTION 
  if( pe::GlobalSection globalSection = true )
```  

pe_GLOBAL_SECTION is a macro that also introduces an if-statement. However, in contrast to the pe_EXCLUSIVE_SECTION, a global section should be unconditionally activated, i.e., executed on all processes. Therefore the local object globalSection of type GLOBAL_SECTION is initialized with true instead of the rank of the current process. Therefore a GlobalSection will always evaluate as true and the code inside the if-statement is executed unconditionally.

Listing 4.16 shows the complete implementation of the GlobalSection class. Similar as in the ExclusiveSection class, the GlobalSection class provides only four functions and has a static boolean variable. The purpose of these functions and the variable is the same as in the case of the exclusive section. The constructor, after having checked the current status of the flag variable and whether the global section is used inside an exclusive section, unconditionally sets the active_flag variable to true. The destructor resets the flag variable to false. Again it is automatically called the moment the scope of the if-statement ends. The conversion operator and the static isActive function both return the current status of the flag variable.

---

2Note that this initialization is necessary in order to be able to create the local object.
4 Smart Scopes

Listing 4.16: Implementation of the pe_GLOBAL_SECTION macro

```cpp
class GlobalSection
{
public:
    GlobalSection( bool activate )
    {
        if( active_ )
            throw std::runtime_error( "Nested global sections detected" );
        if( ExclusiveSection::isActive() )
            throw std::logic_error( "Invalid global inside exclusive section" );
        active_ = activate;
    }

    inline ~GlobalSection() const
    {
        active_ = false; // Resetting the activity flag
    }

    inline operator bool() const
    {
        return active_; // Accessing the activity flag
    }

    static inline bool isActive()
    {
        return active_; // Querying the activity flag
    }

private:
    // Activity flag for the global section.
    // In case a global section is active (i.e., the currently executed code
    // is inside a global section), the flag is set to true, otherwise it is
    // false.
    static bool active_;
};
```

4.4 Conclusion

The pe smart scopes introduced in this chapter are an invaluable tool for various tasks. Their approach to create a new scope with a special effect helps to detect coding errors (as for instance deadlock situations in MPI-parallel environments), reduces the complexity of code and/or offers the ability to enforce programming concepts (as illustrated by means of the pe_GLOBAL_SECTION smart scope), or they merely make certain programming tasks more convenient. Although their implementation is comparatively simple, their application in the pe has a huge positive impact on the reliability and correctness of the framework.
5 The \textit{pe} Logging Functionality

The logging of errors, warnings, or debug information is one of the most fundamental tasks in every software framework. Therefore the requirements on the reliability, correctness, and efficiency of a logging tool are accordingly high. In this chapter I give an overview of the logging functionality of the \textit{pe} framework. This tool offers a reliable and efficient information logging for non-parallel, thread-parallel, and massively MPI-parallel environments.

5.1 Motivation

The policies for the reporting, propagation, and handling of errors within a software framework are fundamental design decisions in every software development process \cite{79, 51}. One aspect of these policies is the logging of errors, warning messages, and other information into log files. The requirements on such a logging tool are high: It should be perfectly reliable, i.e., logged information must be contained in the according log files; it should be perfectly stable, i.e., the tool must not have reliability problems on different architectures and under different compilers and should not provoke errors or exceptional situations by itself; additionally, it should be as efficient as possible while guaranteeing the proper logging functionality.

Depending on the software framework, additional requirements on a logging tool might also be the suitability for threaded as well as massively parallel environments. Whereas threaded environments are nowadays part of every programmers life (especially since the multi-core era), massively parallel environments are less frequently encountered. However, especially in fields where massively parallel programs play an important role, as for instance in the field of numerical simulation in general, the proper logging of errors and/or warnings is of crucial importance.

In this chapter I will introduce the logging functionality of the \textit{pe}. Due to the complexity of the \textit{pe} framework, which is in the range of several hundred thousand lines of C++ code,
and due to the complexity of the systems this engine is used on (as for instance the Jugene supercomputer at the Jülich Supercomputing Centre [59] with a total of 294,912 processor cores and a theoretical peak performance of one petaflops), the reliability of the logging functionality of the pe is essential for its operation.

The pe logging tool is suitable for non-parallel, thread-parallel, as well as massively MPI-parallel simulations [68]. The main focus of the development of this tool is on reliability, the user-interface, and its efficiency. For instance, no runtime and no memory overhead should occur in case the logging functionality is turned off and the task of logging should be performed as efficiently as possible. Additionally, the pe logging module is implemented as an independent tool that can easily be migrated to other C++ software frameworks.

In the following sections, I will illustrate all required C++ classes in detail and will give an insight into all design decisions. Section 5.2 gives a first impression on how the logging is used and how it works. Section 5.3 will focus on the Logger class, which is the core of the logging module, before Section 5.4 will show how the Logger class is used in the context of the LogSection class. Section 5.5 will show how the logging tool is completed by the implementation of the log section macros. It will focus on their implementation and elaborate on the design decisions combining an easy to use interface with reliability and thread-safety. In Section 5.6 I will discuss the limitations of the logging tool in the context of global/static objects. Section 5.7 concludes the chapter.

5.2 A First Impression of the pe Logging Tool

In this section, I provide a first impression of the usage of the pe logging tool. Logging in the pe is performed by the use of so called log sections\(^1\). The following listing gives a first example of two of these log sections in a simple, non-parallel environment. The first of the two log sections is logging a (severe) error message, the second logs a warning message:

```cpp
int main() {
    pe_LOG_ERROR_SECTION( log ) {
        log << " (Severe) error message, which is only printed within an\n" << " active pe_LOG_ERROR_SECTION\n";
    }
    pe_LOG_WARNING_SECTION( log ) {
        log << " Warning message, which is only printed within an active\n" << " pe_LOG_WARNING_SECTION\n";
    }
}
```

The first log section, starting in line 3 in Listing 5.1, is logging an error message to the log file. This log section is only executed if the logging level of the pe is set to error or higher, otherwise the compiler removes this section entirely (see Section 5.5). The parameter log specifies the name of the handle within the log section that can be used as a stream to log

\(^1\) These log sections are also implemented in terms of smart scopes; see Chapter 4.
any kind of streamable information. The second log section, starting in line 8, can be used to log warning messages of any kind. It is only executed if the logging level is at least set to `warning`. Otherwise it is completely removed and creates no runtime overhead.

Each logging section is associated with a certain logging level, which corresponds to the importance of the message. The `pe` logging tool offers several logging levels for different kinds of errors, warnings, and information. These logging levels are organized hierarchically:

- **inactive**: Completely deactivates the logging functionality, i.e., no log file(s) will be written and no runtime overhead occurs. Since this setting can immensely complicate error correction, it is not recommended to use this setting.

- **error**: Only (severe) errors are written to the log file(s).

- **warning**: Extends the `error` setting by warning messages (the default setting).

- **info**: Extends the `warning` setting by additional informative messages.

- **progress**: Extends the `info` setting by progress information.

- **debug**: Extends the `progress` setting by debug information.

- **detail**: Extends the `debug` setting by very fine grained detailed information.

The global logging level of the physics engine is specified via the `loglevel` variable in one of the `pe` configuration files (see Listing 5.2). Depending on the selected logging level information about (severe) errors, warnings, important information, progress reports, debug information, and detailed output is written to the log file.

**Listing 5.2**: Global configuration of the logging level

```cpp
const LogLevel loglevel = warning;
```

Listing 5.3 shows the resulting log file containing the two log messages (assuming the logging level is set at least to warning level):

**Listing 5.3**: Log file resulting from the non-parallel example

```plaintext
3 [ERROR ] (Severe) error message, which is only printed within an
4 active pe_LOG_ERROR_SECTION
5
6 [WARNING ] Warning message, which is only printed within an active
7 pe_LOG_WARNING_SECTION
8
```

Line 1 shows the header line of the resulting log file. It contains a time stamp to indicate at which time the log file was generated. At the end of the file the bottom line can be found, which indicates when the log file was finalized. In between, the two log

---

2In this example, the time stamps of the header and bottom line only match due to the extremely short execution time of the main function.
messages from Listing 5.1 can be seen. Each individual message starts with an according tag, which indicates the logging level of the message, followed by the contents of the message.

In non-parallel environments, a single log file named \texttt{pe.log} is created, containing all the information of the single process. In MPI-parallel environments, each process creates his own log file named \texttt{peX.log}, where \( X \) is replaced by his according process rank in the \texttt{MPI_COMM_WORLD} communicator (see [49] for more information about MPI).

It is also possible to nest several log sections, for instance to extend the logging output by additional information in case of a higher logging level. However, obviously this is only useful in case the inner log section has a higher log level than the outer log section (as for instance an \texttt{info} section nested inside a \texttt{warning} section). Otherwise the inner section would not be executed if the outer log section would not be executed, even if the logging level would be set appropriately. Listing 5.4 shows an example of a \texttt{pe_LOG_DEBUG_SECTION} nested inside a \texttt{pe_WARNING_SECTION}:

\begin{verbatim}
Listing 5.4: Example for two nested log sections

int main ()
{
    pe_LOG_WARNING_SECTION ( warning )
    {
        warning << " Important warning message about a failed computation\n";
    }
    pe_LOG_DEBUG_SECTION ( info )
    {
        info << " Additional debug information about the failed computation\n";
    }
    warning << " Note about how the error was resolved!\n";
}
\end{verbatim}

The resulting log file is shown in Listing 5.5:

\begin{verbatim}
Listing 5.5: Log file resulting from the nested log sections example


[DEBUG     ] Additional debug information about the failed computation

[WARNING   ] Important warning message about a failed computation
Note about how the error was resolved!

\end{verbatim}

Note that the debug information appears before both warning messages, which are combined into a single message. This behavior results from the \texttt{pe} idiom that log operations are considered as atomic operations. The basic reason for this behavior is the ability to guarantee thread-safety. Let us consider for instance the logging example in Listing 5.6, which uses the \texttt{pe threadpool} functionality to create log message from several threads:
5.2 A First Impression of the pe Logging Tool

Listing 5.6: Application of the pe logging tool in a threaded environment

```cpp
void createError()
{
    pe_LOG_ERROR_SECTION( log ) {
        log << " Message" << " about" << " (severe)" << " error!\n";
    }
}

void createWarning()
{
    pe_LOG_WARNING_SECTION( log ) {
        log << " Important" << " warning" << " message!\n";
    }
}

int main()
{
    using namespace pe::threadpool;
    // Creating a threadpool with 4 threads
    ThreadPool threadpool( 4 );
    // Creating a couple of tasks
    for( int i=0; i<100; ++i ) {
        threadpool.schedule( createError );
        threadpool.schedule( createWarning );
    }
    // Waiting for all tasks to finish
    threadpool.wait();
}
```

Listing 5.7 shows an excerpt of the resulting log file:

```
[WARNING] Important warning message!
[WARNING] Important warning message!
[ERROR] Message about (severe) error!
[ERROR] Message about (severe) error!
[WARNING] Important warning message!
[ERROR] Message about (severe) error!
[WARNING] Important warning message!
```

Although the log messages consist of several parts (in this example illustrated by artificially using multiple output operators), the messages appear as a whole in the log file. Additionally, the messages are not intermingled despite the threaded access to the log file.
In order to overcome the disadvantage of the example shown in Listings 5.4 and 5.5, log messages can be manually committed:

Listing 5.8: Manually committing into the log file

```c
int main ()
{
    pe_LOG_WARNING_SECTION ( warning )
    {
        warning << " Important warning message about a failed computation\n";
        // Manually committing the warning message
        warning.commit();
    }
    pe_LOG_DEBUG_SECTION ( info )
    {
        info << " Additional debug information about the failed computation\n";
    }
    warning << " Note about how the error was resolved!\n";
}
```

The explicit call to the `commit` function has basically the same effect as creating a new log warning section. As illustrated in Listing 5.9, the `commit` function has the effect of immediately writing the according message to the file. Everything logged after the invocation of the `commit` function is treated as a new section.

Listing 5.9: Resulting log file for manually committing

```c
[ WARNING ] Important warning message about a failed computation
[ DEBUG ] Additional debug information about the failed computation
[ WARNING ] Note about how the error was resolved!
```

5.3 The Logger Class

In this section I will start to deal with the internal implementation of the `pe` logging functionality. At the core of this tool is the `Logger` class. It is responsible for managing the file stream for the log file(s) and for committing logging messages immediately to the according file(s) (one file for both non-parallel and thread-parallel simulations, several files for MPI-parallel simulations).

Listing 5.10 shows the class definition of the `Logger` class. The class possesses a total of four member functions, where only a single member function is publicly accessible. The first of the four member functions is the constructor. This function tries to open the according log file.
and writes the header line into the file. In case the file cannot be opened, a `std::runtime_error` exception is thrown, indicating that logging will fail. The second function is the destructor, which first writes the bottom line into the log file and afterwards closes the file.

Listing 5.10: Definition of the `Logger` class

```cpp
class Logger : private boost::noncopyable
{
  private:
    Logger();
    ~Logger();

  public:
    template < typename Type > void log( const Type& message );

  private:
    static Logger& instance();

    boost::mutex mutex_; // Synchronization mutex for thread-parallel logging.
    std::ofstream log_; // The log file.

    friend class LogSection;
};
```

The only publicly available function is the `log` function. This function commits a given object of arbitrary type to the log file. Listing 5.11 shows that this operation is performed via the output operator. Therefore it is interesting to note that all types that support the output operator, i.e., which are streamable, can be used as message type. In order to guarantee thread-safety, the access to the file is only provided for one thread at a time. This is done by creating a scoped lock at the beginning of the function before the message is written to the file. Note that this lock is automatically released at the end of the `log` function according to the RAII idiom [97, 96, 53, 34].

Listing 5.11: Implementation of the `log` function

```cpp
template < typename Type > // Type of the log message
void Logger::log( const Type& message )
{
    boost::mutex::scoped_lock lock( mutex_ );
    log_ << message;
    log_.flush();
}
```

The last function of the `Logger` class is the `instance` function (see Listing 5.12). This function implements a classical Meyer's singleton (see [96]): Inside the function, a function local object is created and the function returns a reference to this object. This approach is based on the C++ guarantee that local static objects are initialized the moment their definition is first encountered during an invocation of the function. Note that for the `Logger` class, this local static object is the only `Logger` object that can be created, since the constructor and destructor of the `Logger` class are both `private`. Therefore, via this design pattern in combination with a `private` constructor and destructor it is ensured that during the entire execution time of the program there will only be a single `Logger` object, which handles the only log file for every process. In addition, we achieve that as long as this function is not
called, no Logger object exists, i.e., it does not consume any memory and especially it does not open any file stream. Therefore, as long as this function is not called, no log file will be created.

Listing 5.12: Implementation of the instance function

```cpp
Logger& Logger::instance()
{
    static Logger logger;
    return logger;
}
```

As already mentioned, in MPI-parallel simulations each process creates its own log file called peX.log, where x is replaced by the MPI rank of the process. There are several reasons for this. The most important one is that due to the separate files the MPI processes can handle logging completely independent of each other. Therefore no communication is required and no process is depending on another process. This is especially important if a process detects a local error and has to log the according information. In case this process would depend on at least one other process, the reliability of the logging would be endangered. Another reason is efficiency: Since no communication and/or synchronization is required, processes can write the log files as efficiently as possible. A third reason for creating individual log files is simplicity: Since the essential requirements on the logging tool is reliability, a simpler solution is more suited due to a reduced error-proneness.

5.4 The LogSection Class

The next element of the pe logging functionality is the LogSection class. This class acts as a wrapper around the functionality of the Logger class and provides the atomicity of the logging operation and the formatting of the log output. Listing 5.13 shows the complete class definition of the LogSection class:

Listing 5.13: Definition of the LogSection class

```cpp
class LogSection
{
public:
    LogSection( LogLevel level );
    inline LogSection( const LogSection& ls );
    LogSection();
    inline operator bool() const;
    template < typename Type > inline void log( const Type& message );
    void commit();

private:
    LogLevel level_; //!< The logging level of the log section.
    std::stringstream message_; //!< Intermediate buffer for log messages.
    LogSection& operator=( const LogSection& );
    void* operator new( std::size_t ) throw( std::bad_alloc );
    void* operator new[]( std::size_t ) throw( std::bad_alloc );
    void* operator new( std::size_t, const std::nothrow_t& ) throw();
    void* operator new[]( std::size_t, const std::nothrow_t& ) throw();
};
```
The LogSection provides (next to the copy constructor) only a single constructor that takes a LogLevel as a single argument. Therefore a LogSection object is always corresponding to a specific LogLevel (see Section 5.2). Yet the constructor’s sole responsibility is to properly initialize the two member variables level_ and message_. Listing 5.14 shows the implementation of LogLevel:

```
enum LogLevel
{
    inactive = 0, // Log level for no logging.
    error = 1, // Log level for (sever) errors.
    warning = 2, // Log level for warnings.
    info = 3, // Log level for high-level information.
    progress = 4, // Log level for progress information.
    debug = 5, // Log level for debug information.
    detail = 6 // Log level for detail information.
};
```

Especially interesting is the role of the destructor of the LogSection class:

```
Listing 5.15: The destructor of the LogSection class

LogSection::~LogSection()
{
    commit();
}
```

The destructor of the LogSection class is responsible for committing the log message as a (logically) atomic operation to the log file. Therefore at the moment the lifetime of a LogSection class ends, the log message (buffered in the message_ member variable) is written to the log file. Although this approach (in combination with the thread-safe implementation of the Logger class) guarantees thread-safety, it suffers from the disadvantage that any error (as for instance segmentation faults) and exceptions occurring within a log section might result in the loss of information. However, the combination of thread-safety and fault-tolerance by instantaneous commits is extremely complicated. Due to that and since thread-safety is the more important feature, the pt logging tool does not provide information-safety in presence of uncaught exceptions.

The functionality to commit the log message is implemented in the commit function (see Listing 5.16). The first part of the commit function handles the formatting of the output message. After that, the LogSection acquires a handle to the logger and writes the entire message to the log file in a single step. Note that the first call to the instance function triggers the creation of the single Logger object and with that creates the log file. The last two lines of the function completely reset the message buffer.
5 The pe Logging Functionality

Listing 5.16: Implementation of the commit function

```cpp
void LogSection::commit()
{
    std::ostringstream oss; // Buffer for the formatted log message
    // Formatting of the log message
    // ...
    // Logging the formatted log message
    Logger & logger( Logger::instance() );
    logger << oss.str();
    // Resetting the message buffer
    message_.str( "" );
    message_.clear();
}
```

Similar to the Logger class, the LogSection class also contains a log function. However, since the LogSection class does not have to manage shared resources, the function is much easier to implement:

Listing 5.17: Implementation of the log function

```cpp
template < typename Type > // Type of the log message
inline void LogSection::log( const Type& message )
{
    message_ << message;
}
```

The function appends the message of the given data type Type to the message buffer message_. Again, as in the case of the Logger::log function, this class requires the given data type to be streamable, i.e., to offer a suitable output operator. For convenience, the LogSection class also provides an output operator in order to be able to stream messages to a LogSection.

Listing 5.18: Implementation of the output operator

```cpp
template < typename Type > // Type of the log message
inline LogSection& operator<<( LogSection& logsection, const Type& message )
{
    logsection.log( message );
    return logsection;
}
```

For completeness, Listings 5.19 shows the implementation of the two remaining functions in the public interface of the LogSection class. The bool conversion operator provides a conversion to true, and the explicitly defined copy constructor enables copy operations despite the non-copyable member stream variable. The significance of these two functions will be demonstrated in the next section. It will also give an explanation for the reason behind the private declarations of the copy assignment operator and all possible operator new and operator delete.
5.5 The Logging Sections

The final piece of the logging functionality of the \texttt{pe} are the log sections. The \texttt{pe} provides an accordingly named log section for every logging level:

- \texttt{pe\_LOG\_ERROR\_SECTION}: Log section for (serious) errors.
- \texttt{pe\_LOG\_WARNING\_SECTION}: Log section for warning messages.
- \texttt{pe\_LOG\_INFO\_SECTION}: Log section for important high-level information.
- \texttt{pe\_LOG\_PROGRESS\_SECTION}: Log section for progress information.
- \texttt{pe\_LOG\_DEBUG\_SECTION}: Log section for debug information.
- \texttt{pe\_LOG\_DETAIL\_SECTION}: Log section for detailed information.

All log sections are implemented similarly and only differ in the use of different logging levels. Therefore, we use the \texttt{pe\_LOG\_ERROR\_SECTION} as an illustrative example:

\textbf{Listing 5.20: Implementation of the \texttt{pe\_LOG\_ERROR\_SECTION} macro}

\begin{verbatim}
# define pe_LOG_ERROR_SECTION( NAME ) \ 
  if( pe::logging::loglevel >= pe::logging::error ) \ 
    if( pe::logging::LogSection NAME = pe::logging::error )
\end{verbatim}

As already indicated by the naming convention using capital letters, the \texttt{pe\_LOG\_ERROR\_SECTION} is implemented as a macro (pseudo) function. The macro is replaced by two nested \texttt{if}-statements. The first of them is responsible whether the log section is executed or not. In case of the error section, it is checked whether the global logging level is greater-or-equal than \texttt{error}. If so, the log section is executed, otherwise it is not (which in this particular example would only happen if the global logging level is set to \texttt{inactive}).

Also note the further effects of this first \texttt{if}-statement. Since the global logging level is a constant compile time expression (see Listing 5.1), the first \texttt{if}-statement enables the compiler to completely optimize the entire log section away. This is possible since the condition can be evaluated at compile time and therefore results in either \texttt{if( true )} or \texttt{if( false )}. If the expression evaluates to \texttt{false} the entire content of the \texttt{if}-statement will never be executed and can therefore be optimized away. Therefore a disabled log section will not create any runtime overhead since it is no longer present in the final object file or executable.

It is also worth to mention the advantage of this approach in comparison to the much too often favored preprocessor alternative:
5 The pe Logging Functionality

Listing 5.21: Illustration of the preprocessor alternative

```c
#define LOGLEVEL 1

#if LOGLEVEL >= 2
    // ... Perform logging of a warning message ...
#endif
```

The major advantage of the pe approach is that the entire functionality is made of elements of the programming language itself. The macros are merely used to wrap the functionality in a convenient way. In the preprocessor approach, the preprocessor completely removes the log section in case it is disabled. In comparison, in the pe approach the compiler is able to see the log section before it is removed in an optimization step. Therefore, the compiler can check the code inside the log section for (type-)correctness and consistency each time the compilation unit is compiled. Hence, this effectively helps to remind a programmer to additionally maintain the log sections and prevents the code from becoming out-of-date.

The purpose of the second if-statement is to create a new, exclusive scope for the log section. In this scope, it defines an object of type `LogSection`, which is called according to the specified `NAME` and initialized with the according level. The life-time of this object is bound to the scope of the if-statement: At the end of the if-statement the `LogSection` object is destroyed, which in turn triggers the destructor of the `LogSection` class, which commits the internal message buffer to the log file in a (logically) atomic operation. The size of this scope can be chosen with a pair of brackets following the log section macro (and hence the second if-statement):

```
Listing 5.22: Scope of the pe_LOG_ERROR_SECTION
```

The creation of the `LogSection` object should be examined a little more closely. The initialization shown in Listing 5.20 was chosen as the only way to create a named object of type `LogSection` inside the if-condition. However, although this construct is ultimately resulting in a call to the constructor of the `LogSection` class, it requires the class to allow implicit conversions from `LogLevel` to `LogSection` and to have a defined copy constructor [52], since logically this is equivalent to the following construct:

```
Listing 5.23: Copy initialization of a LogSection object
```

This is the basic reason the copy constructor of the `LogSection` class is defined. However, special care has to be taken since C++ streams cannot be copied. Therefore, the copy constructor must be defined such that the stream is not involved in the copy operation.
5.6 Limits of the Logging Tool

In contrast to the copy constructor, the copy assignment operator is not required. Therefore, this function is declared in the private section of the LogSection class along with all possible operator new and operator delete. The reason for this is the attempt to protect the class from accidental misuse (i.e., from Murphy errors; see [53]). Since the class is exclusively designed for the use within a log section, all further use is hindered and discouraged in order to prevent unforeseen effects. However, although it is well intended, obviously this design does not offer a full protection against Machiavelli errors [53].

The definitions of the log section macros complete the pe logging functionality. Due to them, users as well as developers using the pe can very easily log important messages to the log file(s) without the need to know about the underlying effects (i.e., the Logger and the LogSection classes). Therefore, their application tremendously improves the user interface of the tool since accidental misuse is elegantly prevented. Additionally, their use proves to be a nice solution to combine efficiency and the necessary mechanisms for reliability and thread-safety.

5.6 Limits of the Logging Tool

Due to the elegant combination of the Logger class, the LogSection class, and the log section macros, the pe logging tool is suited for both thread-parallel as well as massively MPI-parallel environments. However, it is important to note that the pe logging tool has a severe limitation in its usefulness: It is not suited for logging information during the construction and destruction of global and static objects. The reason is that in this particular design the life-time of the single Logger object is not guaranteed to persist until the last global object (that might use it to log information) is destroyed.

Consider for instance the following code example in Listing 5.24 that illustrates the problem by means of a simplified Logger class. In this code example, three classes are defined: The Logger class, the Global1 class, and the Global2 class. Whereas the Logger and the according theLogger function represent a simplified Logger as introduced in Section 5.3, Global1 and Global2 both represent classes that are used in a global/static context. The difference between these two classes is that Global1 needs to log information during destruction, whereas Global2 needs to log information during its construction.

Listing 5.24: Order of construction and destruction of global/static objects

```cpp
// Definition of a simplified Logger class
struct Logger
{
  Logger () {
    std::cout << " Inside the Logger constructor\n"
  }

  ~Logger () {
    std::cout << " Inside the Logger destructor\n"
  }
};
```

For more information see the detailed discussion of several singleton patterns in [14].
Listing 5.25 shows the output resulting from the code example compiled with GCC 4.4.1. Both global1 and global2 are constructed prior to the execution of the main function. However, during the construction of the Global2 object, the Logger object gets constructed. Therefore the order of creation of these three objects is Global1, Logger, and finally Global2. Since C++ guarantees a destruction of global objects in the reverse order of the construction [81],
the first object to be destroyed is the Global2 object, before the Logger object, and before the Global1 object. However, this destruction order might (and probably will) result in disaster, since the Global1 object requires the logger during its destruction process. The best we can hope for in this situation is to just lose the probably valuable information that should be logged during the destruction of the Global1 object, but since using an already destroyed object results in undefined behavior this might as well terminate the program in an “ungraceful” way (and result in the loss of more resources).

Listing 5.25: Resulting output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inside the Global1 constructor</td>
</tr>
<tr>
<td>2</td>
<td>Inside the Logger constructor</td>
</tr>
<tr>
<td>3</td>
<td>Inside the Global2 constructor</td>
</tr>
<tr>
<td>4</td>
<td>Logging information!</td>
</tr>
<tr>
<td>5</td>
<td>Starting the main function</td>
</tr>
<tr>
<td>6</td>
<td>Ending the main function</td>
</tr>
<tr>
<td>7</td>
<td>Inside the Global2 destructor!</td>
</tr>
<tr>
<td>8</td>
<td>Inside the Logger destructor!</td>
</tr>
<tr>
<td>9</td>
<td>Inside the Global1 destructor!</td>
</tr>
<tr>
<td>10</td>
<td>Logging information!</td>
</tr>
</tbody>
</table>

Please note that the problem in this example can be removed by switching the construction of the two global objects global1 and global2, but this will not be possible in a real world framework with multiple compilation units, where the order of construction/destruction of global/static objects cannot be controlled. Additionally, the situation is worsened by the fact that the occurrence of the error is depending on the setting of the global logging level. For instance, in case the logging level is set too low, the according log sections are deactivated. By the time the logging level is increased to the level used in the constructors/destructors of the global/static objects, the program might crash mysteriously.

The problem can be solved by using a different singleton pattern (see for instance [14] for a discussion) or for example by using boost::shared_ptr [75, 16, 22]:

Listing 5.26: Implementation of the instance function based on shared_ptrs

```cpp
boost::shared_ptr<Logger> Logger::instance()
{
    static boost::shared_ptr<Logger> logger( new Logger );
    return logger;
}
```

By storing a copy of a shared_ptr the destruction process of the Logger object can be delayed until it is not required anymore. However, the basic reason these techniques are not used in the PE is the requirement of the suitability for massively MPI-parallel environments. In MPI parallel simulations, the name of the log files contains the rank of the according process in order to be able to relate a log file to a certain MPI process. Hence, in order to be able to determine the name of the log file on each MPI process the rank of the process must be known. However, the rank of a MPI process can only be acquired after the MPI_Init function has been executed.
5 The pe Logging Functionality

Listing 5.27: Initialization of the MPI communication system via the MPI_Init function

```c
int main( int argc , char** argv )
{
    MPI_Init( &argc , & argv );
    // ...
}
```

Since the MPI_Init function requires the parameters of the main function, it can only be called after the execution has been transferred to the main function. Thus, in MPI-parallel simulations, the name of the log file(s) can only be determined after the MPI_Init function has been executed and therefore logging can only be performed after the main function has started and not before, for example during the construction of global/static objects.

Logging during the destruction of global/static object could be enabled by a scheme as illustrated in Listing 5.26. However, for symmetry reasons and in order not to break the encapsulation of the design by providing access to the Logger class directly, logging during the destruction process of global/static objects is also not allowed.

5.7 Conclusion

In this chapter I have introduced the logging functionality of the pe framework. I have illustrated in detail the implementation details and design decisions to create a reliable, safe, and efficient logging tool for non-parallel, thread-parallel, and massively MPI-parallel environments. Due to the elegant solution of log sections, a nice and easy to use interface can be combined with the crucial requirements of reliability and thread-safety.
6 The Sandwich Pattern

In this chapter I introduce the C++ design pattern called “Sandwich Pattern”. This design pattern was developed in context of the crucial efficiency, modularity, and extensibility requirements of the pe and is representative of the fundamental software design of the pe framework. The Sandwich Pattern is a variation of plain static polymorphism to replace at compile time certain static as well as dynamic characteristics of a data type while keeping certain other characteristics unchanged. This design pattern is applied in the pe to statically adapt rigid bodies and contacts between these bodies to specific algorithms, which are selected at compile time. Because of the use of this design pattern, the pe offers a remarkable flexibility to integrate various algorithms, while still providing maximum efficiency without runtime overhead due to abstraction.

6.1 Motivation

In certain situations during code development the necessity arises to customize a type depending on a certain configuration. It may be necessary to completely replace certain member data and functions of the type, where other parts should remain a permanent component of the type. It may even be necessary to replace a set of virtual functions by a different set of virtual functions to accommodate for a different configuration.

One of these situations was encountered for the implementation of the \texttt{RigidBody} class that represents the base class for all kinds of rigid, completely undeformable bodies within the pe framework. Depending on a set of statically selected algorithms (for the coarse collision detection, the fine collision detection, the collision response, e.g.) working with the \texttt{RigidBody} type, certain member data and functions of the \texttt{RigidBody} class should be customizable, whereas other components should not be interchangeable. Imagine that all of these algorithms make different demands on the rigid body data structure. Each algorithm requires different, individual data members within each rigid body in order to be able to optimally
perform its task. Additionally, each algorithm requires a certain set of special functions to perform specific tasks or just to access the individual data members.

Listings 6.1 and 6.2 give an impression of the problem for the requirements posed by the LCP-based collision solvers and the PFFD collision solver (see Chapter 10). Whereas the LCP-based solvers require a single virtual function for the time integration step (in this case called move; see Listing 6.1), the PFFD solver requires two virtual functions for its first and second half-step (see Listing 6.2):

**Listing 6.1:** Requirements of LCP-based collision solvers on the RigidBody base class

```cpp
class RigidBody
{
public:
    // LCP-specific function members
    virtual void move(real dt) = 0;

private:
    // LCP-specific data members
    // ...;
};

class Sphere : public RigidBody
{
public:
    virtual void move(real dt) { /* Sphere-specific implementation */ }
    // ...;
};
```

**Listing 6.2:** Requirements of the PFFD collision solver on the RigidBody base class

```cpp
class RigidBody
{
public:
    // PFFD-specific function members
    virtual void firstPositionHalfStep(real dt) = 0;
    virtual void secondPositionHalfStep(real dt) = 0;

private:
    // PFFD-specific data members
    // ...;
};

class Sphere : public RigidBody
{
public:
    virtual void firstPositionHalfStep(real dt) {
        /* Sphere-specific implementation */
    }
    virtual void secondPositionHalfStep(real dt) {
        /* Sphere-specific implementation */
    }
    // ...;
};
```
6.2 The Sandwich Pattern

Obviously, different algorithms may have completely different requirements on the \texttt{RigidBody} base class and according derived classes (in this example the \texttt{Sphere} class). Therefore the question arises how to implement the basic rigid body data structures to accommodate all requirements for the current algorithms and all future algorithms with yet unknown requirements that will have to be integrated into the \texttt{pe} framework. Due to the real-time demands of certain simulation scenarios, these data structures additionally have to be suited for high performance computing, i.e., the data structures themselves should not be too costly in terms of performance.

For the remainder of this chapter, I will use this \texttt{RigidBody} class to illustrate the development of a very flexible design pattern called the “Sandwich Pattern”. This C++ design pattern in its basic form allows to statically change certain characteristics (member data and functions) of a data type while keeping certain others unchanged by a clever combination of static and dynamic polymorphism. In Section 6.2 I will first introduce the general idea of this design pattern by means of the \texttt{RigidBody} class. Section 6.3 will introduce the hierarchical application of this pattern that in particular allows to statically change the dynamic characteristics of the data type, i.e., to customize the set of available virtual functions. To demonstrate the design pattern, Section 6.4 will show an excerpt of the implementation of the Sandwich Pattern as realized in the \texttt{pe} rigid (multi-)body physics engine. Section 6.5 will conclude the chapter.

6.2 The Sandwich Pattern

A first naive approach to solve the problem would be to add all according necessary data and (virtual) function members to the \texttt{RigidBody} base class:

\begin{lstlisting}[language=C++]
class RigidBody
{
private:
  // Data members for algorithm 1
  // Data members for algorithm 2
  // ...
  // Private (virtual) functions for algorithm 1
  // Private (virtual) functions for algorithm 2
  // ...
public:
  // Public (virtual) functions for algorithm 1
  // Public (virtual) functions for algorithm 2
};
\end{lstlisting}

This data structure would certainly only result in negligible performance penalties. However, this approach obviously leads to a bloated \texttt{RigidBody} base class, both in terms of available functionality as well as in terms of the memory footprint of a \texttt{RigidBody} object. All data members of every algorithm are constantly available, although only one algorithm of each category is active at a time. In fact, in our case the configuration consisting of a certain
set of algorithms is selected at compile time since switching between the algorithms only very rarely makes sense (for instance because either real-time aspects or physical accuracy aspects are predominant).

One approach to solve customization problems is the use of static polymorphism via the C++ template mechanism:

Listing 6.4: Template-based implementation of the RigidBody class

```cpp
template < typename C > // Type of the configuration
class RigidBody
{
private:
    // Configuration-specific data members
    // ...

    // Configuration-specific private (virtual) functions
    // ...

public:
    // Configuration-specific public (virtual) functions
    // ...
};
```

With this approach it is possible to specifically add the required data and (virtual) function members to the RigidBody class depending on the type of the configuration (that represents the set of selected algorithms). Additionally, there is no performance penalty involved in this design. However, the downside of this approach is obvious: Since every specialization of the RigidBody class template, which adapts the type to the requirement of a specific configuration/algorithm, additionally needs to define the set of permanent members that should be a constant part of all rigid bodies, the implementation overhead of each specialization might be substantial. Depending on the complexity of the base class, this might very well involve several thousand lines of code that have to be rewritten for every single specialization, i.e., for every new algorithm. However, a simple extension based on dynamic polymorphism improves the situation considerably:

Listing 6.5: Refactoring of the template-based RigidBody class

```cpp
// Implementation of the RigidBodyBase class containing all common data
// and function member
class RigidBodyBase
{
private:
    // Common data members for all rigid bodies
    // ...

    // Common private (virtual) functions for all rigid bodies
    // ...

public:
    // Common public (virtual) functions for all rigid bodies
    // ...
};
```
Due to this refactoring via the introduction of the RigidBodyBase class, which now contains all common functionality for all rigid bodies, the RigidBody class template only has to implement the configuration-specific data and (virtual) function members that are required by the selected algorithms. Additionally, even due to the addition of the RigidBodyBase class, no additional performance bottleneck is introduced in the design. Therefore, apart from some minor criticisms about the design, this approach is already well suited for the requirements of the RigidBody data structure. However, there is still room for additional improvements. Certain common aspects of the data type to be customized might not fit into the new base class, but should remain in the top level class. Therefore, in addition to the RigidBodyBase class, we introduce another class that will represent the top level class of this hierarchy and we will also slightly adjust the names of the classes:

**Listing 6.6: Application of the Sandwich Pattern to the RigidBody base class**

```cpp
// Type of the compile-time selected configuration
typedef /* ... */ Config;

// Implementation of the RigidBodyBase class containing all common data
// and function member
class RigidBodyBase {
private:
  // Common data members for all rigid bodies
  // ...
  // Common private (virtual) functions for all rigid bodies
  // ...
public:
  // Common public (virtual) functions for all rigid bodies
  // ...
};
```

1. We will see the implication and possibilities of the declaration of configuration-specific virtual functions in Section 6.3.
2. See [96] for a discussion about the usage of public virtual functions.
In this class hierarchy the Sandwich Pattern is already fully realized. The RigidBodyBase class contains all data members and (virtual) functions common to all rigid bodies. On top of that the RigidBodyTrait class provides all configuration-specific data and functionality of the rigid bodies. The top level RigidBody class contains all common top level data and functionality that (either logically or from an implementational point of view) cannot be moved to the RigidBodyBase class. Note that the RigidBody class publicly derives from a specific instantiation of the RigidBodyTrait class template according to a configuration Config that is specified at compile time.

![Diagram of the sandwich pattern applied to the RigidBody base class.](image)

**Figure 6.1:** Application of the sandwich pattern to the RigidBody base class.

With Figure 6.1 it becomes obvious that the Sandwich Pattern is a variation of plain static polymorphism. Instead of defining a single class template that would have to carry
the burden to redefine common functionality with every specialization, two additional classes are defined, one as the base class of the center customization class, and one as the top level class. All three classes build a logical compound and still represent the original RigidBody class, but offer a much more convenient way to adapt the class to specific settings. Especially, specializing for a particular algorithm requires much less programming effort than the single template class.

![Image of the Sandwich Pattern](image)

**Figure 6.2:** Multiple customization layers.

Note that the sandwich is not limited to a single templated layer. Just as a real sandwich can be made of several layers of food, a class designed with the Sandwich Pattern can have multiple customization layers. Figure 6.2 gives an impression of the RigidBody class implemented with three templated layers named RigidBodyTrait1, RigidBodyTrait2, and RigidBodyTrait3.

Although the pattern fulfills all requirements on the RigidBody data structure, several potential disadvantages of the Sandwich Pattern should be discussed. One disadvantage in terms of performance might be the additional overhead due to two more constructor and destructor calls during the construction and destruction of a type using this design pattern. Whether or not this results in a real performance bottleneck strongly depends on the application the type is involved in. In case such types are frequently created and destroyed and in case the additional function calls are expensive, the performance penalty might be measurable. However, if either construction/destruction is a rare event (as it is for example in rigid body simulations) or in case the constructors and (non-virtual) destructors can be inlined these additional function calls will not be a performance issue. Another disadvantage is the increased depth of the class hierarchy, which is usually considered bad in class design (for a thorough discussion on this topic see [34]). In case of the Sandwich Pattern, however, the classes build a logical, inseparable unit, i.e., they represent a single class and are only refactored into several classes for implementation reasons. Therefore, this design arguably does not increase the depth of the class hierarchy.

Another disadvantage from a class design point of view is the potential misuse of the "internal" classes (in this example the RigidBodyBase and RigidBodyTrait classes). It is for instance possible to derive other classes from one of these two internal classes since there is no general protection against inheritance. However, in this case one has to distinguish be-
tween “protecting against Murphy, versus protecting against Machiavelli” [53]: Whereas the intentional misuse cannot be prevented (Machiavelli), the accidental misuse via inheritance is rather improbable (Murphy).

The most important drawback of this design pattern is its implications for the rest of the implementation that uses the customized types. Every functionality that uses an algorithm-specific characteristic of the data type has to be transformed into a template, too. This is in particular true for all algorithms using the RigidBody class, since the customized trait classes provide the functionality for these algorithms. Since certain characteristics (as for instance functions) might be completely replaced, using these characteristics even if a different instance of a trait class is selected will result in compilation errors. Let us assume that there is a class called Algorithm, which expects a RigidBody to have a function called doSomething. Then the following example illustrates the problem:

```
Listing 6.7: Non-template implementation of Algorithm

class Algorithm
{
public:
  // ...
  void runAlgorithm( RigidBody* body )
  {
    body->doSomething(); // <- This might result in an error even if
    // Algorithm is not selected
    // ...
  }
  // ...
};
```

The code in Listing 6.7 will only compile, if according trait classes are selected that provide the rigid bodies with the configuration-dependent function doSomething. In case a different configuration is selected, the compiler will flag this as an error, since it is possible to determine that a rigid body does not have the required function. Therefore, the use of these custom characteristics has to be hidden from the compiler by use of template argument dependent types. The simplest approach in this example would be to pass the type of rigid bodies as template arguments:

```
Listing 6.8: Template-based implementation of Algorithm

template< typename BodyType > // Type of the rigid bodies
class Algorithm
{
public:
  // ...
  void runAlgorithm( BodyType* body )
  {
    body->doSomething(); // <- This will always compile since the check
    // for a valid function call is deferred until
    // Algorithm is instantiated
    // ...
  }
  // ...
};
```
In this case, the compiler has to defer the evaluation of the correctness of the function call until the type of the rigid body is known. In case the Algorithm class is not selected, this does not result in an error anymore.

The severity of this disadvantage completely lies in the eye of the beholder. Depending on the situation, it might be considered a minor inconvenience or it might even lead to an immediate rejection of the design pattern. Whether or not the Sandwich Pattern can be applied in a certain situation is therefore completely depending on the actual problem.

### 6.3 The Hierarchical Sandwich Pattern

It is obviously also possible to apply the Sandwich Pattern on several levels of a class hierarchy. For instance, the design pattern could also be applied to classes deriving from the RigidBody class. Let us assume that the Sphere class represents a spherical rigid body and derives publicly from the RigidBody class. By applying the Sandwich Pattern, the implementation of the original Sphere class would be refactored as following:

**Listing 6.9: Application of the Sandwich Pattern to classes derived from RigidBody**

```cpp
// Type of the compile-time selected configuration
typedef /* customization type */ Config;

// ... The classes RigidBodyBase, RigidBodyTrait, and RigidBody are implemented as before

// Implementation of the SphereBase class containing all common data and function member for spherical rigid bodies
class SphereBase : public RigidBody
{
  private:
    // Common sphere-specific private data members and functions
    // Sphere-specific implementation of private virtual functions
  
  public:
    // Common sphere-specific public functions
    // Sphere-specific implementation of public virtual functions
};

// Implementation of the customizable SphereTrait class template
template< typename C > // Type of the configuration
class SphereTrait : public SphereBase
{
  private:
    // Configuration-specific private sphere data members and functions
    // Implementation of configuration-specific private virtual functions
  
  public:
    // Configuration-specific sphere public functions
    // Implementation of configuration-specific public virtual functions
};
```

69
In case the Sandwich Pattern is applied in this way, it is not only possible to change certain static characteristics of a type (i.e., replace a certain set of member data and functions), but it is also possible to customize the dynamic characteristics of the type by completely replacing a set of virtual functions by another set. Depending on the configuration, each specialization of the RigidBodyTrait class can individually define a certain number of virtual functions, that are implemented in the according trait class of the derived class depending on the characteristics of both the derived type and the configuration. Figure 6.3 gives an impression of the hierarchical application of the Sandwich Pattern to a class hierarchy based on the RigidBody base class:

![Diagram](image)

**Figure 6.3:** Hierarchical application of the Sandwich Pattern to the class hierarchy based on the RigidBody class.

### 6.4 Application of the Sandwich Pattern in the pe

One of the major goals of the pe framework is to provide a large selection of different algorithms for all phases of a time step in the rigid body simulation. This includes different algorithms for

(a) the (coarse and fine) collision detection phase,

(b) the contact grouping (batch generation) phase, and

(c) the collision resolution phase.

The algorithms in each category differ in the way they address the different problems. For instance, for certain simulations it is important to resolve collisions as physically accurate as possible whereas in other simulations the real-time characteristics of the algorithms (i.e., the fast but less accurate processing of collisions) are the crucial requirement. Also the selection
of a different coarse collision detection algorithm depending on the characteristics of the simulated scenario might be an important decision for the overall performance of the simulation. However, depending on the selected algorithm in each category the requirements on the RigidBody data structure change. For instance, the UnionFind batch generation algorithm requires additional functionality and data members in each rigid body in order to be able to efficiently create groups of contacting bodies. It can be easily seen why the Sandwich Pattern is one of the essential design patterns applied in the pe framework: via this design pattern it is possible to add the according algorithm-specific function and data members to the RigidBody class.

The selection of the different algorithms happens in a configuration file:

```
# define pe_COARSE_COLLISION_DETECTOR pe::detection::coarse::HashGrids
# define pe_FINE_COLLISION_DETECTOR pe::detection::fine::MaxContacts
# define pe_BATCH_GENERATOR pe::batches::UnionFind
# define pe_CONTACT_SOLVER pe::response::FFDSolver
```

For all four categories, an algorithm has to be defined. In this example the FFDSolver is the selected contact solving algorithm. The selection of all algorithms is combined via the Configuration class to the compile time configuration Config, which is used to instantiate all trait classes:

```
// Definition of the Configuration class
// CD: Type of the coarse collision detection algorithm
// FD: Type of the fine collision detection algorithm
// BG: Type of the batch generation algorithm
// CR: Type of the collision response algorithm
template< template< typename > class CD
    , typename FD
    , template< typename > class BG
    , template< typename , typename , typename > class CR >
struct Configuration
{ /* ... */ };;

// Type definition of the active configuration based on the compile-time
// selection of rigid body algorithms
typedef Configuration< pe_COARSE_COLLISION_DETECTOR
    , pe_FINE_COLLISION_DETECTOR
    , pe_BATCH_GENERATOR
    , pe_CONTACT_SOLVER
    > Config;
```

Listing 6.12 shows an excerpt of the actual realization of the Sandwich Pattern within the pe framework by means of the RigidBody base class and the derived Sphere class:
Listing 6.12: Application of the Sandwich Pattern to the RigidBody class hierarchy

```cpp
// Definition of the rigid body base class
class RigidBodyBase {
private:
  real mass_; // The total mass of the rigid body
  Vec3 gpos_; // The global position of the center of mass of this rigid body
  Vec3 v_;   // The linear velocity of this rigid body
  Vec3 w_;   // Angular velocity of this rigid body
  // ...

public:
  inline real getMass() const { return mass_; }
  inline const Vec3& getPosition() const { return gpos_; }
  inline const Vec3& getLinearVel() const { return v_; }
  inline const Vec3& getAngularVel() const { return w_; }
  // ...
};

// Definition of the default implementation of the rigid body customization
// layer (for instance suited for LCP-based collision solvers)
template <typename C> // Type of the configuration
class RigidBodyTrait : public RigidBodyBase {
public:
  // ...
  virtual void move(real dt) = 0;
};

// Definition of the rigid body top level class
class RigidBody : public RigidBodyTrait<Config> {
private:
  RigidBody* sb_; // The superordinate rigid body
  // ...
public:
  // ...
};

// Definition of the sphere base class
class SphereBase : public RigidBody {
private:
  real radius_;  // ...
public:
  inline real getRadius() const { return radius_; }
  // ...
};
```
6.4 Application of the Sandwich Pattern in the ... in the Sandwi...

```cpp
// Definition of the default implementation of the sphere customization
// layer (for instance suited for LCP-based collision solvers)
template< typename C > // Type of the configuration
class SphereTrait : public SphereBase
{
  public:
    // ...
    virtual void move( real dt ) { /* Sphere-specific implementation */ }
    // ...
};
// Definition of the sphere top level class
class Sphere : public SphereTrait<Config>
{
  public:
    // ...
  private:
    // ...
    friend Sphere* createSphere( size_t uid, const Vec3& gpos, real radius,
    MaterialID material, bool visible );
};
```

In contrast to the abstractions of the previous sections, in this example certain parts of the actual implementation are shown to demonstrate the use of the Sandwich pattern. The RigidBodyBase class contains numerous common data members, as for instance the mass, the global position, the linear and angular velocity of a rigid body, along with the necessary functionality to access those values. These values are common to all rigid bodies and are independent of the functionality contained in the customizable functionality in the next class of the inheritance hierarchy. The RigidBodyTrait class, which inherits all these common characteristics, declares the move function that is used as default to move rigid bodies according to their current velocity and the acting forces for a time interval of \( dt \). The RigidBody class, that publicly derives from the RigidBodyTrait instance for the current configuration, only contains common functionality that cannot be moved to the RigidBodyBase class. For instance, it contains a handle to a superordinate rigid body of type RigidBody*, which refers to a body that may contain this particular rigid body.

SphereBase represents the base class for a spherical primitive and as such defines the necessary geometric information (in this case the radius of the sphere) along with the according access function. The SphereTrait class template implements the move function that was introduced by the according trait class of RigidBody for spherical primitives. On the top level of the hierarchy, the Sphere class provides access for the according sphere setup function.

Depending on the selection of the algorithms (see Listing 6.10), the according specialization of the trait classes is selected. In case of the FFDSolver, the move function does not suit the requirements and a slightly different behavior is required (for more information see Chapter 10 and [68]). Therefore a specialization for all trait classes is created (see Listings 6.13 and 6.14), which defines a different set of virtual functions. In the according specializations of the RigidBodyTrait and the SphereTrait class, the move function is completely replaced by
the two functions `firstPositionHalfStep` and `secondPositionHalfStep`, which perfectly suit the requirements of the FFDSolver. Please note that only due to the application of the Sandwich pattern we are able to replace a single virtual function with two virtual functions. Therefore the data structures are nicely adapted to the compile time selection of the applied algorithms.

**Listing 6.13: Specialization of RigidBodyTrait class for the FFDSolver**

```cpp
// CD: Type of the coarse collision detection algorithm
// FD: Type of the fine collision detection algorithm
// BG: Type of the batch generation algorithm
// C: Type of the configuration
template < template < typename > class CD,
          typename FD,
          template < typename > class BG,
          template < template < typename > class,
          typename,
          template < typename > class
          > class C >
class RigidBodyTrait < C<CD,FD,BG,response::FFDSolver> >
    : public RigidBodyBase {
  public:
    // ...
  virtual void firstPositionHalfStep ( real dt ) = 0;
  virtual void secondPositionHalfStep ( real dt ) = 0;
  // ...
};
```

**Listing 6.14: Specialization of SphereTrait class for the FFDSolver**

```cpp
// CD: Type of the coarse collision detection algorithm
// FD: Type of the fine collision detection algorithm
// BG: Type of the batch generation algorithm
// C: Type of the configuration
template < template < typename > class CD,
          typename FD,
          template < typename > class BG,
          template < template < typename > class,
          typename,
          template < typename > class
          > class C >
class SphereTrait < C<CD,FD,BG,response::FFDSolver> > : public SphereBase {
  public:
    // ...
    virtual void firstPositionHalfStep ( real dt ) {
      /* Sphere-specific implementation */
    }
    virtual void secondPositionHalfStep ( real dt ) {
      /* Sphere-specific implementation */
    }
    // ...
};
```
6.5 Conclusion

In this chapter I have introduced the C++ design pattern called “Sandwich Pattern”, which is representative of the fundamental software design of the pe. Due to this design pattern, the basic data structures for rigid body simulation gain a remarkable flexibility and customizability without the loss of runtime efficiency due to abstraction. Via this pattern it is possible to replace at compile time a specific set of static and dynamic characteristics of a data type while leaving certain other characteristics untouched. I have demonstrated the application of this design pattern by means of an extended example of the pe framework where rigid bodies are statically adapted to a compile time selection of algorithms working on these rigid bodies.
7 Resource Management

The management of resources is one of the most fundamental tasks in every software framework to guarantee the overall correctness and reliability of the software. However, although the C++ programming language provides all features to explicitly handle all kinds of resources perfectly and elegantly, very often the design of appropriate software components responsible for the resource management is neglected. Yet proper resource management requires a more sophisticated software design, especially in the case of complex frameworks. In this chapter I will give insight into the software design of the resource management for rigid bodies within the pe framework. I will illustrate in detail the design decisions made and explain all involved classes and I will additionally demonstrate the impact the design of the user interface has on the correctness of the resource management.

7.1 Motivation

One of the most basic design decisions within a software framework is how programmers have to acquire and release resources. In order to illustrate the consequences of these decisions, let us consider the example code in Listing 7.1, which demonstrates a possible design for the management of the resources in a rigid body simulation framework such as the pe. In this example, the World class represents the environment for a rigid body simulation, and the Sphere class represents one possible instance of a rigid body:

Listing 7.1: Example for a poor interface design (I)

```c++
int main()
{
    // Creating a new environment for a rigid body simulation
    World* world = new World ( /*...*/ );

    // Creating a new sphere primitive
    Sphere* sphere = new Sphere( /*...*/ );
}
```
Hardly anything positive can be said about this particular design\(^1\). Both the simulation world, which represents a kind of container for all rigid bodies in a particular simulation, as well as the rigid bodies have to be explicitly created via the \texttt{new} operator\(^2\). This means that users of this interface are directly confronted with the raw resources behind plain pointers. Additionally, rigid bodies even have to be explicitly added to the simulation world to make them part of the simulation. After the simulation, the rigid bodies have to be manually removed from the simulation world and both the rigid bodies as well as the simulation world have to be explicitly destroyed via the \texttt{delete} operator.

This particular design of the user interface is extremely error-prone. The basic reason for this is that the resources are not handled appropriately, as for instance according to the RAII idiom [97, 96, 53]. Therefore, this might easily lead to resource leaks in case the programmer forgets the according \texttt{delete} statements and in the presence of multiple execution paths due to exceptions and additional return statements (see [33, 34, 105]). Even worse, however, is the fact that the application programmer is responsible for managing the resources manually all by himself. It is generally never a good idea to leave the resource management up to the user of the framework. In this example, it is even necessary to explicitly register and deregister rigid bodies within the simulation world. Such a design leaves much room for potential errors:

\begin{Verbatim}
78

Listing 7.2: Example for a poor interface design (II)

\begin{verbatim}
1 Sphere* sphere = new Sphere( /*...*/ ); // Creating a new sphere primitive
2 world->add( sphere ); // Registering the sphere primitive
3
4 // ... Running part of the rigid body simulation
5 delete sphere; // Destroying the sphere primitive, but not deregistering
6 world->simulationStep( /*...*/ ); // Performing another time step in the
7 // simulation (potential disaster!)
\end{verbatim}

\(^1\)Unfortunately, even sadly, designs like this can still be encountered in some published frameworks.

\(^2\)Note the difference between the \texttt{new} operator and operator \texttt{new}: Whereas the \texttt{new} operator cannot be overloaded since it is a built-in C++ concept that combines memory allocation and construction of an object, operator \texttt{new} is a customizable operator to exclusively handle the memory allocation.
The example in Listing 7.2 demonstrates one of the errors that will eventually happen. In this case, the programmer has forgotten to explicitly remove the sphere primitive from the simulation world via the `remove` function. The already destroyed sphere is therefore still registered as a valid rigid body within the simulation world when the call to the `simulationStep` function is executed. The outcome of this function call is completely undefined. It might deceptively succeed (and therefore remain undetected for a long period), or it might immediately crash the program. Whatever happens, the sole possibility for this error should be convincing enough to show that this particular design is inappropriate for the user interface of a rigid body simulation framework.

Listing 7.3 demonstrates the much cleaner, less verbose, less complicated, and in consequence less error-prone interface of the `pe` framework. The `createSphere` function creates a new sphere primitive and automatically registers this primitive with the simulation world. Instead of a pointer to a raw resource, the function returns a handle of type `SphereID` to the newly created primitive that can be used to manipulate the rigid body (note for instance the call to the `setLinearVel` function in line 4 to set the linear velocity of the rigid body). Since a handle is returned instead of a pointer, the question of whether the resource has to be destroyed via an explicit call to the `delete` operator never arises. In order to destroy the sphere primitive manually, the `destroy` function can be used, which handles both the deregistration process as well as the memory management. In case the primitive is not manually destroyed, it will be destroyed automatically at the end of the simulation. Please also note the use of the `theWorld` function: In the `pe` only a single simulation world exists. Access to this world can be gained via the `theWorld` function, which returns a handle to the simulation world (see Section 7.4).

Listing 7.3: Interface design of the `pe` framework

```c
int main()
{
    SphereID sphere = createSphere( /*...*/ ); // Creating a new sphere
    sphere->setLinearVel( /*...*/ ); // Setting the linear velocity

    // ... Running part of the rigid body simulation
    destroy( sphere ); // Destroying and deregistering the sphere primitive

    WorldID world = theWorld();
    world->simulationStep( /*...*/ ); // Performing another time step in
    // the simulation
}
```

In contrast to the design illustrated in Listing 7.1, the `pe` design of the user interface is focused on clarity and on preventing accidental misuse [53], and therefore favors the correctness of user code. Additionally, the `pe` design is much easier to maintain, and completely exception safe without the loss of runtime efficiency.

---

3“Proper interface design is as much a question of psychology and empathy as technical prowess.” (Steven Dewhurst) [33]

4And if this question does arise then the compiler will reject the code complaining that it is impossible to destroy a rigid body via the `delete` operator; see Section 7.2.
The following sections will explain this particular design in more detail. Section 7.2 will start by giving a short introduction of how the user interface should be designed to prevent misuse and by that protect the correctness of the simulation. Additionally, it will explain how resources are abstracted in the context of the user interface of the pe. After that, Section 7.3 will answer the question of how rigid bodies can be automatically managed and how the resource management is transferred from the user to the pe system. The section will give an in-depth overview of the BodyManager class, which represents the general interface for all entities managing rigid bodies. Section 7.4 will focus on the implementation of the World class, which represents the most important rigid body management entity in the pe. The chapter is concluded in Section 7.5.

7.2 Resource Abstraction

Although the example interface illustrated in Listing 7.1 is a particularly bad example, certain basic concepts of this code have to be adopted in the pe framework. In order to provide a flexible setup of rigid body scenarios it is necessary to dynamically allocate rigid bodies on the free store at runtime. Therefore, if the pe framework is supposed to automatically destroy all rigid bodies at the end of the simulation, it is inevitable that the pe system has to take the responsibility for dynamic memory resources. Additionally, it is necessary to return some kind of handle to newly created objects to enable manipulations of rigid bodies.

The generally simplest and most straightforward handles to dynamic resources are plain pointers. They are easy to use, provide direct access to a specific resource, they have a small memory footprint, and they provide the possibility to exploit dynamic polymorphism. However, plain pointers possess several intrinsic disadvantages if used in the interface of a framework. One of the most severe disadvantages is that plain pointers do not clearly convey the message how users have to deal with the resource, i.e., whether or not users have to handle the resource by themselves. This is especially complicated by the fact that with plain pointers users have the ability to interfere with the resource management:

```
1 // Creating a new sphere primitive and automatically registering
2 // it within the simulating world
3 Sphere* sphere = createSphere( /*...*/ );
4
5 // ...
6
7 // Manually destroying the sphere.
8 // This is unintended by the framework and thus might result in
9 // disaster since the sphere has not been properly deregistered!
10 delete sphere;
```

The basic problem is that users are directly confronted with plain pointers and are therefore perfectly aware what options they have. Additionally, the resource is not properly encapsulated and therefore cannot protect itself from misuse [35]. A better approach in this regard is the use of smart pointers [75], which are based on the RAII idiom:
7.2 Resource Abstraction

Listing 7.5: Example for `shared_ptr` handles

```cpp
// Creating a new sphere primitive.
boost::shared_ptr<Sphere> sphere = createSphere(/* ... */);

// ...

// Compilation error: use of the delete operator is not possible
delete sphere;

// Still, the resource can be manually destroyed
sphere.reset();
```

In contrast to plain pointers it is not possible to (accidentally) destroy the resource by a call to the `delete` operator (i.e., we are safe of Murphy errors [53]). It might still be possible to manually destroy the resource by an explicit call to the `reset` function, but this falls into the category of intentional misuse (i.e., Machiavelli errors [53]). However, depending on which type of smart pointer is used for this task, this might result in a severe performance penalty due to additional functionality, as for instance in the case of `boost::shared_ptr`, which performs reference counting. This is especially true in case the smart pointer is very frequently copied. Even for very simple smart pointer implementations, which merely wrap the pointer resource into an object (as for instance the `pe::SimplePtr`), a performance penalty can be measured in comparison to plain pointers. Listing 7.6 shows a small benchmark program to measure the performance impact of smart pointer copy operations. In order to simulate frequent copy operations, $10^8$ function calls are executed for each type of (smart) pointer.

Table 7.1 summarizes the according benchmarking results.

Listing 7.6: Benchmark program for the performance penalty of smart pointers

```cpp
size_t counter (0);

// Definition of the Sphere class
class Sphere { /* ... */ };

// Definition of the test functions
void doSomething( Sphere* sphere ) { ++counter; }
void doSomething( SimplePtr<Sphere> sphere ) { ++counter; }
void doSomething( shared_ptr<Sphere> sphere ) { ++counter; }

int main()
{
    const size_t N( 100000000 );

    Sphere* plain( new Sphere() );
    pe::SimplePtr<Sphere> simple( new Sphere() );
    boost::shared_ptr<Sphere> shared( new Sphere() );
```

Note that the functions expect their arguments by value and not by reference. Passing the smart pointers by reference removes the performance penalty since for instance in case of the `boost::shared_ptr` no reference counting has to be performed. However, the purpose of this test is to evaluate the performance penalty of the necessary copy operation. Also note that the sole task performed within the `doSomething` functions is to increment the global variable `counter`. In general, it is a bad idea to use global variables, but in this case it serves for illustration purposes only.
// Benchmarking the performance of plain pointers
pe::timing::WcTimer timer1;
for ( size_t i =0; i<N; ++i ) {
    doSomething( plain );
}
timer1.end();

// Benchmarking the performance of SimplePtr
// ...

// Benchmarking the performance of shared_ptr
// ...

<table>
<thead>
<tr>
<th></th>
<th>Runtime [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>boost::shared_ptr</td>
<td>6.82824</td>
</tr>
<tr>
<td>pe::SimplePtr</td>
<td>0.65166</td>
</tr>
<tr>
<td>plain pointer</td>
<td>0.316259</td>
</tr>
</tbody>
</table>

Table 7.1: Performance comparison for $10^8$ function calls using different (smart) pointers as handles, which are passed to the function as arguments. The test machine for all performance tests was an Intel Core2 Quad Q6700 CPU at 2.66 GHz. All executables are compiled with the GNU G++ 4.4.1 compiler (branch revision 150839).

Clearly the additional functionality of the boost::shared_ptr can cause a serious performance decrease. Even for a smart pointer implementation without additional functionality, a measurable performance drop is caused in comparison to plain pointers. For this reason, and in order to preserve pointer semantics, i.e., in order to be able to use dynamic polymorphism, the pe framework uses plain pointers for all dynamically allocated resources that are very frequently used. However, this implementation detail is elegantly hidden via the ID concept (which is similar to the concept used in the ODE [63]):

Listing 7.7: Illustration of the pe ID concept

```cpp
class BodyManager;
class RigidBody;
class Sphere;
class World;
// ...

typedef BodyManager* ManagerID;
typedef const BodyManager* ConstManagerID;
typedef RigidBody* BodyID;
typedef const RigidBody* ConstBodyID;
typedef Sphere* SphereID;
typedef const Sphere* ConstSphereID;
typedef boost::shared_ptr<World> WorldID;
typedef boost::shared_ptr<const World> ConstWorldID;
// ...
```
All resource handles, whether they are plain pointers or smart pointers, are unified via convenient type definitions. For instance, the type BodyID abstracts from the fact that a rigid body handle is implemented in terms of a plain pointer. In comparison, WorldID hides the fact that a simulation world handle is implemented in terms of a boost::smart_ptr. Yet, hiding the implementation details alone is not enough. It is still possible to accidentally destroy resources manually via the delete operator. In order to prevent that, the destructors of these classes are declared in a non-public section. Listing 7.8 illustrates this for the RigidBody base class:

```
Listing 7.8: Extensions of the RigidBody class (I)

class RigidBody : public RigidBodyTrait<Config>
{
   // ...
   protected:
   RigidBody( /* ... */ );
   virtual ~RigidBody() = 0;
   // ...
   private:
   friend void destroy( BodyID body );
};
```

In case of the RigidBody base class, both the constructor as well as the destructor are declared in a protected section. Therefore, only derived classes are allowed to access these functions and it is not possible to destroy an object of type RigidBody manually via the delete operator. In order to grant access to the destructor for the destroy function, a friend declaration is added to the class definition (see Section 7.3 for the definition of the destroy function).

### 7.3 The BodyManager Class

In order to provide the feature of automatic management of rigid bodies, i.e., their registration, deregistration, and destruction, the pe framework requires a suitable mechanism for the management of dynamically allocated rigid bodies. The requirements on this mechanism are comparatively high: It must guarantee a perfectly stable and reliable rigid body management even in the presence of exceptions, and it should be extensible and adaptable to yet unknown future requirements. In the pe framework, this mechanism is based on the BodyManager base class. This class represents the abstract interface for all entities responsible for the management of rigid bodies. Listing 7.9 shows the complete interface of the BodyManager class:

---

6See Chapter 6 for information about the concept and the implementation of the RigidBodyTrait base class.
7This concept is also applied to all classes derived from the RigidBody base class, as for instance the Sphere class.
7 Resource Management

Listing 7.9: Class definition of the BodyManager base class

```cpp
class BodyManager {
public:
  virtual void add ( BodyID body ) = 0;
  virtual void remove ( BodyID body ) = 0;

inline void setDefaultManager();

protected:
  ~BodyManager();

  inline void setManager ( BodyID body );
  inline void resetManager( BodyID body );
  static inline void destroyBody ( BodyID body );

private:
  // The currently active default body manager.
  // This body manager handle refers to the currently active default body
  // manager. Per default, this role is taken by the simulation world (see
  // the World class). However, every body manager can be instated as the
  // default body manager by using the setDefaultManager() member function.
  static ManagerID default_;

friend ManagerID theDefaultManager();
};
```

The public interface of the BodyManager class merely provides three member functions. The first of these functions is the add function, which represents the abstract interface to add a rigid body to the body manager. A call to this function additionally transfers the responsibility for the resource to the corresponding body manager. Accordingly, the remove function represents the interface to remove a rigid body, and with it the responsibility for the resource, from the body manager. Both functions are declared as pure virtual and have therefore to be defined by classes deriving from the BodyManager class (as for instance the World class; see Section 7.4).

The protected interface of the BodyManager class provides additional functionality, which is necessary to properly manage rigid bodies, to all classes deriving from BodyManager. A body manager can register itself as the manager of a particular rigid body via the setManager function, or reset the body manager of a rigid body with the resetManager function. Additionally, body managers can destroy rigid bodies by use of the destroyBody function, i.e., they are allowed to manage the dynamically allocated resources. Also note the fact that the non-virtual destructor of the BodyManager class is declared within a protected section of the class definition. Therefore it is not possible to destroy any body manager via a base class pointer, but derived classes are granted the required access to this function.

The purpose of the setDefaultManager function is to set the corresponding body manager as the current default manager. Via this feature it is possible to automatically register a new rigid body with a specific body manager. Per default, the simulation world acts as the default manager, but it might be temporarily replaced by other body managers. For
instance, it is convenient to temporarily register a `Union` compound geometry as the default body manager. Via the `pe_CREATE_UNION` smart scope\(^8\) this can be achieved automatically:

**Listing 7.10:** Implementation of the `destroy` function

```csharp
// ...  
pe_CREATE_UNION( union2, 2 )
{
    // Creating the oak sphere 3 with a radius of 1.2 at the global
    // position (0.2, -5.0, 6.3). Since the sphere is created inside
    // the scope of a `pe_CREATE_UNION`, the sphere is automatically
    // and directly added to the newly created union instead of the
    // simulation world.
    createSphere( 3, Vec3( 0.2, -5.0, 6.3 ), 1.2, oak );
}
// ...  
```

The currently active default manager can be accessed via the `theDefaultManager` function:

**Listing 7.11:** Implementation of the `theDefaultManager` function

```csharp
ManagerID theDefaultManager ()
{
    if( BodyManager::default_ == NULL )
        BodyManager::default_ = theWorld().get();
    return BodyManager::default_;
}
```

In case no default manager has been defined (which usually happens the first time a rigid body needs to be registered), the simulation world is used as the default manager. After that a handle of type `ManagerID` is returned. Note the use of the `get` function on the return value of the `theWorld` function. The `theWorld` function returns a handle to the simulation world of the `pe`. Since world handles are implemented as smart pointers (`boost::shared_ptr<World>`), it is necessary to explicitly acquire the encapsulated resource. In case the `BodyManager` destructor would be publicly accessible and it would therefore be possible to destroy body managers via base class pointers, this would result in a severe resource leakage. Due to the non-public destructor, this leakage of resources cannot be accidentally or intentionally exploited.

Listing 7.12 shows the necessary extensions of the `RigidBody` class for the automatic resource management by the `BodyManager` mechanism:

**Listing 7.12:** Extensions of the `RigidBody` class (II)

```csharp
class RigidBody : public RigidBodyTrait<Config>
{
    public:
        inline bool hasManager() const;
        inline ManagerID getManager();
        inline ConstManagerID getManager() const;
        // ...
```

\(^8\)See Chapter 4 for more information about the `pe` smart scopes.
Via the three public functions hasManager and getManager (one non-const and one const version) it is possible to acquire the current body manager responsible for a particular rigid body. In order to grant access to the non-public members for the special functionality of the BodyManager, the RigidBody class definition additionally contains an according friend declaration. Although a friend declaration expresses the strongest possible relation between two classes, in this context this declaration is perfectly justified. It merely reflects the special relation between rigid bodies and the BodyManager class and provides the controlled management of rigid bodies by the according authorized entities.

With the described concepts in place, the rigid bodies can safely and automatically be registered with the currently active body manager. This is for instance done in all rigid body setup functions. Listing 7.13 shows the essential parts of the createSphere function, which is used to create a new sphere primitive:

```
Listing 7.13: Implementation of the createSphere function

SphereID createSphere( id_t uid, const Vec3& gpos, real radius,
    MaterialID material, bool visible )
{
    // Checking the radius of the sphere
    if( radius <= real(0) )
        throw std::invalid_argument( "Invalid sphere radius" );

    // Creating a new sphere
    SphereID sphere = new Sphere( Sphere::createUniqueID(), uid,
        gpos, radius, material, visible );

    // Registering the new sphere with the default body manager
    try {
        theDefaultManager()->add( sphere );
    }
    catch( std::exception& ) {
        delete sphere;
        throw;
    }

    return sphere;
}
```

The function initially checks all input arguments whether the function preconditions have been fulfilled. In case they have not, a std::invalid_argument exception is thrown. After that the sphere primitive is allocated and initialized according to the passed arguments. This primitive is then added to the currently active default body manager. Since the sphere
The World Class

The primitive has not been encapsulated within a RAII class, it is necessary to embed the call to
the `add` function within a `try-catch` block in order to make the function exception safe. The
last line returns the handle to the newly created sphere.

Conceptually, rigid bodies cannot be manually destroyed. Instead, the `destroy` function
has to be used. Listing 7.14 illustrates the implementation of the `destroy` function. The
assertion in line 3 checks the basic assumption that each individual rigid body must have a
valid manager. After that the rigid body is removed from its current body manager (i.e., the
responsibility for its management is transferred to the `destroy` function), and is destroyed.

**Listing 7.14: Implementation of the `destroy` function**

```cpp
donotcopyable
void destroy( BodyID body )
{
   assert( body->hasManager(), "Body has no rigid body manager" );
   body->getManager()->remove( body );
   delete body;
}
```

7.4 The World Class

So far the entire resource management has been based on the abstract `BodyManager` class. All
rigid bodies (i.e., all classes deriving from the `RigidBody` base class) can now conveniently be
managed automatically, independent of any concrete body manager implementation. There-
fore, the only missing component are classes implementing the given abstract interface ac-
cordingly. One of these classes, which represents the most important body manager in the
pe system, is the `World` class. From a logical point of view, the `World` class represents the
rigid body simulation: All rigid bodies participating in the simulation are contained in the
simulation world. From an implementational point of view, the `World` is a container for an ar-
bitrary number of rigid bodies. Listing 7.15 shows an overview of the essential functionality
of the `World` class in the context of rigid body resource management:

**Listing 7.15: Class definition of the `World` class**

```cpp
class World : public BodyManager, private boost::noncopyable
{
private:
   explicit World();
   // ...
public:
   virtual ~World();
   virtual void add( BodyID body );
   // ...
private:
   virtual void remove( BodyID body );
   // ...
   friend WorldID theWorld();
};
```
The `World` class inherits publicly from the `BodyManager` class, which makes the world a body manager. Additionally, the `World` inherits privately from the `boost::noncopyable` class, which prohibits any copy operations (i.e., copy construction and copy assignment; see [75]). According to the interface the `World` class inherits from the `BodyManager` class, the `World` class implements both the `add` function as well as the `remove` function depending on its internal implementation details. Listing 7.16 illustrates the essential part of the `World::add` function:

```
void World::add( BodyID body )
{
    // Removing the body from its old body manager
    if( body->hasManager() )
    {
        if( body->getManager() == ManagerID( this ) ) return;
        ManagerID manager = body->getManager();
        manager->remove( body );
    }
    // Registering the world as manager for the rigid body
    setManager( body );
    // Adding the rigid body to the world data structures
    // ...
}
```

Note that whereas the `add` function is declared in the `public` section of the `World` class, the `remove` function is declared in the `private` section. This implementation detail results from the fact that adding a rigid body to one body manager should automatically remove it from another body manager (line 8). Therefore the `remove` function is exclusively called within the `pe` via a base class pointer and can be made `private` in the `World` class in order to prevent any (accidental) misuse.

The most intricate implementation detail of the `World` class, however, is the automatic destruction of all rigid bodies contained in the simulation at the end of the `main` function:

```
int main()
{
    // Rigid body simulation
}
// Automatic destruction of the pe simulation world
// including all contained rigid bodies
```

This behavior can be efficiently modeled by the singleton design pattern [44, 14] for two reasons. The first reason is the observation that there only needs to be a single simulation world at the same time. The second reason is the fact that the singleton pattern is designed for object life time management and is therefore perfectly suited for this particular task. For the purpose of the `World` class, the singleton pattern in combination with `shared::ptrs` is especially beneficial since this combination allows an explicit life time control of the single `World` instance. Listing 7.18 illustrates the implementation of the `World` singleton pattern within the `theWorld` function.
7.5 Conclusion

Listing 7.18: Implementation of the `theWorld` function

```cpp
inline WorldID theWorld()
{
    static WorldID world( new World() );
    return world;
}
```

The function can be implemented in only two lines of code. The first line creates a function local `shared_ptr` object, which immediately takes control of the one `World` instance\(^9\). Only the first time the `theWorld` function is called, this line is executed and the function local object is created. However, this object persists until the end of the program, i.e., until after the `main` function is finished. It is even possible to postpone the destruction of the single `World` instance, for instance in order to guarantee its destruction after a certain global/static object, by storing a `shared_ptr` handle within the global/static object. With this it is possible to guarantee that all rigid bodies contained in the simulation world are accessible as long as necessary.

The second line of the `theWorld` function merely returns the `shared_ptr` as handle to the simulation world. In the context of the `theWorld` function, also note the declarations of the constructor and destructor of the `World` class: Whereas the destructor is declared in a `public` section, the constructor is declared `private`. This implementation detail is part of the singleton pattern and guarantees that it is not possible to create a second `World` instance. In order to grant the `theWorld` function access to the constructor, an according `friend` declaration is added to the `World` class.

7.5 Conclusion

This chapter has given insight into the sophisticated software and interface design of the `pe` framework, which is necessary to create a reliable, correct, and (exception) safe resource management. I have demonstrated the possible pitfalls resulting from a badly designed interface and emphasized the benefits of a carefully designed interface on the correctness of the resource management.

\(^9\)Note that this single line of code is perfectly exception safe. In case the `new` operator throws an exception, no resources are lost since none have been acquired. In case the `World` constructor throws, the standard guarantees that the allocated memory is freed again. No resources are lost in this case either. The last operation, the construction of the `shared_ptr` cannot throw. In summary, no resource leaks are possible in this single line of code.
8 The Math Library of the $pe$ framework

Performance optimized linear algebra operations are an essential ingredient for all numerical packages. Therefore several standard frameworks are focused on the performance optimized implementation of these operations, as for instance several packages implementing the BLAS standard. However, with the predominance of the efficiency aspect, other crucial software aspects, such as reliability, usability, and maintainability tend to play only a subordinate role in the development of numerical packages.

In this chapter I introduce the math library of the $pe$ framework. Whereas its main focus is also efficiency, crucial software aspects such as code clarity, usability and maintainability are not forgotten. This library combines the usability and maintainability of C++ code and the performance of an optimized BLAS implementation. Additionally, it unifies operations between dense as well as sparse vectors and matrices of mixed element types. This is achieved by a smart expression template based implementation that allows expression dependent evaluation and optimization of mathematical vector and matrix operations.

The following Section 8.1 will motivate the development of the $pe$ math library. In Section 8.2, the basic idea of expression templates is explained in detail, followed by an in-depth analysis of smart expression templates in Section 8.3. Section 8.4 presents performance results for the $pe$ math library in comparison to the Boost uBLAS library.

8.1 Motivation

A lot of CPU time in numerical simulations is spent in the execution of basic linear algebra subprograms (BLAS), such as vector additions, matrix-vector multiplications, and matrix-matrix multiplications. Obviously, the performance optimized implementation of these operations is crucial for efficient simulations. For this purpose the BLAS standard was introduced to define a set of the most important linear algebra operations. BLAS is focused on providing an interface for highly optimized linear algebra functionality. Since
performance is platform dependent, the BLAS standard has been implemented in various libraries, as for instance the ATLAS [20] library, Intel’s IMKL[71], AMD’s ACML[18], or NVidia’s CUBLAS [88]. All of these frameworks implement the set of BLAS operations defined by the BLAS standard according to its interface. However, all of them are primarily focused on performance, yet neglect several other important aspects of software frameworks, such as usability and maintainability. The following code example shows the signature of the C interface function for the multiplication of two dense, double precision matrices according to the expression \( C = \alpha \cdot A \cdot B + \beta \cdot C \):

**Listing 8.1:** Signature of the BLAS function for the matrix-matrix multiplication for double precision matrices

```c
void cblas_dgemm ( const enum CBLAS_ORDER Order ,
const enum CBLAS_TRANSPOSE TransA ,
const enum CBLAS_TRANSPOSE TransB ,
const int M, const int N,
const int K, const double alpha ,
const double *A, const int lda ,
const double *B, const int ldb ,
const double beta ,
double *C, const int ldc );
```

The trouble with these BLAS functions is the lack of usability and maintainability. The generality of the function formulation due to the language deficiencies of C, and the goal to enable all kinds of matrix-matrix multiplications makes it harder to use this function and still harder for maintainers to understand the performed operation. Usability is also limited due to the general restriction to the two data types `float` and `double` and the restriction to dense data structures. Integral data types and mixed types, as for instance for the multiplication between a `float` and a `double` matrix and sparse data structures are not supported at all. Additionally, it is up to the programmer to make sure the sizes of the matrices match, because the function has no possibility to check for possible errors.

**Listing 8.2:** Usage of the BLAS matrix-matrix multiplication

```c
// Definition of the double precision matrices A, B, and C
double* A = ...;
double* B = ...;
double* C = ...;

// Initialization of the matrices
// ...

// Performing the multiplication
// Are the parameters correct? Are the matrix sizes (still) valid?
// How maintainable is the code?
cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans ,
3, 2, 3, 1.0 , A, 3, B, 2, 0, C, 2 );
```

A different approach is taken by the Boost framework’s uBLAS library [29, 28]. This library is not primarily focused on high performance but on providing user-friendly and maintainable BLAS level 1, 2, and 3 functionality for dense, packed, and sparse vectors and sparse matrices.

---

1In case you also consider complex numbers as a data type, single and double precision complex numbers are also supported.
matrices. Their C++ design and implementation unifies mathematical notation via operator overloading and efficient code generation via expression templates.

Listing 8.3: Matrix-matrix multiplication using the Boost uBLAS library

```cpp
using boost::numeric::ublas;

// Definition of the matrices A, B, and C
matrix<double> A, B, C;

// Initialization of the matrices
// ...

// Performing the multiplication
C = A * B;
```

In contrast to the multiplication via the `cblas_dgemm` function, the intent of the programmer is perfectly clear in this case: The matrices A and B are multiplied and the result is written to matrix C. The complexity of the operation, including all performance optimizations, is hidden behind this intuitive interface. Another usability advantage, in comparison to the BLAS standard, is that even if the matrices are changed to single precision matrices (`matrix<float>`), the multiplication itself remains unchanged. This is even true for mixed type matrix-matrix multiplications:

Listing 8.4: Mixed type matrix-matrix multiplication using the Boost uBLAS library

```cpp
using boost::numeric::ublas;

// Definition of the matrices A, B, and C
matrix<double> A, C;
matrix<float> B;

// Initialization of the matrices
// ...

// Performing the mixed type multiplication
C = A * B;
```

In addition, even if the sizes of the matrices are changed in the initialization part, the overloaded multiplication operator makes sure that only matrices of matching size are multiplied. This gives an important security factor that cannot be provided by the BLAS functions. However, even for the default BLAS functionality for dense vectors and matrices, the performance of the Boost uBLAS implementation severely lags behind the other BLAS implementations.

8.2 Expression Templates

Expression templates are a C++ template programming technique to improve the performance degrading characteristics of the standard operator overloading technique. The expression template technique was concurrently invented by Todd Veldhuizen and David Vandevenorde in 1995 [101, 102, 104]. The basic idea of expression templates is the delay of the computation of overloaded operators by creating intermediate expression objects that represent the result of a numerical operation. Only if these intermediate objects are assigned to
8 The Math Library of the pe

their destination, the evaluation of the expression is performed. This approach circumvents
the creation of expensive temporary objects, which usually consists of a memory allocation, a
copy operation, and a memory deallocation. Since these expression objects can be efficiently
inlined by the compiler, expression template based C++ code can attain 95-99.5% efficiency
in comparison to hand-crafted C code and speed improvements of 2-15 times in comparison
to conventional C++ code [55].

To illustrate the technique of expression templates, the following example will first recall
the classical approach of operator overloading, before demonstrating the addition of dense
vectors via expression objects.

Listing 8.5: Basic implementation of a vector class

```cpp
1 template< typename Type > // Type of the vector elements
2 class Vector
3 {
4     public:
5         explicit Vector( std::size_t n, Type value=Type() )
6             : n_(n), v_(new Type[n])
7         {
8             std::fill( v_, v_+n, value );
9         }
10
11         Vector( const Vector& v )
12             : n_(v.n_), v_(new Type[n_.n_])
13         {
14             std::copy( v.v_, v.v_+n_, v_ );
15         }
16
17         ~Vector() { delete[] v_; }
18
19         Vector& operator=( const Vector& rhs )
20         {
21             if( &rhs == this ) return * this ;
22             resize( rhs.size_ );
23             std::copy( rhs.v_, rhs.v_+n_, v_ );
24             return * this ;
25         }
26
27         std::size_t size() const { return n_; }
28
29         Type& operator[]( size_t i ) { return v_[i]; }
30         const Type& operator[]( size_t i ) const { return v_[i]; }
31
32         void resize( std::size_t n ) { /* ... */ }
33     }
34
35     private:
36         std::size_t n_; // The current size/dimension of the vector
37         Type* v_; // The dynamically allocated vector elements
38     };
```
8.2 Expression Templates

Listing 8.5 shows the relevant part of the basic implementation of a classical `Vector` class for elements of arbitrary type. A `Vector` can be either created by explicitly specifying its size \(n\) or via copy construction. Additionally, the class provides a copy assignment operator, functions for the direct access to the vector elements, and a function to change the size of a vector.

**Listing 8.6: Addition operator for the addition of two vectors**

```cpp
template<typename Type>
const Vector<Type> operator+(const Vector<Type>& lhs, const Vector<Type>& rhs)
{
    if(lhs.size() != rhs.size())
        throw std::invalid_argument("Vector sizes do not match");

    Vector<Type> tmp(lhs.size());
    for(std::size_t i=0; i<lhs.size(); ++i)
        tmp[i] = lhs[i] + rhs[i];
    return tmp;
}
```

Listing 8.6 shows the according addition operator for the addition of two `Vectors`. The first step is a check whether the sizes of the two vectors match; if not, a `std::invalid_argument` exception is thrown. Otherwise, a temporary vector `tmp` is created and filled with the sum of the two involved vectors, before it is returned.

It is this temporary value that is made responsible for the performance penalty of the C++ implementation in comparison to a C or Fortran implementation [80]. The creation of the temporary value results in a dynamic memory allocation and deallocation, and in a vector copy operation during the initialization of the function return value by the return expression, which is accomplished with copy initialization [34]. However, note that this is only half of the story since the temporary value does not have to be created in case the return value is used to create another vector. In this case, the “named return value optimization” (NRV) as described in section 12.1.1c of the Annotated C++ Reference Manual (ARM) [37], is used to optimize the code:

**Listing 8.7: Application of NRV during copy construction and copy assignment**

```cpp
Vector<double> a, b;
Vector<double> c(a + b); // No temporary value necessary due to NRV
```

If a compiler applies the NRV to a code (which is triggered by the presence of an explicit copy constructor), the local variable `tmp` will be replaced by a reference to the eventual destination of the return value in the caller. It is as if the function were written as follows:

---

2Note that this particular definition of the addition operator is only able to add two vectors of the same element type. By using the `MathTrait` class template (see Appendix C) it would be easily possible to add two vectors of different element types.
In Listing 8.7, during the creation of \( \mathcal{C} \), the compiler is able to directly copy the result of the addition into vector \( \mathcal{C} \). However, this is not possible in the case of the assignment, since the copy assignment operator is a member function that performs operations similar to the destruction of \( \mathcal{C} \) followed by a reinitialization of it. Optimizing the assignment could therefore severely alter the semantics of the program (i.e., remove potential side effects of the copy assignment operator). Therefore the compiler is forced to create a temporary object, which results in code similar to the following one:

**Listing 8.9:** Compiler generated code for the copy assignment of vectors

```cpp
Vector<double> a, b, c;
// NRV optimized addition of a and b into the temporary tmp
Vector<double> tmp( a + b );
// Assignment of the temporary to the vector c
c = tmp;
```

The performance penalty from the generation of temporary objects even increases if several additions are concatenated:

**Listing 8.10:** Concatenation of multiple vector additions

```cpp
Vector<double> a, b, c, d;
d = a + b + c;
```

In this case a temporary vector is created for every single expression. Therefore the overhead of the vector addition grows considerably: One memory allocation and deallocation, one copy operation, and one loop execution for every addition expression. This overhead is not avoidable for the classical C++ formulation of the vector addition, although the final result could be calculated in a single \texttt{for}-loop (as it is usually done in a plain C implementation):

**Listing 8.11:** C-like implementation of multiple vector additions

```cpp
Vector<double> a, b, c, d;
for( size_t i=0; i<size; ++i ) { d[i] = a[i] + b[i] + c[i]; }
```
The approach of expression templates is to remove the creation of the costly temporary objects entirely and to delay the execution of the expression until it is assigned to its target. Therefore the addition operator no longer returns the (computationally expensive) result of the addition, but a small temporary object that acts as a placeholder for the addition expression [55]:

Listing 8.12: Expression template formulation of the vector addition

```cpp
template< typename A, typename B >
class Sum {
public:
  explicit Sum( const A& a, const B& b )
    : a_( a ), b_( b )
{} 
  std::size_t size() const {
    return a_.size();
  }
  double operator[]( std::size_t i ) const {
    return a_[i] + b_[i];
  }
private:
  const A& a_; // Reference to the left-hand side operand
  const B& b_; // Reference to the right-hand side operand
};

template< typename A, typename B >
Sum<A,B> operator+( const A& a, const B& b )
{
  if( a.size() != b.size() ) {
    throw std::invalid_argument( "Vector sizes do not match" );
  }
  return Sum<A,B>( a, b );
}
```

Instead of calculating the result of the addition of two vectors, the addition operator now returns an object of type `Sum<A,B>`, where A and B are the types of the left- and right-hand side operands. The only requirements the addition operator poses on A and B are the existence of a subscript operator and a size function. The Sum class merely has two data members, which are references-to-const to the two operands of the addition operation. Therefore this object is very cheap to create and copy in comparison to the complete result vector. Since the Sum class represents the result of an addition, it must provide access to the resulting elements. For this purpose, it defines two access functions: The size function to access the size of the resulting vector and the subscript operator to access the individual elements.

The Sum class now temporarily represents the addition, until an assignment operator is encountered:
This assignment operator is the only other assignment operator of the `Vector` class next to the copy assignment operator (which is strictly necessary due to the management of the dynamically allocated memory). Every time an expression object is assigned to a `Vector`, the assignment operator from Listing 8.13 is used to handle the assignment. It first resizes the vector accordingly and afterwards traverses the elements of the given expression within a single `for`-loop. Note that this traversal triggers the evaluation of the expression due to the access to the values via the subscript operator. Also note that this `for`-loop is the only `for`-loop necessary to evaluate the entire expression.

Via this formulation based on the inline formulation of all functions and the evaluation within a single `for`-loop hidden in the assignment operator it is possible to attain the performance of a C-like implementation as illustrated in Listing 8.11. It is even possible to concatenate several additions without the creation of any temporary object (and still a single `for`-loop evaluation):

```
Listing 8.14: Expression template optimized addition of multiple vectors

Vector< double > a, b, c, d;

d = a + b + c; // Evaluated in a single for-loop
```

The expression instantiation graph that is created during the compilation of the right-hand side expression `a + b + c` is illustrated in Figure 8.1. For the addition of the first two vectors, a temporary object of type `Sum< Vector<double>, Vector<double> >` is instantiated, that represents the addition of the two vectors `a` and `b`. For the addition with of third vector `c` to the result of the addition of `a` and `b`, a temporary object of

---

3 The thorough reader might notice that due to the signature of this assignment operator all non-vector objects assigned to a vector that do not fit the signature of the copy assignment operator will use this assignment operator. The problems associated with this fact will be handled in the remainder of this section.
8.2 Expression Templates

type Sum< Sum< Vector<double>, Vector<double> >, Vector<double> > is created, that represents the addition of all three vectors.

Figure 8.1: Expression template instantiation graph for the addition of three vectors.

Note that the expression objects, which are returned instead of the immediate operation result are temporary objects as well. However, their creation and the according copy operations do not hurt the performance for two reasons. The first reason is the fact that these temporary expression objects do not contain any member data that are expensive to copy. Instead they only contain references to the operands of the according numerical operation. The second reason is the aforementioned NRV. Since the expression objects are not assigned but instead created, the compiler has leeway to optimize the copy operation away entirely. Listing 8.15 demonstrates this for the addition of three vectors as shown in Listing 8.14.

Listing 8.15: Compiler generated temporary objects the addition of three vectors

```c++
Vector<double> a, b, c, d;

// Compiler generated temporary object for the expression object
// returned by the addition of vector a and b. Due to NRV no copy
// operation is required!
const Sum< Vector<double>, Vector<double> > __tmp1__( a + b );

// Compiler generated temporary object for the expression object
// returned by the addition of the first addition of a and b and
// the third vector c. Again, due to NRV no copy operation is
// required!
const Sum< Sum< Vector<double>, Vector<double> >, Vector<double> >
__tmp2__( __tmp1__ + c );

// Assignment of the right-hand side expression to the left-hand side
// vector d
vector d = __tmp2__;
```

In terms of the vector addition, the explained expression template formulation is already complete and working. Figure 8.2 shows a performance comparison between the classical operation overloading technique and the demonstrated expression template formulation for the addition of three vectors as performed in Listing 8.14. In the direct comparison, the expression template based formulation, which requires no temporary vectors (i.e., does not perform any allocations or copy operations) and used only a single for-loop, outperforms the classical operator overloading technique, which involves two memory allocation, deallocations, copy operations, and for-loops, by a factor of 2.25.
Figure 8.2: Performance comparison between the classical operation overloading technique and expression templates for the addition of three vectors. The performance was measured on an Intel Core i7 940 CPU at 2.93 GHz (Bloomfield core) with 8 MByte of shared level three cache using double precision vector elements. The executables were compiled with the GNU G++ 4.4.1 compiler (branch revision 150839).

However, due to the general formulation of the addition operator and the assignment operator of the `Vector` class, this addition operator now also qualifies for additions between completely unrelated types:

Listing 8.16: Imperfection of the general expression template formulation

```cpp
Vector<double> x, y;
Matrix<double> A;

y = A + x;
```

Although it should not be possible to add a matrix and a vector, the current formulation of the addition operator allows this operation. In order to prevent the arbitrary addition of unrelated types, an `Expr` base class is introduced [55]:

Listing 8.17: Final expression template formulation of the vector addition

```cpp
template< typename A >
struct Expr {
    const A& operator~() const {
        return *static_cast<const A*>(this);
    }
};

template< typename Type > // Type of the vector elements
class Vector : public Expr<Vector<Type>> {
    public:
        // ...

    template< typename A >
    Vector& operator=( const Expr<A>& expr ) {
        const A& a( ~expr );
        resize( a.size() );
        for( std::size_t i=0; i<a.size(); ++i )
            v_[i] = a[i];
        return *this;
    }
    // ...
};
```
Both the `Vector` and the `Sum` class now publicly derive from the `Expr` base class and are therefore considered valid operands for the addition operator. The `Expr` base class is based on the “Curiously Recurring Template Pattern” (CRTP) [104], which enables a type-safe downcast to the dynamic type of the expression object. In the implementation of the `pe`, this downcast is explicitly performed via the `operator~`, as for example used in the addition operator. Note that in this improved expression template formulation, both the addition operator as well as the assignment operator are now only accepting `Expr` objects and therefore effectively restrict the number of possible arguments to the allowed expressions.

### 8.3 Smart Expression Templates

Smart expression templates are an extension of the expression template formulation explained in Section 8.2. The term “smart” in this sense refers to the fact that smart expression templates (in contrast to standard expression templates) can optimize the expression evaluation depending on the type of the operands (see 8.3.1). Additionally, the smart expression templates, as implemented in the `pe`, are able to detect aliasing effects and adapt the expression evaluation accordingly (Section 8.3.2). Furthermore, smart expression templates allow an integration of performance optimized BLAS functionality in order to combine the usability and maintainability of the expression templates with the high performance of BLAS (see Section 8.3.3).
8.3.1 Expression Dependent Evaluation

The expression template formulation as shown in the previous chapter works fine for simple expressions, as for instance the addition of vectors, the multiplication between a matrix and a vector, or a matrix-matrix multiplication. However, for complex or composite expressions the standard formulation can lead to severe performance penalties. One example is the following matrix-vector multiplication:

```
Listing 8.18: Complex matrix-vector multiplication
Matrix<double> A;
Vector<double> a, b, c;
c = A * (a + b);
```

Using the expression template formulation as shown before, this right-hand side expression would result in an expression object that contains references to its two operands: The matrix \(A\) and the vector addition \(a + b\). However, for the evaluation of this expression, the entire result of the vector addition is needed for every single element of the \(c\) vector. Therefore each element of the vector addition is evaluated \(M\) times, where \(M\) is the number of rows of the matrix. This unfortunately decreases the performance of the expression template formulation considerably. Figure 8.3 shows the performance results of the composite matrix-vector multiplication for the expression in Listing 8.18. Obviously, the standard formulation of expression templates is slower than the classical approach that involves two temporary vectors (one for the vector addition, one for the matrix-vector multiplication), although during the expression template evaluation no memory allocations/deallocations or expensive copy operations take place. Also included in the figure are the performance results for the smart expression template implementation as explained in this section and the result for the combination of smart expression templates with a performance optimized BLAS implementation.

```
Figure 8.3: Performance comparison between the four implementations of a dense matrix-vector multiplication. The performance was measured on an Intel Core i7 949 CPU at 2.93 GHz (Bloomfield core) with 8 MByte of shared level three cache and using double precision elements for both the matrix and the vectors. All executables were compiled with the GNU G++ 4.4.1 compiler (branch revision 150839).
```

Apparently it is necessary to distinguish between a plain matrix-vector multiplication between a matrix and a vector and a complex matrix-vector multiplication between a matrix and a vector expression. Therefore the expression classes are updated by a mechanism to detect whether or not the operands are expressions or plain objects. Listing 8.19 shows part of the smart expression template implementation of the matrix-vector addition as implemented in the \(pe\):
The `DMatDVecMultExpr` class template represents the multiplication of a dense matrix and a dense vector (either plain matrices/vectors or matrix/vector expressions). This class is publicly derived from the `DenseVector` class template that works in the same way as the `Expr` base class of the standard expression template formulation and allows a restriction of operands to
dense vectors. Additionally, the DMatDVecMultExpr class derives privately from the Expression base class. The purpose of this base class is solely to mark the DMatDVecMultExpr class as a vector expression in comparison to a plain vector.

The major difference in comparison to standard expression templates is an additional set of nested type definitions that are required for a smart evaluation of expressions. The MRT and VRT defined in a private section of the class are merely shortcuts for the result types of the left-hand side and right-hand side operands defined by the nested ResultType type definition. The ResultType represents the resulting data type in case the expression object is evaluated. Note that these types are never expression types but always vector or matrix types that have to be supported by the MathTrait class template (see Appendix C). The result type of an expression is defined as the data type resulting from the according operation between the left-hand side operand of type RT1 and the right-hand side operand of type RT2. The nested type ElementType represents the element type of the expression, that is derived from the ResultType of the expression. MET and VET are shortcuts to the element type of the left-hand side and right-hand side result types.

The nested CompositeType type represents the data type that is used in case the expression is used in another expression (as for instance another addition, subtraction, etc.). This data type is either a reference to the expression itself in case it is not necessary to evaluate the expression in case of a composite expression, or it is an according concrete type (in most cases the ResultType) in case it is necessary or beneficial to immediately evaluate the expression within a composite expression. In case of the DMatDVecMultExpr class, the CompositeType is a reference to DMatDVecMultExpr since it is not necessary to evaluate a matrix-vector multiplication prior to any composite evaluation and is not beneficial in terms of performance. However, for instance in case of an addition between a dense and a sparse vector, the CompositeType is a concrete dense vector type, since it is a huge performance advantage to evaluate the expression immediately:

Listing 8.20: Composite type of the smart expression object for the dense vector-sparse vector addition

```cpp
template<typename VT1, typename VT2> // Type of the left-hand side dense vector
class DVecSVecAddExpr : public DenseVector<DVecSVecAddExpr<VT1, VT2>>, private Expression
{
  public:
    // ...

    typedef typename MathTrait<RT1, RT2>::AddType ResultType;

    typedef const ResultType CompositeType;

  // ...
};
```

In the DMatDVecMultExpr class, the CompositeType is for instance used to determine the member data type of the left-hand side operand (Lhs). In case an expression needs to be immediately
evaluated, \textit{Lhs} becomes a temporary that holds the result of the left-hand side expression. In case the expression does not have to be evaluated immediately, it corresponds to a reference to the left-hand side matrix operand.

The left-hand side dense matrix operand does not require any special treatment. In the matrix-vector multiplication every element of the matrix is used exactly once. Therefore the decision whether or not to evaluate the matrix expression is handled by the nested \texttt{CompositeType}. The right-hand side dense vector expression on the other hand requires a special treatment. Depending on whether or not the right-hand side vector is a plain vector or a vector expression, the right-hand side member \texttt{Rhs} has to be chosen differently: In case the right-hand side vector operand is a plain vector, it is enough to store a reference to the vector inside the matrix-vector multiplication object. However, if the right-hand side vector is an expression type (i.e., derived from the \texttt{Expression} base class) a temporary object is created to hold the result of the vector expression in order to guarantee an efficient evaluation of the matrix-vector multiplication.

The type definition of the member data type for the right-hand side dense vector operand \texttt{Rhs} demonstrates how the type dependent evaluation of expressions works. The \texttt{Rhs} type is defined as either the result type of the vector expression (\texttt{VRT}) or a reference to the right-hand side plain vector via the \texttt{SelectType} class template (see Appendix B). The selection between these two types is performed by the \texttt{IsExpression} type trait:

\begin{verbatim}
1 template< typename T >
2 struct IsExpressionHelper
3 {
4   enum { value = boost::is_base_of<Expression,T>::value &&
5           !boost::is_base_of<T,Expression>::value }; 
6   typedef typename SelectType<value,TrueType,FalseType>::Type Type;
7};
8
9 template< typename T >
10 struct IsExpression : public IsExpressionHelper<T>::Type
11 {
12 {
13   public:
14   enum { value = IsExpressionHelper<T>::value }; 
15   typedef typename IsExpressionHelper<T>::Type Type;
16};
\end{verbatim}

The \texttt{IsExpression} type trait class tests whether or not the given type \texttt{Type} is a \textit{re} expression template or not. In order to qualify as a valid expression template, the given type has to derive (publicly or privately) from the \texttt{Expression} base class. This relationship is tested via the Boost \texttt{is_base_of} type trait. In case the given type is a valid expression template, the \texttt{value} member enumeration is set to 1, the nested type definition \texttt{Type} is \texttt{TrueType}, and the class derives from \texttt{TrueType}. Otherwise \texttt{value} is set to 0, \texttt{Type} is \texttt{FalseType}, and the class derives from \texttt{FalseType}. 

---

\textbf{Listing 8.21:} Type trait class for expressions
For the evaluation of the right-hand side member type of the matrix-vector multiplication object, the `SelectType<IsExpression<VT>::value,const VRT, const VT&>` results either in `const VRT` in case the right-hand side vector `VT` is an expression or in `const VT&` in case the right-hand side vector is a plain vector.

An obvious difference in comparison to standard expression templates is the creation of temporary objects as members of the expression objects. At first sight this might be a potentially expensive endeavor since the expression objects are returned by value (i.e., by a copy operation) from the overloaded operators. However, also in this case, the NRV optimization removes any potential copy operation and directly constructs the object into a compiler generated, temporary value:

Listing 8.22: Compiler generated temporary objects for the matrix-vector multiplication

```cpp
Matrix<double> A;
Vector<double> a, b, c;

// Compiler generated temporary object for the expression object returned by
// the addition of vector a and b. No copy operation is required due to NRV!
const DVecDVecAddExpr<Vector<double>, Vector<double>> __tmp1__(a + b);

// Compiler generated temporary object for the expression object returned by
// the multiplication of a dense matrix with a dense vector expression. Due
// to NRV, no copy operation is required!
const DMatDVecMultExpr<Matrix<double>, Vector<double>> __tmp2__(A * __tmp1__);

// Assignment of the right-hand side expression to the left-hand side
// vector c
const DMatDVecMultExpr<Matrix<double>, Vector<double>> __tmp2__(A * __tmp1__);
```

### 8.3.2 Detection of Aliasing Effects

The standard expression template formulation has a second weakness that is directly correlated to the correctness of the calculations. Consider the multiplication of a dense matrix and a dense vector as illustrated in Listing 8.23:

Listing 8.23: Aliasing problem for a matrix-vector multiplication

```cpp
Matrix<double> A;
Vector<double> x;

// This statement causes a serious aliasing effect!!
x = A * x;
```

In the standard formulation, this calculation would most certainly result in a wrong output. During the evaluation of the right-hand side multiplication expression, the values of the left-hand side vector are directly updated without the creation of an intermediate, temporary result. Therefore the values of `x` are changed during the expression evaluation although the old values of `x` are still required. In the standard expression template formulation this problem can only be solved by an explicit generation of a temporary object, since it is not possible to detect such aliasing problems:
8.3 Smart Expression Templates

Listing 8.24: Explicit resolution of aliasing problems

```cpp
Matrix< double > A;
Vector< double > x;
Vector< double > tmp( A * x ); // Explicit creation of a temporary vector, 
// optimized by NRV!
x = tmp; // Update of vector x!
```

However, it is potentially very dangerous to allow the formulation of incorrect calculations. Therefore it is necessary to incorporate the detection of aliasing effects into the expression template evaluation. In the pe implementation, this is solved by an additional member function in every vector, matrix, and expression class, called 

Listing 8.25: Member function to detect aliasing effects

```cpp
template< typename MT // Type of the left-hand side dense matrix 
    , typename VT > // Type of the right-hand side dense vector
class DMatDVecMultExpr : public DenseVector< DMatDVecMultExpr<MT,VT> > 
    , private Expression
{
    // ...

    public:
    // ...

    template< typename T >
    inline bool isAliased( const T* alias ) const {
        return vec_.isAliased( alias );
    }

    private:
    Lhs mat_; // Left-hand side dense matrix of the multiplication expression
    Rhs vec_; // Right-hand side dense vector of the multiplication expression
    // ...
};
```

The isAliased function of the DMatDVecMultExpr class returns whether any operand of the expression is aliased with the given address alias. In case of the matrix-vector multiplication, aliasing effects can only occur in case the expression is assigned to a vector that is part of the multiplication expression. Therefore the function relays the query to the vector (expression). Listing 8.26 illustrates the implementation of the isAliased function in the Vector class:

Listing 8.26: Implementation of the isAliased function in the Vector class

```cpp
template< typename Type > // Data type of the vector
template< typename Other > // Data type of the foreign expression
inline bool Vector<Type>::isAliased( const Other* alias ) const
{
    return static_cast<const void*>(this) == static_cast<const void*>(alias);
}
```

---

4Even a perfect documentation of this issue could not prevent an accidental use of this formulation, even by experienced programmers.
The initial call to the `isAliased` function is contained in the assignment operator of the vector and matrix classes. For instance, consider the according assignment operator of the `Vector` class:

```cpp
template< typename VT > // Type of the right-hand side dense vector
inline Vector & Vector::operator=( const DenseVector<VT>& rhs )
{
    using pe::assign;

    if( IsExpression<VT>::value && (~rhs).isAliased( this ) ) {
        Vector tmp( rhs );
        swap( tmp );
    } else {
        resize( (~rhs).size(), false );
        assign( *this, (~rhs );
    }

    return *this;
}
```

This smart expression assignment operator represents the direct replacement of the standard expression template assignment operator demonstrated in Listing 8.17. It exclusively accepts dense vectors and dense vector expressions, i.e., classes deriving from the `DenseVector` class. The assignment operation distinguishes between a default assignment and an aliased assignment. In case the given right-hand side dense vector is an expression, an alias test is performed with the address of the left-hand side vector. In case this test returns `true`, it is necessary to create a temporary object and afterwards update the values of the vector via the “temporary swap” idiom [53, 52]. If no aliasing is detected, the vector is resized accordingly and the right-hand side is directly assigned without any temporary (the `assign` functionality is discussed in the next section).

Note that self-assignment between two vectors (`x = x`) is handled by the copy assignment operator. Therefore the `IsExpression<VT>::value` expression should be considered a performance optimization since it is not possible to encounter any aliasing effects in case the right-hand side dense vector is not an expression. Also note that this implementation of the detection of aliasing effects is based on the assumption that the addresses of the vectors and matrices are uniquely identifying a specific vector or matrix. In the current implementation this is particularly true since both base classes of the `Vector` class are empty base classes and therefore do not occupy any memory on their own in case the compiler supports the “Empty Base Class Optimization” [105]. However, this property is assured by several compile time constraints (see Chapter 3).

### 8.3.3 Integration of BLAS Functionality

In order to achieve a maximum level of performance, the smart expression template formulation has to allow the expression dependent customization of the assignment. This would for instance enable the integration of BLAS functionality to specific operations. For the purpose of this section, let us consider the multiplication between two dense matrices since
this particular operation profits the most from the use of a performance optimized BLAS library. Listing 8.28 shows the assignment operator of the \texttt{Matrix} class that is very similar to the assignment operator of the \texttt{Vector} class (Listing 8.27):

\begin{lstlisting}[language=C++]
// Type of the matrix elements
template < typename Type >
// Type of the right-hand side dense matrix
template < typename MT >
inline Matrix<Type>& Matrix<Type>::operator=( const DenseMatrix<MT>& rhs )
{
  using pe::assign;
  if( !IsExpression<MT>::value && (!rhs).isAliased( this ) ) {
    MatrixMxN tmp( rhs );
    swap( tmp );
  }
  else {
    resize( (!rhs).rows(), (!rhs).columns() );
    assign( *this, !rhs );
  }
  return *this;
}
\end{lstlisting}

Depending on the result of the \texttt{if} statement that handles the detection of aliasing effects, either a temporary object is created and afterwards “assigned” via the “temporary swap” idiom \cite{53, 52}, or the matrix is resized and afterwards assigned the right-hand side dense matrix (expression). The \texttt{assign} function is the key to the expression dependent optimization of the assignment. \texttt{assign} is a free function that expects two arguments: the left-hand side target matrix and the right-hand side dense matrix (expression) to be assigned to the left-hand side matrix. The default implementation of \texttt{assign} is illustrated in Listing 8.29:

\begin{lstlisting}[language=C++]
// Type of the left-hand side dense matrix
// Type of the right-hand side dense matrix
inline void assign( DenseMatrix<MT1>& lhs, const DenseMatrix<MT2>& rhs )
{
  // Internal check of the number of rows and columns
  pe_INTERNAL_ASSERT( (~lhs).rows() == (~rhs).rows() );
  pe_INTERNAL_ASSERT( (~lhs).columns() == (~rhs).columns() );
  (~lhs).assign( rhs );
}
\end{lstlisting}

The \texttt{assign} function expects the target matrix to have the appropriate size for the assignment. Therefore the \texttt{resize} operation is performed before the assignment in the assignment operator. Afterwards, the default assignment strategy for a dense matrix is selected, which is depending on the implementation of the left-hand side dense matrix and is therefore a member function of the left-hand side matrix. Thus the strategy for the assignment is a ping-pong approach: The assignment operator calls a free function, that allows an expression dependent customization (for instance via overloading or specialization) and in the default case the default assignment is selected, which is again a member function of the \texttt{Matrix} class:
The default assignment applies an element-wise strategy to assign the new values. However, this strategy is a bad choice for the evaluation of a matrix-matrix multiplication. Therefore it will be necessary to customize this strategy explicitly. Note that even this function allows a certain level of optimization by applying explicit loop unrolling.

In order to optimize the assignment of a matrix-matrix multiplication, the expression class has to provide an according assign function. The following two code excerpts explain in detail how this optimization is implemented in case of the DMatDMatMultExpr class that represents the multiplication between two dense matrices:

```
Listing 8.31: Smart expression object for the matrix-matrix multiplication

template< typename MT1 , typename MT2 > // Type of the left-hand side dense matrix
class DMatDMatMultExpr : public DenseMatrix< DMatDMatMultExpr<MT1,MT2> >
{
private:
    typedef typename MT1::ResultType RT1;
    typedef typename MT2::ResultType RT2;
    typedef typename MT1::CompositeType CT1;
    typedef typename MT2::CompositeType CT2;

public:
    typedef MathTrait<RT1,RT2>::MultType ResultType;
    typedef const DMatDMatMultExpr& CompositeType;
}
```
8.3 Smart Expression Templates

Just as any other expression class of the \texttt{pe}, the \texttt{DMatDMatMultExpr} class defines the nested types \texttt{ResultType}, \texttt{CompositeType}, and \texttt{ElementType}. Additionally, it defines the two member data types for the left-hand and right-hand side operands \texttt{Lhs} and \texttt{Rhs} depending on the given types. This process is similar to the type evaluation of the \texttt{DMatDVecMultExpr} class, since for a matrix-matrix multiplication the creation of temporary values in case either of the two operands is an expression also increases the performance. As a dense matrix, the \texttt{DMatDMatMultExpr} class also provides a function call operator for accessing the matrix elements, a \texttt{rows} and a \texttt{columns} function, and the \texttt{isAliased} function to detect aliasing effects.

Listing 8.32 shows the customized optimization of the \texttt{assign} function. For the purpose of providing a customized \texttt{assign}, the \texttt{DMatDMatMultExpr} class defines four nested friend functions. At the point of instantiation of the class, these function are injected into the surrounding namespace and are considered by the compiler during the selection of matching candidates for \texttt{assign} function calls (this approach is also known as the Barton-Nackman trick [104]):

**Listing 8.32: Assign functions for the evaluation of the matrix-matrix multiplication**

```cpp
// Specialized assign function injected into the surrounding namespace
template< typename MT > // Type of the target dense matrix
friend inline void assign( DenseMatrix<MT>& lhs,
                           const DMatDMatMultExpr& rhs )
{
    // Internal check of the number of rows and columns
    pe_INTERNAL_ASSERT( (~lhs).rows() == rhs.rows() );
    pe_INTERNAL_ASSERT( (~lhs).columns() == rhs.columns() );
```
The first of the four assign functions is a template for an expression dependent optimization of the assignment of a specific expression. The function is an overload to the default assign function. In contrast to the default function, the second argument is not a template argument, but an instance of a DMatDMatMultExpr. In case a matrix-matrix multiplication is assigned to another dense matrix, this function is selected to handle the assignment to the right-hand side matrix instead of the default function. In case of the DMatDMatMultExpr, this function acts as a dispatcher. It merely calls one of the three provided assign member functions of the DMatDMatMultExpr class. These functions are overloaded such that, in case
the two operands of the multiplication match the arguments expected from one of the two BLAS matrix-matrix multiplication functions `cblas_sgemm` or `cblas_dgemm`, the according member function is called and the BLAS functions are executed. In case the BLAS functions cannot be used, the default performance optimized `assign` function is called. In contrast to the default `assign`, which employs an element-wise strategy for the assignment, this function implements a cache optimized calculation of the matrix values.

Note that the same strategy can also be used for addition assignments and subtraction assignments:

```
Listing 8.33: Addition and subtraction assignment of a matrix-matrix multiplication
1 Matrix<double> A, B, C, D;
2
3 C += A * B;
4 D -= A * B;
```

The according functions are called `addAssign` and `subAssign`, respectively, and follow the same implementation approach as the `assign` function family.

### 8.4 Performance Results

In this section, the performance of the smart expression templates, as implemented in the math library of the `pe`, are examined in more detail. The focus of this section is the performance of composite expressions of dense vectors and matrices that can make use of BLAS optimized functions. For a complete overview of possible operations between dense and sparse vectors and matrices refer to Appendix D. The `pe` performance results are compared to the performance of the uBLAS library of the Boost framework. This library also offers an expression template based implementation of several dense and sparse vector and matrix data structures and can be considered the current standard in C++ math libraries. However, so far Boost uBLAS is primarily focused on usability and maintainability, and on providing a rich set of features for all kinds of mathematical operations. In contradiction to the name of the library, it does not yet support any high performance BLAS operations\(^5\).

Listing 8.34 shows a code excerpt for the performance benchmark of the complex expression \((A + B) \cdot (C - D)\) using the uBLAS library. Prior to the performance measurement, all matrices are accordingly sized and initialized. Afterwards, a certain number of repetitions are executed, each time measuring the passed wall clock time with the timing functionality of the `pe`. Each operation is executed several times to guarantee runtimes of several seconds per measurement. Additionally, special care is taken to prevent the compiler to optimize the calculations away.

```
Listing 8.34: Benchmark for the matrix-matrix multiplication using Boost uBLAS
1 // Definition of the dense double-precision matrices
2 boost::numeric::ublas::matrix<double> A(N, N), B(N, N), C(N, N),
3     D(N, N), E(N, N);
```

\(^5\)Although this might change with any new release of the Boost library.
The Math Library of the pe

```cpp
// Definition of a timer for wall clock time measurements
pe::timing::WcTimer timer;

// Initialization of the matrices
// ...

// Initial calculation of the matrix E
noalias(E) = prod(A + B, C - D);

for (std::size_t rep = 0; rep < reps; ++rep) {
  timer.start();
  for (std::size_t step = 0; step < steps; ++step) {
    noalias(E) = prod(A + B, C - D);
  }
  timer.end();
}
```

The test machine for all performance tests was an Intel Core i7 940 CPU at 2.93 GHz (Bloomfield core) with 8 MByte of shared level three cache. All executables are compiled with the GNU G++ 4.4.1 compiler (branch revision 150839). The data types of the vector and matrix elements is double for all performance tests. I used Boost 1.39 for all performance comparisons and the used BLAS library is ATLAS in the version 3.9.17.

### 8.4.1 Complex Expression \( A \cdot (a + b) \)

**Listing 8.35:** Implementation of the expression \( A \cdot (a + b) \) in the pe

```cpp
// Definition of a double precision dense square matrices
// and three double precision dense vectors
pe::MatN A( N, N );
pe::VecN a( N ), b( N ), c( N );

// Initialization
// ...

// Evaluation of the expression
c = A * ( a + b );
```

**Figure 8.4:** Performance of the pe and Boost for the complex expression \( A \cdot (a + b) \).

The complex expression \( A \cdot (a + b) \), as already used in the motivation part, benefits from the smart expression template formulation due to the intermediate evaluation of the vector expression \( a + b \) into a temporary object. In comparison to the uBLAS library, the smart expression template formulation of the pe in combination with the BLAS cblas_dgemv function for double precision matrix-vector multiplications performs better by a factor of 1.41 for small matrices and vectors \((N = 50)\) and by a factor of 1.47 for large matrices and vectors.
(N = 1000). The major performance improvement in this case results from the creation of the temporary vector. The additional application of a performance optimized BLAS library only slightly improves the performance.

### 8.4.2 Complex Expression \( A \cdot (a + b + c) \)

Listing 8.36: Implementation of the expression \( A \cdot (a + b + c) \) in the \( \text{pe} \)

```cpp
1 // Definition of a double precision dense square matrices
2 // and four double precision dense vectors
3 pe::MatN A( N, N );
4 pe::VecN a( N ), b( N ), c( N ), d( N );
5 // Initialization
6 // ...
7 // Evaluation of the expression
8 d = A * ( a + b + c );
```

Figure 8.5: Performance of the \( \text{pe} \) and Boost for the complex expression \( A \cdot (a + b + c) \).

For the multiplication of a matrix with a vector expression representing the addition of three vectors, the performance gain of the smart expression template formulation is even more prominent. In comparison to the uBLAS library, the \( \text{pe} \) performs better by a factor of 1.84 for small operands (\( N = 50 \)) and by a factor of 1.88 for large operands (\( N = 1000 \)). Again, the application of a performance optimized BLAS library only slightly improves the performance in comparison to the performance gain from the smart expression template formulation.

### 8.4.3 Complex Expression \((A \cdot B) \cdot (a + b)\)

Listing 8.37: Implementation of the expression \((A \cdot B) \cdot (a + b)\) in the \( \text{pe} \)

```cpp
1 // Definition of two double precision dense square matrices
2 // and three double precision dense vectors
3 pe::MatN A( N, N ), B( N, N );
4 pe::VecN a( N ), b( N ), c( N );
5 // Initialization
6 // ...
7 // Evaluation of the expression
8 c = ( A * B ) * ( a + b );
```

115
The third complex expression represents a matrix-vector multiplication between a dense matrix and a dense vector, where the dense matrix is the result of a matrix-matrix multiplication and the dense vector results from the addition of two dense vectors. Although even without usage of the optimized BLAS functions the pe is faster than the BOOST uBLAS library by a factor of 1.15 for small operands ($N = 50$) and by a factor of 1.11 for larger operands ($N = 400$), the main performance improvement results from the ATLAS library. Note that although each element of the matrix resulting from the matrix-matrix multiplication is required only once during the subsequent matrix-vector multiplication, the creation of a temporary matrix for the result in order to be able to apply the BLAS functionality is very beneficial for the overall performance of the expression. Also note that the performance gain is higher the larger the operands of the expressions are.

### 8.4.4 Complex Expression $(A \cdot B) + C$

**Listing 8.38:** Implementation of the expression $(A \cdot B) + C$ in the pe

```cpp
// Definition of four double precision dense square matrices
pe::MatN A( N, N ), B( N, N ), C( N, N ), D( N, N);

// Initialization

// Evaluation of the expression
D = ( A * B ) + C;
```

The fourth complex expression $(A \cdot B) + C$ also greatly profits from the performance optimized BLAS functions. Whereas without the BLAS functions, the pe and uBLAS exhibit a similar performance, the application of ATLAS greatly improves the performance, especially for large operands. In this case, the performance gain is also only possible by creating a temporary matrix for the result of the matrix-matrix multiplication in order to be able to apply the BLAS functionality.
8.4.5 Complex Expression \((A + B) \cdot (C - D)\)

Listing 8.39: Implementation of the expression \((A + B) \cdot (C - D)\) in the \textit{pe} framework.

```cpp
1 // Definition of five double precision dense square matrices
2 pe::MatN A(N, N), B(N, N), C(N, N), D(N, N), E(N, N);
3
4 // Initialization
5 // ...
6
7 // Evaluation of the expression
8 E = (A + B) * (C - D);
```

Figure 8.8: Performance of the \textit{pe} and Boost for the complex expression \((A + B) \cdot (C - D)\).

The complex expression \((A + B) \cdot (C - D)\) impressively demonstrates the advantage of smart expression templates for the evaluation of composite expressions. The matrix addition \(A + B\) is multiplied with the result of the matrix subtraction \(C - D\). Since for the matrix-matrix multiplication all elements from both matrices are required several times, both operands are evaluated into temporary matrices. This approach alone already results in a performance difference of a factor of 1.48 for small matrices \((N = 50)\) and a factor of 14.36 for large matrices \((N = 500)\) in favor of the \textit{pe} in comparison to Boost uBLAS. In case ATLAS is used, another factor of 3.52 for small matrices and 4.78 for large matrices is gained. In total, this results in a performance difference of a factor of 68.62 between the \textit{pe} and uBLAS.

8.5 Conclusion

In this chapter I have introduced the smart expression template programming technique as implemented in the math library of the \textit{pe} framework. In comparison to standard expression templates, as they are used to circumvent the performance degrading characteristics of the C++ technique of operator overloading, they offer an expression dependent evaluation and optimization. Possible optimizations include the integration of BLAS functionality and the automatic detection of aliasing effects. I presented an in depth explanation of the implementation details of smart expression templates and gave an overview of the extensions necessary to update standard expression templates. In order to demonstrate the achieved performance gain of this approach, I compared the performance of several complex expressions between the \textit{pe} math library and the Boost uBLAS library. In comparison to uBLAS, which offers similar features as the \textit{pe} math library and is also based on expression templates, smart expression templates in combination with BLAS may result in tremendous performance improvements depending on the given expression.
8 The Math Library of the pe
Part III

Massively Parallel Rigid Body Dynamics
In this part I will focus on the unique feature of the \textit{pe} framework to perform massively parallel rigid body dynamics simulations. The primary goal of these massively parallel simulations is to be able to simulate an arbitrary number of rigid bodies on an arbitrary number of processor cores. In order to achieve this objective, it is strictly necessary to distribute the rigid bodies among the involved MPI processes due to basic computational and memory restrictions. Thus each process is assigned part of the global simulation domain along with all rigid bodies contained in this subdomain.

In comparison to shared-memory simulations (either non-parallel or thread-parallel), the distributed memory parallelization of rigid body dynamics causes several additional problems that need to be solved. Chapter 9 deals with the general problems encountered during the parallelization of rigid body simulations. It will explain the basic idioms of the domain partitioning as applied in the \textit{pe} framework. Additionally, it will cover the problems caused by the fact that rigid bodies are volumetric objects and can therefore not be as easily be related to a specific process as for instance point masses. A third focus of this chapter is the MPI communication, which has to deal with heterogeneous geometry types, heterogeneous data types as well as heterogeneous MPI messages.

Chapter 10 will introduce the algorithm for massively parallel rigid body dynamics, the Parallel Fast Frictional Dynamics Algorithm (PFFD). In contrast to other collision response algorithms, which are extremely difficult to parallelize due to the global problem formulation or an iteration scheme, the PFFD uses a non-iterative, local collision update scheme that enables an efficient parallelization. The chapter will explain the algorithm in detail and will compare it to other collision solvers. Additionally, I will present performance and scaling results on up to 131,072 processor cores on the world’s currently largest supercomputer, the Jugene system at the Jülich Supercomputing Centre \cite{59}, and validation results for granular material simulations.

Since the PFFD algorithm for the first time enables massively parallel rigid body dynamics simulations, it offers completely new possibilities for granular media simulations. As the authors in \cite{91} state, “it is [their] firm believe that Rigid-Body Dynamics represents an interesting complement to Molecular Dynamics, which could provide new insights into many systems that are notoriously hard to model using Molecular Dynamics methods.” In Chapter 11 I will demonstrate the applicability of our algorithm to granular material simulations, thus enabling simulations of billions of arbitrarily shaped granular particles with frictional contact mechanics.
This chapter deals with the general problems resulting from the distributed memory parallelization of rigid body simulations. Section 9.1 will give an overview of the conceptual design of domain partitioning in the \textit{pe}. Additionally, it will give some examples how parallel rigid body simulations are set up. Section 9.2 will cover the problems arising from the fact that rigid bodies – in contrast to point masses – occupy a volume and can therefore be physically present on several processes. It also contains the solution approaches chosen in the \textit{pe} framework. In Section 9.3 I will explain the problems involved in the MPI communication of a parallel rigid body simulation. I will give a detailed overview of the MPI communication concept as realized in the \textit{pe} framework.

9.1 Domain Partitioning

The primary goal of the large-scale rigid body simulations in the \textit{pe} is the treatment of an arbitrary number of interacting rigid bodies on an arbitrary number of processor cores. This parallelization enables simulation scenarios that could not be performed on a single processor due to basic computational and memory restrictions. Figure 9.1 and 9.2 show two examples of possible large-scale rigid body simulations\(^1\).

In order to achieve the objective of a massively parallel simulation, one of the most basic principles of the \textit{pe} MPI parallelization is that there is no global data. Each process is responsible for a subdomain of the complete simulation domain. Depending on the domain partitioning, it only knows about the rigid bodies contained in its local domain and the

\(^1\)Please note that both examples only use a small number of processor cores. The limitation of these simulations is the subsequent ray-tracing visualization.
Figure 9.1: Simulation of 500,000 spheres and boxes falling into a well built from 3,000 fixed boxes. The patches indicate the domains of the 91 involved MPI processes. Due to the hexagonal setup of the domains, each MPI process has a maximum of six neighboring processes.

Figure 9.2: Simulation of a granular medium consisting of 108,570 spherical particles. Using 64 MPI processes, the runtime of the simulation is approximately 3 hours for 379,300 time steps.

boundaries to the neighboring processes. It is not necessary to explicitly define the domain of a process, since the local domain, i.e., the domain of a specific process, is implicitly defined by the boundaries to the neighboring processes. However, this philosophy makes it necessary to be able to clearly partition the simulation domain into a local and a remote part with each process boundary. It is then for instance possible to check whether the center of mass of a specific rigid body is contained in the local process by querying each process boundary whether it contains the coordinate. Only if all boundaries do not contain the point, it is contained in the local domain.

Figure 9.3 shows an example of the static partitioning of the 64 processes used for the granular medium simulation. Due to the size of the granular particles in this scenario, it is only necessary to communicate with the directly neighboring processes in order to exchange rigid bodies. Therefore each process within the granular medium outlet structure has a maximum of 8 neighboring MPI processes. For instance, as illustrated in Figure 9.4, process 23 has process 22 and 24 as direct neighbors in the x-direction, process 14 and 32 as direct neighbors in the z-direction, and processes 13, 15, 31, and 33 as diagonal neighbors. Each of the processes has to communicate with each of its neighbors about rigid bodies crossing the process boundary in order to detect and handle all collisions between the rigid bodies.

The communication to handle the rigid bodies in a parallel simulation is automatically handled by the $p_e$. However, the MPI communication has to be explicitly established. The setup of a connection between the local and a remote MPI process is done via the connect functions:
9.1 Domain Partitioning

Figure 9.3: Partitioning of the MPI processes used in the granular medium simulation. The patches indicate the domain of each of the 64 processes.

Figure 9.4: Detailed illustration of the domain partitioning of the MPI processes used in the granular medium simulation.

Listing 9.1: The connect() function family

```cpp
1 connect( int rank, real a, real b, real c, real d, real dx );
2 connect( int rank, const Vec3& normal , real d, real dx );
3 connect( int rank, real a, real b, real c, real x, real y, real z, real dx );
4 connect( int rank, const Vec3& normal , const Vec3& gpos , real dx );
5 connect( int rank, const Geometry& geometry );
```

Each of these functions establishes a dedicated communication connection between the local and the specified remote process. It is important to note that there is no grid involved in the simulation of rigid body dynamics. Therefore, it is not necessary to align the process boundaries to any kind of grid. Connections to remote processes are created by specifying the geometry of the remote process from the point of view of the local process. The first
argument for all five functions is the rank of the remote process to be connected to the local process. The following arguments are used to define the remote process geometry. In the simplest case, from the point of view of the local process, the remote process occupies a complete half-space separated by a simple plane-shaped boundary from the half-space of the local process (see Figure 9.5). The first four connect functions define such a plane-shaped boundary by specifying a normal vector ((a,b,c) or normal) and either the displacement of the plane from the origin of the global world frame (d) or a surface point on the surface of the plane ((x,y,z) or gpos).

Figure 9.5: Plane-shaped process boundary between the local and a remote process.

Figure 9.6 shows a two dimensional illustration for a possible configuration of a plane-shaped process boundary between process 0 and process 1. The corresponding Listing 9.2 shows the setup of these two processes by use of one of the connect functions.

Figure 9.6: Plane-shaped process boundary between process 0 and process 1.

Listing 9.2: Establishing a MPI connection between two processes

```c
// Connecting two MPI processes
pe::EXCLUSIVE_SECTION( 0 ) {
    // Connecting the local process 0 with the remote process 1
    pe::connect( 1, Vec3( 1.0, 1.0, 0.0 ), -std::sqrt(2.0) );
}
```

126
Due to the “Single Program Multiple Data” (SPMD) paradigm of MPI, each process runs exactly the same program. Therefore it is only possible to distinguish between the different processes based on the rank of the processes. In this example this is done by using the `pe_EXCLUSIVE_SECTION` scope: Process 0 and process 1 both start an exclusive section to connect to each other individually. The `pe_EXCLUSIVE_SECTION` introduces a section that is only executed by exactly one process. For instance, the first of the two sections is executed exclusively by process 0. All other processes skip this section entirely. Within the exclusive section, process 0 is connecting to process 1 by calling a `connect` function and specifying the normal and displacement of the plane-shaped boundary. The same is true for the second exclusive section that is exclusively executed by process 1. Inside, process 1 connects to process 0 by defining exactly the same boundary but from the point of view of process 1 (i.e., the normal vector and the displacement are reversed)\(^2\).

The fifth `connect` function allows to define arbitrarily shaped processes and process boundaries. This function can, for instance, also be used to define the boundary between process 0 and process 1 as illustrated in Figure 9.6:

```cpp
// Connecting two MPI processes
pe_EXCLUSIVE_SECTION( 0 ) {
    // Connecting the local process 0 with the remote process 1
    pe::connect( 1, pe::HalfSpace( Vec3( 1.0, 1.0, 0.0 ), -std::sqrt(2.0) ) );
}

pe_EXCLUSIVE_SECTION( 1 ) {
    // Connecting the local process 1 with the remote process 0
    pe::connect( 0, pe::HalfSpace( Vec3( -1.0, -1.0, 0.0 ), std::sqrt(2.0) ) );
}
```

Instead of directly specifying the normal vector and the displacement of the separating plane-shaped boundary, this function expects an object representing the geometry of the remote process. The `HalfSpace` class represents such a process geometry. For this example it becomes obvious that the first four `connect` functions are just simplifications for the case that two processes are separated by a single plane. However, this `connect` function can also be used to define arbitrarily complex process boundaries and geometries. Currently, the `pe` only supports combinations of plane shaped boundaries via the `merge` and `intersect` functions\(^3\).

Figure 9.7 shows an example of a possible configuration of two processes:

\(^2\)The `pe_EXCLUSIVE_SECTION` scope is one example of a `pe` smart scope. The concept of smart scopes is explained in detail in Chapter 4. For more information about the internal details of the `pe_EXCLUSIVE_SECTION` refer to Section 4.2.

\(^3\)Note however, that arbitrarily formed geometries/boundaries can easily be incorporated into the framework. These geometries solely have to be able to answer whether or not a coordinate and/or rigid body is contained in the remote process. For this purpose, geometries with an infinite expansion, as for instance a plane, are most suited.
Figure 9.7: Combination of two plane-shaped boundaries between process 0 and process 1.

Listing 9.4 illustrates the setup of the two MPI processes by use an explicit merging/intersection of two half-spaces and the fifth `connect` function:

```cpp
1 // Connecting process 0
2 pe_EXCLUSIVE_SECTION( 0 ) {
3     // Connecting the local process 0 with the remote process 1
4     pe::connect( 1, pe::intersect( pe::HalfSpace( 1.0, 0.0, 0.0, 2.0 ),
5                     pe::HalfSpace( 0.0, 1.0, 0.0, 1.0 ) ) );
6 }
7
8 // Connecting process 1
9 pe_EXCLUSIVE_SECTION( 1 ) {
10    // Connecting the local process 1 with the remote process 0
11    pe::connect( 0, pe::merge( pe::HalfSpace( -1.0, 0.0, 0.0, -2.0 ),
12                        pe::HalfSpace( 0.0, -1.0, 0.0, -1.0 ) ) );
13 }
```

From the point of view of process 0, the process boundary to the neighboring process 1 and the space occupied by process 1 are defined by an intersection of two half-spaces. The first half-space is defined by the normal $(1, 0, 0)$ and a displacement of 2, whereas the second half-space is specified by the normal $(0, 1, 0)$ and a displacement of 1. Process 0 intersects these two half-spaces via the `intersect` function, whereas process 1 merges the two half-spaces via the `merge` function. Note again the fact that process 1 defines its half-spaces contrariwise: both the normals and the displacements are reversed.

For a correct parallel setup it is also necessary to explicitly connect adjacent remote processes that are no direct neighbors of the local process. Figure 9.8 demonstrates the setup of a 2-dimensional (for reasons of simplicity) scenario with 4 MPI processes. Process 0 is considered to be the local process. Although process 2 is only diagonally connected to process 0 and has no common face/edge to this process, it is still necessary to connect the two processes (by a combination of two half-spaces) since rigid bodies may move directly from process 0 to process 2 (and vice versa).

Depending on the size of the rigid bodies in comparison to the expansion of the process domains it might also be necessary to extend the neighborhood of the processes and to ad-


9.2 Parallel Simulation of Volumetric Bodies

In discrete element methods as well as rigid body dynamics we are simulating rigid, completely undeformable, volumetric objects of arbitrary shape. However, although the full resolution of the shape and volume of an object is one of the major strengths of these methods, it causes some problems in comparison to particle-based simulation approaches.
The first problem results from the fact that rigid bodies cannot be uniquely related to exactly one process due to their expansion. In contrast to for instance molecular dynamics simulations [48], where the particles are considered to be point masses that do not occupy any space and can therefore be uniquely related to exactly one process based on their current position⁴, rigid bodies may be physically present on an arbitrary number of processes at the same time as illustrated in Figure 9.10.

There are only two possible solutions how to handle the problem of rigid bodies overlapping a process boundary: Either each process updates all rigid bodies that are (partially) contained in its part of the simulation domain based on their current state, or exactly one process is responsible for each rigid body, calculates all necessary updates, and notifies all other processes about the new position, orientation, and velocities. Although the first solution seems to be the simpler and more efficient one, it has several intrinsic problems that are either very hard to solve or that would degrade the performance of the algorithm.

The first problem involves general numerical problems such as roundoff errors. In case each process updates a rigid body, the results on different processes might differ due to small but important deviations of the calculation results. For instance, in case the result of the force synchronization for a specific rigid body differs on several processes due to roundoff errors, the resulting velocities, positions, and orientations of the body will differ. However, this might result in a different evaluation of the collision response process. For instance, one process might find a collision and resolve accordingly, whereas due to a slightly different position resulting from roundoff errors another process will not detect the collision and will therefore not treat it. The outcome will be a completely different behavior of the rigid body on different processes. This problem can only be addressed by performing exactly the same calculations on each involved process. However, this task is immensely complicated on heterogeneous architectures such as for instance the Roadrunner supercomputer at Los Alamos National Laboratory (LANL) [61]. On such machines, it is very hard, if not impossible, to guarantee the same numerical results of the computations on different processor cores.

Problem number two involves the collision response calculations of the PFFD algorithm itself. As it will be explained in chapter 10, the PFFD algorithm determines a feasible post-collision velocity for rigid bodies involved in collisions with other rigid bodies. However, the solution of this calculation is not unique but only one of many possible solutions. Therefore, also in case of the PFFD algorithm, it has to be assured that exactly the same calculations based on exactly the same motion constraints are performed on all processes, which calculate the collision response for a particular rigid body, in order to come to the same result. As already discussed, this cannot be assured in general, especially for heterogeneous architectures.

Obviously, on closer inspection, the first option to leave the task to update a rigid body to each process the body is (partially) contained in, easily results in non-recoverable simulation errors. Therefore, in case of the pe option two, to manage each rigid body exclusively by exactly one process, was chosen as a basic parallelization idiom. This choice makes it nec-

⁴This assumption is only valid in case the domains of the MPI processes do not overlap. However, this can be safely considered to be the usual rule.
9.2 Parallel Simulation of Volumetric Bodies

It is necessary to create a rule in order to be able to attach each rigid body to exactly one process that takes responsibility for the body. For this purpose, the pe uses the center of mass of each rigid body: In case the center of mass is contained in the domain of the local process, the body is considered to reside in this process, otherwise it is a remote body. The choice of the center of mass (or any other point within the rigid body, as for instance the geometric center, which might differ from the center of mass in case of heterogeneous densities) makes it possible to uniquely relate each rigid body to exactly one MPI process.

Note the expression "otherwise it is a remote body". In the pe framework, every MPI process knows only about his own local domain and about the process boundaries to his neighbors. The global partitioning of the simulation domain is therefore unknown to the processes and it is in fact never stored. From the point of view of a particular MPI process, a rigid body can therefore only be local (i.e., contained in the local domain) or remote (not contained in the local domain). It is not possible (and also not necessary) to relate a remote rigid body to a specific remote process, since the domain of remote processes is unknown. Figure 9.10 illustrates the setup of two MPI processes from the point of view of both involved processes.

![Figure 9.10: Simulation setup for two MPI processes. Both processes only know about the boundary to the neighboring process, but nothing about the domains these remote processes span. Rigid bodies are exclusively managed by the process their reference point (in our case the center of mass) belongs to. In case they are partially contained in the domain of a remote process, they have to be synchronized with the other process, where they are treated as remote bodies.](image)

Another problem that is only encountered in a parallel rigid body simulation is that every rigid body in the simulation must be uniquely identifiable. In case a rigid body is overlapping a process boundary (i.e., is visible for several MPI processes), all involved processes should logically work with the same rigid body, although physically, due to the distributed memory parallelization, they work with copies of the same body.

One way to identify a rigid body is to create a globally unique ID for every single rigid body. The first idea for such an ID is a counter, shared among all processes, that is increased every time a rigid body is created. However, it is virtually impossible to efficiently create an unique ID in an environment with an arbitrary number of MPI processes and several million rigid bodies. This would result in an all-to-all communication for every single rigid body.
body that is created, which would be devastating for the performance of the setup process on massively parallel machines.

**Figure 9.11:** Example of a 64-bit unique ID created as a combination of the MPI rank and a local body counter. In the *pe* implementation, the number of bits used for the MPI rank is chosen dynamically according to the number of used MPI processes. The remaining bits can be used for the local counter.

In the *pe* framework, this ID is created locally as a combination of the rank of the local MPI process and the number of locally created rigid bodies. (for an example see Figure 9.11). This approach guarantees that every rigid body is assigned a unique ID within the whole rigid body simulation. The downside of this approach is that every rigid body may only be created by exactly one process, since there is no communication between processes during the setup of a rigid body. Otherwise the same rigid body would be assigned different IDs on all processes it is created in and could not be identified as the same body. This design concept is enforced in the *pe* by all setup functions: In case the newly created rigid body’s center of mass is not contained in the local domain, an exception is thrown.

Listing 9.5 gives an example of the creation of a single sphere primitive in a MPI-parallel simulation:

```cpp
// Defining the global position of the center of mass of the sphere primitive
const Vec3 gpos( 1.2, -2.4, 3.5 );

// Acquiring a handle to the local simulation world/domain
World world = theWorld();

// Creating the iron sphere primitive
// Assuming that the process setup was performed properly, and since all processes are executing the code, the if-condition will only return true in case of exactly one process. The attempt to create the sphere on a process, which does not contain the specified center of mass, will result in an exception being thrown.
if( world->containsPoint( gpos ) )
   createSphere( 1, gpos, 2.0, iron );
```

After the local setup process, the rigid bodies are only known by the process of their creation. This is also true for bodies initially overlapping process boundaries. In order

---

5Note that communication during the setup of a rigid body, specifically between the processes the rigid body is created in, could very easily result in a deadlock situation. Therefore this is also not considered a valid option.
to make all rigid bodies visible on all processes they are contained in, the local simulation
domains have to be synchronized after the setup of all rigid bodies is complete. Note that this
synchronization is also necessary if rigid bodies are locally modified (for instance translated
or rotated).

Listing 9.6: Synchronization of the simulation domain

```c
// Synchronizing the simulation domain among the MPI processes.
// Afterwards all processes know about all rigid bodies that are
// (partially) contained in their domain.
world->synchronize();
```

In contrast to the setup of mobile rigid bodies, it is not necessary with large immobile
rigid bodies to extend the communication neighborhood (see Section 9.1). Instead it is
assumed that, since these bodies are not moving, a synchronization via communication is not
necessary. For the purpose of setting up large, immobile rigid bodies, such as for instance all
infinite primitives like planes, or fixed bodies as the outlet structure of the granular medium
simulation (see Figure 9.4), the pe offers a special section, the pe_GLOBAL_SECTION. Rigid
bodies created inside a global section are considered global bodies, which are known on all
processes and are fixed per default. Listing 9.7 shows an example of the use of a global section:

Listing 9.7: Application of the pe_GLOBAL_SECTION

```c
// Default parallel region
// ...

// Global parallel region
pe_GLOBAL_SECTION
{
    // Setup of a global sphere
    createSphere( 1, Vec3( 10.0, 20.0, -40.0 ), 5.0, oak );

    // Setup of a global union
    pe_CREATE_UNION( union3, 2 ) {
        // ...
    }

    // Synchronization of the non-global rigid bodies
    world->synchronize();
```

In this example, the pe_GLOBAL_SECTION macro starts a global section for the setup of global
rigid bodies. Inside the global section the oak sphere primitive 1 at position (10, 20, -40) with
a radius of 5 is created as well as a union with the user-specific ID 2. Both rigid bodies
are considered global, i.e., they are known on all involved MPI processes and their global
position is automatically fixed. Note that, since the rigid bodies are known on all processes,
they are not communicated and therefore also not synchronized by the synchronize function.
All modifications applied to a global rigid body have to be applied on all processes. There-
fore it is prohibited to modify (translate, rotate, destroy, etc.) a global rigid body inside an
exclusive section. The attempt to do this results in an exception.

---

*6For more information on the pe global section refer to section 4.3.*
9.3 MPI communication

Even more basic than the question of when to send which data (which is answered in Section 10.2) is the question of how to send data from one MPI process to another. In order to parallelize a rigid body simulation, it is necessary to handle several fundamental problems concerning the MPI communication itself. These problems, including heterogeneous data types, heterogeneous geometry types, and arbitrarily sized messages, are covered in this section.

The first problem stems from the fundamental performance rule that the number of messages in a MPI communication should be kept as small as possible, since every MPI message involves a fixed, considerable latency. Therefore one goal for an efficient parallelization has to be to send a single MPI message to a remote process in every communication step. However, this results in the requirement to send different data types (e.g., integral data and floating point data) at the same time. The solution to this problem in the pe framework is the definition of a fixed message layout and to send all necessary data as raw bytes to the remote processes. Each remote process is now responsible to convert the bytes back to their original data types, which is only possible due to the fixed message layout.

A second problem are the heterogeneous geometry types (as for instance spheres, boxes, or arbitrary triangle meshes). Different geometry types involve a different number of values to be sent. For example, for the geometric description of a sphere primitive, it is sufficient to send only the global position and the radius of the sphere. For a box primitive, the position, its orientation and the lengths of the three sides have to be sent. This problem can be easily solved by starting every data segment in a message with a particular message tag. Depending on this tag, the receiving process knows the content, size and structure of the following bytes and is able to convert them back to their original types and values.

The third problem is the variable size of a MPI message in a rigid body simulation. The number of rigid bodies that have to be sent to a neighboring process is only depending on the current position and orientation of the bodies. In combination with the heterogeneous data types, heterogeneous geometry types, and arbitrarily sized messages, are covered in this section.

...
types and the heterogeneous geometry types, the size of a particular message is unknown for the receiving process (see for example Figure 9.12). It is even possible that a message contains no data, i.e., the message size is zero (see for example Figure 10.3(a)). However, since it is not possible to tell beforehand whether a message has to be sent or not, the receiving process will always expect a message. Therefore it is not possible to omit empty messages. In order to receive messages of arbitrary size, the receiving process has to acquire the size of a message before receiving it. This is performed by the MPI_Probe() function with a subsequent call to the MPI_Get_count() function.

For every communication step during the parallel rigid body simulation, the send buffers for the neighboring processes have to be filled individually with the according information. This encoding step is succeeded by the message exchange via MPI, which is then followed by the decoding phase of the received messages. The complete MPI communication used by the $pe$ is shown in Algorithm 1. The first step during communication is the initialization of the transfer of the according byte-encoded messages to the neighboring MPI processes via the non-blocking message passing function MPI_Isend(). While the MPI system handles the send operations, the receive operations are initiated. Since the size of the messages is unknown, the MPI_Probe() and MPI_Get_count() functions are used to test any incoming message. After that, the size of the receive buffer is adjusted and the blocking message passing function MPI_Recv() is used to receive the message. After all messages have been received, it is necessary to wait for all non-blocking send operations to finish. Only then the send buffers can be cleared for the next communication step.

Algorithm 1: MPI communication

1. for each neighboring process do
   2. Encode the message to the neighboring process
   3. Start a non-blocking send operation of the according byte encoded message via MPI_Isend()
4. end
5. for $i$ from 0 to the number of neighboring processes-1 do
   6. Check for any incoming message via MPI_Probe()
   7. Acquire the total size of the message via MPI_Get_count()
   8. Adjust the buffer size of the receiving buffer
   9. Receive the message via MPI_Recv()
10. Decode the received message
11. end
12. Wait until all non-blocking send operations are completed
13. Clear the send buffers

9.4 Conclusion

This chapter has covered the problems related to the distributed memory parallelization of rigid body simulations. I have explained the problems resulting from the fact that rigid
bodies are fully resolved and therefore are physically present on several MPI processes. Additionally, I have given insight into the MPI communication and the domain partitioning concepts as realized in the \texttt{pe} framework. These were demonstrated by means of several examples of how to set up parallel rigid body simulations.
The simulation of interacting rigid bodies with unilateral contacts is a very important field of research for many engineering applications and natural phenomena. Therefore a variety of different approaches exist to handle the frictional contacts between rigid bodies. The currently most accurate approaches to predict frictional contact forces, which really treat the rigid bodies as completely undeformable, are the rigid body dynamics (RBD) formulations based on linear complementarity problems (LCP) [32, 94, 73, 19] [6]. In this approach, all constraints are gathered in a global LCP problem, which is then solved by simplex methods (also known as direct or pivoting methods; for instance the algorithms of Lemke and Dantzig [32]) or variations of LSE solvers, as for instance the Projected Gauss-Seidel (PGS), the Projected Gauss-Jacobi (PGJ), or Conjugate Projected Gradient methods [6]). Due to the high complexities of these solvers (they have a complexity ranging from $O(N\log N)$ to even $O(N^4)$), the computational efficiency of LCP problems resulting for instance from millions of interacting granular particles is a problem even on the most efficient supercomputers. Additionally, the parallelization itself is very difficult due to the unstructured nature of rigid body dynamics.

One approach to large-scale RBD simulations was presented by Tasora et al [103]. Their approach is focused on graphical processing units and is based on a variation of the general LCP-formulation, the cone complementarity problem (CCP), for the simulation of frictional contact dynamics. Due to the computational power of GPGPUs, their approach is suited for several million rigid bodies. However, due to the notorious difficulty parallelizing LCP-based collision response algorithms, their approach is focused on the shared memory parallelization on graphics hardware and is not (yet) suited for distributed memory systems. Simulations are therefore limited to the available graphics memory.
A second, straightforward approach to solve this class of problems consists of regularization schemes, as for instance the Discrete Element Method (DEM) [27, 43], that transforms the discontinuities of rigid body collisions into a stiff force field. With this approach the collision resolution calculations becomes completely local (in contrast to the global LCP problem), which highly facilitates the parallelization for distributed memory machines. However, the regularization approach requires very small time steps to keep interpenetrations between rigid bodies small and to achieve numerical stability. Therefore this approach results in very high computational costs and long runtimes – even on supercomputers.

Next to these two approaches favored in the engineering community to deal with rigid body collisions, a third group of algorithms, focused on the real-time requirements of computer games and virtual reality environments, has become (especially in recent years) increasingly popular [54, 87, 85, 50, 26, 84]. Although the frameworks focused on computer games (so called “physics engines”, such as for instance the Havok engine [58] or the PhysX framework [62]) are sometimes also using fast implementations of LCP-based formulations, they usually offer collision response algorithms with a strictly linear complexity in the number of contacts between rigid bodies. Due to their focus on efficiency, these algorithms are usually less accurate than the LCP approach (i.e., they are interested in calculating feasible collision responses, not necessarily physically accurate responses, for instance in terms of momentum and energy conservation), but they avoid the setup of a global problem and deal with the collisions more locally. Therefore they are in general more suited for large-scale parallelizations.

So far, from all the aforementioned approaches to simulate rigid bodies with unilateral contacts, only the DEM approach has already been parallelized successfully [41, 42]. The primary reason for this is that in the DEM approach collisions between rigid bodies are handled completely locally and non-iteratively by only taking into account the current state of the directly contacting rigid bodies. In contrast to that, all other approaches are either much harder and more complicated to parallelize due to a global problem formulation or iterative algorithms, or less promising for parallelization due to a high computational complexity or a high expected communication overhead. The choice for a first approach towards massively parallel RBD is therefore primarily driven by the suitability for parallelization and less by the aspect of physical accuracy (i.e., momentum and energy conservation; see Section 10.5).

The algorithm of choice for the \( pf \) framework is an algorithm from the group of fast, local collision solvers favored by the computer graphics and computer games communities. Section 10.1 will shortly introduce the original Fast Frictional Dynamics (FFD) algorithm as proposed by Kaufman et al. [76] in 2005, which is focused on non-deformable rigid bodies and restricted to contact forces, i.e., does not consider joints. Subsequently, the parallelization of the FFD algorithm, the Parallel Fast Frictional Dynamics (PFFD) algorithm, which represents the first massively parallel RBD algorithm suited for massively parallel supercomputers, will be discussed in Section 10.2. Section 10.3 will provide a scenario-based comparison between LCP-based collision solvers and the PFFD algorithm. The scalability of the PFFD for massively parallel simulations will be demonstrated in Section 10.4, where I will present performance and scaling results and illustrate the simulation of 2 billion particles on 131 072 processor cores. Section 10.5 will present results of a validation experiment for a granular gas scenario.
10.1 The Serial Algorithm

The Fast Frictional Dynamics Algorithm was introduced by Kaufman et al. [76] and improved by Wengenroth et al. [11]. Due to its modeling approach, this algorithm belongs to the group of fast, real-time collision response algorithms. The FFD employs a novel contact model that uses mass, location and velocity information from all contacts at the moment of maximum compression to constrain rigid body velocities and to achieve “approximate momentum conservation” [76]. It also uses a new friction model in the configuration space of rigid bodies, which unifies rolling and sliding friction based on the principles of maximum dissipation. The two most novel and distinctive attributes of the FFD are its low computational complexity and the fact that the simulation problem is solved exclusively in the velocity space.

Algorithm 2 shows the complete FFD algorithm as published in [76]. Every simulation step starts with a position half-step (line 2) and velocity half-step (line 3) in order to calculate the approximate position and velocity of all objects midway through the time step. The position half-step is a simple explicit Euler step using the current velocity $\vec{\phi}$ of an object. In Kaufman et al. [76] an exponential map $W^E_i$ is used to update the affine map $W_i$ representing the configuration of body $i$:

$$ W_{i}^{\text{new}} = W_{i}^{E} \exp \left( \frac{1}{2} \begin{bmatrix} w_{i} & v & 0 \\ 0 & 0 & 0 \end{bmatrix} \right). $$ (10.1)

where $\vec{v}$ represents the linear velocity of body $i$, $\vec{w}$ represents the angular velocity, and $h$ represents the size of the current time step. The square brackets around a three-dimensional vector denote the $3 \times 3$ matrix form of the cross product, i.e.:

$$ [\vec{a}] = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}, \text{ or: } [\vec{a}] \vec{b} = \vec{a} \times \vec{b} = -\vec{b} [\vec{a}] \text{.} $$ (10.2)

In the pe implementation of the FFD (see [11]), rotations of rigid bodies are handled by quaternions instead [13].

In the velocity half-step, as described in Kaufman et al. [76], which operates on body frame values only, the Coriolis force $\vec{c}$ has to be taken into account as well as external, non-contact forces, as for instance gravity:

$$ \ddot{\vec{\phi}}_i = \mathcal{M}^{-1} \vec{f}_i + \mathcal{M}^{-1} \vec{c}_i = \mathcal{M}^{-1} \vec{f}_i + \mathcal{M}^{-1} \begin{bmatrix} \vec{w}_i \\ \vec{v}_i \end{bmatrix} = \mathcal{M} \begin{bmatrix} \vec{w}_i \\ \vec{v}_i \end{bmatrix} = \mathcal{M} \begin{bmatrix} \vec{w}_i \\ \vec{v}_i \end{bmatrix} \mathcal{M}^{-1} \vec{f}_i - \mathcal{M}^{-1} \begin{bmatrix} \vec{w}_i \\ \vec{v}_i \end{bmatrix} \mathcal{M} \begin{bmatrix} \vec{w}_i \\ \vec{v}_i \end{bmatrix} = \mathcal{M}^{-1} \vec{f}_i - \mathcal{M}^{-1} \begin{bmatrix} \vec{w}_i \times (\vec{I} \vec{w}_i) \\ m_i (\vec{w}_i \times \vec{v}_i) \end{bmatrix} \text{.} \quad (10.3)

$$

$$ \vec{\phi}_i^{\text{new}} = \vec{\phi}_i + \frac{1}{2} h \ddot{\vec{\phi}}_i \text{.} \quad (10.4)$$

---

1The motivation to use exponential maps to update the configuration of an object is described in [47].
Algorithm 2: The FFD-Algorithm

for each body $B$ do
  position half-step ($\phi^+_B$)
  $\delta^+_B$ = velocity half-step ($B$)
end

for each body $B$ do
  find all contacts $C(B)$
  for each contact $k \in C(B)$ do
    calculate twist $\vec{n}_k$ induced by contact normal
    calculate constraint offset $d_k$
    if constraint is violated ($B$, $\vec{\phi}^-_B$, $\vec{n}_k$, $d_k$) then
      add collision constraint $\vec{n}_k$, $d_k$ to $T(B)$
      for $m = 1 \ldots N_{\text{samples}}$ do
        calculate twist $\vec{s}_m$ induced by $m^{th}$ sample $\perp$ to contact normal
        add friction constraint $\vec{s}_m$, $\mu_m$ to $S(B)$
      end
    end
  end
end

for each body $B$ do
  if $B$ has violated constraints then
    find post-collision velocity $\vec{\phi}^*_B$ on the border of $T(B)$ closest to $\vec{\phi}^-_B$
    $\vec{r}_B = \vec{\phi}^*_B - \vec{\phi}^-_B$
    select friction response $\vec{\delta}_B$ from $S(B)$ minimizing $\vec{\delta}_B + \vec{\phi}^+_B$
    $\vec{\phi}^+_B = \vec{\phi}^*_B + \epsilon \cdot \vec{r}_B + \vec{\delta}_B$
  end
  else
    $\vec{\phi}^+_B$ = velocity half-step ($B$)
  end
position half-step ($B$)
end

where $I$ is the $3 \times 3$ inertia tensor and $M$ is the generalized mass matrix (with $I_3$ being the $3 \times 3$ identity matrix):

$$\mathcal{M} = \begin{pmatrix} I & 0 \\ 0 & m \cdot I_3 \end{pmatrix}.$$  \hfill (10.4)

After the first position and velocity half-step, in the collision detection phase (Algorithm 2, line 6), a list of currently active contacts between the rigid bodies is created. Every contact is treated as an independent, totally inelastic collision. A set of minimal constraints on the involved object’s velocities and the friction contribution is deduced from every contact (lines 7-17). When all contacts have been taken into consideration, a post-collision velocity $\vec{\phi}^*_B$ (line 21) and a friction response $\vec{\delta}$ (line 23) are selected for every colliding body to obtain its post-
resolution velocity $\vec{\phi}^+$. For all non-colliding bodies, a second velocity half-step is applied (line 27). Finally the new, locally feasible velocity is used to compute another position half-step (line 29). This way, feasible velocities can be guaranteed at all contact points, limiting inter-penetration to the moment (or rather half-step) of impact. Note that for free moving objects, the staggered arrangement of the half-steps is equivalent to using the midpoint method to determine the object velocity. For more details about the $p$e implementation of the FFD, including the calculation of the post-resolution velocity $\vec{\delta}$ and the friction response $\vec{\phi}^\tau$, see [11].

10.2 The Parallel Algorithm

The FFD solver is particularly suited for a large-scale parallelization due to its strictly local collision treatment: In order to calculate a post-collision velocity for a colliding rigid body, only the states of the contacting rigid bodies are considered. Therefore it is not necessary to set up and solve a LCP in parallel as it would be necessary in case of the LCP-based collision response algorithm. Additionally, the complexity of the FFD algorithm is linearly depending on the number of rigid bodies and the number of contact points between these bodies.

Figure 10.1 shows the parallel version of the FFD algorithm, the Parallel Fast Frictional Dynamics (PFFD) algorithm. The parallelization of this algorithm is based on the parallelization concepts discussed in Chapter 9. In contrast to the nonparallel algorithm, the parallel version contains a total of four MPI communication steps to handle the distributed computation (see the lines 1, 6, 13 and 24). The rest of the algorithm remains unchanged in comparison to the nonparallel formulation.

Instead of immediately starting with the first position and velocity half-step for every rigid body, the first step in every time step of the parallel algorithm is the synchronization of the external forces (Figure 10.1, line 1). In case a rigid body is overlapping a process boundary, part of the external forces can be contributed by the remote processes. This scenario happens for instance in case the rigid body is immersed in a fluid flow (as illustrated in Section 1.2, Figure 1.6). Part of the hydrodynamic forces are calculated by the local process, another part by the remote processes. Therefore the involved processes have to synchronize their forces and calculate a total force for every rigid body.

After the force synchronization, all local rigid bodies (i.e., all rigid bodies whose reference point is contained in the local process) are updated with the first position half-step and first velocity half-step. Note that only local rigid bodies are treated. Every rigid body is updated by exactly one process, i.e., by the process its center of mass is contained in. Remote rigid bodies (i.e., all rigid bodies whose reference points are not contained in the local process) are updated in the second MPI communication step (Figure 10.1, line 6). This communication step involves a synchronization of the position, the orientation and the velocities of remote rigid bodies, where the process that contains the reference point of the body sends the updated values to all neighboring remote processes that only contain part of the body. Due to the position and orientation change of all rigid bodies, it may also occur that a rigid
body now crosses a process boundary of a particular neighboring remote process. In this case the entire rigid body has to be sent to the remote process, including information about its geometry (as for instance the radius of a sphere, the side lengths of a box, the triangle mesh of an arbitrary geometry) and its material (containing information about its density, the coefficient of restitution and frictional parameters). With this information, the remote process is now able to create a copy of the rigid body.

Note that, due to efficiency reasons, updates should be preferred to sending the complete rigid body. Only in case the remote process does not know the rigid body, sending the full set of information cannot be avoided. Also note, that it may be necessary to send updates for a rigid body that is no longer overlapping a particular boundary in order to give the remote process the information that the rigid body has left the domain of the remote process and can therefore be destroyed on the remote process.
The second communication step is followed by the collision detection that creates contact points for every pair of rigid bodies that is in contact. Directly afterwards, the setup of all collision constraints can be performed (see Figure 10.1, line 9-11): For every contact point attached to the rigid body, a single collision constraint and several friction constraints are created in case the collision constraint is violated.

![Figure 10.2: Some collisions can only be detected on remote processes. In the illustrated example, process 0 only knows the local sphere 1, for which collision constraints have to be calculated. Process 1, on the other hand, knows both sphere 1 and 2 (where sphere 1 is remote and 2 is local). Therefore only process 1 can setup the constraints for the collision between spheres 1 and 2 and has to send these constraints to process 0, where they are used to handle the collision between the two spheres.](image)

In the consecutive third communication step (Figure 10.1, line 13), the MPI processes exchange collision and friction constraints among each other such that every process knows all active constraints on all local rigid bodies. This exchange is necessary since some constraints can only be detected on remote processes, as illustrated in Figure 10.2. These constraints can now be used to treat all collisions and to update the velocities of the colliding rigid bodies. In case a rigid body is not involved in a collision, the velocity is updated in a second velocity half-step. After the velocity update (either due to collision handling or a velocity half-step), the second position half step is performed. In order to cope with this second change of the position and orientation of the rigid bodies, the fourth communication step (Figure 10.1, line 24) performs the same synchronization as the second communication step. After this communication, all rigid bodies on all processes are synchronized and in a consistent state and the time step is finished.

![Figure 10.3 gives an example of a single moving rigid body that crosses the boundary between two processes. The example explains in detail when the rigid body has to be sent by which process and when the responsibility for a rigid body is transferred to another process. The reference point of the rigid body is indicated by the cross in its center, its linear velocity by the arrow. In Figure 10.3(a), the rigid body is solely contained in process 0. Process 1 does not know the rigid body. In Figure 10.3(b), the rigid body has moved into the overlap region between the two processes and therefore has to be sent to process 1. Only in the first communication, the entire rigid body, including information about its geometry and its material have to be sent. In subsequent communications, it is sufficient to send updates for the body. In Figure 10.3(c), the rigid body’s reference point is no longer contained in process 0, but has moved to process 1. Therefore process 1 now considers the](image)

143
rigid body as local, process 0 only as remote. In Figure 10.3(d), the rigid body has left the overlap region between the two processes and does not have to be communicated anymore. Therefore process 0 no longer contains any part of the body, so the body is locally destroyed. Note that the choice of the size of $dx$ is arbitrary, but it should be larger than the minimum distance between two colliding bodies.

![Diagram](image1.png)

**Figure 10.3:** Example of the communication of a single rigid body.

As discussed, the parallel version of the FFD algorithm contains four distinct MPI communications. We believe that this is the minimal number of MPI communications in case we do not want to change the fact that in between two time steps the user may manipulate rigid bodies by for instance adding external forces to the bodies. Tab. 10.1 lists the dependencies of the individual communication steps from the steps in the non-parallel algorithm.

<table>
<thead>
<tr>
<th>Step</th>
<th>Purpose</th>
<th>After</th>
<th>Before</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Force synchronization</td>
<td>Force calculation</td>
<td>1. velocity half-step</td>
</tr>
<tr>
<td>2</td>
<td>Update of remote and notification of new bodies</td>
<td>1. position half-step</td>
<td>Collision detection</td>
</tr>
<tr>
<td>3</td>
<td>Collision constraint synchronization</td>
<td>Constraint setup</td>
<td>2. velocity half-step</td>
</tr>
<tr>
<td>4</td>
<td>Update of remote and notification of new bodies</td>
<td>2. position half-step</td>
<td>Force calculation</td>
</tr>
</tbody>
</table>

**Table 10.1:** Dependencies of the MPI communication steps.
10.3 Scenario-based Evaluation

So far, only LCP-based collision solvers, as well as the PFFD algorithm, have been integrated into the *pe* rigid body simulation framework as collision response algorithms. Although these solvers only represent a fraction of all possible collision response algorithms, this still allows for a qualitative comparison between LCP-based collision solvers and the PFFD algorithm. Table 10.2 gives an overview over the experiences gained with the *pe* implementations of both approaches for four relevant simulation problems.

The first problem is the classical billiard scenario. Due to the fact that this scenario only involves a very small number of rigid bodies, both approaches are suited to simulate this scenario. However, in comparison to the LCP-based solvers, whose only disadvantage is the fact that coefficients of restitution larger than zero (i.e., for elastic collisions) cannot be properly modeled with LCPs (see [19] and [6]), the PFFD can only accurately resolve this scenario in case the collisions are treated as a series of successive collisions in several time steps. Therefore LCP-based collision solvers are in practice more suited for this particular scenario.

A very similar impression of the two collision solver approaches can be gained by the second simulation scenario, the well-known Newton’s cradle scenario. For this scenario the LCP-based solvers are also more suited than the PFFD algorithm due to the better accuracy of the LCP-based solvers. However, although in practice the LCP-based solvers are able to simulate the effect that the momentum of the colliding sphere is transferred to the opposite sphere, this behavior cannot be guaranteed since there are multiple solutions to this problem. Additionally, as in the case of the billiard example, the elastic collisions cannot be properly modeled by the LCP formulation. The PFFD solver is only able to recover the effect in case the collisions are treated as a series of successive collisions in several time steps.

Stacking problems are a specific class of scenarios that involve stackable rigid bodies, as for instance boxes, cylinders, etc. These particular problems cannot be correctly simulated by the PFFD solver. Since the collision constraints acting on a stacked body result in infeasible responses (see [11]), the stack appears to be completely indestructible and resembles a stack built from fixed objects. LCP-based solvers work well for this class of problems, but are limited in the number of rigid bodies due to the size of the resulting LCPs.

For granular media simulation scenarios involving huge numbers of sphere-like rigid bodies, the PFFD solver can show its full potential. Due to the purely local collision handling and the linear complexity in the number of contacts, the PFFD solver is perfectly suited for massively parallel simulations and therefore allows for the simulation of billions of interacting particles (see for instance Chapter 11). Although the LCP-based solver can simulate granular media scenarios as well, they are extremely limited in the number of rigid bodies due to the size of the resulting (global) LCPs.

---

2 Please note that this comparison is purely based on the experiences with the *pe* framework and is therefore not representative for other frameworks. Further comparisons to other solvers and closer investigations of advantages and disadvantages in certain scenarios are left for future research.
<table>
<thead>
<tr>
<th>Illustration</th>
<th>Description</th>
<th>LCP-based solvers</th>
<th>PFFD solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Billiard</td>
<td>Simulation of a small number (usually 16) perfectly elastic balls on a billiard table. Results in a very small number of contacts.</td>
<td>In practice physically very accurate results, but modeling problems for elastic bodies.</td>
<td>Cannot be simulated accurately within a single time step, but only as successive collisions in several time steps.</td>
</tr>
<tr>
<td>Newton’s Cradle</td>
<td>Simulation of a row of equally sized spheres of same mass. Very small number of bodies and contacts.</td>
<td>Works in practice, but physically accurate result is not guaranteed due to multiple possible solutions. Additionally, modeling problems for elastic bodies.</td>
<td>Cannot be simulated accurately within a single time step, but only as successive collisions in several time steps.</td>
</tr>
<tr>
<td>Stacking Problems</td>
<td>Simulation of large numbers of stackable objects (boxes, cylinders, etc). Usually results in multiple contacts between two rigid bodies.</td>
<td>Stable and accurate, but limited in the number of bodies due to the size of the resulting LCP problems.</td>
<td>Incorrect simulation results: Stack appears to be completely indestructible.</td>
</tr>
<tr>
<td>Granular Media</td>
<td>Simulation of a huge number of sphere-like rigid bodies with an enormous number of contacts.</td>
<td>Physically very accurate, but extremely limited in the number of bodies due to the number of contacts.</td>
<td>Physically less accurate, but feasible results for large numbers of bodies. Suitable for large-scale parallelization. Scales linearly with the number of contacts.</td>
</tr>
</tbody>
</table>

**Table 10.2**: Scenario-based comparison between LCP-based collision solvers and the (P)FFD solver.
10.4 Performance Results

In [68] I have already demonstrated the strong and weak scaling behavior of the PFFD with up to 512 cores on the Xeon cluster of the Regional Computing Center Erlangen (RRZE) [65]. In this section I will demonstrate the scaling behavior of the PFFD algorithm on two machines that are listed in the Top500 list [66]. The first machine is the HLRB-II supercomputer (SGI Altix 4700 platform) (see Appendix A.1) at the Leibnitz computing center Munich (LRZ) [60]. The second machine for the scalability experiments is the Jugene supercomputer (see Appendix A.2) at the Jülich Supercomputing Centre [59]. In case of the HLRB-II, I am using up to 9 120 cores in order to simulate up to 1.14 billion rigid bodies in a single simulation\(^3\). In case of Jugene, I will demonstrate the scaling behavior of the PFFD on up to 131 072 processor cores and up to 2 billion rigid bodies\(^4\).

The scenario I have chosen for the scaling experiments is illustrated in Figure 10.4: I am simulating a granular gas consisting of (non-)spherical particles in an evacuated box-shaped volume without external forces. For this a specified number of uniformly distributed particles is created, each with a random initial velocity. Although this scenario only demonstrates a fraction of the capabilities of the framework, it is well suited for the scaling experiments, since it provides statistically a good load balancing and communication balance with the initial domain partitioning.

![Simulation scenario for the scaling experiments](image)

**Figure 10.4:** Simulation scenario for the scaling experiments: The specified number of (non-)spherical rigid bodies are moving randomly in a closed, evacuated, box-shaped domain without external forces.

10.4.1 Serial Performance

In order to be able to analyze the parallel performance of the algorithm, it is first necessary to investigate the serial performance of the \(pe\) framework. Figure 10.5 shows a serial scaling experiment with up to 27 000 spherical particles in the scenario described above using

---

\(^3\)On the HLRB-II, all executables are compiled with the Intel 10.1 compiler (using the optimization flag -O3) and I am using SGI MPI for ccNUMA Altix systems

\(^4\)On Jugene, all executables are compiled with GCC 4.1.2 (using the optimization flag -O3) and I am using the default IBM MPI for Blue Gene Systems
two different coarse collision detection algorithms. Apparently in both cases the rigid body simulation exhibits a scaling behavior linear with the number of bodies. However, the performance obviously differs depending on the selected coarse collision detection algorithm.

The top graph shows the simulation performance with the Sweep-And-Prune (SNP) algorithm \cite{21, 38, 39}. Although this algorithm is very popular in the rigid body community, it strongly depends on a very high spacial and temporal coherence, which is not optimal in this particular simulation scenario. Additionally, for parallel simulations, SNP suffers from the overhead resulting from the fact that rigid bodies may appear or disappear on process boundaries, which decreases the coherence (see \cite{8} for a complete discussion).

The bottom graph shows the performance with the Hierarchical Hash Grids (HHG) algorithm \cite{38, 84}. Obviously this algorithm is much more suited for the requirements of this simulation scenario and additionally does not suffer from the effects of the parallelization \cite{8}. Therefore this algorithm was chosen for the strong and weak scaling experiments.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure105}
\caption{Serial performance of 1000 time steps in the scenario for the strong and weak scaling experiments.}
\end{figure}

\section*{10.4.2 Strong Scaling Results}

Figure 10.6 and Table 10.3 give an impression for a strong scaling experiment on Jugene with 2188232 spherical particles initially arranged in a 523 \times 523 \times 8 grid. In this experiment, the spheres use approximately 6.5\% of the total volume of the domain (which makes this a rather dense granular gas). In this experiment, I am using 256 up to 4096 processor cores and a strictly 2-dimensional partitioning (i.e., exactly one process in the z-direction).
The graph exhibits the usual strong scaling behavior: Increasing the number of processor cores asymptotically decreases the runtime. The reason for the asymptotic behavior is the growing ratio between surface of a process domain and the enclosed volume: By decreasing the volume of a process domain, the ratio of particles that have to be communicated grows. This becomes especially apparent for the last two simulations with 3 136 and 4 096 processor cores. In these two cases the ratio between computation and communication is particularly bad due to the small number of remaining rigid bodies.

![Scaling graph for a strong scaling experiment on Jugene of 1 000 time steps with 2 188 232 spherical particles.](image)

**Figure 10.6:** Scaling graph for a strong scaling experiment on Jugene of 1 000 time steps with 2 188 232 spherical particles.

<table>
<thead>
<tr>
<th># Cores</th>
<th># Spheres per core</th>
<th>Partitioning</th>
<th>Runtime [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>8 548</td>
<td>16 × 16</td>
<td>211.229</td>
</tr>
<tr>
<td>576</td>
<td>3 799</td>
<td>24 × 24</td>
<td>122.281</td>
</tr>
<tr>
<td>1024</td>
<td>2 137</td>
<td>32 × 32</td>
<td>84.9902</td>
</tr>
<tr>
<td>1600</td>
<td>1 368</td>
<td>40 × 40</td>
<td>61.8609</td>
</tr>
<tr>
<td>2304</td>
<td>950</td>
<td>48 × 48</td>
<td>50.152</td>
</tr>
<tr>
<td>3136</td>
<td>698</td>
<td>56 × 56</td>
<td>45.4584</td>
</tr>
<tr>
<td>4096</td>
<td>534</td>
<td>64 × 64</td>
<td>41.0598</td>
</tr>
</tbody>
</table>

**Table 10.3:** Strong scaling experiment of 1 000 time steps with 2 188 232 spherical particles initially arranged in a $523 \times 523 \times 8$ grid. The total volume of the spheres occupies approximately 6.5% of the domain volume.
10.4.3 Weak Scaling Results

Weak scaling in a rigid body simulation proves to be more complicated than strong scaling. In contrast to, for instance, grid-based simulations, where increasing the number of unknowns usually just means an increase of the resolution of the problem, increasing the number of rigid bodies inevitably changes the simulated scenario due to additional interactions. Weak scaling of a rigid body simulation also raises the question of where to create the new bodies. Therefore I chose to combine weak scaling with an analysis of the performance influence of the number of communication neighbors and to demonstrate simulations with over a billion rigid bodies.

In the weak scaling experiments, each process owns a cubic subdomain of initially $25 \times 25 \times 25$ or $50 \times 50 \times 50$ (non-)spherical particles, respectively. These blocks of particles are now put together depending on the arrangement of the processes. Note that, due to this setup, the weak scaling experiments cannot be directly compared to the strong scaling experiments: Whereas in the strong scaling experiment the shape of the domain was fixed and the shape of the subdomains was adapted according to the arrangement of the processes, in the weak scaling experiments the shape of a subdomain is fixed and the shape of the domain results from the arrangement of the processes.

Weak scaling experiments of the \( p_e \) framework for spherical particles on the HLRB-II are shown in Table 10.4 and 10.5. The first observation is that the scaling behavior clearly depends on the number of communication neighbors: When choosing a particular number of processes, a one-dimensional partitioning is clearly the fastest, whereas a two-dimensional partitioning increases the maximum number of communication neighbors from two to eight. In case a three-dimensional partitioning is chosen, the maximum number of communication neighbors even increases to 26. This observation is supported by a detailed measurement of the ratios of the individual phases of the FFD algorithm (see Table 10.6). Note that this results primarily from the choice of cubic blocks of spheres per process, which directly leads to an increased communication overhead in case of more neighbors, since for every neighboring process, messages have to be individually encoded and decoded. The second observation is that different partitionings with a similar communication overhead result in approximately the same runtime. Considering the fact that every single measurement represents a different scenario, this is a result that demonstrates the generally good scaling behavior of the MPI parallelization of the \( p_e \) framework.

Table 10.7 (spherical particles) and Table 10.8 (randomly created, non-spherical granular particles) show the results for a weak scaling experiment on Jugene with 128 up to 131,072 processor cores. Although the particles occupy approximately 6% of the domain volume (which again makes this a rather dense granular gas) and approximately 10% of the particles have to be communicated in every time step (which results in a non-trivial communication overhead) the efficiency of the parallelization for a large number of cores never drops below 98.6% in reference to the smallest simulation with 128 cores. This clearly shows the generally good scaling behavior of the MPI parallelization of the PFFD.
### 10.4 Performance Results

<table>
<thead>
<tr>
<th># Cores</th>
<th># Spheres</th>
<th>Partitioning</th>
<th>Runtime [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1000000</td>
<td>64 × 1 × 1</td>
<td>267.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 × 8 × 1</td>
<td>394.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 × 4 × 4</td>
<td>547.4</td>
</tr>
<tr>
<td>128</td>
<td>2000000</td>
<td>128 × 1 × 1</td>
<td>266.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 × 8 × 1</td>
<td>392.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 × 4 × 4</td>
<td>545.3</td>
</tr>
<tr>
<td>256</td>
<td>4000000</td>
<td>256 × 1 × 1</td>
<td>269.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 × 16 × 1</td>
<td>393.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 × 8 × 4</td>
<td>547.5</td>
</tr>
<tr>
<td>448</td>
<td>7000000</td>
<td>448 × 1 × 1</td>
<td>265.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28 × 16 × 1</td>
<td>391.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 × 8 × 7</td>
<td>565.8</td>
</tr>
<tr>
<td>960</td>
<td>15000000</td>
<td>960 × 1 × 1</td>
<td>273.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 × 30 × 1</td>
<td>402.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 × 8 × 8</td>
<td>560.4</td>
</tr>
<tr>
<td>1920</td>
<td>30000000</td>
<td>1920 × 1 × 1</td>
<td>268.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60 × 32 × 1</td>
<td>415.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 × 15 × 8</td>
<td>585.5</td>
</tr>
<tr>
<td>3840</td>
<td>60000000</td>
<td>3840 × 1 × 1</td>
<td>267.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64 × 60 × 1</td>
<td>432.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 × 16 × 15</td>
<td>611.5</td>
</tr>
<tr>
<td>7680</td>
<td>120000000</td>
<td>7680 × 1 × 1</td>
<td>263.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>120 × 64 × 1</td>
<td>432.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32 × 16 × 15</td>
<td>599.0</td>
</tr>
<tr>
<td>9120</td>
<td>142500000</td>
<td>9120 × 1 × 1</td>
<td>264.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>96 × 95 × 1</td>
<td>432.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24 × 20 × 19</td>
<td>600.3</td>
</tr>
</tbody>
</table>

**Table 10.4:** Weak scaling experiment of 1000 time steps with up to 142 500 000 spheres. Each process initially owns a block of 25 × 25 × 25 spheres. The total volume of the spheres occupies approximately 1% of the domain volume.

The simulations with 131 072 cores deserve some special attention. Although the achieved scaling results do not show the limits of the \( pe \) framework (it has already been used on up to 294 912 processor cores on the Jugene system in combination with the Lattice Boltzmann framework WALBERLA [46], which impressively demonstrates the scalability of the \( pe \) over the maximal currently possible range), they show that it is possible to simulate several billion interacting, fully resolved rigid bodies in reasonable runtimes. Additionally, it demonstrates that the PFFD can even be used on the largest supercomputers available.
### Table 10.5: Weak scaling experiment of 1000 time steps with up to 1,140,000,000 spheres. Each process initially owns a block of $50 \times 50 \times 50$ spheres. The total volume of the spheres occupies approximately 1% of the domain volume.

<table>
<thead>
<tr>
<th># Cores</th>
<th># Spheres</th>
<th>Partitioning</th>
<th>Runtime [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>8,000,000</td>
<td>$64 \times 1 \times 1$</td>
<td>1054.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$8 \times 8 \times 1$</td>
<td>1149.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$4 \times 4 \times 4$</td>
<td>1400.56</td>
</tr>
<tr>
<td>128</td>
<td>16,000,000</td>
<td>$128 \times 1 \times 1$</td>
<td>1059.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$16 \times 8 \times 1$</td>
<td>1139.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$8 \times 4 \times 4$</td>
<td>1400.87</td>
</tr>
<tr>
<td>256</td>
<td>32,000,000</td>
<td>$256 \times 1 \times 1$</td>
<td>1056.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$16 \times 16 \times 1$</td>
<td>1145.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$8 \times 8 \times 4$</td>
<td>1406.18</td>
</tr>
<tr>
<td>448</td>
<td>56,000,000</td>
<td>$448 \times 1 \times 1$</td>
<td>1049.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$28 \times 16 \times 1$</td>
<td>1141.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$8 \times 8 \times 7$</td>
<td>1405.81</td>
</tr>
<tr>
<td>960</td>
<td>120,000,000</td>
<td>$960 \times 1 \times 1$</td>
<td>1051.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$32 \times 30 \times 1$</td>
<td>1148.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$15 \times 8 \times 8$</td>
<td>1408.58</td>
</tr>
<tr>
<td>1920</td>
<td>240,000,000</td>
<td>$1920 \times 1 \times 1$</td>
<td>1065.06</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$60 \times 32 \times 1$</td>
<td>1157.27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$16 \times 15 \times 8$</td>
<td>1414.46</td>
</tr>
<tr>
<td>3840</td>
<td>480,000,000</td>
<td>$3840 \times 1 \times 1$</td>
<td>1055.39</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$64 \times 60 \times 1$</td>
<td>1157.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$16 \times 16 \times 15$</td>
<td>1418.86</td>
</tr>
<tr>
<td>9120</td>
<td>1,140,000,000</td>
<td>$9120 \times 1 \times 1$</td>
<td>1066.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$96 \times 95 \times 1$</td>
<td>1359.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$24 \times 20 \times 19$</td>
<td>1613.73</td>
</tr>
</tbody>
</table>

### Table 10.6: Runtime ratios of the individual phases of the PFFD algorithm for the weak scaling experiments for 1D, 2D, and 3D-partitioning.

<table>
<thead>
<tr>
<th>Phase</th>
<th>1D-Partitioning</th>
<th>2D-Partitioning</th>
<th>3D-Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication 1</td>
<td>1.81 %</td>
<td>2.0 %</td>
<td>3.6 %</td>
</tr>
<tr>
<td>First Half Step</td>
<td>12.1 %</td>
<td>11.13 %</td>
<td>9.0 %</td>
</tr>
<tr>
<td>Communication 2</td>
<td>7.04 %</td>
<td>9.52 %</td>
<td>14.7 %</td>
</tr>
<tr>
<td>Contact Detection</td>
<td>47.5 %</td>
<td>43.88 %</td>
<td>35.58 %</td>
</tr>
<tr>
<td>Communication 3</td>
<td>1.3 %</td>
<td>1.23 %</td>
<td>1.24 %</td>
</tr>
<tr>
<td>Second Half Step</td>
<td>17.13 %</td>
<td>15.82 %</td>
<td>12.8 %</td>
</tr>
<tr>
<td>Contact Resolution</td>
<td>0.02 %</td>
<td>0.02 %</td>
<td>0.02 %</td>
</tr>
<tr>
<td>Communication 4</td>
<td>13.1 %</td>
<td>16.4 %</td>
<td>23.06 %</td>
</tr>
</tbody>
</table>
### Performance Results

<table>
<thead>
<tr>
<th># Cores</th>
<th># Particles</th>
<th>Partitioning</th>
<th>Runtime [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>2 000 000</td>
<td>$8 \times 4 \times 4$</td>
<td>598.565</td>
</tr>
<tr>
<td>256</td>
<td>4 000 000</td>
<td>$8 \times 8 \times 4$</td>
<td>600.39</td>
</tr>
<tr>
<td>512</td>
<td>8 000 000</td>
<td>$8 \times 8 \times 8$</td>
<td>601.587</td>
</tr>
<tr>
<td>1 024</td>
<td>16 000 000</td>
<td>$16 \times 8 \times 8$</td>
<td>603.268</td>
</tr>
<tr>
<td>2 048</td>
<td>32 000 000</td>
<td>$16 \times 16 \times 8$</td>
<td>603.533</td>
</tr>
<tr>
<td>4 096</td>
<td>64 000 000</td>
<td>$16 \times 16 \times 16$</td>
<td>604.132</td>
</tr>
<tr>
<td>8 192</td>
<td>128 000 000</td>
<td>$32 \times 16 \times 16$</td>
<td>601.364</td>
</tr>
<tr>
<td>16 384</td>
<td>256 000 000</td>
<td>$32 \times 32 \times 16$</td>
<td>604.414</td>
</tr>
<tr>
<td>32 768</td>
<td>512 000 000</td>
<td>$32 \times 32 \times 32$</td>
<td>606.961</td>
</tr>
<tr>
<td>65 536</td>
<td>1 024 000 000</td>
<td>$64 \times 32 \times 32$</td>
<td>606.472</td>
</tr>
<tr>
<td>131 072</td>
<td>2 048 000 000</td>
<td>$64 \times 64 \times 32$</td>
<td>604.845</td>
</tr>
</tbody>
</table>

**Table 10.7:** Weak scaling experiment of 1 000 time steps with up to 2 048 000 000 spherical particles. Each process initially owns a block of $25 \times 25 \times 25$ particles. The total volume of the particles occupies approximately 6.5% of the domain volume.

<table>
<thead>
<tr>
<th># Cores</th>
<th># Particles</th>
<th>Partitioning</th>
<th>Runtime [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>2 000 000</td>
<td>$8 \times 4 \times 4$</td>
<td>727.096</td>
</tr>
<tr>
<td>256</td>
<td>4 000 000</td>
<td>$8 \times 8 \times 4$</td>
<td>726.991</td>
</tr>
<tr>
<td>512</td>
<td>8 000 000</td>
<td>$8 \times 8 \times 8$</td>
<td>727.15</td>
</tr>
<tr>
<td>1 024</td>
<td>16 000 000</td>
<td>$16 \times 8 \times 8$</td>
<td>727.756</td>
</tr>
<tr>
<td>2 048</td>
<td>32 000 000</td>
<td>$16 \times 16 \times 8$</td>
<td>727.893</td>
</tr>
<tr>
<td>4 096</td>
<td>64 000 000</td>
<td>$16 \times 16 \times 16$</td>
<td>728.593</td>
</tr>
<tr>
<td>8 192</td>
<td>128 000 000</td>
<td>$32 \times 16 \times 16$</td>
<td>728.666</td>
</tr>
<tr>
<td>16 384</td>
<td>256 000 000</td>
<td>$32 \times 32 \times 16$</td>
<td>728.921</td>
</tr>
<tr>
<td>32 768</td>
<td>512 000 000</td>
<td>$32 \times 32 \times 32$</td>
<td>729.094</td>
</tr>
<tr>
<td>65 536</td>
<td>1 024 000 000</td>
<td>$64 \times 32 \times 32$</td>
<td>728.674</td>
</tr>
<tr>
<td>131 072</td>
<td>2 048 000 000</td>
<td>$64 \times 64 \times 32$</td>
<td>728.32</td>
</tr>
</tbody>
</table>

**Table 10.8:** Weak scaling experiment of 1 000 time steps with up to 2 048 000 000 non-spherical particles. Each process initially owns a block of $25 \times 25 \times 25$ particles. The total volume of the particles occupies approximately 6% of the domain volume.
10.5 Validation

This section is dedicated to the validation of the PFFD in context of large-scale granular media simulations. Although the local collision treatment of the PFFD enables large-scale parallelization in the first place, it is this local treatment that might result in feasible, yet globally physically inaccurate results: Since for the resolution of collisions only the state of the directly contacting rigid bodies is taken into account, the local collision response does not have to correspond to the global, physically accurate collision response (see [11]).

Unfortunately, the validation of the PFFD for complex granular media simulations (as for instance shown in Chapter 11) is very difficult. Even for simpler scenarios, the validation tends to be a non-trivial task. Therefore, the scenario I have chosen for the validation is the granular gas scenario that is also used for the scaling experiments (see Section 10.4). It is assumed that all collisions are perfectly elastic. Due to this and due to the lack of external forces the kinetic energy of the system should remain constant.

I measure the initial and the final total kinetic energy of my system for several gas densities. What I expect to find is a perfect energy conservation for rarefied gases. In these simulations there should mainly be collisions between two rigid bodies, which are perfectly resolved by the PFFD. However, for dense gases I expect to find some deviations of the final kinetic energy from the initial energy. This is due to the fact that with an increasing gas density the probability for collisions between more than two rigid bodies increases.

<table>
<thead>
<tr>
<th>Volume fraction</th>
<th>0.419%</th>
<th>0.818%</th>
<th>1.939%</th>
<th>6.545%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total collisions</td>
<td>279144</td>
<td>498533</td>
<td>1122523</td>
<td>4056797</td>
</tr>
<tr>
<td>Pairwise collisions</td>
<td>77.2%</td>
<td>51.8%</td>
<td>19.2%</td>
<td>2.8%</td>
</tr>
<tr>
<td>Multi-collisions</td>
<td>22.9%</td>
<td>48.2%</td>
<td>80.8%</td>
<td>97.2%</td>
</tr>
<tr>
<td>Energy deviation</td>
<td>–</td>
<td>$3.83 \cdot 10^{-4}$</td>
<td>$-2.3 \cdot 10^{-3}$%</td>
<td>$-1.95 \cdot 10^{-2}$%</td>
</tr>
</tbody>
</table>

Table 10.9: Measurement of the energy deviation of the granular gas test scenario with 125,000 particles and 500,000 time steps.

Table 10.9 shows the results of my kinetic energy measurements involving 125,000 granular particles. I have chosen particle volume fractions ranging from 0.419% to 6.545% by varying the free space in between the particles. It becomes obvious that for a larger volume fraction, the number of collisions grows considerably. However, whereas for rarefied gases most collisions happen between exactly two spherical particles (77.2% for a volume fraction

---

In [11], a complete chapter is dedicated to a thorough discussion of all problems involved with the local collision resolution of the FFD algorithm. This chapter also includes a discussion about the energy conservation characteristics of the FFD as well as an analysis of simulation scenarios with multiple contacts, such as the classical Newton's cradle scenario and stacking problems.
of 0.419%, and 51.8% for a volume fraction of 0.818%), for dense gases the majority of collisions happens between more than two particles (80.8% for a volume fraction of 1.939%, and 97.2% for a volume fraction of 6.545%). A comparison between the initial kinetic energy of the system with the resulting final kinetic energy reveals that although for dense gases most of the collisions are multi-collisions, the energy deviation is only $1.95 \cdot 10^{-2}\%$ after 500,000 time steps of the simulation and a total of 4,056,797 collisions. For rarefied gases with a volume fraction of 0.419%, no energy deviation can be measured. Therefore I come to the conclusion that despite its purely local collision update the PFFD is well suited for large-scale granular media simulations.

### 10.6 Conclusion

The PFFD algorithm introduced in this chapter represents the first massively parallel rigid body dynamics algorithm. Although the choice of this algorithm was primarily driven by its suitability for large-scale parallelization and not by the important aspect of physical accuracy, the scaling results, the validation experiment, and especially the granular media simulations demonstrated in Chapter 11 impressively show its capabilities to simulate huge numbers of interacting, completely undeformable rigid bodies with frictional contact mechanics. This scientific achievement of the pe framework has already been recognized by the computer science community: For the achieved results of the PFFD algorithm, the last contribution to the ISC conference series has been awarded the PRACE award [69].

Still, although the local collision treatment of the FFD algorithm is a perfect precondition for the parallelization of this algorithm, this modeling approach unfortunately also causes inaccuracies and problems, which limits the generality of the approach. Therefore it will be necessary to conduct more research into developing a rigid body dynamics algorithm that is also suitable for massively parallel simulations but has better characteristics in terms of physical accuracy.
In this section I will present several large-scale granular media simulations to demonstrate the capabilities of the PFFD algorithm. The simulations will also demonstrate the flexibility in connecting processes with neighboring MPI processes. Please note that for visualization purposes the number of MPI processes is comparatively low since the size of the granular particles was chosen as a compromise, trying to make the simulation large but still being able to distinguish individual particles in the visualization.

11.1 The Cube Scenario

Figure 11.1: Illustration of the cube scenario with 344,960 spherical particles performed on 64 MPI processes.
The first example of a (still small scale) granular medium simulation is the “cube scenario”: 344,960 spherical particles are initially held within a transparent cubical container until the side walls swing open and release the particles (see Figure 11.1). The patches indicate which particles are handled by which of the 64 MPI processes that are arranged in a circular pattern around the center of the cube. Due to this arrangement of processes the computational load is evenly distributed among the processes, even after the particles flow out from the cube.

For this scenario, a total of 60,300 time steps with a time step size of 0.001 s were simulated with a runtime of approximately 3.6 hours on 64 Xeon 5160 “Woodcrest” cores running at 3.0 GHz [107]. Therefore, on average, a single time step was computed in approximately 0.22 seconds. On average, each process had to handle 5,390 particles.

11.2 The Staircase Scenario

Figure 11.2: Simulation of 720,484 spherical particles rolling down the steps of an ancient spiral staircase. The 32 MPI processes are arranged in a circular fashion around the center of the staircase.
11.3 The Silo Scenario

The second scenario is the “staircase scenario”. Figure 11.2 shows the simulation of 720 484 spherical particles that are rolling and bumping down the steps of a spiral staircase. Again, the processes are arranged in a circular pattern around the center pillar of the staircase (see Figure 11.3 for an example with 9 processes). Since the particles cannot penetrate this pillar each process has a maximum of two neighboring processes.

These 12 000 time steps with a size of 0.01s were performed in approximately 6.18 hours by 32 Xeon 5160 “Woodcrest” cores running at 3.0 GHz. On average, a single time step of this simulation was performed in 1.85 seconds and on average every process had to handle 22 515 granular particles. However, our current static load balancing is not perfect, and better results could be obtained using a dynamic rebalancing of the load. Especially the processes at the top of the staircase had to handle a larger number of particles as for instance the processes at the bottom of the staircase.

![Staircase Scenario Diagram](image)

**Figure 11.3:** Partitioning of the simulation domain in the staircase scenario for 9 MPI processes.

11.3 The Silo Scenario

The final simulation scenario is the “silo scenario”. In this scenario, the flow behavior of a granular medium contained in a silo-like structure is analyzed. Note that this is not the simulation of a round silo, but a flat, elongated 2-dimensional version of a silo. As illustrated in Figure 11.4 I am simulating 27 270 randomly generated non-spherical particles. Each particle is created by overlapping two to five random spheres which leads to particles that resemble pebbles or chocolate raisins. The particles are initially held in the silo until
Figure 11.4: Simulation of 27,270 randomly generated non-spherical particles on 256 MPI processes. Each granular particle is shaped from 2 to 5 spheres.

Figure 11.5: Detailed illustration of the domain partitioning of the MPI processes used in a granular medium simulation with 64 processes. Each process within the silo has a maximum of 8 neighboring MPI processes. The bottom processes outside the silo (in this example only process 0) are communicating with the processes directly above and below.
the bottom hatch opens and the granular medium flows out of the silo. For this simulation a total of 256 MPI processes were used. The arrangement of the processes is illustrated in Figure 11.5 for an example of 64 processes. In these simulations, 18 processes are used in the horizontal direction and 14 processes in the z-direction. 4 processes handle the particles falling from the bottom hatch.

Figures 11.6, 11.7, 11.8, 11.9, 11.10, and 11.11 show several simulation runs with different inclinations ranging from $0^\circ$ to $56.25^\circ$. In all cases the typical granular medium behavior can be observed: Primarily the central particles are flowing towards the bottom opening. The void created due to this movement is filled by the particles at the top of the granular medium. The bottom left and bottom right particles are the last to flow through the hatch.

For these simulations a total of 379 300 time steps with a time step size of 0.0005s second were used. Although on average a single process had only to handle 107 particles, due to the compression of the granular medium, the small time step, and the lack of load balancing, the overall simulation time was 16.4 hours for the simulation depicted in Figure 11.6. On average, a single time step of the simulation was performed in 0.156 seconds.

Table 11.1 shows a detailed analysis of the runtimes of the individual phases of the PFFD algorithm for the silo scenario after 1 000 time steps. It becomes obvious that the four communications steps (including encoding of messages, message transfer via MPI, and decoding of messages) are the most expensive part of this algorithm with a total of 56.15% of the runtime. However, in contrast to the granular gas scaling scenario (see Section 10.4), where the collision resolution only took a negligible percentage of the runtime, in this scenario the collision resolution phase is the single most expensive phase of the PFFD algorithm (closely followed by the contact detection phase). Also note that the large discrepancy between the minimum and the maximum runtimes of a particular phase result from the fact that no load balancing is used for this simulation. Because of that the subdomains of the top-most processes are emptied during the course of the simulation.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication 1</td>
<td>0.0677</td>
<td>5.8343</td>
<td>1.5408 (10.11%)</td>
</tr>
<tr>
<td>First Half Step</td>
<td>0.0019</td>
<td>0.1113</td>
<td>0.0642 (0.42%)</td>
</tr>
<tr>
<td>Communication 2</td>
<td>0.0688</td>
<td>5.9244</td>
<td>1.5827 (10.38%)</td>
</tr>
<tr>
<td>Contact Detection</td>
<td>0.0017</td>
<td>7.7821</td>
<td>3.1736 (20.82%)</td>
</tr>
<tr>
<td>Communication 3</td>
<td>0.0559</td>
<td>8.6949</td>
<td>2.7747 (18.21%)</td>
</tr>
<tr>
<td>Second Half Step</td>
<td>0.0053</td>
<td>0.3285</td>
<td>0.1846 (1.21%)</td>
</tr>
<tr>
<td>Contact Resolution</td>
<td>0.0</td>
<td>3.2614</td>
<td>3.2614 (21.4%)</td>
</tr>
<tr>
<td>Communication 4</td>
<td>0.2712</td>
<td>6.0572</td>
<td>2.6597 (17.45%)</td>
</tr>
</tbody>
</table>

Table 11.1: Runtime ratios (in seconds) of the individual phases of the PFFD algorithm for the silo scenario after 1 000 time steps
Figure 11.6: The silo scenario with no inclination of the bottom walls at time step 0, 30,000, 90,000, and 120,000.

Figure 11.7: The silo scenario with an inclination of 11.25° at time step 0, 30,000, 90,000, and 150,000.
11.3 The Silo Scenario

Figure 11.8: The silo scenario with an inclination of $22.5^\circ$ at time step 0, 30 000, 90 000, and 180 000.

Figure 11.9: The silo scenario with an inclination of $33.75^\circ$ at time step 0, 30 000, 120 000, and 180 000.
11 Large-Scale Granular Media Simulations

Figure 11.10: The silo scenario with an inclination of 45° at time step 0, 30 000, 120 000, and 180 000.

Figure 11.11: The silo scenario with an inclination of 56.25° at time step 0, 60 000, 120 000, and 180 000.
Part IV

Conclusions and Future Work
12 Conclusion and Future Work

12.1 Conclusion

The main task of this work was the software engineering effort to create the pe multi-purpose rigid body simulation framework. Whereas the framework is also well suited for virtual reality environments, the primary target applications are the simulation of natural processes and industrial applications, such as granular media, sedimentation, segregation, or fluidization processes. Due to the enormous complexity of the applications involving various physical effects, huge numbers of interacting particles, and interdisciplinary aspects between different sciences, the requirements on the software framework are particularly high. In order to satisfy the demands on reliability, correctness, modularity, extensibility, efficiency, portability, and maintainability, all of which are essential aspects in the pe software development process, a systematic software design is crucial.

In several chapters I have talked about the beauty of the elegant C++ code used for the pe implementation that leads to a design that favors less coding errors. I have demonstrated the sophisticated design of several pe features such as for instance compile time constraints, which are focused on improving the reliability and correctness of the implementation, and the concept of smart scopes, which was invented during the development of the pe. I have explained in detail the logging functionality of the pe, which is suited for non-parallel, thread-parallel as well as massively parallel environments. By introducing the Sandwich Design Pattern, which represents the core concept of the pe’s flexibility and customizability, I have given a detailed insight into the software structure of the pe framework. The strive for efficiency has been demonstrated by explaining in detail the structure of the pe math library, which represents one of the most powerful and efficient C++ math libraries today.

An entire part of this thesis was devoted to the software concepts for massively parallel rigid body simulations. This feature of the pe, in combination with its ability to easily be
12 Conclusion and Future Work

coupled to massively parallel fluid solvers, enables realistic simulation scenarios of several billion fully resolved, non-spherical, interacting rigid bodies and therefore offers new possibilities to gain insight into the physical characteristics of granular media applications. As a demonstration of the capabilities of the PFFD algorithm, I have shown several large-scale granular media simulations with spherical as well as non-spherical granular particles, and I have presented scaling experiments with up to 131,072 cores on two massively parallel supercomputers, among them the world’s largest supercomputer in terms of number of cores, the Jugene.

12.2 Suggestions for Future Work

Although in the four years of active work on the pe the framework has already reached a size of approximately two hundred thousand lines of C++ code and several hundred source files, the work on this framework is far from being finished. On the contrary, now that the fundamental structure of the design has been finished and basic tools and features are ready for use, the pe looks forward to a very active and interesting future.

So far a lot of features that are supported by other rigid body frameworks have been neglected or only rudimentally been implemented. For instance, the feature of joints has only recently been added to the pe, but is not yet usable in large-scale simulations. Extending the scope of joints to parallel simulations would create completely new possibilities, for instance for the simulation of huge numbers of complex proteins.

So far the PFFD algorithm as developed in the context of the pe is the only massively parallel rigid body dynamics algorithm. However, due to its modeling it is limited to certain kinds of simulations: Whereas it is perfectly suited for granular media simulations, it is not a good choice for stacking problems. Therefore the integration of other collision solvers suited for massively parallel simulations should be pursued in order to extend the capabilities of the pe beyond large-scale granular media simulations. An important aspect of this work must also be the focus on an improved physical accuracy: Due to the origin of the PFFD algorithm in the computer games and computer graphics communities, its suitability for massively parallel simulations is achieved at the expense of momentum and energy conservation. First steps into this direction have already been initiated by a selection process for potential solvers suited for large-scale parallelization.

The collision resolution via LCP-based formulations has not been a topic of this thesis. However, the pe certainly also supports several LCP-based collision problem formulations and several friction models. It also offers several linear complementarity solvers, such as the projected Gauss-Seidel algorithm and the Conjugate Projected Gradient method. Based on fact that up to date this modeling approach is still one of the best available approaches today in terms of physical accuracy, work in this direction should definitely be continued. Due to the notoriously difficulty parallelizing these methods for distributed memory machines, it will also be necessary to examine parallelization concepts such as already used to port these
solvers on GPGPUs. Further work should also be invested to pursue the idea of multigrid algorithms for multibody problems.

A completely new direction for the pe could be the integration of deformable bodies. In contrast to the current limitation to completely rigid, undeformable bodies, this feature would greatly extend the scope of possible simulations. Possible scenarios involve for instance the simulation of red blood cells: in combination with a large-scale fluid flow solver, this would result in completely new prospects for realistic medical simulations. Some initial works in this direction have already been conducted (see for instance [100] and [2]), and an implementation of a particle-based model is currently developed. However, also finite element based models are promising.

Another, completely different, aspect will be the publication of the pe. Developed under the GNU Public License 3.0, the basic idea is to make the pe available to researchers and scientists of all fields for all kinds of rigid body simulations. Although the pe is in an exceptional and exemplary state (in comparison to other published frameworks), a lot of work remains until the framework will be a pleasant and successful experience for everyone.

Last but not least, a lot of work will be spent in optimizing the implementation of the pe in terms of performance on various architectures (including CPUs as well as GPUs and other accelerators as for instance the Cell Broadband Engine). Much work will also have to be invested in updating the pe implementation to the new C++ standard [99], which is expected in 2011.
Part V

Appendix
A Test Machines

A.1 HLRB II – SGI Altix 4700

Figure A.1: The HLRB-II supercomputer at the Leibnitz computing center Munich (LRZ).

The first machine for the massively parallel scaling experiments of the pe is the HLRB-II supercomputer (SGI Altix 4700 platform) at the Leibnitz computing center Munich (LRZ) [60]. Placed on rank 82 in the November 2009 Top500 list [66], this machine with a total of 9,728 processor cores and its fast network, which was installed in 2007, is still a perfectly suited mid range supercomputer for the scaling experiments of the PFFD. The HLRB-II features the following properties:

- 9,728 Intel Itanium2 Montecito Dual Core cores running at 1.6 GHz, 9 MB Level 3 Cache per core, and 4 GByte of RAM per core
A Test Machines

- 39 TByte of total main memory
- NUMAlink 4 interconnect fabric with 6.4 GByte/s bandwidth of one link (bidirectional)
- Overall peak performance of 62.3 TFlop/s (LINPACK result: 56.5 TFlop/s)

### A.2 Jugene – IBM BlueGene/P

![Jugene supercomputer](image)

**Figure A.2:** The Jugene supercomputer at the Jülich Supercomputing Center (JSC).

The second machine for the strong and weak scalability experiments of the *pe* framework is the Jugene supercomputer at the Jülich Supercomputing Centre [59]. Installed in 2009 as the third fastest supercomputer worldwide and the third machine with a theoretical peak performance of one PFlop/s, Jugene still is the world’s largest supercomputer in terms of number of cores. Therefore it provides the perfect environment for scaling tests over the maximal currently possible range.

The Jugene provides 73,728 quad-core Power 450 processor located in 72 racks. Each Power 450 processor running at 850 MHz delivers 13.6 GFlop/s peak performance, summing up to the theoretical peak performance of 1 PFlop/s for the whole system and achieving 825.5 TFlop/s for the LINPACK benchmark. Each processor is attached to a 8 MB cache and 2 GB of main memory, resulting in 144 TB of aggregated main memory for the overall system. The maximum achievable bandwidth of the main memory is 13.6 GB/s. The entire system is connected by different networks. For general-purpose, point-to-point and multicast operations, a torus network is used, which interconnects all compute nodes. Each node in the torus has six nearest-neighbor connections with a target hardware bandwidth of 425 MB/s in each direction for a total of 5.1 GB/s bidirectional bandwidth per node. This torus network also supports virtual cut-through hardware routing. Up to 256 compute nodes, the processors can only be connected in a mesh structure. A low power consumption compared to other supercomputers, the high-speed interconnect and the scalable design makes the Blue Gene architecture one of the most successful contemporary supercomputing architectures.
The SelectType class template is a template metaprogramming class to select one of two given types depending on a boolean compile time constant expression. The implementation is based on a class template and a single specialization:

```
template< bool Select, typename T1, typename T2 >
struct SelectType
{
    typedef T1 Type; // The selected type.
};

template< typename T1, typename T2 >
struct SelectType<false,T1,T2>
{
    typedef T2 Type; // The selected type.
};
```

In case the first given template argument `Select` is `true`, the base template is selected, which has a nested type definition for the second template argument (the first of the two given types to select from). In case `Select` is `false`, the specialization is selected, which defines a nested type for the third template argument (the second of the two given types). Note that this class merely selects one of two given types. It is therefore not necessary for the compiler to instantiate any of the given types.
B The SelectType class template
C The MathTrait class template

This chapter explains in detail the implementation of the MathTrait class template as it is used in the math library of the pe (see Chapter 8). The MathTrait class template offers the possibility to select the resulting data type of a generic mathematical operation. Listing C.1 demonstrates the use of MathTrait for the addition of two values. Via MathTrait it is possible to specify the return type of the add function without knowledge about the two given types T1 and T2:

Listing C.1: Application of the MathTrait class template

```
1 template< typename T1, typename T2 > // The type of the left operand
2    typename MathTrait<T1,T2>::AddType // The type of the right operand
3    add( T1 t1, T2 t2 ) // The resulting generic return type
4    {
5        return t1 + t2; // The function 'add' returns the sum of the two given values
6    }
```

Per default, the MathTrait template provides specializations for all built-in data types (except void and bool, which are both not considered numeric data types, but instead including std::size_t and std::ptrdiff_t for several compilers) and all mathematical classes of the pe. Specifying the resulting data type for a specific operation is done by specializing the MathTrait template for this particular type combination. In case a certain type combination is not defined in a MathTrait specialization, the base template is selected, which is left undefined and therefore stops the compilation process:

Listing C.2: Undefined base template of the MathTrait class

```
1 template< typename T1, typename T2 >
2 struct MathTrait;
```

Each specialization of MathTrait defines the data types HighType that represents the high-order data type of the two given data types and LowType that represents the low-order data type. Additionally, each specialization defines the types AddType, SubType, MultType, and
DivType, that represent the type of the resulting data type of the corresponding mathematical operation. The following example shows the specialization for operations between the double and the integer type:

Listing C.3: Specialization of the MathTrait class template

```cpp
template <> struct MathTrait< double, int >
{
  typedef double HighType;
  typedef int LowType;
  typedef double AddType;
  typedef double SubType;
  typedef double MultType;
  typedef double DivType;
};
```

In case of operations between built-in data types, the MathTrait class defines the more significant data type as the resulting data type. For this selection, signed data types are given a higher significance. It is also possible to specialize the MathTrait template for additional user-defined data types, such as vectors and matrices. However, it is possible that a specific mathematical operation is invalid for the particular type combination. In this case, the INVALID_NUMERICAL_TYPE can be used to fill the missing type definition\(^1\). The INVALID_NUMERICAL_TYPE represents the resulting data type of an invalid numerical operation. It is left undefined to stop the compilation process in case it is instantiated. The following example shows the specialization of the MathTrait template for MatrixMxN and VectorN. In this case, only the multiplication between the matrix and the vector is a valid numerical operation. Therefore for all other types the INVALID_NUMERICAL_TYPE is used.

Listing C.4: Specialization of the MathTrait class template

```cpp
template< typename T1, typename T2 >
struct MathTrait< MatrixMxN<T1>, VectorN<T2> >
{
  // Invalid, no common high data type
  typedef INVALID_NUMERICAL_TYPE HighType;

  // Invalid, no common low data type
  typedef INVALID_NUMERICAL_TYPE LowType;

  // Invalid, cannot add a matrix and a vector
  typedef INVALID_NUMERICAL_TYPE AddType;

  // Invalid, cannot subtract a vector from a matrix
  typedef INVALID_NUMERICAL_TYPE SubType;

  // Multiplication between a matrix and a vector
  typedef VectorN< typename MathTrait<T1, T2>::MultType > MultType;

  // Invalid, cannot divide a matrix by a vector
  typedef INVALID_NUMERICAL_TYPE DivType;
};
```

\(^1\)Instead of using the INVALID_NUMERICAL_TYPE it is also possible to leave a certain type undefined. However, using the INVALID_NUMERICAL_TYPE usually results, depending on the compiler, in clearer, more expressive error messages due to the explicit reference to the INVALID_NUMERICAL_TYPE.
Note the recursive instantiation of `MathTrait` for the definition of the data type resulting from a multiplication between a matrix and a vector. Both classes are defined as templates to enable arbitrary data types as elements:

**Listing C.5:** Declarations of the `Matrix` and `Vector` classes

```cpp
1 template< typename T > class MatrixMxN;
2 template< typename T > class VectorN;
```

Therefore it is possible to combine matrices and vectors with different element types. In the simplest case, a matrix of `double` values might be multiplied with a vector of `int` values. In this case, the resulting data type would be a vector of `double` values:

**Listing C.6:** Mixed-type matrix-vector multiplication

```cpp
1 MatrixMxN<double> A;
2 VectorN<int> v;
3 VectorN<double> x = A * v;
```

A more complex example might involve a matrix of matrices of `float` values and a vector of vectors of `long` values:

**Listing C.7:** Mixed-type matrix-vector multiplication

```cpp
1 MatrixMxN< MatrixMxN<float> > A;
2 VectorN< VectorN<long> > v;
3 VectorN< VectorN<float> > x = A * v;
```

Still the operation is well defined and the correct return type of the multiplication can be evaluated by the recursive use of the `MathTrait` class template.
C The MathTrait class template
This section contains further performance comparisons between the *pe* math library and Boost uBLAS. It presents a complete evaluation of all possible operations between dense and sparse vectors and matrices to demonstrate the suitability and efficiency of the smart expression template approach. For all measurements, two different vector and matrix sizes were chosen to demonstrate the performance for both small and large vectors and matrices. The following data types from the Boost uBLAS library were used: `vector` for dense vectors, `matrix` for dense matrices, `compressed_vector` for sparse vectors, and `compressed_matrix` for sparse matrices. The according data types of the *pe* are `VectorN` for dense vectors, `MatrixMxN` for dense matrices, `SparseVectorN` for sparse vectors, and `SparseMatrixMxN` for sparse matrices. The test machine for all performance tests was an Intel Core i7 940 CPU at 2.93 GHz (Bloomfield core) with 8 MByte of shared level three cache. All executables are compiled with the GNU G++ 4.4.1 compiler (branch revision 150839). The data type of the vector and matrix elements is `double` for all performance tests. I used Boost 1.39 for all performance comparisons and the applied BLAS library is ATLAS in the version 3.9.17. All performance results will be presented normalized to the fastest contestant to simplify the comparison.

A general observation of the performance results is that the *pe* always performs better than the Boost uBLAS library (except for the dense matrix-dense matrix multiplication with small matrices). For all operations between dense data structures, the performance difference between *pe* and uBLAS is either very small or within reasonable bounds. However, all operations involving a sparse data structure exhibit a tremendous performance advantage for the *pe*. The reason for this is based in the general expression template formulation of uBLAS as well as in the optimization efforts of the *pe*.
D Performance Results of the pe Math Library

Dense Vector-Dense Vector Addition/Subtraction

Figure D.1: Performance comparison between the pe and Boost uBLAS for the dense vector-dense vector addition/subtraction.

Dense Vector-Sparse Vector Addition/Subtraction

Figure D.2: Performance comparison between the pe and Boost uBLAS for the dense vector-sparse vector addition/subtraction.

Sparse Vector-Dense Vector Addition/Subtraction

Figure D.3: Performance comparison between the pe and Boost uBLAS for the sparse vector dense vector addition/subtraction.

Sparse Vector-Sparse Vector Addition/Subtraction

Figure D.4: Performance comparison between the pe and Boost uBLAS for the sparse vector-sparse vector addition/subtraction.
Dense Vector-Scalar Multiplication

Figure D.5: Performance comparison between the pe and Boost uBLAS for the multiplication of a dense vector and a scalar.

Sparse Vector-Scalar Multiplication

Figure D.6: Performance comparison between the pe and Boost uBLAS for the multiplication of a sparse vector and a scalar.

Dense Matrix-Dense Vector Multiplication

Figure D.7: Performance comparison between the pe and Boost uBLAS for the dense matrix-dense vector multiplication.

Dense Matrix-Sparse Vector Multiplication

Figure D.8: Performance comparison between the pe and Boost uBLAS for the dense matrix-sparse vector multiplication.
D Performance Results of the \texttt{pe} Math Library

**Sparse Matrix-Dense Vector Multiplication**

Figure D.9: Performance comparison between the \texttt{pe} and Boost uBLAS for the sparse matrix-dense vector multiplication.

**Sparse Matrix-Sparse Vector Multiplication**

Figure D.10: Performance comparison between the \texttt{pe} and Boost uBLAS for the sparse matrix-sparse vector multiplication.

**Dense Matrix-Dense Matrix Addition/Subtraction**

Figure D.11: Performance comparison between the \texttt{pe} and Boost uBLAS for the dense matrix-dense matrix addition/subtraction.

**Dense Matrix-Sparse Matrix Addition/Subtraction**

Figure D.12: Performance comparison between the \texttt{pe} and Boost uBLAS for the dense matrix-sparse matrix addition/subtraction.

**Sparse Matrix-Dense Matrix Addition/Subtraction**

Figure D.13: Performance comparison between the \texttt{pe} and Boost uBLAS for the sparse matrix-dense matrix addition/subtraction.
Sparse Matrix-Sparse Matrix Addition/Subtraction

Figure D.14: Performance comparison between the pe and Boost uBLAS for the sparse matrix-sparse matrix addition/subtraction.

Dense Matrix-Scalar Multiplication

Figure D.15: Performance comparison between the pe and Boost uBLAS for the multiplication of a dense matrix and a scalar.

Sparse Matrix-Scalar Multiplication

Figure D.16: Performance comparison between the pe and Boost uBLAS for the multiplication of a sparse matrix and a scalar.

Dense Matrix-Dense Matrix Multiplication

Figure D.17: Performance comparison between the pe and Boost uBLAS for the dense matrix-dense matrix multiplication.

Dense Matrix-Sparse Matrix Multiplication

Figure D.18: Performance comparison between the pe and Boost uBLAS for the dense matrix-sparse matrix multiplication.
D Performance Results of the \textit{pe} Math Library

**Sparse Matrix-Dense Matrix Multiplication**

![Graph](image1)

**Figure D.19:** Performance comparison between the \textit{pe} and Boost uBLAS for the sparse matrix-dense matrix multiplication.

**Sparse Matrix-Sparse Matrix Multiplication**

![Graph](image2)

**Figure D.20:** Performance comparison between the \textit{pe} and Boost uBLAS for the sparse matrix-sparse matrix multiplication.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Two examples of small-grained granular media.</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Medium-sized granular media simulation of 972,000 spherical particles.</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Small-sized of an opening cube containing 344,960 spherical particles.</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>Granular media simulation of 27,270 randomly generated non-spherical particles.</td>
<td>8</td>
</tr>
<tr>
<td>1.5</td>
<td>Coupled LBM/rigid body fluidization scenario (Courtesy of Jan Götz).</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>Segregation simulation of 242,200 spheres (Courtesy of Jan Götz).</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Example simulation of the classical Newton’s cradle scenario.</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Simulation of a wrecking ball destroying a wall.</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Simulation of a rigid body generator.</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Simulation of a well containing 5,000 tri-star unions.</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Simulation of a long metal chain.</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Simulation of the destruction of a castle.</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Large-scale simulation of 500,000 spheres and boxes falling into a well.</td>
<td>17</td>
</tr>
<tr>
<td>6.1</td>
<td>Application of the sandwich pattern to the RigidBody base class.</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Multiple customization layers.</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>Application of the Sandwich Pattern to the RigidBody class hierarchy.</td>
<td>70</td>
</tr>
<tr>
<td>8.1</td>
<td>Expression template instantiation graph for the addition of three vectors.</td>
<td>99</td>
</tr>
<tr>
<td>8.2</td>
<td>Performance comparison: Operator overloading vs. expression templates.</td>
<td>100</td>
</tr>
<tr>
<td>8.3</td>
<td>Performance comparison of matrix-vector multiplication implementations.</td>
<td>102</td>
</tr>
<tr>
<td>8.4</td>
<td>Performance of the pe and Boost for the complex expression $A \cdot (a + b)$.</td>
<td>114</td>
</tr>
<tr>
<td>8.5</td>
<td>Performance of the pe and Boost for the complex expression $A \cdot (a + b + c)$.</td>
<td>115</td>
</tr>
<tr>
<td>8.6</td>
<td>Performance of the pe and Boost for the complex expression $(A \cdot B) \cdot (a + b)$.</td>
<td>116</td>
</tr>
<tr>
<td>8.7</td>
<td>Performance of the pe and Boost for the complex expression $(A \cdot B) + C$.</td>
<td>116</td>
</tr>
<tr>
<td>8.8</td>
<td>Performance of the pe and Boost for the complex expression $(A + B) \cdot (C - D)$.</td>
<td>117</td>
</tr>
<tr>
<td>9.1</td>
<td>Simulation of 500,000 spheres and boxes falling into a well.</td>
<td>124</td>
</tr>
<tr>
<td>9.2</td>
<td>Simulation of a granular medium consisting of 108,570 spherical particles.</td>
<td>124</td>
</tr>
<tr>
<td>9.3</td>
<td>Partitioning of the MPI processes used in the granular medium simulation.</td>
<td>125</td>
</tr>
<tr>
<td>9.4</td>
<td>Detailed domain partitioning illustration.</td>
<td>125</td>
</tr>
<tr>
<td>9.5</td>
<td>Plane-shaped process boundary between the local and a remote process.</td>
<td>126</td>
</tr>
<tr>
<td>9.6</td>
<td>Plane-shaped process boundary between two processes.</td>
<td>126</td>
</tr>
<tr>
<td>9.7</td>
<td>Combination of two plane-shaped boundaries between two processes.</td>
<td>128</td>
</tr>
<tr>
<td>9.8</td>
<td>Connection of diagonally neighboring processes.</td>
<td>129</td>
</tr>
<tr>
<td>9.9</td>
<td>Extension of the process neighborhood for large rigid bodies.</td>
<td>129</td>
</tr>
<tr>
<td>9.10</td>
<td>Setup of a simulation of two MPI processes.</td>
<td>131</td>
</tr>
</tbody>
</table>
## List of Figures

9.11 Example of a 64-bit unique rigid body ID ........................................... 132
9.12 Illustration of two MPI rigid body messages ........................................ 134

10.1 Schematic illustration of the parallel FFD algorithm ................................. 142
10.2 Collision detection on remote processes ............................................... 143
10.3 Example of the communication of a single rigid body ............................... 144
10.4 Simulation scenario for the strong and weak scaling experiments ............... 147
10.5 Serial performance results .................................................................. 148
10.6 Strong scaling on Jugene with 2,188,232 spherical particles .................... 149

11.1 Illustration of the cube scenario with 344,960 spherical particles ............. 157
11.2 Simulation of 720,484 spherical particles rolling down a spiral staircase .... 158
11.3 Partitioning of the simulation domain in the staircase scenario ................ 159
11.4 Simulation of 27,270 randomly generated non-spherical particles .......... 160
11.5 Detailed illustration of the domain partitioning ....................................... 160
11.6 The silo scenario with no inclination of the bottom walls ....................... 162
11.7 The silo scenario with an inclination of 11.25° ...................................... 162
11.8 The silo scenario with an inclination of 22.5° ....................................... 163
11.9 The silo scenario with an inclination of 33.75° ...................................... 163
11.10 The silo scenario with an inclination of 45° ........................................ 164
11.11 The silo scenario with an inclination of 56.25° .................................... 164

A.1 The HLRB-II supercomputer at the Leibnitz computing center Munich (LRZ) 173
A.2 The Jugene supercomputer at the Jülich Supercomputing Center (JSC) ... 174

D.1 Performance of the dense vector-dense vector addition/subtraction ........ 182
D.2 Performance of the dense vector-sparse vector addition/subtraction ....... 182
D.3 Performance of the sparse vector-dense vector addition/subtraction ...... 182
D.4 Performance of the sparse vector-sparse vector addition/subtraction ....... 182
D.5 Performance of the multiplication of a dense vector and a scalar .......... 183
D.6 Performance of the multiplication of a sparse vector and a scalar .......... 183
D.7 Performance of the dense matrix-dense vector multiplication ............... 183
D.8 Performance of the dense matrix-sparse vector multiplication ............... 183
D.9 Performance of the sparse matrix-dense vector multiplication ............... 184
D.10 Performance of the sparse matrix-sparse matrix multiplication .......... 184
D.11 Performance of the dense matrix-dense matrix addition/subtraction ...... 184
D.12 Performance of the dense matrix-sparse matrix addition/subtraction .... 184
D.13 Performance of the sparse matrix-dense matrix addition/subtraction .... 184
D.14 Performance of the sparse matrix-sparse matrix addition/subtraction .... 184
D.15 Performance of the multiplication of a dense matrix and a scalar .......... 185
D.16 Performance of the multiplication of a sparse matrix and a scalar ......... 185
D.17 Performance of the dense matrix-dense matrix multiplication ............. 185
D.18 Performance of the dense matrix-sparse matrix multiplication ............. 185
D.19 Performance of the sparse matrix-dense matrix multiplication .......... 186
D.20 Performance of the sparse matrix-sparse matrix multiplication .......... 186
List of Tables

7.1 Performance comparison for different (smart) pointers .......................... 82

10.1 Dependencies of the MPI communication steps ................................. 144
10.2 Comparison between LCP-based solvers and the FFD solver .................. 146
10.3 Strong scaling experiment with 2 188 232 spherical particles ................. 149
10.4 Weak scaling experiment with up to 142 500 000 spherical particles ....... 151
10.5 Weak scaling experiment with up to 1 140 000 000 spherical particles .... 152
10.6 PFFD runtime ratios for 1D, 2D, and 3D-partitioning ......................... 152
10.7 Weak scaling experiment with up to 2 048 000 000 spherical particles .... 153
10.8 Weak scaling experiment with up to 2 048 000 000 non-spherical particles 153
10.9 Measurement of the energy deviation ........................................... 154

11.1 PFFD runtime ratios for the silo scenario ...................................... 161
Listings

3.1 First example of a compile time error .................................. 23
3.2 Second example of a compile time error ................................. 24
3.3 Proper use of static assertions ........................................... 24
3.4 First approach for static assertions ..................................... 25
3.5 Improvement of the first approach for static assertions .......... 25
3.6 Second approach for static assertions ................................ 26
3.7 Third approach for static assertions .................................. 26
3.8 Fourth approach for static assertions .................................. 26
3.9 The Boost static assertion .................................................. 27
3.10 Instantiation of the STATIC_ASSERT_FAILURE template .......... 27
3.11 First example of bad error messages ................................... 28
3.12 Second example of bad error messages ................................ 28
3.13 Example for a pe compile time constraint ......................... 29
3.14 Application of a pe compile time constraint ....................... 30
3.15 The MUST_BE_BASE_OF constraint .................................... 32
3.16 The MUST_BECTSTRRICTLY_BASE_OF constraint ................. 32
3.17 Implementation of the MUST_BE_BASE constraints ............. 32
3.18 The MUST_BE_SAME_TYPE constraints ............................... 33
3.19 The IsSame type traits ..................................................... 33
3.20 Implementation of the MUST_BE_SAME_SIZE constraints ....... 34
4.1 First example of the use of a pe smart scope ....................... 35
4.2 Second example of the use of a pe smart scope ..................... 36
4.3 Implementation concept of the smart scopes ....................... 36
4.4 Unexpected behavior of a pe_EXCLUSIVE_SECTION (I) .......... 37
4.5 Unexpected behavior of a pe_EXCLUSIVE_SECTION (II) ......... 38
4.6 Unexpected behavior of a pe_EXCLUSIVE_SECTION (III) ....... 38
4.7 Application of the pe_EXCLUSIVE_SECTION ....................... 38
4.8 Default implementation of behavioral differentiation .......... 39
4.9 Default implementation of behavioral differentiation .......... 39
4.10 Recommended implementation of behavioral differentiation .... 40
4.11 Implementation of the pe_EXCLUSIVE_SECTION macro ........ 40
4.12 Implementation of the ExclusiveSection class ..................... 40
4.13 Application of the isActive function inside the simulationStep function .......... 42
4.14 Implementation of the pe_GLOBAL_SECTION macro ........... 42
4.15 Implementation of the pe_GLOBAL_SECTION macro ........... 43
4.16 Implementation of the pe_GLOBAL_SECTION macro ........... 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>First impression of the <em>pe</em> logging tool</td>
</tr>
<tr>
<td>5.2</td>
<td>Global configuration of the logging level</td>
</tr>
<tr>
<td>5.3</td>
<td>Log file resulting from the non-parallel example</td>
</tr>
<tr>
<td>5.4</td>
<td>Example for two nested log sections</td>
</tr>
<tr>
<td>5.5</td>
<td>Log file resulting from the nested log sections example</td>
</tr>
<tr>
<td>5.6</td>
<td>Application of the <em>pe</em> logging tool in a threaded environment</td>
</tr>
<tr>
<td>5.7</td>
<td>Log file resulting from the threaded example</td>
</tr>
<tr>
<td>5.8</td>
<td>Manually committing into the log file</td>
</tr>
<tr>
<td>5.9</td>
<td>Resulting log file for manually committing</td>
</tr>
<tr>
<td>5.10</td>
<td>Definition of the Logger class</td>
</tr>
<tr>
<td>5.11</td>
<td>Implementation of the log function</td>
</tr>
<tr>
<td>5.12</td>
<td>Implementation of the <em>instance</em> function</td>
</tr>
<tr>
<td>5.13</td>
<td>Definition of the LogSection class</td>
</tr>
<tr>
<td>5.14</td>
<td>The destructor of the LogSection class</td>
</tr>
<tr>
<td>5.15</td>
<td>The destructor of the LogSection class</td>
</tr>
<tr>
<td>5.16</td>
<td>Implementation of the commit function</td>
</tr>
<tr>
<td>5.17</td>
<td>Implementation of the log function</td>
</tr>
<tr>
<td>5.18</td>
<td>Implementation of the output operator</td>
</tr>
<tr>
<td>5.19</td>
<td>Implementation of the copy constructor and the conversion operator</td>
</tr>
<tr>
<td>5.20</td>
<td>Implementation of the <strong>pe_LOG_ERROR_SECTION</strong> macro</td>
</tr>
<tr>
<td>5.21</td>
<td>Illustration of the preprocessor alternative</td>
</tr>
<tr>
<td>5.22</td>
<td>Scope of the <strong>pe_LOG_ERROR_SECTION</strong></td>
</tr>
<tr>
<td>5.23</td>
<td>Copy initialization of a LogSection object</td>
</tr>
<tr>
<td>5.24</td>
<td>Order of construction and destruction of global/static objects</td>
</tr>
<tr>
<td>5.25</td>
<td>Resulting output</td>
</tr>
<tr>
<td>5.26</td>
<td>Implementation of the <em>instance</em> function based on <em>shared_ptr</em></td>
</tr>
<tr>
<td>5.27</td>
<td>Initialization of the MPI communication system via the <strong>MPI_Init</strong> function</td>
</tr>
<tr>
<td>6.1</td>
<td>Requirements of LCP-based collision solvers</td>
</tr>
<tr>
<td>6.2</td>
<td>Requirements of the PFFD collision solver</td>
</tr>
<tr>
<td>6.3</td>
<td>Naive approach to integrate the requirements of numerous algorithms</td>
</tr>
<tr>
<td>6.4</td>
<td>Template-based implementation of the Rigibdy class</td>
</tr>
<tr>
<td>6.5</td>
<td>Refactoring of the template-based Rigibdy class</td>
</tr>
<tr>
<td>6.6</td>
<td>Application of the Sandwich Pattern to the Rigibdy base class</td>
</tr>
<tr>
<td>6.7</td>
<td>Non-template implementation of Algorithm</td>
</tr>
<tr>
<td>6.8</td>
<td>Template-based implementation of Algorithm</td>
</tr>
<tr>
<td>6.9</td>
<td>Application of the Sandwich Pattern to classes derived from Rigibdy</td>
</tr>
<tr>
<td>6.10</td>
<td>Compile time selection of rigid body algorithms</td>
</tr>
<tr>
<td>6.11</td>
<td>Definition of the Configuration class</td>
</tr>
<tr>
<td>6.12</td>
<td>Application of the Sandwich Pattern to the Rigibdy class hierarchy</td>
</tr>
<tr>
<td>6.13</td>
<td>Specialization of RigibdyTrait class for the FFDSolver</td>
</tr>
<tr>
<td>6.14</td>
<td>Specialization of <em>SphereTrait</em> class for the FFDSolver</td>
</tr>
<tr>
<td>7.1</td>
<td>Example for a poor interface design (I)</td>
</tr>
<tr>
<td>7.2</td>
<td>Example for a poor interface design (II)</td>
</tr>
<tr>
<td>7.3</td>
<td>Interface design of the <em>pe</em> framework</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7.4</td>
<td>Example for plain pointer handles</td>
</tr>
<tr>
<td>7.5</td>
<td>Example for <code>shared_ptr</code> handles</td>
</tr>
<tr>
<td>7.6</td>
<td>Benchmark program for the performance penalty of smart pointers</td>
</tr>
<tr>
<td>7.7</td>
<td>Illustration of the <code>pe</code> ID concept</td>
</tr>
<tr>
<td>7.8</td>
<td>Extensions of the <code>RigidBody</code> class (I)</td>
</tr>
<tr>
<td>7.9</td>
<td>Class definition of the <code>BodyManager</code> base class</td>
</tr>
<tr>
<td>7.10</td>
<td>Implementation of the <code>destroy</code> function</td>
</tr>
<tr>
<td>7.11</td>
<td>Implementation of the <code>theDefaultManager</code> function</td>
</tr>
<tr>
<td>7.12</td>
<td>Extensions of the <code>RigidBody</code> class (II)</td>
</tr>
<tr>
<td>7.13</td>
<td>Implementation of the <code>createSphere</code> function</td>
</tr>
<tr>
<td>7.14</td>
<td>Implementation of the <code>destroy</code> function</td>
</tr>
<tr>
<td>7.15</td>
<td>Class definition of the <code>World</code> class</td>
</tr>
<tr>
<td>7.16</td>
<td>Implementation of the <code>World::add</code> function</td>
</tr>
<tr>
<td>7.17</td>
<td>Automatic destruction of all rigid bodies</td>
</tr>
<tr>
<td>7.18</td>
<td>Implementation of the <code>theWorld</code> function</td>
</tr>
<tr>
<td>8.1</td>
<td>Signature of the BLAS function for the matrix-matrix multiplication</td>
</tr>
<tr>
<td>8.2</td>
<td>Usage of the BLAS matrix-matrix multiplication</td>
</tr>
<tr>
<td>8.3</td>
<td>Matrix-matrix multiplication using the Boost uBLAS library</td>
</tr>
<tr>
<td>8.4</td>
<td>Mixed type matrix-matrix multiplication using the Boost uBLAS library</td>
</tr>
<tr>
<td>8.5</td>
<td>Basic implementation of a vector class</td>
</tr>
<tr>
<td>8.6</td>
<td>Addition operator for the addition of two vectors</td>
</tr>
<tr>
<td>8.7</td>
<td>Application of NRV during copy construction and copy assignment</td>
</tr>
<tr>
<td>8.8</td>
<td>Standard implementation of the vector addition operator</td>
</tr>
<tr>
<td>8.9</td>
<td>Compiler generated code for the copy assignment of vectors</td>
</tr>
<tr>
<td>8.10</td>
<td>Concatenation of multiple vector additions</td>
</tr>
<tr>
<td>8.11</td>
<td>C-like implementation of multiple vector additions</td>
</tr>
<tr>
<td>8.12</td>
<td>Expression template formulation of the vector addition</td>
</tr>
<tr>
<td>8.13</td>
<td>Expression template assignment operator</td>
</tr>
<tr>
<td>8.14</td>
<td>Expression template optimized addition of multiple vectors</td>
</tr>
<tr>
<td>8.15</td>
<td>Compiler generated temporary objects during the addition of three vectors</td>
</tr>
<tr>
<td>8.16</td>
<td>Imperfection of the general expression template formulation</td>
</tr>
<tr>
<td>8.17</td>
<td>Final expression template formulation of the vector addition</td>
</tr>
<tr>
<td>8.18</td>
<td>Complex matrix-vector multiplication</td>
</tr>
<tr>
<td>8.19</td>
<td>Smart expression object for the matrix-vector multiplication</td>
</tr>
<tr>
<td>8.20</td>
<td>Composite type of the smart expression object</td>
</tr>
<tr>
<td>8.21</td>
<td>Type trait class for expressions</td>
</tr>
<tr>
<td>8.22</td>
<td>Compiler generated temporary objects for the matrix-vector multiplication</td>
</tr>
<tr>
<td>8.23</td>
<td>Aliasing problem for a matrix-vector multiplication</td>
</tr>
<tr>
<td>8.24</td>
<td>Explicit resolution of aliasing problems</td>
</tr>
<tr>
<td>8.25</td>
<td>Member function to detect aliasing effects</td>
</tr>
<tr>
<td>8.26</td>
<td>Implementation of the <code>isAliased</code> function in the <code>Vector</code> class</td>
</tr>
<tr>
<td>8.27</td>
<td>Smart expression template assignment operator of the <code>Vector</code> class</td>
</tr>
<tr>
<td>8.28</td>
<td>Smart expression template assignment operator of the <code>Matrix</code> class</td>
</tr>
<tr>
<td>8.29</td>
<td>Default implementation of the <code>free assign</code> function</td>
</tr>
<tr>
<td>8.30</td>
<td>Default implementation of the assignment of a dense matrix</td>
</tr>
</tbody>
</table>

193
# List of Algorithms

1. MPI communication .......................... 135
2. The FFD-Algorithm ............................ 140
<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Description</th>
</tr>
</thead>
</table>

196
Supervised Student Theses


Bibliography


200


Bibliography


[99] B. Stroustrup. The Design of C++0x. May 2005.


