TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Engineering and Design

CoSimulation and Mapping for large scale engineering applications

Philipp Lukas Bucher

Vollständiger Abdruck der von der TUM School of Engineering and Design der Technischen Universität München zur Erlangung eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Die Dissertation wurde am 11.07.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Engineering and Design am 11.03.2024 angenommen.

Schriftenreihe des Lehrstuhls für Statik
TU München

Band 63

Philipp Bucher

CoSimulation and Mapping for large scale engineering applications

München 2023

Veröffentlicht durch

Kai-Uwe Bletzinger
Lehrstuhl für Statik
Technische Universität München
Arcisstr. 21
80333 München

Telefon: +49(0)89 289 22422
Telefax: +49(0)89 289 22421
E-Mail: kub@tum.de
Internet: www.cee.ed.tum.de/st

ISBN: 978-3-943683-76-9

Schmerzen vergisst du,
aufgeben nie!
(Motto, von *triathlon.de*)

Abstract

This work presents solutions for coupled simulations (also referred to as *Co-Simulation*) of large-scale engineering applications. The partitioned approach is employed, which enables to use existing and mature solvers. A multiphysics framework is chosen as the basis for the developments, differing in many ways from CoSimulation via dedicated coupling tools. One advantage is the very efficient usage of internal solvers. The details of the implementation and integration are shown, including the requirements for distributed memory environments.

External solvers can be used by a newly developed detached interface technology, which allows for efficient and flexible integration. Additionally, a remote-controlled approach for CoSimulation is presented, in which the coupling is orchestrated centrally. This approach simplifies the setting up of different coupling algorithms and is less prone to deadlocking compared to existing solutions. A detailed study of methods for data exchange between solvers/tools is conducted, resulting in TCP sockets as the best technique.

Mapping algorithms are studied, and solutions for large-scale applications are proposed. The developments are successfully tested on a supercomputer with over 10,000 cores and applied to the examples presented later.

Practical experiences are presented for approaching, setting up, and running coupled simulations. They were gathered and systematically structured during this work. A particular focus is set on using cluster systems for running large-scale CoSimulation.

Numerous examples, ranging from academic benchmarks to large-scale engineering applications, are used to highlight and showcase the capabilities of the presented work. The culmination of this work is a Fluid-Structure interaction simulation with the roof of the Olympic Stadium in Munich.

Finally, conclusions are provided, summarizing the highlights of this work. An outlook with possible future work is shown.

Kurzfassung

Diese Arbeit beschäftigt sich mit Lösungen für gekoppelte Simulationen von großen Ingenieuranwendungen. Der partitionierte Ansatz wird angewandt, welcher die Nutzung von bestehenden und ausgereiften Lösern ermöglicht. Ein Mehrfeldphysik-Framework bildet die Basis für die Entwicklungen, was sich in vielerlei Hinsicht von gekoppelten Simulationen mittels spezialisierten Kopplungstools unterscheidet. Ein Vorteil ist die sehr effiziente Nutzung von internen Lösern. Details der Implementierung und Integration werden vorgestellt, inklusive der Anforderungen für Rechensysteme mit verteiltem Speicher.

Externe Löser können mithilfe einer neu entwickelten losgelösten Schnittstelle genutzt werden, die eine sehr effiziente und flexible Integration ermöglicht. Weiterhin wird ein ferngesteuerter Ansatz für gekoppelte Simulationen vorgestellt, bei dem die Kopplung zentral gesteuert wird. Dieser Ansatz vereinfacht das Einrichten verschiedener Kopplungsalgorithmen und führt zu weniger Deadlocks, verglichen mit existierenden Methoden. Umfangreiche Untersuchungen bezüglich des Datenaustauschs zwischen verschiedenen Lösern wird durchgeführt, wobei sich TCP-Sockets als die beste Wahl erweisen.

Mapping-Algorithmen werden untersucht und Lösungsansätze für große Probleme werden aufgezeigt. Die Entwicklungen sind auf einem Supercomputer mit über 10.000 Kernen erfolgreich getestet sowie bei den später folgenden Beispielen angewandt.

Praktische Erfahrungen in Bezug auf Herangehensweise, Aufsetzen und Durchführung von gekoppelten Problemen sind dargestellt. Diese wurden während der Durchführung dieser Arbeit gesammelt und systematisch aufbereitet. Ein Fokus liegt hierbei insbesondere auf der Durchführung von großen gekoppelten Problemen auf Rechenclustern.

Die Fähigkeiten und Möglichkeiten der vorgestellten Methoden sind anhand mehrerer Beispiele gezeigt, von wissenschaftlichen Benchmarks bis zu großen Ingenieuranwendungen. Den Höhepunkt der Arbeit stellt eine gekoppelte Fluid-Struktur-Interaktionssimulation mit dem Dach des Olympiastadions in München dar.

Abschließend werden die wichtigsten Ergebnisse und Errungenschaften dieser Arbeit zusammengefasst. Weiterhin ist ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

Acknowledgements

This work is by no means only my achievement. Without the support of colleagues, friends, and family, I would not have finished it successfully.

I want to thank my supervisor Prof. Dr.-Ing. Kai-Uwe Bletzinger for providing me with the opportunity to conduct this work at his institute. Many thanks also to Prof. Dr.-Ing. habil. Roland Wüchner, for mentoring me over a long time, starting with my master studies. Especially at times when I thought my work went nowhere.

My colleagues also deserve a big thank you, starting with my office-mates Andreas and Mátè. Furthermore, thanks to Aditya, Iñigo, Klaus, and Tobias for many years of fruitful collaboration and lots of fun.

Giorgia, thanks for sharing the pain and suffering that we had to endure while working on the turbine. Your persistence was no doubt a big help in this!

Of course, the Kratos-community and the Barcelona guys need to be mentioned here, particularly Jordi, Riccardo, Charlie, Ruben, and Pooyan. Thanks for the great and productive discussions we have had over the years, I learned a lot from you!

Finally, I want to thank my family and friends. Without their continuous support, this work would surely not have been finished, possibly never even started. In particular, my partner Anna, who has suffered for many years with me being in my head over the thesis and spending way too much time at work.

I also gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this work (grants pr94va and pn56ba) by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (www.lrz.de).

Philipp Bucher
July 2023

Contents

List of Abbreviations

AI	Artificial Intelligence.
ALE	Arbitrary Lagrangian-Eulerian.
API	Application Programming Interface.
CAD	Computer Aided Design.
CFD	Computational Fluid Dynamics.
CHT	Conjugate Heat Transfer.
CPU	Central Processing Unit.
CSD	Computational Structural Dynamics.
DEM	Discrete Element Method.
DES	Detached Eddy Simulation.
DNS	Direct Numerical Simulation.
DOF	Degree of Freedom.
FEM	Finite Element Method.
FSI	Fluid-Structure Interaction.
FVM	Finite Volume Method.
HPC	High Performance Computing.
IPC	Interprocess Communication.
LES	Large Eddy Simulation.
ML	Machine Learning.
MPI	Message Passing Interface.
NNet	Neural Network.
OS	Operating System.
RANS	Reynolds-Averaged Navier-Stokes.
RBF	Radial Basis Function.

SDOF	Single Degree of Freedom.
SSD	Solid State Drive.
TCP	Transmission Control Protocol.
TS	Timestep.
URANS	Unsteady Reynolds-Averaged Navier-Stokes.
VMS	Variational Multiscale.
WSI	Wind-Structure Interaction.
WSL	Windows Subsystem for Linux.

Chapter 1

Introduction

Numerical simulations are a well-established tool in engineering practice nowadays. They help push the boundaries of technology, and some innovations would be impossible without them. The development and usage of simulations have been traditionally focused on standalone physical phenomena, such as structural mechanics for civil engineers or fluid dynamics for aerospace engineers. However, with more ambitious and advanced projects and applications, it is no longer sufficient to neglect the interaction of a component with relevant effects of other physical disciplines. Prominent examples are lightweight structures such as tents subjected to wind loading, where the interaction must be considered for an efficient and safe design.

Simulations considering more than one physical domain are referred to as multi-physics simulations. Many new challenges arise with these applications, from the individual domains and the coupling between them. Furthermore, the computational effort can be a multiple of the individual solution times, depending on several factors such as coupling setup or degree of interaction between the domains. The increased effort requires efficient methods to keep the simulation times within acceptable and practical bounds.

Mainly two approaches exist for solving coupled problems: Monolithic and partitioned. The first combines the individual disciplines within one solution method, whereas the latter combines existing solvers via a dedicated coupling tool. This work employs the latter, as it allows reusing existing solvers. Many single-domain tools have been developed for decades, and their technology is very mature. These consolidated solvers are essential to simulate and predict the behavior of large-scale engineering applications, as is done in this work. Furthermore, coupling various solvers from different physical disciplines offers more flexibility.

Previous works that dealt with partitioned coupled simulations oftentimes developed dedicated tools for bringing together existing solvers, such as [11],

[46], [28] and [81]. While the execution and implementation differ, most share this fundamental concept. In contrast, this work develops coupling features and functionalities within a multiphysics framework, which also features several solvers. This approach has several advantages: better computational efficiency, easier usage because the user only needs one software, and more efficient deployment on High Performance Computing (HPC) systems. Coupling only internal solvers however would limit the applicability, and thus coupling to external tools is also developed. Detailed and systematic comparisons between the approaches are performed, with a representative large-scale engineering example.

The main motivation of this work is to develop methods that enable engineers to conduct large-scale coupled simulations in a robust, stable, efficient, and accurate way. For this, many developments are realized within the scope of this work, enabling the coupling with internal and external solvers of the multiphysics framework. One of the most important ones is integrating the coupling features into the existing and established workflows. Additionally, two new complementary techniques for coupling external solvers are developed. First, a detached interface is proposed to simplify the general data exchange and communication. Secondly, an approach is presented in which the coupling orchestration is done centrally in one place. Finally, mapping algorithms are revisited for large applications, including new developments for distributed systems.

The examples presented in this work demonstrate the capabilities of the developments. They are carefully selected, each highlighting at least one aspect of CoSimulation. The largest examples are Fluid-Structure Interaction (FSI) simulations of a full-scale wind turbine and the roof of the Olympic Stadium in Munich.

One of the goals of this work is to make coupled simulations as accessible as possible. Choosing a multiphysics framework over a dedicated coupling tool has the inherent advantage that it has internal solvers. Those can be used with little additional effort if the framework is already used. Of course, the choice of solver is entirely up to the user, thus coupling to internal and external tools is developed and made possible. Another important point to consider for accessibility is the free choice of computational hardware to conduct the simulations. It should be chosen based on the requirements of the respective simulations and not limited by shortcomings of the employed software. This is particularly important when it comes to parallelization and cross-platform support. Lastly, also the means of distribution are important. Straight-forward ways to obtain and run simulation software enables engineers to integrate numerical analysis into their daily workflows.

The open-source framework Kratos Multiphysics [15] (available on GitHub[1]) fulfills all the above requirements and is thus chosen as the basis for this work. All presented developments are fully integrated and publicly available.

[1] `https://github.com/KratosMultiphysics`

1.1 Contributions of this work

This section highlights novelties, contributions, and advances made in this work. The focus lies in particular on developing methods to enable the stable, robust, accurate, and efficient simulation of large-scale coupled problems, which are relevant in engineering practice.

- The developments of this work for simulating coupled problems are applied to several large-scale engineering applications. These represent real-world problems and are aimed to push the boundaries of modern computational methods in engineering.
- The realization of the newly developed coupling is done within a multiphysics framework. A modular approach makes it possible to exchange, extend and customize the different building blocks of CoSimulation. All developments consider large-scale problems and thus fully support distributed memory architectures and HPC.
- Coupling of the solvers inside the multiphysics framework and coupling to tools not part of this software is presented and compared in detail.
- A new way for CoSimulation with external tools is presented, in which the orchestrator takes full control over the external tools. The external tool provides a predefined set of functions to the coupling tool, allowing it to execute different operations remotely. This approach is titled *remote-controlled CoSimulation*.
- A detached interface is developed, which simplifies the integration of a coupling interface into an external solver greatly. It works in shared and distributed memory environments as well as with different Operating Systems (OSs).
- A detailed study of methods for Interprocess Communication (IPC) in CoSimulation is conducted, with focus on a large range of possible scenarios for coupled simulations. Different OSs are considered, as well as large-scale cases in distributed memory environments.
- Proposing a new generic interface for solvers, which aims to unify the integration into CoSimulation.
- Mapping algorithms for surface-to-surface and volume-to-volume are revisited for the simulation of large-scale coupled problems. In particular efficient strategies for distributed memory environments are investigated, and new concepts are proposed and realized. Their performance is tested with over 10,000 cores on a supercomputer.
- Practical experiences and guidelines for setting up and running coupled problems are presented. Furthermore, dealing with different methods of parallelism of the tools involved in CoSimulation is considered, as well as conducting large-scale coupled simulations on clusters and other HPC systems.

1.2 Outline

This work is subdivided into several chapters. Following is a short outline of each chapter to give an overview and orientation.

Chapter 1 provides the introduction and motivation for this work. An overview of the topic of coupled simulation is provided, as well as a list of contributions and advances made in this work.

Chapter 2 introduces the relevant theoretical basics and background information, which is used in the following chapters. This consists mainly of the building blocks of CoSimulation, the governing equations of some solution techniques, and a review of existing tools for coupled simulations.

Chapter 3 is the core of this work. The details of realizing CoSimulation *in* and *with* a multiphysics framework are presented. A focus is set on realizing large-scale coupled problems efficiently, including HPC systems. A new approach for remote-controlled CoSimulation and its implementation is shown.

Chapter 4 is about mapping for non-matching grids, particularly for large-scale cases. The basics of mapping are revisited, and efficient methods for mapping in distributed memory environments are proposed and systematically assessed. Surface-to-surface and volume-to-volume mapping algorithms are considered.

Chapter 5 consists of a collection of practical experiences for setting up and running coupled simulations. It is the condensed knowledge obtained and developed during this work. The peculiarities of CoSimulation on HPC systems are addressed as well.

Chapter 6 presents a large range of examples from different applications. Academic benchmarks are considered to validate the developments, as well as large-scale engineering problems. They show the flexibility and efficiency of the presented work in handling various real-world applications.

Chapter 7 concludes this work by summarizing the advances and developments, as well as providing an outlook for further work and potential improvements.

Chapter 2

Components and fundamental concepts of CoSimulation

The goal of simulations is to model and predict the behavior of real-world systems. A large variety of tools and techniques exists for this purpose, differing, among others, in accuracy, modeling/simulation and computational effort. Usually, the desired accuracy of the results dictates which technique is used. In the past many tools have been developed and specialized to solve problems involving one physical phenomenon, in this work referred to as *single-physics*. However, the real world is always an interaction between different physical phenomena, and it is a modeling choice to neglect their interaction. Therefore, depending on the desired accuracy of the simulation, it might be required to consider this interaction through different coupling techniques. Problems involving several physical phenomena will be referred to as *multi-physics* in this work, see also [44].

Two fundamental approaches exist for modeling the interaction between different physical phenomena, namely the monolithic and the partitioned approach. In the monolithic approach, all phenomena are solved together simultaneously. With the partitioned approach, the individual phenomena are solved independently and in a specific sequence. Both approaches have their distinct advantages and disadvantages, as has been thoroughly discussed in the literature (see [36] or [19]). The main advantage of the monolithic approach is the higher accuracy than the partitioned approach. Its main disadvantage is that it is difficult or even impossible to reuse existing tools, which makes it necessary to develop new tools for each new type of problem. With the partitioned approach, existing tools are used and coupled together through different methods and techniques, which makes it possible to reuse tools that have been thoroughly developed over many years, and hence most of the development time can be spent on the coupling. The disadvantage is the reduced accuracy, and it is more sensitive to coupling instabilities than the monolithic approach. However, different techniques have been developed to mitigate these inherent disadvantages. Another technical difficulty that practical applications need to consider is how well any coupling can be integrated into a particular tool.

In order to use existing tools, much effort has been invested in the partitioned approach to find solutions to make it robust and accurate enough for real-world applications. This work will focus purely on the partitioned approach for solving multi-physics problems.

This chapter serves as an introduction to solving coupled problems and paints the big picture motivating the further chapters of this work.

2.1 Spectra of CoSimulation

CoSimulation can be done in various ways. An overview of different scenarios of CoSimulation can be found in [77, Chapter 1.2]. A small review alongside an extension of this list is given in this section.

2.1.1 Monolithic - all in one

In the monolithic approach, all physical phenomena are solved together in one large system of equations. This means that it is mostly impossible to reuse existing tools which limits the applicability for practical problems. Still, it is the most stable and accurate approach, which means that it can be a viable option for some problems.

2.1.2 Partitioned solver coupling with dedicated coupling tools

The partitioned approach couples existing tools in a black box way and is hence the other end of the spectrum of CoSimulation. Dedicated coupling tools have been developed for this purpose. The main advantage is that existing tools can be reused, which reduces the development and verification/validation time a lot. However, especially for problems with strong interaction between the fields, special treatment is required in order to ensure stable solutions.

2.1.3 Multiphysics tools

While traditionally tools have been developed for solving single-physics problems, in recent years, also frameworks dedicated to solving multi-physics problems such as [15], [35], [55] have been developed. These tools can combine some advantages of both approaches while at the same time limiting the number of disadvantages. This work focuses on CoSimulation in and with a multi-physics framework.

2.2 Building blocks of CoSimulation

CoSimulation using the partitioned approach consists of several basic building blocks that will be introduced in this chapter and then explained from a practical point of view in the later chapters of this work. Figure 2.1 shows the three main building blocks: synchronization, data exchange, and data transfer.

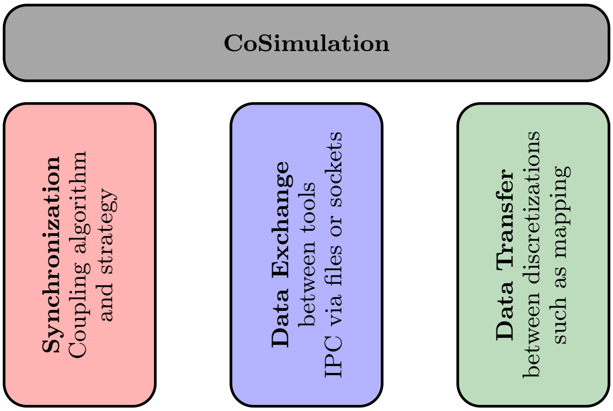


Figure 2.1: The building blocks of partitioned CoSimulation.

2.2.1 Synchronization and solution techniques

The first building block of CoSimulation, as displayed in Figure 2.1, is the synchronization, including the coupling algorithm and strategy. It defines the order in which the participating tools are used, as well as which additional components (such as relaxation techniques) are employed. Naturally, the choice of these procedures has a significant influence on the robustness, accuracy, performance, efficiency, and stability of the coupled simulation and must therefore be chosen according to the problem to be solved.

It shall also be referred to the extensive existing literature on this topic: [60], [49], [29], [81], [77], [46].

2.2.1.1 Question / Challenge

Coupled simulations where the solution of one solver affects the solution of another, and vice versa can be sensitive to stability and convergence problems. This becomes more severe the stronger the interaction becomes, i.e. the more impact the solution of one solver has on the solution of another. The sequence in which the tools are executed can also have an impact on the stability of the coupled solution. Therefore, the coupling algorithm, which dictates this order, should be chosen carefully. Different coupling algorithms with their distinct properties are known in the literature. The aspects that are important for this work will be explained in the following.

2.2.1.2 Methodological Answer

The application and the type of interaction dictates which coupling algorithm to be used. It becomes more complex and elaborate the stronger the interaction becomes. This ranges from one-way coupling in case the influence is only from one side to the other, to strongly two-way coupled with advanced methods to assist in the convergence of the coupled solution, e.g. utilizing relaxation or prediction techniques.

2.2.1.3 Technical Realization

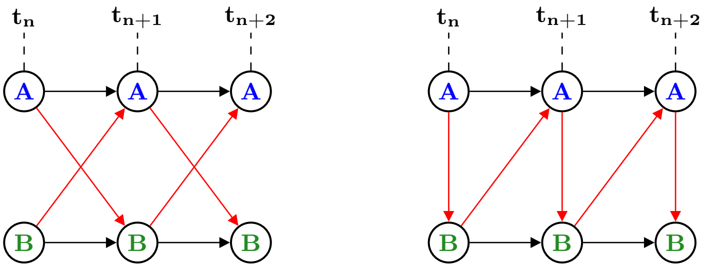
The coupling tool/orchestrator dictates when and which tool executes which part of the coupling. This can be tasks like solving a linear system of equations or the import/export of data. A wide range of techniques to aid in the successfully coupled simulation has been developed. An overview of commonly used methods and techniques is given in the following.

Communication patterns
The communication pattern specifies the flow of information between the coupling partners, in particular, which state of the data is used. The two most used patterns are the Jacobi (see Figure 2.2a) and the Gauss-Seidel pattern (see Figure 2.2b). The main difference is that with the Gauss-Seidel pattern always the newest data is used, whereas, with the Jacobi pattern, the data from the previous timestep or iteration is used. The flow of data is visualized in Figure 2.2.

The Gauss-Seidel pattern converges faster than the Jacobi pattern, but the coupling partners cannot be run at the same time due to the data dependencies. Furthermore, a time lag between the coupling partners exists. On the other hand, with the Jacobi pattern no data dependencies exist, hence the execution can be done concurrently. This also means that the coupling partners have to share the available computing resources. With the staggered execution of the Gauss-Seidel pattern, the coupling partners are executed sequentially and thus do not share the computing resources.

Coupling methods
For a two-way coupling, in which both partners influence each other, two main methods are distinguished: Weak/explicit and strong/implicit coupling. With strong coupling, the convergence of the coupled solution is checked at the end of a timestep, and if it is not achieved, the timestep is repeated. No convergence check is performed in the weak coupling the timestep is never repeated. Figure 2.3 shows the basic schematics of weak and strong coupling.

Weak coupling is also known as explicit, staggered, or loose coupling. It is a basic coupling method in which the interface quantities are exchanged once per timestep. Since the timestep is never repeated, it is inherently cheaper in terms of computational effort than the strong coupling, especially when

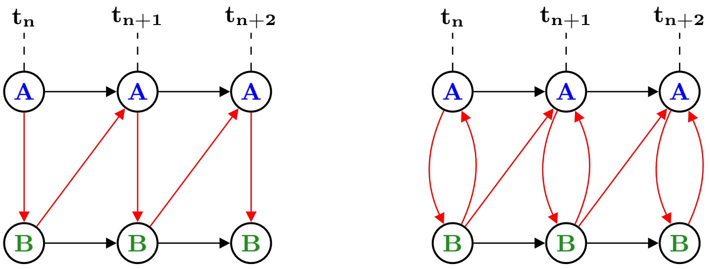


(a) Jacobi pattern (b) Gauss-Seidel pattern

Figure 2.2: Jacobi and Gauss-Seidel communication patterns. The Jacobi pattern uses the data from the previous timestep, whereas the Gauss-Seidel pattern uses the latest available data.

considering that the execution of a coupling partner is usually the most costly part of a coupled simulation. On the downside, it suffers from accuracy issues, especially when the interaction between the physical fields is strong. Prediction techniques can mitigate these problems, but in some applications with very strong interaction, it is not possible to achieve convergence with weak coupling.

Strong coupling is also known as implicit or iterative coupling. It tries to achieve convergence of the coupled solution by minimizing an interface residual to fulfill the interface compatibility and equilibrium conditions. Thus, it is more accurate and stable than the weak coupling at the cost of being computationally more expensive. Relaxation techniques are used to support and accelerate the convergence within a timestep to reduce the number of coupling iterations.



(a) Weak/Explicit coupling (b) Strong/Implicit coupling

Figure 2.3: Weak and strong coupling methods (using the Gauss-Seidel pattern) for a two-way coupling. The strong coupling algorithm checks for convergence of the coupled solution and might repeat the current timestep if it is not achieved.

Prediction

Prediction or extrapolation techniques use the information from previous Timesteps (TSs) to provide a better initial guess for the solution at the beginning of a timestep. The goal is to improve and accelerate the convergence, both of the coupled solution but also of the individual solvers. They can be used both with weak and strong coupling methods.

Different methods are available, ranging from linear extrapolation to more advanced methods such as presented by [21]. [46] provides an extensive overview and comparison of prediction techniques.

Relaxation and convergence acceleration

Relaxation is often used to increase the stability of a strongly coupled simulation. Instead of applying the full update of the solution, it is scaled with a (relaxation) factor. Factors smaller than 1 give under-relaxation, larger than 1 is over-relaxation. Practical cases typically require under-relaxation to ensure stability of the coupled solution, which leads to slower convergence and thus higher computational cost. Therefore, the choice of the relaxation factor is crucial for efficient simulations.

Converge acceleration techniques aim to optimize the computation of the relaxation factor for a given problem. The goal is to reduce the number of coupling iterations while maintaining stability. A large variety of methods has been proposed in the literature, and it shall be referred to [46] and [77] for an extensive overview of this matter.

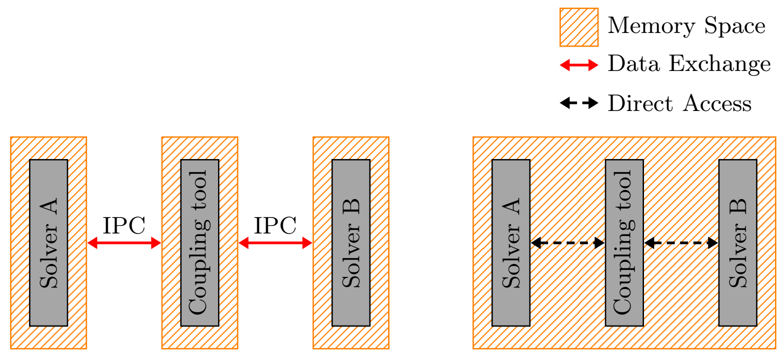
Convergence criterion

The convergence in strongly coupled simulations is checked with a convergence criterion. In many cases, this is realized by checking the difference of a specific value or field between coupling iterations. Once this difference drops below a predefined threshold, the TS is considered to be converged, and the solution advances in time. The choice of threshold and criteria can have a large impact on the performance and convergence behavior of the coupled simulation and is therefore an important component. [49] and [29] provide an overview over commonly used criteria.

2.2.2 Data exchange between tools

Column two in Figure 2.1 represents the data exchange between tools, which is required as soon as more than one software is involved in the CoSimulation. Typically, each tool runs in its own address/memory space, which makes it not possible for the coupling tool to directly access the data. Therefore, data has to be exchanged between them. In software terms, this is the exchange of data between different processes, which is generally referred to as IPC in computer science. This concept is shown in Figure 2.4a.

Multiphysics tools have the inherent advantage that the data of the solvers is directly accessible by the coupling tool and hence no data exchange is required. This concept is shown in Figure 2.4b.



(a) Coupling tool can access the data only after it was transferred via IPC.

(b) When the solver uses the same address/memory space as the coupling tool (e.g. multiphysics framework), then it can directly access the relevant data/memory without any IPC.

Figure 2.4: Different ways the coupling tool can access the data of the solvers.

2.2.2.1 Question / Challenge

IPC in the context of CoSimulation needs to fulfill various requirements. Typically, the exchange of data happens several times per TS and can involve large amounts of data, depending on the type of coupling. This means that the data exchange needs to be fast and efficient. It is also beneficial to support many different systems, since nowadays, the tools can be deployed on various hardware, ranging from small microcomputers such as the *Raspberry PI* to large supercomputers using distributed memory environments. This is especially important as IPC is usually implemented on a very low level and oftentimes differs greatly between different OSs. Stability and reliability are other selection criteria, in particular the initial handshake during the connection phase. The exchanged data can range from scalar values to entire meshes and geometries. Hence, it is required to support different types of data, which again, depends strongly on the type of coupling.

From a developer's perspective, more criteria are important. The implementational effort can be a deciding factor if the complexity of implementing a method outweighs its benefits. Also, it is beneficial if the method is easy to understand and debug, especially when it comes to the flow of information. In many coupled simulations, the data exchange between tools is also used for synchronization between them. Here being able to easily understand the data exchange can greatly reduce the time that is required to achieve a successful coupling.

The requirements for IPC in CoSimulation contexts can be compiled to the following list:

- Performance

- Robustness
- Portability
- Usability
- Implementation
- Dependencies

2.2.2.2 Methodological Answer

As IPC is a commonly required technology for many different applications in computer science, it has been well studied, and many different methods have been developed, see e.g. [73]. Based on the selection criteria introduced above, different methods can be analyzed and evaluated regarding their suitability for CoSimulation. See also [29].

2.2.2.3 Technical Realization

Many different methods of IPC exist, which are listed and briefly introduced in the following. More information can be found in the documentation of the respective OSs, e.g. for Linux[1] and Windows[2].

- Shared Memory: Data exchange through a (usually specially allocated) part of the memory, which can be accessed by different processes.
- Files: Data exchange via files that are written to the filesystem.
- Anonymous pipes: Data exchange via a data channel between related (parent/child) processes. The data that is sent first, is received first.
- Named pipes: Similar to anonymous pipes, but between unrelated processes.
- Network socket: An endpoint to send and receive data through the network. Therefore, it works across machines.
- Unix domain socket: Similar to the network socket, but uses the kernel directly to communicate, within one machine.
- Message queues: Similar to pipes but the receiving order can deviate from the sending order.
- Signals: a small message usually used to send a signal for control rather than more complex data.
- Memory mapped file: A section of the RAM that acts like a file, but can also be accessed by memory addresses.

[1] `https://linux.die.net/`

[2] `https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications`

Even though most OSs provide all or a subset of IPC methods listed above, it is not guaranteed that they work in the same way or are compatible. Since only a few methods are standardized, it is oftentimes required to use low-level, OS specific Application Programming Interface (API) functions. Platform-specific solutions and implementations (pipes for example work differently on Windows and Unix and are thus incompatible) further complicate cross-platform data exchange, only a subset of methods is suitable for this, such as files. Some libraries like the *ASIO* library[3] aim to hide this complexity by providing high-level interfaces.

Furthermore, due to the large variety of data that might be exchanged, it is profitable to find solutions that work for each of them, i.e. are agnostic of what data is exchanged. One approach to unify the treatment of data is *serialization*, which is explained in the next section.

2.2.2.4 Serialization

Sending an object of a complex type such as a mesh through a network to another process requires it to be represented by a stream of bits. The same holds when saving it into a file. The layout of the object in memory might not allow this, in most cases only very basic data structures like arrays (which store their data contiguously in memory) can be sent/saved directly. Therefore, it is necessary to have a way of converting an object into a bitstream, as well as the inverse operation, reconstructing the original object from a stream. This is generally known as serialization and deserialization. The first takes an object, and saves it to a stream or file, while the second one loads the object. Many different techniques can be used, depending on the complexity of the objects that are involved[4].

Other common applications of serialization in numerical simulation are restart/checkpoint files. The internal state of the solver is saved to a file so that it can later be used to continue a simulation. This is particularly helpful for conducting long simulations on HPC systems, where the runtime is typically limited, and therefore the total simulation needs to be divided into several parts.

2.2.3 Data transfer between solution techniques

Each coupling partner uses its own solution technique, depending on what suits best for the respective application. This also means that each tool has its own way of handling and storing data in a certain format. For CoSimulation, it is necessary to get the data of one coupling partner to another. In many cases, this requires the data exchange as explained in Section 2.2.2. Additionally, the data has to be translated from one format to another, which is referred to as data transfer in this work, see column three of Figure 2.1. The flow of information is from the *origin* to the *destination* as visualized in Figure 2.5.

The quantities on both sides of the interface should be the same, as shown in equation 2.1:

[3] `https://think-async.com/Asio/`
[4] `https://isocpp.org/wiki/faq/serialization`

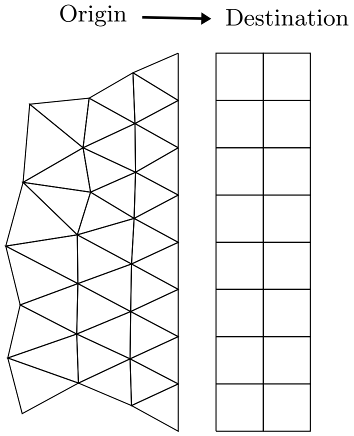


Figure 2.5: Flow of information during data transfer between solution techniques, from the *Origin* to the *Destination*. Here displayed for a Finite Element Method (FEM) mesh.

$$\mathbf{x}_d = \mathbf{x}_o \tag{2.1}$$

Here the vector of quantities on the destination $\mathbf{x}_d$ is the same as the one on the origin $\mathbf{x}_o$. A conversion between the domains is required if the discretizations are not matching, see equation 2.2:

$$\mathbf{x}_d = \mathbf{H}_{od}\mathbf{x}_o \tag{2.2}$$

The transfer matrix $\mathbf{H}_{od}$ is used to transform the quantities on the origin into the ones on the destination.

Mapping is the most prominent example of data transfer techniques. It is used when a tool stores its data in relation to a geometrical location. Numerical tools that use the FEM or Finite Volume Method (FVM) subdivide their solution domain into meshes consisting of small finite elements/volumes, respectively, connected by nodes at their corners. They store the solution data at the nodes and in the elements/volumes, hence it has geometrical information attached to it. The discretizations of the coupling partners are usually different on the coupling interfaces, which makes it necessary to map the data with special mapping techniques.

On the other end of the spectrum of data transfer techniques is the exchange of a single scalar value without any geometrical information. This can be seen as the most basic form of data transfer, in which the value on the origin is assigned to the value on the destination.

Other types of data transfer can be the direct copy of N-to-N values without taking geometrical information into account. Furthermore, 1-to-N and *sum*(N)-to-1 transfers can be done if the coupling partners require it. An example is the exchange of data between coupling partners that discretize their solution domain and tools that only use single scalar values like Single Degree of Freedom (SDOF) solvers.

2.2.3.1 Question / Challenge

The translation of data from one format to another is one of the fundamental challenges of CoSimulation. The coupling partners use different formats of data that, in most cases, are not directly compatible with each other. Therefore, special techniques for transferring data from one format to another are required.

The data transfer technique is specific to each pair of data formats which requires the development of new techniques each time a coupling partner with a new solution technique joins the CoSimulation.

Data transfer is usually conducted several times per TS (in particular for a strong coupling) and can impact the overall solution time significantly. Speed and efficiency are therefore one of the most crucial requirements. This becomes particularly important when the coupling partners employ distributed computing, meaning that also the data transfer needs to be able to handle this efficiently, ideally without expensive gather-scatter operations of data.

Additional requirements can be the conservation of energy during the data transfer.

2.2.3.2 Methodological Answer

The layout of the data involved in the coupling drives the selection or development of which data transfer technique to be used. The specifics of each coupling partner need to be taken into account, such as the solution techniques it uses. Developing new techniques, therefore, requires detailed knowledge of the internals of the coupling partner. Defining common/standard interfaces and methods for similar tools can thus significantly reduce the time required to establish a coupled solution.

Common traits and challenges can be identified, which leads to the specification of a set of requirements for a data transfer technique.

2.2.3.3 Technical Realization

Defining the requirements is the first step toward realizing a data transfer technique. Due to the different ranges of applications, from copying of scalar values to mapping on non-matching meshes, the development, and implementational effort varies greatly. More information can be found in Chapter 4, which is exclusively dedicated to this complex topic.

2.3 Theoretical background of solution techniques

This section briefly introduces the relevant theory and notations for most of the solution techniques and physical fields and phenomena considered in this work. These are Computational Structural Dynamics (CSD), Computational Fluid Dynamics (CFD) and FSI. References for further reading are provided.

2.3.1 Computational Structural Dynamics (CSD)

CSD is the numerical simulation of structural problems. The strong form of the governing equations based on the displacements **u** is formulated as follows:

$$\rho\ddot{\mathbf{u}} - \nabla\boldsymbol{\sigma} - \rho\mathbf{b} = 0 \tag{2.3}$$

Herein ρ is the density, $\ddot{\mathbf{u}}$ the acceleration, $\boldsymbol{\sigma}$ the stress tensor, and **b** the body forces.

The principle of virtual work is used to fulfill the equilibrium of forces in a weak sense. The general formulation is:

$$\delta W = \delta W_{kin} + \delta W_{int} + \delta W_{ext} = 0 \tag{2.4}$$

Where δW_{kin} is the kinetic/inertial/dynamic, δW_{int} is the internal, and δW_{ext} is the external virtual work.

After applying the boundary conditions and considering the constitutive behavior of the structure, finally, the weak form of the governing equations can be expressed as:

$$\underbrace{\int_\Omega \rho\ddot{\mathbf{u}} \cdot \delta\mathbf{u}\, d\Omega}_{\delta W_{kin}} + \underbrace{\int_\Omega \boldsymbol{\sigma} : \delta\mathbf{e}\, d\Omega}_{\delta W_{int}} - \underbrace{\int_\Omega \rho\mathbf{b} \cdot \delta\mathbf{u}\, d\Omega - \int_\Gamma \rho\mathbf{b} \cdot \delta\mathbf{t}\, d\Gamma}_{\delta W_{ext}} = 0 \tag{2.5}$$

The FEM is used to transform the continuous formulation of the weak form from equation 2.5 into a discrete one. The displacements **u** are hereby approximated by means of shape functions **N** and discrete values at the nodes $\hat{\mathbf{u}}$:

$$\mathbf{u} = \mathbf{N}\hat{\mathbf{u}} \tag{2.6}$$

The discretization leads ultimately to the equation of motion in discrete form:

$$\mathbf{M}\hat{\ddot{\mathbf{u}}} + \mathbf{D}\hat{\dot{\mathbf{u}}} + \mathbf{K}\hat{\mathbf{u}} = \mathbf{F_{ext}} \tag{2.7}$$

where the matrices $\mathbf{M}, \mathbf{D}, \mathbf{K}$ are the mass, damping, and stiffness of the system, respectively. The derivation of mass **M** and stiffness **K** is based on the governing equations as shown above. Velocity-proportional Rayleigh damping [75] is used in this work to approximate the damping of the system. Hereby the damping **D** is computed from mass and stiffness, see equation 2.8:

$$\mathbf{D} = \alpha\mathbf{M} + \beta\mathbf{K} \tag{2.8}$$

The parameters α and β can be computed from the two most relevant eigenfrequencies of the system, see [3] for more details.

Most examples in this work involve large displacements and rotations. Nonlinear kinematics are required to accurately model them, which leads to a nonlinear stiffness of the system. It then depends on the displacements $\mathbf{u}$, resulting in $\mathbf{K}(\mathbf{u})$. A Newton-Raphson strategy (see [3]) is therefore applied to solve equation 2.7. Implicit time integration is performed using the BDF2 scheme [30]. An overview of time integration schemes for structural simulations is presented in [48].

More information about FEM and its application to CSD can be found in the standard textbooks [1], [39], [3] and [37].

2.3.2 Computational Fluid Dynamics (CFD)

The behavior of fluid flows can be simulated with CFD. Within this work, the governing equations for these problems are the Navier-Stokes equations. For most examples, an incompressible flow is considered, for which they are formulated as follows:

$$\begin{aligned} \rho\dot{\mathbf{v}} + \rho\mathbf{v}\cdot\nabla\mathbf{v} - \nabla\cdot\boldsymbol{\sigma} &= \mathbf{f} \\ \nabla\cdot\mathbf{v} &= 0 \end{aligned} \tag{2.9}$$

where ρ is the density of the fluid, $\mathbf{v}$ the velocity, $\boldsymbol{\sigma}$ the stress tensor and $\mathbf{f}$ the external forces.

A wide variety of solution methods is available for CFD, concerning the treatment of turbulence and transient effects as well as discretization. Direct Numerical Simulation (DNS) is used to fully resolve turbulence and transient effects but is rarely applied for large problems due to its immense computational cost. Time averaging together with a turbulence model with Reynolds-Averaged Navier-Stokes (RANS) is on the other end of the spectrum regarding its computational requirements. The Large Eddy Simulation (LES) approach models the small scales but resolves large vortices, positioning it in between the aforementioned approaches in terms of accuracy and computational cost. More methods such as Unsteady Reynolds-Averaged Navier-Stokes (URANS) and Detached Eddy Simulation (DES) also exist, which are classified between RANS and LES.

Most examples in this work that involve CFD are using the LES approach with FEM as discretization. More information about this can be found in [79], [12], and [22].

2.3.3 Fluid-Structure Interaction

FSI is a coupled problem where the effects of a fluid flow on a structure and vice versa are considered in a numerical simulation. The fluid $\mathbf{F}$ depends on the structural displacements $\mathbf{u}$, whereas the structure $\mathbf{S}$ depends on the fluid forces $\mathbf{f}$:

$$\begin{aligned} \mathbf{f} &= \mathbf{F}(\mathbf{u}) \\ \mathbf{u} &= \mathbf{S}(\mathbf{f}) \end{aligned} \tag{2.10}$$

The interface conditions for a FSI problem can be formulated as follows:

$$\begin{aligned} \mathbf{u}_{\Gamma,S} &= \mathbf{u}_{\Gamma,F} \\ \sum \mathbf{f}_{\Gamma,S} &= \sum \mathbf{f}_{\Gamma,F} \end{aligned} \tag{2.11}$$

On the interface Γ, the displacements are equal, and the sum of forces of both solvers needs to be in equilibrium.

The solutions introduced in Section 2.2.1 for solving coupled problems can be used, including prediction or relaxation techniques to accelerate the convergence of the coupling. In a standard Dirichlet-Neumann coupling, the loads computed by the fluid solver are mapped to the structure, which uses them to compute new displacements. These are then mapped back to the fluid domain, where the mesh-solver moves the mesh to avoid a collapse of elements near the boundary. The inner loop of a standard FSI problem is shown in algorithm 1.

Algorithm 1: Inner coupling loop of an FSI problem, using Dirichlet-Neumann coupling.

```
1 Solve Fluid
2 Map loads to Structure
3 Solve Structure
4 Map displacements to Fluid
5 Move Fluid Mesh
```

After the structural displacements are mapped to the fluid interface, a mesh moving technique (see [74]) is used to propagate them from the interface into the domain. This is required to maintain a good element quality, especially near the interface. Those regions are typically meshed finely to resolve the boundary layer accurately and thereby compute accurate loading on the structure. Any distortion could lead to convergence problems. The Arbitrary Lagrangian-Eulerian (ALE) approach is used to consider the mesh velocity resulting from the motion of the mesh in the fluid solution.

Mapping is required in most practical cases due to non-matching discretizations on the coupling interface Γ.

2.4 Parallel Computing

Numerical simulations of large real-world applications oftentimes require a very high computational effort. Even more so when several problems are solved together in a coupled simulation. This can lead to very long simulation times on regular computers, thus limiting the applicability of numerical simulations for engineering applications and developments.

One way to reduce the solution time is by employing parallel computing. Here the workload of the simulation is split between many compute resources to reduce the solution time. The more resources are used, the smaller the simulation time becomes, theoretically. The speedup S is defined in equation 2.12, it indicates how much faster the computing time becomes when more resources are used. $t(1)$ refers to serial execution, $t(N)$ to parallel execution with N compute resources.

$$S = \frac{t(1)}{t(N)} \tag{2.12}$$

Fundamentally two types of parallel computing are distinguished: Shared and distributed memory. As the name suggests, with the first one, the Central Processing Unit (CPU) has access to the entire memory of the machine. In the context of numerical simulations, this means that the entire computational model can be accessed. This type of parallelization is usually done using a single CPU with many cores. Typically, up to 128 cores are used with this type of parallelization.

Even more computing resources can be used with distributed computing. Many CPUs are combined into one large system, which is then referred to as a cluster (or even a supercomputer for very large systems). Every CPU (in this context also called compute node) has access only to its local memory since the memory is distributed. This requires splitting the model into subdomains (using domain decomposition). For numerical simulations, this means that it is no longer possible to access the entire model from each CPU.

The main differences between the two variants are the communication that is required in distributed computing between the different CPUs, for example during collective operations. Furthermore, the algorithms can be vastly different, since in shared memory only part of the execution is done in parallel, whereas with distributed memory, the entire execution happens in parallel.

Shared and distributed memory parallelization can be combined in hybrid approaches, on large systems with many compute nodes. Within one node/CPU, shared memory is used, whereas distributed parallelism is used across nodes for the communication of data.

The performance of a parallel algorithm is typically assessed with scaling studies. Execution times are measured with different numbers of compute resources and problem sizes. Strong scaling studies vary the number of threads/cores/processes/ranks while keeping the size of the problem constant. Weak scaling studies do the same, but additionally, the size of the problem increases, to keep the local size (per thread/core/process/rank) constant.

2.4.1 Shared Memory

Most modern CPUs have several cores, which can be used to process data in parallel. Leveraging this computing power requires the usage of special programming techniques. Independent of programming language, this is typically achieved with a forking model, where at some point the execution is split into several parts (usually threads, within the same process). These are combined again after the parallel execution has been completed. Each part is processed by a separate thread, which is why shared memory parallelization is often referred to as threading.

One challenge with this type of parallelization is to develop thread-safe code without race conditions. These occur when several threads compete for the same data. It needs to be ensured that the threads performing the computations access the data in a safe and regulated manner, otherwise, the behavior of the program is undefined and will yield different results in every execution.

The implementation of shared memory parallelization strongly depends on the programming language and technology employed. Commonly used are OpenMP[5] and pthreads[6] in C/C++, or multiprocessing[7] in Python.

2.4.2 Distributed Memory

Using large computer systems such as clusters or even supercomputers is often referred to as HPC. It is used for very large computational requirements, for example, the coupled numerical simulation of real-world applications, as presented in this work. These systems are typically built from many individual processors that are connected by a high-speed network, hence a CPU/processor can only access its local memory. If data from remote memory/other processors is required, then it can only be accessed by communicating it through the network. This distributed memory architecture requires the development of specialized algorithms.

Communicating data between different processors is a crucial difference to algorithms developed for conventional/shared memory architectures. The time spent communicating does not contribute to the solution of the problem, it is therefore considered overhead and should be avoided as much as possible. The less time an algorithm spends on communication, the more efficient it is in distributed computing.

The exchange of data among the processes is commonly done with the Message Passing Interface (MPI), see [58]. MPI offers many functionalities that facilitate the exchange of data with different methods. It is already extensively used for many years in the development of algorithms in distributed environments. Currently, it is the de facto standard for the development and execution of distributed algorithms. The features and concepts which are important in this work are briefly introduced in the following.

MPI also refers to processes as ranks. Several ranks are usually grouped within an *MPI-Communicator*. The global communicator including all ranks is *MPI_COMM_WORLD*. Communicators are used for function calls to MPI, only the ranks where a certain communicator is defined participate in the function calls. Each rank has a unique identifier within a communicator.

Commonly used functionalities are the communication between different ranks, where one rank acts as the sender, and the other ranks act as receivers of the data. Also, collective operations including several ranks such as gather, scatter, or broadcast operations are available.

In the context of numerical simulations, using distributed computing usually implies that the computational domain is split/partitioned into several subdomains, one for each processor/core/process. The communication happens mostly on the domain interfaces. METIS [43] and SCOTCH [62] are

[5] `https://www.openmp.org/`
[6] `https://man7.org/linux/man-pages/man7/pthreads.7.html`
[7] `https://docs.python.org/3/library/multiprocessing.html`

commonly used libraries for this task. The main goal is to split the domain into similar chunks, to distribute the computational load equally. Furthermore, the interfaces between the subdomains are to be minimized, in order to reduce the communication among them.

Even though HPC is not strictly necessary for CoSimulation, it is oftentimes required due to the high computational effort of real-world applications. Hence, neglecting it during the development of algorithms and tools for CoSimulation can lead to a significant performance penalty. Without the support for distributed environments, some computationally expensive simulations might not be feasible. Alternatively, if no support for distributed environments is available, the coupling can be performed on a single rank. This however requires a lot of communication because of the gathering and scattering of data to and from a single rank. This bottleneck becomes more severe with an increasing number of processes.

Despite the clear advantages, distributed memory parallelization also has many drawbacks and introduces challenges, both during development and execution. A main obstacle is the development of special algorithms due to the different underlying hardware layouts. Furthermore, fewer libraries support it, for example, linear solvers. Some technologies such as pipes for IPC only work in shared memory. Debugging large cases can present another obstacle, especially since algorithms are mainly developed on local machines which use shared memory.

Conducting computations on a cluster usually means that the jobs cannot be started directly, they need to be submitted to a special queuing system (e.g. SLURM [84]). This implies that one typically needs to wait for some time until the job is run. Furthermore, in most cases the runtime is limited, meaning that long simulations need to use restarts in order to complete. This further increases the complexity and potential failures.

2.5 Existing Tools

Partitioned multi-physics simulations can be performed in different ways (see Section 3.4). Many academic and commercial solutions have been developed over the years, focussing on various applications. The existing tools can roughly be divided into two categories: Dedicated coupling tools to connect and couple solvers, and multiphysics tools with coupling functionalities for their internal solvers.

The solutions developed in this work blur the lines between these two categories. Coupling of internal solvers is possible, as well as external solvers, with minimal changes in the workflow.

Dedicated Coupling Tools

These tools are characterized by focusing purely on the coupling of external solvers. Enabling coupled simulations is usually done by integrating the API of the coupling tool in the solver. This requires access to the source code in most cases, which is typically not available for commercial solvers. Some solvers provide API access, which can be used to integrate the coupling tool.

Since they do not offer solution techniques themselves, the capabilities of dedicated coupling tools for solving multi-physics problems depend strongly on the external tools used. A selection of tools is listed in the following, including a brief summary of their scope and functionalities. See [29] and [77] for a broader overview.

- *EMPIRE / CoMA*: EMPIRE [80] and its predecessor CoMA [28] were developed in-house and are part of the long history of research in coupled simulations at the author's institute. The experiences collected during their development and usage were used as a basis and inspiration for this work. Both tools employ the server-client library approach and use MPI to exchange data. Additionally, advanced mapping methods are available.
- *precice*: precice [11] is a library for partitioned black-box coupling, open-source and licensed with LGPL. A special focus is set on HPC, with tested scalability of over 10,000 cores. This is realized through a peer-to-peer communication approach. It provides adapters to connect various solvers, offers a wide range of coupling functionalities, and has strong community support. More information can be found in [29] and [77].
- *MpCCI*: MpCCI [42] is a commercial coupling tool. It offers application-specific solutions such as for FSI as well as generalized coupling functionalities. Furthermore, its interface is integrated by some commercial solvers, which enables the realization of coupled simulations.
- *comana*: Comana [47] is a closed-source framework for the generic solution of multi-physics problems. More information about this tool can be found in [46]. Its focus is on the efficient integration of external solvers and offers a wide range of coupling features to stabilize and accelerate convergence. It has been tested with several large-scale maritime applications.

Multiphysics Tools

The collection and unification of solution techniques for different physical applications is the purpose of these tools. They oftentimes also provide capabilities and processes to couple their internal solvers for performing coupled multiphysics simulations. These however are usually limited to internal solvers, and cannot be used for integrating external solvers into the CoSimulation. If available, the API can in some cases be used to realize the integration, but this generally comes at the price of additional development effort and computational overhead.

Some multiphysics tools are introduced in the following:

- *FreeFEM*: FreeFEM [35] offers solution procedures for non-linear structural, fluid and FSI applications, in 1D, 2D and 3D. It is open-source, and licensed with GPL. Capabilities for coupling external solvers are not provided.

- FEBio: The FEBio[55] software suite is a FEM tool with a special focus on biomechanics and biophysics with large deformations. In addition to structural problems, it also offers solutions for fluid dynamics and heat transfer. Coupling features with the internal solvers are provided, e.g. for FSI. It is open-source with an MIT license.
- Commercial solvers like Abaqus[8], COMSOL[9] and Ansys Workbench[10] provide CoSimulation capabilities, which are primarily focussed on their respective internal solvers. Coupling with external solvers is sometimes possible by using their APIs, including the drawbacks listed above.

[8] https://www.3ds.com/products-services/simulia/products/abaqus/

[9] https://www.comsol.com/

[10] https://www.ansys.com/products/ansys-workbench

Chapter 3

CoSimulation in and with a multiphysics framework

Chapter 2 presented the theoretical basics for partitioned coupled simulations. This chapter continues on a more practical path by presenting the realization of CoSimulation *in* and *with* a multiphysics framework. Performing CoSimulation *in* the framework means that solvers of the framework are used. CoSimulation *with* the framework means that it acts as a coupling tool for coupling external tools/solvers.

The open-source code *Kratos Multiphysics* ([15], in the following referred to as *Kratos*), is chosen as the framework to integrate and perform CoSimulation with. The main reasons for using Kratos are the cross-platform support, efficient parallelization and memory management, wide selection of internal solution techniques, modern software architecture, and permissive license. This combination makes it the ideal basis for this work.

After a brief introduction of Kratos, the different aspects of CoSimulation are presented. Then the integration of CoSimulation in Kratos is explained, including the treatment of distributed memory environments. An overview of technical realizations is shown afterward, pointing out the differences between coupled simulations with a dedicated coupling tool to a multiphysics tool. The integration of internal solvers is outlined as well as the coupling to external solvers. For the latter one, a new detached interface is presented, with detailed studies of IPC for CoSimulation.

The work presented in this chapter was done in collaboration with Aditya Ghantasala. The design and concepts were developed together and are presented in [31]. The contribution of this work is the implementation and realization of CoSimulation, and in particular the developments for the coupling to external solvers. Furthermore, [9] presents some intermediate findings and developments.

3.1 Overview of Kratos Multiphysics

Kratos serves as the basis for this work, therefore this section explains and introduces its features that are important for this work. This is required to motivate the decisions taken in the later parts of this chapter, especially when it comes to implementational aspects.

Kratos is an open-source multiphysics framework using the FEM. The source code is freely available on GitHub[1]. It contains solution techniques for several physical phenomena such as fluid dynamics, (CFD) structural mechanics (CSD) or particle methods (e.g. Discrete Element Method (DEM)). Aside from the solution techniques that are strongly focused on the physics of an application, other methods like shape optimization or the CoSimulation presented in this work are available.

To avoid overhead and code duplication, Kratos employs a core/applications approach. Common functionalities like data structures, in-/output, linear solvers, FEM assembly or search structures are located in the core of Kratos. The applications depend on the core, and in some cases also on each other. The latter is especially important for CoSimulation, as it brings together individual solvers to perform coupled simulations.

An overview of the applications used in this work is given in the following.

- **FluidDynamicsApplication**: This application contains most of the CFD functionalities of Kratos and is used for FSI simulations. Unlike many other CFD solvers, it uses the FEM in combination with the Variational Multiscale (VMS) method. More details can be found in [12]. Due to the typically large computational cost of CFD simulations, support for distributed environments via MPI parallelization is provided.

- **StructuralMechanicsApplication**: Solid and structural simulations (CSD) can be performed with this application. Among others, it features solid, shell, membrane, beam and truss elements along with the corresponding boundary and loading conditions. It also supports distributed simulations.

- **DEMApplication**: Solution techniques based on the DEM method are implemented in this application, for solving the motion and interaction of discrete particles.

- **MeshMovingApplication**: In a two-way coupled FSI simulation, the structural deformations are mapped to the fluid domain on the interface. Typically, the vicinity of the structure is finely discretized to accurately represent the surrounding flow. If the structural deformations are of the same magnitude or larger as the fluid elements, they can distort severely which oftentimes leads to a diverging simulation. To avoid this, the deformations are smoothened and pushed out to the domain. This is oftentimes referred to as mesh moving, for more details see [59].

[1] `https://github.com/KratosMultiphysics`

- **MappingApplication**: Different techniques for mapping between non-matching meshes are implemented in this application. Chapter 4, is entirely dedicated to this topic.
- **CoSimulationApplication**: A large part of this work was the implementation of coupling features, which is done within this application. Coupling strategies, coupling to external codes and implementational aspects are discussed in this chapter.
- **LinearSolversApplication**: Linear solvers are a key component for many numerical simulations, as they can significantly influence solution times and stability, as well as memory usage. This application provides an interface for solvers from external libraries, most notably to Eigen [32] and the Intel-MKL[2].
- **MetisApplication**: Distributed simulations require partitioning the simulation domain among the different computing cores. Domain decomposition with the METIS library [43] is done with this application.
- **TrilinosApplication**: The trilinos library [76] is a collection of functionalities including distributed linear algebra. Kratos uses it as the backend for the distributed simulations and distributed FEM assembly routines. All commonly used components using trilinos can be found in this application.

Kratos uses mainly two programming languages: *Python* for high-level scripting tasks, as well as controlling and steering the overall flow of execution due to its simplicity and flexibility. On a low level, C++ is used for the main data structures and computationally expensive tasks such as linear solutions or the assembly of the system of equations.

This approach combines the advantages of both languages and mitigates their disadvantages. Python provides users with easier accessibility to the framework, as well as scripting options, whereas the performance of C++ can be fully leveraged if necessary.

3.1.1 Data container for CoSimulation

Handling data is one of the main tasks in CoSimulation. This includes but is not limited to, the synchronization of data among the coupling partners and the mapping of data from one discretization to another. As those are critical and often occurring tasks, the data access and storage need to be done in a fast and efficient way. The *ModelPart* is the main data container in Kratos and was designed based on these requirements. It is a generic and flexible container for storing different types of data and entities that are related to different numerical solution techniques. Storing plain data such as scalar values is possible, as well as meshes with nodes and elements or entire geometries such as NURBS patches which are used in Computer Aided Design (CAD) models.

[2] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`

Details of the implementation can be found in [14]. It is used by the different physical solvers of Kratos to store their solution data.

The existing usages, available features and flexibility make the ModelPart the ideal data container for CoSimulation. This has several significant advantages:

- The data of the Kratos solvers can be directly accessed, without any overhead such as copying of memory. This is a crucial advantage of performing CoSimulation within a multiphysics tool over using dedicated coupling tools. Due to the direct access, it is also not necessary to exchange the data with IPC.
- The ModelPart is a sophisticated and optimized data container that is extensively used in the Kratos solvers.
- Many auxiliary functionalities like in-/output are available for the ModelPart and can directly be used and integrated into CoSimulation.
- Different types of data and entities required in numerical solution techniques can be stored: From plain data to meshes with nodes and elements which contain geometrical information. The geometrical information is required for establishing geometrical relations between coupling partners, as is required for mapping.

3.1.2 Kratos in distributed environments

Many numerical solution techniques like CFD can have a high computational cost, especially for the detailed simulation of real-world problems. The consequence is long simulation times, which is not feasible for industrial usage. As discussed in Section 2.4.2, a solution for this problem is to use large computer systems such as clusters or supercomputers, or generally, HPC systems. These systems usually employ distributed memory models, which means that not all data/memory can be easily accessed anymore. Non-local/remote data has to be requested and can be fetched by exchanging messages among the processes. MPI is a standard for passing messages among processes that has been adapted by most HPC systems. [13] shows how support for distributed computing is implemented in Kratos.

Compared to traditional shared memory machines, distributed computing brings many additional challenges on the algorithmic, soft- and hardware side. This means that even though it is very powerful and heavily used for some applications, it is not used for every numerical simulation. One of the main features of Kratos is the support both for shared and distributed memory systems to provide maximum flexibility for accommodating different application cases and computer systems. Functionalities are provided that allow the developer to write algorithms that are agnostic of the employed memory model. Common requirements are the computation of reductions such as minima or maxima or synchronization of data across ranks.

Mainly two objects are used for the agnostic implementation of handling the memory model, the *(MPI)Communicator* and the *(MPI)DataCommunicator*. Depending on whether Kratos is used in a shared or distributed memory environment, the corresponding object is used. In shared memory, no communication is required since - as the name suggests - all the data is directly accessible in the shared/common memory. Here the Communicator/DataCommunicator is used. They serve as base classes for their MPI-variants and have mostly trivial implementations since no communication is required. In distributed memory, the MPICommunicator/MPIDataCommunicator are used. They perform the necessary computation if it is required. The functionalities of each object are briefly listed in the following:

- **(MPI)DataCommunicator**: The MPI interface mostly uses C and no modern C++, therefore the DataCommunicator serves as a small wrapper for the most used MPI-functions. This is done so that it can be directly used with the C++ data types such as *std::vector*. The interface includes reductions as well as sending and receiving of data among different ranks. Each *MPIDataCommunicator* has its own *MPI_Comm* communicator such as *MPI_COMM_WORLD* for MPI function calls. Depending on the setup it is also possible that the MPI_Comm is not defined on a certain rank which means that it is not part of/used for communication. This is particularly relevant for CoSimulation if a solver does not use all MPI ranks.
- **(MPI)Communicator**: This object is owned by a ModelPart and handles the communication and the synchronization of local and ghost entities such as nodes when the ModelPart is partitioned across several ranks. Furthermore, access to local, ghost and interface entities is provided. Internally it has a DataCommunicator for (MPI) communication.

3.1.3 Common solver interface

Despite using vastly different physical solution techniques, all Kratos solvers follow the same top-level interface: the *AnalysisStage*. It is designed flexible and generic to accommodate the requirements of different solution techniques, both for Kratos internal and external solvers. Furthermore, it provides features to adapt and customize the behavior if necessary. This unified interface simplifies the integration of Kratos solvers into CoSimulation greatly since it allows them to be treated as black-box. It can therefore be considered one of the basics developed in this work. The main part of the interface that is important for CoSimulation is explained in the following, and visualized in 3.1.

The three main functions are:

- **Initialize**: This function is executed once at the beginning of the simulation. The solver is prepared for the solution procedure, which usually starts with reading and validating input. Afterward, the internal data structures are prepared, and the initial conditions are applied. Final checks are done before the start of the solution loop.
- **RunSolutionLoop**: Main function of the solution, executes the TSs. It can be customized to accommodate different solution algorithms like iterative solution procedures.

- **Finalize**: Cleanup and finalization after the simulation. Can be used to close output files or perform statistical evaluation of the obtained solution.

Additionally, the *RunSolutionLoop* method is split up into six more functions for fine-grained control over the different parts of the solution.

- **AdvanceInTime**: Increment the time and perform time integration, along with the preparation of the data structure for the following step.

- **InitializeSolutionStep**: Application of boundary conditions and other custom conditions that can vary in space and time.

- **Predict**: Prediction techniques to improve and accelerate the solution by providing an improved initial guess.

- **SolveSolutionStep**: Main function which is responsible for the solution of the current step. It is the only function that can be executed multiple times in an iterative solution procedure, usually until convergence is achieved.
  More abstractly, this function obtains a new solution, by different means. These can be a numerical solution, communication with an external tool to get new values, or predicting the new solution with a Neural Network (NNet).

- **FinalizeSolutionStep**: After the solution is performed, the TS is finalized by updating internal data structures.

- **OutputSolutionStep**: Used for writing output and other postprocessing.

```
Initialize()

# RunSolutionLoop
while time < end_time:
    AdvanceInTime()
    InitializeSolutionStep()
    Predict()
    SolveSolutionStep()
    FinalizeSolutionStep()
    OutputSolutionStep()

Finalize()
```

Listing 3.1: Call sequence of simulation, including the main solution loop.

The developed common interface can be used by other applications besides CoSimulation. A prominent use case is numerical optimization (such as shape optimization), which also treats the solvers as black-box.

3.2 Integration of CoSimulation into a multiphysics framework

The implementation and realization of CoSimulation in Kratos is done inside the *CoSimulationApplication*. This application contains the different solution components and procedures for partitioned coupled simulations. The integration was realized by adhering to the workflows that are established in Kratos. The solvers in Kratos can be used as well as the external coupling partners.

The building blocks for partitioned CoSimulation were introduced in Section 2.2. The technical realization of these components is presented in the following. Figure 3.1 gives an overview of the implementation of the CoSimulationApplication, and how the individual components are connected and related to each other.

A large part of the application is written in Python, to allow for a simple extension and modification of the various components of CoSimulation. Each of them can be changed, adapted and customized to suit the needs of the corresponding use case. The high flexibility allows it to cover a wide range of engineering problems and is extensible for future requirements.

3.2.1 CoSimulationAnalysis

The *CoSimulationAnalysis* derives from and implements the interface of the *AnalysisStage* as was explained in Section 3.1.3. It is the top-level component in the hierarchy and the first interface with the user.

3.2.2 SolverWrapper

This is the interface for solvers and other tools to be integrated as coupling partners into CoSimulation. It implements the interface of the previously introduced *common solver interface* (see Section 3.1.3), with extensions to accommodate further functionalities related to CoSimulation.

The SolverWrapper is also the manager of the data used in CoSimulation. It owns a *Model*, which in turn holds the *ModelPart*s of the coupling partner. Generic access to the data is provided utilizing the *CouplingInterfaceData* (see Section 3.2.5).

Derived classes implement the specific requirements of a particular solver. These can vary strongly between the tools. For Kratos internal solvers, the calls are mostly forwarded to the corresponding *AnalysisStage*s of the solvers. External solvers on the other hand use their dedicated *IO* to communicate with the running solver instance. Typically, the *SolveSolutionStep* method is used to update the data structures in Kratos with the ones of the external tools. This oftentimes also serves as synchronization between the tools.

Remark on Kratos terminology: The *AnalysisStage* is the top-level interface to most solvers/applications in Kratos. It is thus the logical choice to be used as black-box inside the *SolverWrapper* of the *CoSimulationApplication*. However, it can lead to confusion with another important object in Kratos, the *PythonSolver*. It is owned by the AnalysisStage, and is in charge of solving the physical problem, for example, a dynamic or a static solver for a

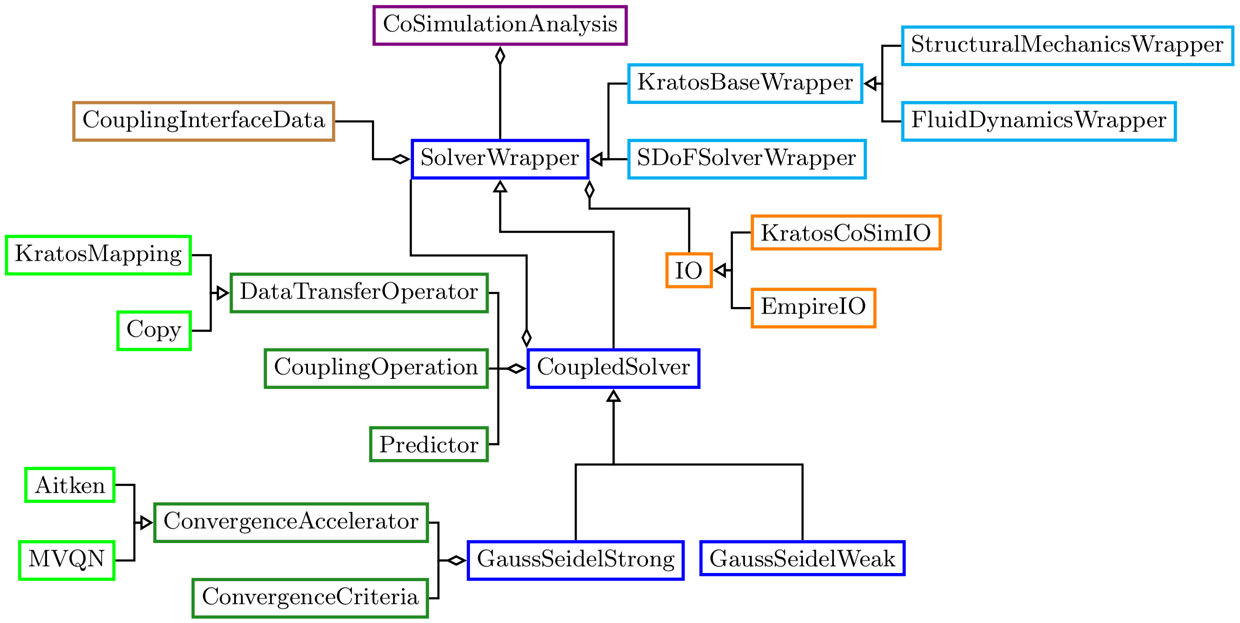


Figure 3.1: Class diagram of the implementation of the *CoSimulationApplication*. Only a subset of the important classes and their relations is shown.

structural problem. Other important tasks of the simulation are handled by the AnalysisStage, such as boundary conditions as well as input/output. Therefore, the PythonSolver is not suitable to be used within the SolverWrapper.

3.2.3 IO

The communication with external coupling tools is implemented in the *IO*. The main task is to synchronize the data of the tool with the data inside Kratos so that it can be used in the coupling. Typically, this means that different variants of IPC are implemented as presented in Section 2.2.2. As for the *SolverWrapper*, the implementation is highly specific to the external partner. Unified approaches like *CoSimIO* (see Section 3.7) can be used as well as customized solutions.

3.2.4 CoupledSolver

Different coupling strategies and communication patterns as presented in Section 2.2.1 are realized in this class. It is thus one of the most important components of a coupled simulation.

It derives from the *SolverWrapper*, which means that each coupled simulation can itself be used in another coupled simulation. This nesting leads to multi-coupling scenarios, as presented in [31] and [83].

Examples of realizing weakly and strongly coupled solvers, using the Gauss-Seidel communication-pattern, are shown in 3.2 and 3.3, respectively. 3.4 shows the details of the synchronization of data among the solvers.

```
def SolveSolutionStep():
    # initialize coupling iteration ...

    for solver in solvers:
        synchronize_input_data(solver)
        solver.SolveSolutionStep()
        synchronize_output_data(solver)

    # finalize coupling iteration ...
```

Listing 3.2: Overview of implementation for a weakly coupled Gauss-Seidel coupled solver. The synchronization of data among the solvers is shown in 3.4.

```
def SolveSolutionStep():
    for k in range(num_coupling_iterations):
        # initialize coupling iteration ...

        for solver in solvers:
            synchronize_input_data(solver)
            solver.SolveSolutionStep()
            synchronize_output_data(solver)

        # finalize coupling iteration ...

        if convergence_criteria.IsConverged():
            return True

        if k+1 >= num_coupling_iterations:
            # convergence was not achieved in this time step
            return False

        convergence_accelerator.ComputeAndApplyUpdate()
```

Listing 3.3: Overview of implementation for a strongly coupled Gauss-Seidel coupled solver. The synchronization of data among the solvers is shown in 3.4. The integration of *ConvergenceAccelerator* (see Section 3.2.8) and *ConvergenceCriteria* (see Section 3.2.9) is shown.

```
def synchronize_data():
    for data in data_list:
        # from solver
        from_solver_data = data(...)

        # to solver
        to_solver_data = data(...)

        coupling_operations.Execute("before_data_transfer")

        # conduct the transfer, e.g. with Mapping
        data_transfer_operator.TransferData(from_solver_data,
     to_solver_data)

        coupling_operations.Execute("after_data_transfer")
```

Listing 3.4: The synchronization of data among the solvers involves a *DataTransferOperator* (see Section 3.2.6) and optionally *CouplingOperations* (see Section 3.2.10).

Special coupling sequences can be realized by creating a dedicated version of the *CoupledSolver*.

3.2.5 CouplingInterfaceData

Access to the *ModelPart* of the coupling partner, which is the main container for data (see Section 3.1.1), is done through this component. Due to the large amounts of data involved in CoSimulation, efficient access to data is crucial which is ensured by avoiding copies of data if possible.

By using this intermediate object it is possible to decouple the storage of the data (e.g. nodes or elements) from the usage in the coupling algorithms. Hence, this class can be considered as an interface between the physical data that is stored depending on the physics and the generic algorithms in CoSimulation that work on *data*.

Support for distributed environments is implemented by using the *DataCommunicator* that is owned by the *ModelPart*. It is in particular important for determining whether a ModelPart exists on a certain rank or not, which is used for many coupling algorithms. Furthermore, it can be used to implement algorithms that rely on global information, agnostic of the underlying memory model.

3.2.6 DataTransferOperator

The concept of transferring data between different solution techniques was introduced in Section 2.2.3. In the CoSimulationApplication this concept is realized with the *DataTransferOperator*. It is used to implement different methods, mapping between non-matching grids being one variant.

An important aspect of distributed simulations is to have a *MPI_ Comm* that contains all ranks of the interface on both sides.

3.2.7 Predictor

Prediction/extrapolation techniques as introduced in Section 2.2.1 are used to improve the convergence of a (coupled) solution, by providing an initial guess for the solution of the new TS. This is particularly important and useful for weakly coupled simulations because no convergence check is performed like in strongly coupled simulations. This component is realized within the *Predictor* class, which can be used in different coupled solvers.

3.2.8 ConvergenceAccelerator

Strongly coupled simulations often require relaxation to achieve stable solutions. The usage of convergence acceleration techniques (see Section 2.2.1) can greatly speed up the convergence, which decreases the number of coupling iterations. Therefore, the total simulation time gets reduced. Various methods have been developed over the years, this work implements the fixed point methods constant under-relaxation and *Aitken* [50] as well as the quasi-newton methods *IQN-ILS* [20] and *MVQN* [7].

The implementation is realized with the *ConvergenceAccelerator* class. It has access to the necessary data structures and has interfaces for the integration of strongly coupled solving strategies.

3.2.9 ConvergenceCriteria

For a strongly coupled solution strategy, it is required to determine whether convergence has been achieved for the current TS. This is implemented as *ConvergenceCriteria*, where different ways of checking for convergence can be selected and specified. The necessary interfaces to access the data structures are available, as well as synchronization for distributed memory simulations.

3.2.10 CouplingOperation

Many coupled simulations require the same or a similar set of functionalities, which can be composed of the components introduced before. However, for some application cases, these functionalities might not be sufficient, and special treatment is required. In order to provide the maximum possible flexibility to accommodate as many coupled simulation scenarios, the *CouplingOperation* provides a way to customize many aspects of the coupling. The use cases range from input/output of values and fields, manipulation of data structures, and debugging output to introducing additional coupling components. Python as a programming language helps to ease the integration of custom extensions into CoSimulation.

3.3 CoSimulation in distributed environments

The support for distributed memory environments is one of the most important features necessary for solving large-scale real-world problems. Naturally, this means that also CoSimulation must be able to be usable in those cases. Special methods are used and implemented to support these systems.

3.3.1 Bringing together different parallelization methods

Aside from solvers like CFD that rely largely on large computing systems, there are many solvers and solution techniques that do not offer support for distributed systems, or where is technically not feasible to implement them. This also includes coupling partners that do not have a high computational effort and would hence not benefit a lot from the available computing resources. CoSimulation implements methods to integrate these partners such that it is completely up to them how many of the available compute resources they use.

Figure 3.2 shows the possible scenarios of how different solvers can be deployed in distributed environments in a CoSimulation context. Firstly it is important to mention that CoSimulation uses all available ranks to achieve maximum performance and simplify the integration of highly parallel solvers.

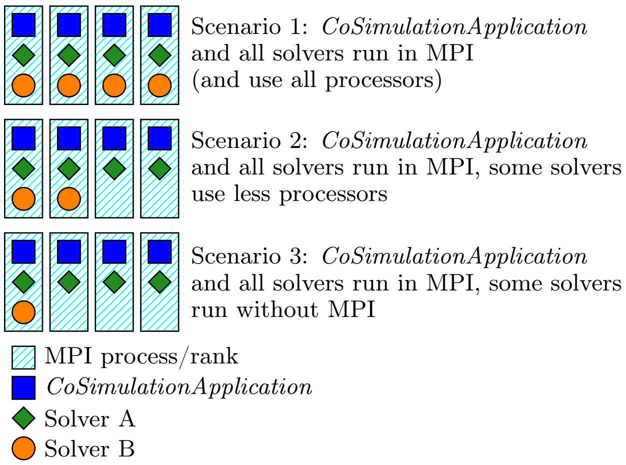

Scenario 1: *CoSimulationApplication* and all solvers run in MPI (and use all processors)
Scenario 2: *CoSimulationApplication* and all solvers run in MPI, some solvers use less processors
Scenario 3: *CoSimulationApplication* and all solvers run in MPI, some solvers run without MPI
MPI process/rank
CoSimulationApplication
Solver A
Solver B

Figure 3.2: Different scenarios for running CoSimulation in distributed environments (4 MPI-processes).

In the first scenario, all coupling partners use all available MPI processes. This is typically used when the different partners have similar computational requirements. It is also the most efficient scenario, as none of the processes is idle at any given time, since all partners and CoSimulation make use of all available resources.

In the second scenario, one or more partners use fewer processors than are available but still use MPI. This is oftentimes used when a solver performs better with fewer processors due to the size of the problem. In those cases, the problem is not large enough and the communication overhead outweighs the computational effort. Thus, the solver uses fewer cores to run at optimal performance.

This scenario is implemented in CoSimulation by assigning a dedicated *MPI_ Comm* MPI-Communicator to each solver. The parent coupled solver includes all ranks that its solvers use. Using different ranks poses a challenge mainly for data transfer, particularly mapping. The details of how this is addressed are presented in Section 4.3.

The ranks that are idle during the computation of this solver might be used by a different coupling partner depending on the employed coupling algorithm. Here especially Jacobi-type communication patterns (see Figure 2.2a) could be used which have no data dependencies among each other within one TS or coupling iteration. In this case, it can also be important to check how the jobs can be deployed on the cluster, to ensure that they can be executed at the same time.

Scenario three is probably the most common one. Here, one or more of the coupling partners do not make use of distributed computing, and run purely serial or shared memory parallel. This can have several reasons which are specific to the solver. Nevertheless, their integration into a coupled simulation needs to be done in a way that they can still be combined with solvers that employ distributed computing. This mainly affects the data transfer between the solvers, e.g. the mapping between them. Also, same as in scenario two, some ranks can be idle as no solver uses them at a time.

Example 6.5 compares different versions for parallelization, using large-scale FSI simulations of the Olympic Tower in Munich. Finally, Section 5.2 illustrates how to submit jobs on a cluster with different means of parallelization.

3.3.2 Algorithms for distributed environments

The requirements for algorithms that work in distributed environments are different since only a part of the total memory can be accessed directly. Remote memory can only be accessed by communicating the data, e.g. via MPI. This communication is overhead that should be avoided as much as possible. Conducting numerical simulations in distributed environments typically involves partitioning/distributing the computational domain (see Section 2.4.2), so that each rank can compute a part of it. This part of the domain can be directly accessed, accessing other parts of the domain can only be done via communication.

Avoiding communication is a crucial task for developing algorithms that work efficiently in distributed environments. This means that computations should be executed locally as much as possible, and the exchange of data among the ranks and synchronization should be reduced as much as possible.

One application of pure local computations is the predictors, which can be implemented to work purely on their local part of the domain, without the need for communicating or synchronizing with the other ranks. The same holds for *CouplingOperations* such as the scaling of values, which also can happen purely on local data.

On the other hand, some algorithms might require operations on the entire interface, such as the computation of norms over the entire interface for checking convergence. Here communication can in most cases not be avoided. Even more so, for more advanced algorithms such as are used for convergence accelerators, the algorithm might not be easy to implement in distributed memory architectures. In such cases, a last resort method is to gather all the data on one rank and treat it as if it were shared memory parallel. After the computation, the new data is then scattered back to its original ranks. This is slow as a lot of communication is required, and only one rank performs the heavy computations. Additionally, depending on the setup of the coupling, the memory consumption of the computing rank can increase significantly.

3.3.3 Practical considerations

When developing algorithms, it is usually beneficial to make use of established solutions, especially for complex components like linear algebra or data structures. Most programming languages provide a standard library that offers commonly used functionalities. If something is not part of the standard library, then external libraries can be used. This shortens and simplifies the development process. However, relying on external libraries that are not part of the standard library, complicates the setting up of the software. With more and more dependencies, also the risk that something can not be installed on the system that is to be used increases. This should especially be taken into account on HPC infrastructure, where it might not be possible to install some dependencies. Therefore, limiting dependencies can help to achieve maximum portability and flexibility when it comes to using software on different architectures.

From the point of view of developing algorithms for distributed architecture, it might be better in some cases to use simpler techniques and methods to limit the complexity and reduce less stable components. Furthermore, the number of libraries that support distributed architectures is much smaller, since the vast majority of computers use shared memory. This means, that the choice of libraries to use is already limited, and introducing a dependency should be carefully considered and evaluated.

3.4 Spectrum of technical realizations for CoSimulation

Partitioned CoSimulation can be done in various ways, differing in the way the coupling partners are integrated. One end of the spectrum requires none or only a small amount of modifications to the tool, but has significant overhead, both in terms of computing time and memory usage. More efficient versions of integration can be achieved, but these are typically more intrusive, and might not be possible for some solvers. Figure 3.3 offers an overview of the spectrum of integrations.

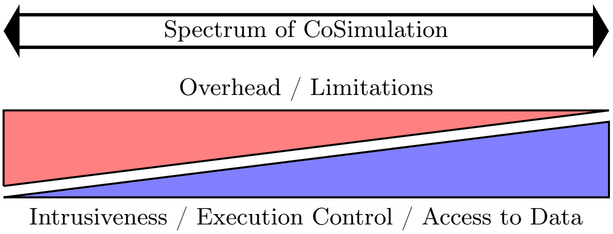


Figure 3.3: Spectrum of technical realizations of partitioned CoSimulation. The left end is characterized by large overhead and little flexibility, while the other end of the spectrum is highly intrusive, with low-level access to data structures and execution control.

This work proposes the separation of principles into four categories, which will be introduced and explained in the following.

3.4.1 Integration without any modification

The most basic approach for using a solver in a coupled simulation is by restarting it for every step of the solution. The communication happens purely with files, i.e. the input is modified before each step, and the output is read by the coupling tool, as illustrated in Figure 3.4.

This procedure is highly specific to each tool (in the way the input is written, and the output is read), and requires special (solver-) wrappers. Furthermore, it has a significant overhead, due to the continuous reading and writing of files, as well as the starting of the entire solution procedure. This approach can be used for commercial/proprietary solvers, where it is not possible to access/modify the source code, and also does not provide any CoSimulation-related functionalities like functions to modify the internal data structures.

3.4.2 Integration via data exchange routines

Implementing a dedicated API for the data exchange between the coupling partner and the coupling tool is a more elaborate way of enabling CoSimulation.

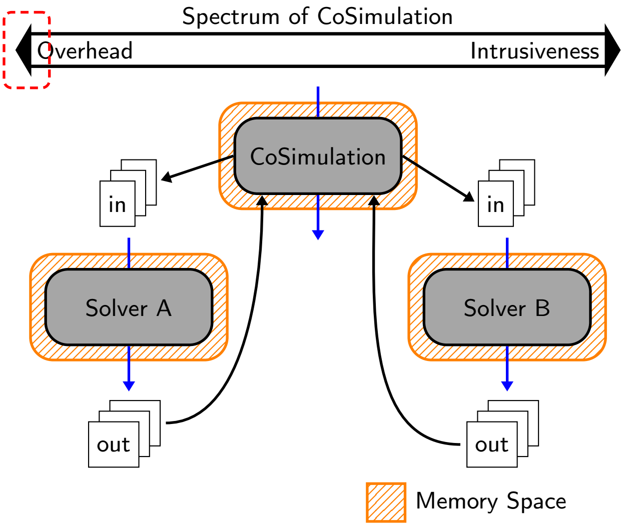


Figure 3.4: Using a solver for a coupled simulation, without any modifications to it. The solver is restarted for each solution step, based on modified input files. The output files are read by the coupling tool.

The solver no longer needs to be restarted for each solution step, hence no expensive re-initialization of data structures is necessary. The data exchange (**I**nput-**O**utput) serves both as a way for communicating the coupling data to the coupling tool, as well as for synchronization. In practical cases, this means that after the data is communicated to the coupling tool, it waits until an updated solution data is available. Figure 3.5 visualizes this way of integration.

A significant limitation of this approach is the lack of flexibility for the coupling algorithm. If the coupling algorithm is changed e.g. from a weak to a strong coupling, then the integration of the API has to be changed, e.g. by providing an additional check if the time step needs to be repeated in case no convergence of the coupled solution is achieved.

This way of integration is usually done by dedicated coupling tools. The coupling partners implement the interface of the respective tool to communicate with it. Even though this is a largely improved approach compared to the restarting approach, it still lacks flexibility. Furthermore, the duplication of data between the coupling tool and the coupling partner can cause memory problems for large problems.

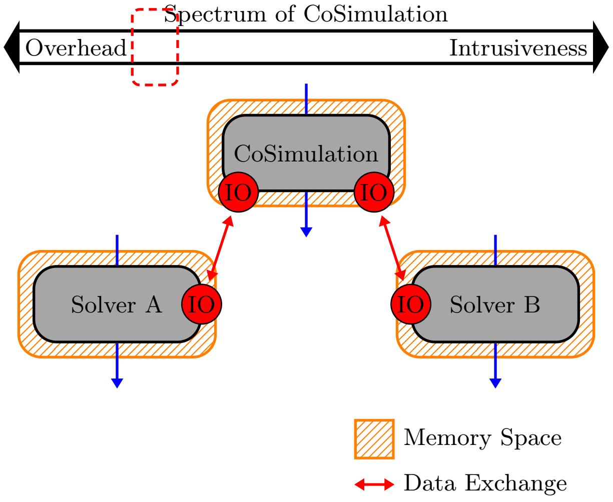


Figure 3.5: Integration via a dedicated API. The solvers implement the coupling sequence and run in separate memory spaces. IO can be done with different methods of IPC.

3.4.3 Integration by providing control for CoSimulation

This work proposes a new approach, in which the synchronization with the coupling partner is no longer done via the data exchange. Instead, full execution control is given to CoSimulation, by providing a set of functions that can perform different tasks. This means, that the coupling algorithm is no longer reflected in the source code of the coupling partner. The order in which the individual parts of the solution procedure are executed is left completely to CoSimulation, and thus any coupling algorithm/solution procedure can be realized without any changes in the source code of the solver. Furthermore, the possibility for deadlocking is greatly reduced, since the flow of execution is controlled from one single component. This approach is titled *remote-controlled CoSimulation*, as it has full execution control to orchestrate the coupling. It requires a high flexibility of the coupling partner, which is a disadvantage of this approach.

The different parts of the solution procedure are the functions offered by the interface of the *SolverWrapper*, as well as procedures for the exchange of data. Hence, this approach can be seen as an improvement and superset of the integration via data exchange. Figure 3.6 shows this approach.

Some tools like MpCCI (see Section 2.5) also allow registering functions for different tasks in the coupling tool. However, these are focussed on the coupling functionalities like data exchange (and thus only a different realization of the approach presented in Section 3.4.2), and not the top-level

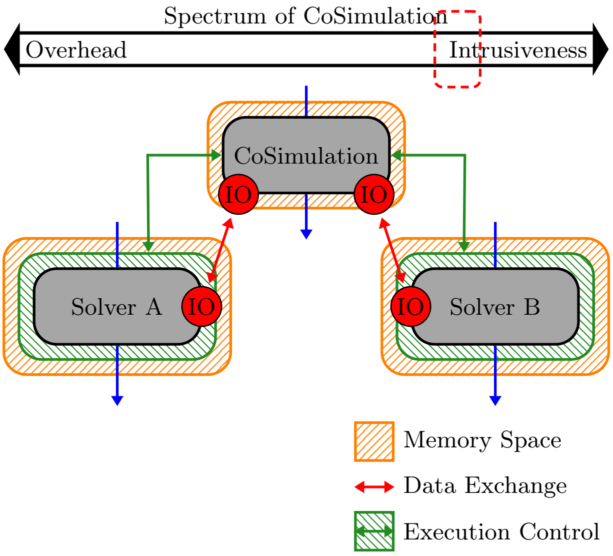


Figure 3.6: Remote controlled CoSimulation: Each partner provides a set of functions to execute different parts of the solution procedure. CoSimulation has full execution control and orchestrates the coupling. The data exchange is not used for synchronization.

control of the entire solver, as is proposed here.

3.4.4 Integration within the same framework

This approach is the most flexible yet most intrusive. The coupling partner and coupling tool are part of the same framework, which is the case for most multiphysics tools. The data of the coupling partner can be directly accessed by the coupling tool, without any communication via IPC. Due to the shared memory space, it is also the most memory-efficient approach, which makes a difference, especially for large coupled problems. Figure 3.7 presents this approach. Same as in the previous approach, the solvers can be directly controlled by the coupling tool if they share the same interface.

3.5 CoSimulation with Kratos internal solvers

Kratos offers numerical solution techniques for different types of physical problems. Enabling multiphysics requires CoSimulation functionalities which are developed and implemented within the scope of this work.

One of the main algorithmic developments is the unification of the solver interfaces with the *AnalysisStage*, see Section 3.1.3. This is a first step

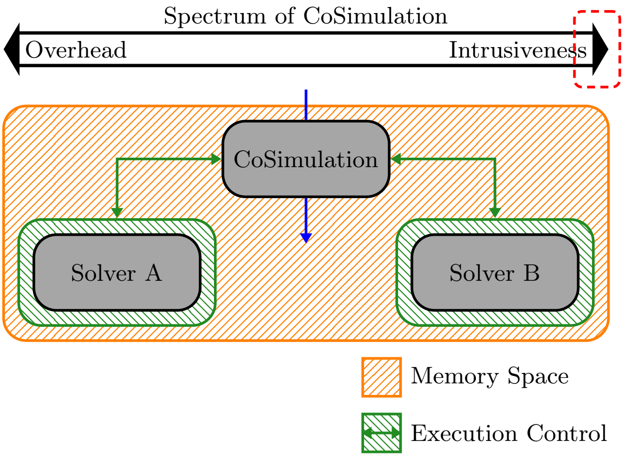


Figure 3.7: The solvers and the coupling tool are part of the same framework. They run in the same memory space, thus no data communication via IPC is necessary. Furthermore, data duplication is avoided. Only possible with multiphysics tools.

towards the realization of CoSimulation within Kratos. The interface of solvers to be integrated into CoSimulation is the *SolverWrapper*. Because the interfaces of SolverWrapper and AnalysisStage are compatible, the integration of Kratos solvers is achieved with minimal effort. The CoSimulation can directly access the data structures of the solvers in the form of the *ModelPart*, which completes the integration. This means that no data exchange between different processes via IPC is required. Furthermore, no memory overhead occurs as no duplication of data is necessary.

Accessing the ModelPart directly also has a disadvantage: Since the data of the solvers is accessed directly it can also be dangerous if the wrong data is modified at the wrong time. I.e. some data may be overwritten that is not supposed to be changed. Even though the approach is black-box, still the boundaries become blurry in the case of directly accessing the solvers' database.

3.6 Coupling to external solvers/tools

Even though Kratos offers many solution techniques, providing CoSimulation only for the solvers of Kratos would severely limit the applicability, as many external tools exist which can be included to perform coupled simulations. Hence, solutions are developed for the integration of Kratos-external coupling partners into CoSimulation.

As for Kratos solvers, the interface for coupling partners is the *Solver-Wrapper*. The provided interface can be used to perform solver-specific tasks, which depends highly on the tool to be integrated.

Unlike the Kratos internal solvers, the data structures of external coupling partners can not be directly accessed. This means that the data involved in the coupling needs to be communicated to Kratos via IPC. This is done with the *IO*. Standard solutions are provided, but it is also possible to use solutions custom for each solver.

The data of an external coupling partner is saved within a *ModelPart*, which makes the uniform treatment of solvers internal or external to Kratos possible. This means, that after the data is communicated to Kratos and stored in a ModelPart, it no longer matters where the data originally came from. This leads to a uniform treatment of internal and external solvers in CoSimulation.

3.6.1 Detached Interface

Integrating an external library or framework and its interface/API is a challenging task. This applies generally and is not limited to CoSimulation. The larger and more complicated the interface and the library are, the more effort needs to be spent on the integration. Additionally, if the library has dependencies itself, then those also need to be available. Large codes like Kratos have many dependencies, which can be tough to install, especially on special infrastructures like HPC systems. Also, different versions of dependencies might not be compatible which needs to be taken into account. Therefore, it is best to avoid dependencies as much as possible to reduce possible issues with integrating and using an external library. Of course, this does not hold for all cases, especially when avoiding a dependency would result in large development efforts. Hence, adding a new dependency needs to be carefully evaluated and assessed each time.

For the reasons mentioned above, a special interface is developed to simplify the integration of CoSimulation capabilities. As explained in [31], the proposed interface is independent of Kratos and thus fully *detached* from the coupling framework. To keep it as small and lightweight as possible, it contains the least amount of functionality required to enable CoSimulation: The API and IPC. The connection and communication with the CoSimulationApplication is done by using IPC. All the functionalities required for CoSimulation such as mapping or coupled solvers are part of Kratos. Without the tight coupling of the tools, it is furthermore possible to deploy them independently. The connection is made only at runtime. Figure 3.8 visualizes the concept of the detached interface.

3.7 Realization of detached interface: CoSimIO

CoSimIO is an implementation of the detached interface as presented in Section 3.6.1. It is a small tool that offers data exchange via IPC for CoSimulation contexts. The goal was to create a tool that provides a very easy integration. This way it becomes very simple to realize CoSimulation.

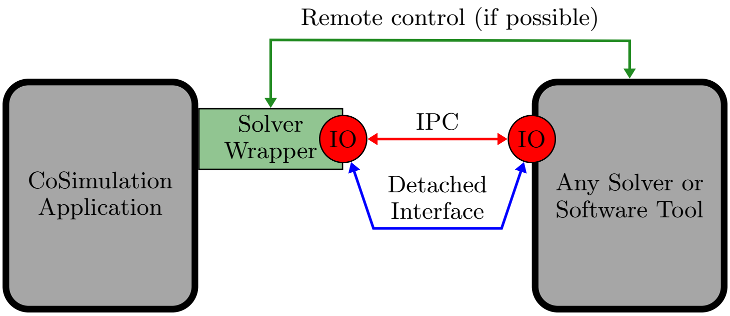


Figure 3.8: Detached interface: A small interface which is implemented by solvers/tools, without dependencies between each other. The connection is made at runtime, the data can be exchanged with different methods of IPC.

It follows the same open-source concept as Kratos, and is freely available[3], with a permissive BSD-3 license.

3.7.1 API of CoSimIO

Aside from the integration of CoSimIO from the software engineer's point of view, integrating the API is crucial for enabling CoSimulation. The right calls have to be made at the correct places. The clear and crisp interface supports this, hence the elaborate and well-thought-through design of the public API of CoSimIO is presented and explained.

Independent of the type of information to be exchanged, the first step in any coupled simulation is to establish a connection between the tools. At the end of the simulation, this connection is closed by disconnecting. Once a connection is established, then the exchange of information can occur. This is realized through the *Connect* and *Disconnect* functions. Both partners need to call both functions.

The exchange of data occurs once a connection is established. One partner exports the information, while the other one imports it. This is equivalent to a sender/receiver concept. It is important to decide on the data and its types. CoSimIO supports three basic types of data: Data, Mesh and Info.

For remote-controlled CoSimulation as introduced in Section 3.4.3, two additional functions are necessary. The *Register* function is used to register the procedure calls supported by the external tool. These are then called during the execution. After the necessary functions are registered, the *Run* method executes the entire coupled simulation.

This set of functionalities leads to the following API of CoSimIO:

[3] `https://github.com/KratosMultiphysics/CoSimIO`

- **Connect**: This method needs to be called first, it prepares CoSimIO for communication. At first, it does some basic checks with the connection partner to check for compatibility, for example, version and other options. The second step is to initialize the selected method of IPC, for example by opening network ports. The third and final step is to make the connection with the partner, for example by connecting to the port that was opened by the partner. Since this procedure requires both partners to be ready for connection, the function only returns once the connection is fully established. This way, any kind of race conditions with the following functions that communicate data are avoided.

- **Disconnect**: At the end of the coupled simulation, after all the data is exchanged and both partners are done, the last step is to end the connection. The main part is to close the IPC, which entails for example disconnecting and closing ports.

- **Data exchange** (*ExportData* and *ImportData*): An variable-sized array of data represents the most basic form of information. Usually, the data type is a floating point number such as *double*. This information can be associated with a mesh and represent a field (e.g. Pressure or Displacement). It can also be independent, and represent quantities such as sensor input or other signals which are not associated with a geometrical location.

- **Mesh exchange** (*ExportMesh* and *ImportMesh*): Numerical simulations that deal with spatial discretization often use meshes for storing geometrical information. The domain is discretized into small elements or cells, which are connected by nodes at their corner points. This type of information is required for the mapping of data between different mesh sizes and is hence commonly exchanged in coupled simulations.

- **Info exchange** (*ExportInfo* and *ImportInfo*): Information other than data or meshes can be collected with a special *Info* object. It uses a key-value map to store different information associated with an identifier. The key is a string, while the value can be of type *integer*, *double*, *bool* or *string*. Nesting of *Info* objects is also possible. Typical examples are settings, parameters and other meta-data that need to be exchanged. This general mechanism for exchanging information means that no special methods need to be created for every new kind of data that needs to be exchanged. With the *MPI_Info*, the MPI standard defines a very similar object[4]. The main difference is the type-safety and MPI indepencence of the CoSimIO implementation, which simplifies the usage significantly.

- **Register**: For remote-controlled CoSimulation it is required by the external solver to register functions that can perform various steps in the solution procedure. Examples are solving the time step, or the import/export of data. These need to be registered such that they

[4] `https://www.mpi-forum.org/docs/mpi-2.1/mpi21-report-bw/node194.htm`

can be called by the controlling instance, depending on the solution algorithm.

- **Run**: After all the functions required for the solution are registered, the external tool calls this method to relinquish control to the remote instance. Internally it continuously waits for a signal requesting the execution of a specific, previously registered function. Which calls are being made and in which order hereby depends on the chosen solution algorithm. For the calling tool, this function only returns after the entire CoSimulation has finished.

The last step towards integrating the API is to decide on the level of integration of the external tool. As explained in Section 3.3, different approaches can be used. CoSimIO provides two options:

3.7.1.1 Classical CoSimulation

This approach implements what was introduced in Section 3.4.2. The synchronization between the tools happens with the information exchange routines. No additional API calls are therefore necessary. It is a basic approach that lacks flexibility, as the coupling sequence is duplicated and needs to match the one in the coupling tool. Hence, every change such as weak to strong coupling requires changes in the source code.

3.5 illustrates how the integration of CoSimIO is done for weak coupling. In contrast, 3.6 shows the same but for strong coupling. Their comparison indicates that changing the coupling algorithm requires several changes in the implementation, and thus is not flexible. It is of course possible to implement several known algorithms, but exploring other ways of coupling is not possible without source code modifications.

```
# solver initializes ...

CoSimIO::Connect(...) # establish connection

CoSimIO::ExportMesh(...) # send meshes to the
    CoSimulationApplication

# start solution loop
while time < end_time:
    CoSimIO::ImportData(...) # get interface data

    # solve the time step

    CoSimIO::ExportData(...) # send new data to the
    CoSimulationApplication

CoSimIO::Disconnect(...) # stop connection
```

Listing 3.5: Integration of CoSimIO using the classical approach (weak coupling)

```
# solver initializes ...

CoSimIO::Connect(...) # establish connection

CoSimIO::ExportMesh(...) # send meshes to the
     CoSimulationApplication

# start solution loop
while time < end_time:
    is_converged = false
    while(!is_converged): # convergence iterations
        CoSimIO::ImportData(...) # get interface data

        # solve the time step

        CoSimIO::ExportData(...) # send new data to the
     CoSimulationApplication

        # receive convergence signal
        info = CoSimIO::ImportInfo(...)
        is_converged = info["is_converged"]

CoSimIO::Disconnect(...) # stop connection
```

Listing 3.6: Integration of CoSimIO using the classical approach (strong coupling)

3.7.1.2 Remote controlled CoSimulation

Providing full execution control to CoSimulation as explained in Section 3.4.3 has distinct advantages over the classical approach: The external tool does no longer need to implement any coupling logic, as everything is controlled by CoSimulation. This implies maximum flexibility in terms of the choice of coupling algorithm. The solver provides procedures that perform different parts in the coupling logic, for example, the solution of the TS, or the export of data. After these are registered, the *Run* method is used to execute the simulation. It receives a signal from CoSimulation which part of the coupling needs to be executed, and then calls the provided routines. Thus, the order of execution is left entirely to CoSimulation. Any coupling logic can be used without having to change the implementation. 3.7 gives an example of this approach. The implementation is valid both for weak and strong coupling and does not require changes to the source code, unlike in the classical approach. Due to the centrally managed flow of execution, the chance for deadlocks (e.g. when two partners are waiting for data due to a wrongly configured coupling) is greatly reduced. This helps the debugging and setting up of new cases greatly.

The downside of this approach is that it not only offers a high flexibility when it comes to the choice of coupling algorithm, but it requires also a high level of flexibility by the solver. The implementation and software design needs to be done in a way that allows the arbitrary execution of different steps in the solution procedure. In particular, the memory allocation and layout are affected, as in most simulation scenarios the order of execution is the same.

```
# solver initializes ...

CoSimIO::Connect(...) # establish connection

# defining functions to be registered
def SolveSolution():
    # external solver solves timestep
    solver.solve()

def ExportData():
    # external solver exports data to the CoSimulationApplication
    CoSimIO::ExportData(...)

# after defining the functions they can be registered in CoSimIO:

CoSimIO::Register(SolveSolution)
CoSimIO::Register(ExportData)
# ...

# After all the functions are registered and the solver is fully
# initialized for CoSimulation, the Run method is called
CoSimIO::Run() # this function runs the coupled simulation. It
    returns only after finishing

CoSimIO::Disconnect(...) # stop connection
```

Listing 3.7: integration of CoSimIO using the remote control approach

This new approach for conducting coupled simulations was successfully applied by [54], where a FSI simulation of a flexible membrane blade is performed. Kratos was used both for the structural solution and the coupling with the CoSimulationApplication and CoSimIO. The CFD solver TAU [72] was coupled by registering the functions required for the different solving stages within CoSimIO. The contribution of the author to this work is assisting in the development of the coupling between the solvers.

3.7.2 Programming language and software architecture

The programming language C++ is used for the implementation of CoSimIO since it is a good combination of efficiency, support on modern systems, the complexity of implementation, and availability of features required for CoSimulation. Furthermore, it is also the language in which the core of Kratos is written. Hence, no additional requirements are necessary for using CoSimIO alongside Kratos.

To provide maximum portability, the C++11 standard (version 2011) is used. It is already over 10 years old at the time of this writing, which means that it is very stable and mature by now and available on most systems that are relevant for conducting numerical simulations. Even though newer versions of the language are available, they are deliberately not used. This is because the benefits of the new features do not outweigh the added installation effort, in particular on older systems. This aligns also with the requirements for providing a simple integration into existing tools.

Despite the popularity of header-only libraries in C++, a classical library approach is used: CoSimIO is compiled into a shared library with the headers included by the external tool, and afterward linking against the library. While this requires the additional linking step, it also has several important advantages. The compilation times are smaller, but even more importantly no large headers (like *windows.h*) are leaked into the integrating code which could cause code bloat and other issues.

The implementation is based mostly on the standard library of C++. The base version of CoSimIO only has two dependencies which are very carefully chosen and checked for maximum portability:

- *filesystem*: Interacting with and handling files and directories is often done in CoSimIO. The C++17 standard offers these capabilities within *std::filesystem*[5], but for earlier versions it is necessary to use OS specific function calls for performing the same tasks. Therefore, the *filesystem* library[6] is used, which is a C++11 compatible implementation of C++17-filesystem. This header-only library facilitates the interaction with files and directories with the same API, meaning that it can be exchanged for the integrated version of C++ once version 17 is used.

- *ASIO*: Networking is used in CoSimIO for IPC via sockets. Similar to the interaction with files and directories, C++ does not offer a native way to do networking. At the time of this writing, this applies also to the latest version, which currently is C++20. Therefore, the *ASIO* library[7] is used for the communication of data via sockets. This header-only library is being developed for almost 20 years and is the de facto standard for networking in C++. It might be integrated into C++ in the future as *std::net*[8].

3.7.3 Implementation details

Section 3.7.2 contained a brief overview of the overall software architecture and general design decisions. This section complements it by presenting and explaining the crucial parts of the implementation in detail.

The main functionality of a detached interface is the data exchange via IPC. Several methods are implemented, and they can be selected at runtime. All of them share a common interface, which is implemented with the *Communication* class. Connecting and disconnecting are implemented in dedicated functions. Furthermore, procedures for exchanging data, mesh and info are provided. This follows the API as presented in Section 3.7.1. In principle, any data structure that supports serialization (see Section 2.2.2.4) can be exchanged.

The main outline of the implementation is shown in 3.8. The base class implements two crucial tasks, namely the initial handshake and the serialized data exchange: Before any connection is established, the two partners perform a handshake, which consists of synchronization among them as well as

[5] `https://en.cppreference.com/w/cpp/filesystem`
[6] `https://github.com/gulrak/filesystem`
[7] `https://think-async.com/Asio/`
[8] `https://cplusplus.github.io/networking-ts/draft.pdf`

exchanging some basic information which is used for checking compatibility. This is explained in more detail in Section 3.7.6.

IPC can only handle basic data types for the data exchanges, such as arrays of doubles or characters. Complex data structures, however, such as meshes or info, need to be serialized to a stream of bytes for the exchange. Afterward, they need to be reconstructed by deserializing them back into their original data structure. This is generally slower as it introduces the additional serialization/deserialization step, but it is much more flexible in the sense of the data to be exchanged. Any type of data structure can be sent and received via a network in this way.

```cpp
class Communication
{
public:
    // Constructor
    Communication(...);

    Info Connect(...);
    Info Disconnect(...);

    Info ExportInfo(...);
    Info ImportInfo(...);

    Info ImportData(...);
    Info ExportData(...);

    Info ImportMesh(...);
    Info ExportMesh(...);

private:
    void HandShake(...);

    // direct data exchange
    void SendData(...);
    void ReceiveData(...);

    // serialized data exchange
    void SendString(...);
    void ReceiveString(...);
};
```

Listing 3.8: Base class of Communication class. Only the main functions are shown

Serialization in CoSimIO

Serialization as presented in Section 2.2.2.4 is extensively used in CoSimIO for communicating data between the two connection partners. Mesh and Info have complex memory layouts, and thus cannot be directly exchanged via IPC. A serializer similar to the one of Kratos is used, which can use ASCII or binary format.

3.7.3.1 Container for Data

Arrays of data are among the most commonly exchanged formats of data in a coupled simulation. Usually, field data such as forces and displacements in

FSI fall in this category. Not only can they be exchanged multiple times per TS, but also they can contain lots of data. The size of the data is equal to the number of nodes on the interface times the dimensionality. Therefore, handling this type of data efficiently is crucial for performing CoSimulation.

The solution developed in this work is a generic container that can not only store different types of data but **does not require any copying while interfacing between different programming languages**. The interface as shown in 3.9 is inspired by the commonly used C++ data container *std::vector*[9]. It is a variable-sized container storing its data contiguously in memory and provides $\mathcal{O}(1)$ access to its data by index. This abstraction of access to the data enables to use the container inside the other parts of CoSimIO agnostic of the underlying data storage. The actual storage of data is implemented by the derived classes. For C++ (and Python), *std::vector* is used, while for C and Fortran raw arrays are used. Methods for serialization are provided, even though the contiguously stored data can be directly exchanged by most IPC methods.

```cpp
template<typename TDataType>
class DataContainer
{
public:
    // Constructor
    DataContainer();

    virtual std::size_t size();
    virtual void resize(...); // potentially (re-)allocates memory

    virtual TDataType* data(); // access to the underlying raw data

    TDataType& operator[](...); // index based access

private:
    // Serialization
    virtual void save(...);
    virtual void load(...);
};
```

Listing 3.9: DataContainer for storing raw data, inspired by std::vector. Provides storage agnostic access to the underlying data for usage in CoSimIO. Data is not copied while interfacing languages.

3.7.3.2 Container for Mesh

Meshes are used by most simulation tools since discretization techniques such as FEM or FVM split the computational domain into many subdomains. These are commonly referred to as elements or cells, which are connected through nodes at their corners. Due to the crucial role that meshes play in numerical simulation, a dedicated data structure was created. The *Mesh* consists of *Elements* and *Nodes*, which in turn have their own data types. Each of them is briefly described in the following. The composition of different objects makes the *Mesh* a hierarchical data structure, which represents the

[9] https://en.cppreference.com/w/cpp/container/vector

geometry of the simulation model. Serialization is used when exchanging data through IPC due to the complexity of the data structure.

The corresponding data structure of Kratos was used as a reference and inspiration. Not only to ensure compatibility but also due to the established and mature API. This object is called *ModelPart* and was introduced in Section 3.1.1.

Meshes play an important role in CoSimulation when it comes to the data transfer between them. Values need to be mapped from one mesh/solver to another. Many different methods exist, see Chapter 4. In particular non-matching mapping is important, since it enables the solvers to use different discretizations. This means that they can use what is best suited for their solution method, without being limited by CoSimulation. Even for matching meshes the access to the discretization is usually required since the ordering of the nodes and elements between solvers is different.

Nodes represent the corners of the elements. They are small objects that store coordinates along with a unique identifier. Their basic implementation is shown in 3.10. The basic access functions are complemented with serialization methods.

```cpp
class Node
{
public:
    // Constructor
    Node(...);

    // Member access
    IdType Id();
    double X();
    double Y();
    double Z();
    CoordinatesType Coordinates();

private:
    // Serialization
    void save(...);
    void load(...);
};
```

Listing 3.10: Node, representing the corner of elements.

Elements as (typically small) parts of the computational domain have a geometrical shape associated. This depends strongly on the original model and the meshing technique used for the discretization. Typical shapes include, but are not limited to: lines (1D), triangles and quadrilaterals (2D) as well as tetrahedrons and hexahedrons (3D). Their topology dictates the number of nodes required. Hence, an element has an associated type/geometry/topology, a list of nodes (which can be shared with other elements) and a unique identifier. Serialization and element also involve the serialization of its nodes and are therefore hierarchical. An outline of the implementation can be seen in 3.11.

```cpp
class Element
{
public:
    // Constructor
    Element(...);

    // Member access
    IdType Id();
    ElementType Type();
    std::size_t NumberOfNodes();
    NodesContainerType& Nodes();

private:
    // Serialization
    void save(...);
    void load(...);
};
```

Listing 3.11: Element, small part of the computational domain. Nodes for the corners.

The *ModelPart* contains both nodes and elements and thus a part, or the entire simulation model. It is used in the mesh exchange functions *ExportMesh* and *ImportMesh*. Different ways of creating and accessing nodes and elements are provided, depending on the needs of the integrating tool, see 3.12. Same as for the elements, the serialization of the *ModelPart* invokes the same methods hierarchically for its nodes and elements. The interface for distributed simulations with local and ghost nodes is shown too. This will be explained in more detail in Section 3.7.5.

```
class ModelPart
{
public:
    // Constructor
    ModelPart(...);

    // Member access
    std::string& Name();

    std::size_t NumberOfNodes();
    std::size_t NumberOfLocalNodes();
    std::size_t NumberOfGhostNodes();

    std::size_t NumberOfElements();

    Node& CreateNewNode(...);
    Node& CreateNewGhostNode(...);

    Element& CreateNewElement(...);

    NodesContainerType& Nodes();
    NodesContainerType& LocalNodes();
    NodesContainerType& GhostNodes();

    ElementsContainerType& Elements();

private:
    // Serialization
    void save(...);
    void load(...);
};
```

Listing 3.12: ModelPart, which contains part of the simulation model and consists of many nodes and elements.

3.7.3.3 Container for MetaData: Info

Data exchange does not only involve arrays of floating-point numbers or meshes. Metadata is an important type of data that is especially relevant for tools other than numerical simulation software. These tools usually do not work with meshes, hence their data might not be associated with a geometrical location. A unique identifier helps to distinguish the values. Settings or other configuration inputs are further examples that can be represented as metadata. Metadata can therefore be classified as key-value pairs.

The proposed *Info* can be used to store metadata. It stores different types of data as key-value pairs. The interface is inspired by the C++ *std::map*[10] and the Python *dict*[11] implementations. Another example of key-value pair data is the *json*[12] file format, which is often used for input files where many different settings are specified. Kratos uses this format for all non-mesh input files.

Getting and setting values by their key are the two main functionalities. Basic data types such as int, bool and double are supported. Additionally, it is possible to nest the objects, enabling to use complex hierarchical structures. Serialization is provided, which uses the previously introduced mechanisms.

```
class Info
{
public:
    // Constructor
    Info();

    // Access functions
    template<typename TDataType>
    TDataType& Get(Key);

    // returns the provided default if the key does not exist
    template<typename TDataType>
    TDataType& Get(Key, Default);

    template<typename TDataType>
    void Set(Key, Value);

    // Helper functions
    bool Has(Key);
    void Erase(Key);
    void Clear();
    std::size_t Size();

private:
    // Serialization
```

[10]https://en.cppreference.com/w/cpp/container/map

[11]https://docs.python.org/3/tutorial/datastructures.html#dictionaries

[12]https://www.json.org

```
    void save(...);
    void load(...);
};
```

Listing 3.13: Info, storing metadata in the form of key-value pairs

3.7.4 Exposing to other programming languages

Solvers and tools in general that are involved in CoSimulation are written in various languages. Hence, enabling the usage of CoSimIO in programming languages other than C++ greatly expands its usability. The goal hereby was to not only make it available but to offer the functionalities of CoSimIO in a way that is native and intuitively integrated with the respective language. This entails for example no manual memory management when integrating with Python or using native data types and structures. Furthermore, any duplication of data or other overhead when interfacing between languages shall be avoided to maximize performance and minimize memory footprint.

Some of the most commonly used languages for simulation tools are Python, C and Fortran. The latter two are comparatively old low-level languages and have been used extensively to write numerical software. Nowadays, Python is very popular due to its simple syntax and a large ecosystem of libraries. It is a very high-level language, which can be combined with other languages to achieve high performance with easy usage. Kratos is one example, which combines a user interface written in Python with a highly efficient and fast core implemented in C++.

3.7.4.1 C

C is one of the oldest and commonly used low-level programming languages, it is compiled and type-safe. It is heavily used for complex applications such as the Linux kernel and other highly performant software. It is the predecessor of C++ and was first standardized in 1989 (known as the C89 standard) for the first time. Since then the language has had several revisions, however nowadays, over 30 years later, using the first standard is not uncommon due to its portability. C is considered a *lingua franca* (common/bridge language), it offers interfaces to many other programming languages such as C++, Fortran, Python, Rust and Java. This makes it an ideal choice as a basis for applications or libraries that are then used in different languages. This makes C a perfect choice for making it available through CoSimIO, as it can then be either directly used by other languages, or in native C codes. The compatibility with C++ is very good given it is based on C. Still, C programming is different from C++, especially when it comes to memory management. Since this is a crucial part of CoSimulation and can severely affect the performance of a coupled simulation, this work presents solutions for memory management in CoSimulation. Additionally, the interfaces of C++ CoSimIO are ported to a native version for C, assisting a smooth integration.

Exchanging data to and from CoSimIO is realized without any copies between data structures and hence without overhead. The *DataContainer* introduced in Section 3.7.3.1 operates with the raw pointers from C internally by abstracting them to a high-level interface. This enables native usage in C++ without compromising performance. For the export of data, the array of data together with its size is passed to CoSimIO, which can directly use it. For the import of data, however, it is not possible upfront to allocate memory by the integrating solver, since the size of the incoming data is not known. The *DataContainer* handles this situation by allocating the memory internally before the data is received. This avoids any overhead and realizes a consistent interface for importing and exporting data.

The data structures *Mesh* and *Info* have their C-versions, which underlying use the C++ counterparts. Again this means that any expensive copying of data is avoided while providing a language-native interface. Finally, also the functions required for CoSimulation orchestrated simulations can be registered in CoSimIO by passing raw function pointers.

3.7.4.2 Fortran

Similar to C, Fortran is an old and established language, compiled and type-safe, and developed for over 50 years. It is used in particular for numerical calculations in scientific and high-performance computing. Before the rise of more modern languages such as C++ or Rust, many simulation codes were written in Fortran. Even nowadays it is still heavily used, in particular in the backends of solvers, which have matured over many decades. Providing integration with this language opens many usage opportunities for CoSimulation via CoSimIO.

The Fortran interface is based on the C interface, which serves as a bridge to C++. This realization was selected because Fortran offers interoperability with C through the *iso_c_binding* module (since Fortran 2003). Therefore, no additional implementation is required, and the routines developed for the C interface can be directly reused and made available in Fortran.

Same as for the C interface, the memory management is again realized without the overhead. Memory of raw data can be passed through pointers, the objects *Mesh* and *Info* have their Fortran native versions, which under the hood use the C++ counterparts. Pointers to functions for orchestrated CoSimulation can be passed by using a dedicated data type offered by the interoperability functionalities.

3.7.4.3 Python

Python has grown very popular in recent years and is heavily used by many different applications. Unlike C and Fortran, it is a high-level and interpreted language. Therefore, it does not offer the same level of performance, but instead a simpler syntax and larger ecosystem of auxiliary libraries. Memory management is done automatically, including garbage collection. The inferior performance compared to many compiled languages is compensated to a degree by using libraries such as numpy[13]. They implement computationally expensive functionalities very efficiently in low-level languages and then expose

[13] `https://numpy.org/`

their interface to Python. This concept has been successfully applied by many libraries, including Kratos.

Due to its popularity, it was used for many projects, among them are also tools that can be used in CoSimulation. These are either simple numerical solvers, NNet codes like TensorFlow[14], or other codes. Furthermore, what makes it especially useful for CoSimulation, is that more and more solvers develop Python interfaces on top of their core functionality written in low-level languages. A Python interface simplifies the integration for such solvers, as it can be done by scripting and does not require access to the source code. Commercial solvers without a dedicated API for their low-level functionality can still be used in CoSimulation if they provide a Python interface.

Given the manifold of use cases, a Python interface was developed for CoSimIO. It follows the same goals of the integration in the lower-level languages, namely a native interface with high performance and no memory overhead. The pybind library[15] (which is also used by Kratos) was employed for exposing the C++ core of CoSimIO to Python. This again required some special solutions for managing the memory when interfacing with the languages. Python holds a pointer to the C++ objects which is passed between the languages and therefore involves no copies. The objects are complemented with native interfaces.

3.7.5 Extension to distributed environments

Parallel computing via MPI as introduced in Section 2.4.2 is commonly used in numerical analysis to reduce the simulation times. The natural choice was therefore to provide support for distributed environments with CoSimIO. The main differences to the serial version are the communication among the computing ranks, the integration of the MPI-library without affecting the serial version, the handling of distributed meshes, and most importantly the IPC methods for distributed environments.

The communication and synchronization among the ranks of CoSimIO are done with a similar concept to Kratos (see Section 3.1.2). The raw *MPI_Comm* is wrapped in an auxiliary object, which in serial does nothing, but in MPI acts as an abstract wrapper for MPI calls. This is the *DataCommunicator*, as explained in Section 3.1.2.

The MPI extension is built on top of the serial version, with only one directional dependency. It is implemented in a separate shared library, which links against the serial version. This is crucial for deploying on systems without the MPI libraries present. Even though CoSimIO was compiled with MPI support, it will only use the serial part since the MPI part is separated. It is important to mention that CoSimIO as a library does not interfere with the initialization and finalization of MPI. This is left to the including code to avoid runtime and memory problems. The *MPI_Comm* to be used by CoSimIO is being passed from the application. It needs to contain at least the ranks that have a part of the interface.

[14]https://www.tensorflow.org/

[15]https://github.com/pybind/pybind11

MPI is by design used for exchanging messages among different ranks/processes of the same program. It also provides functionalities to exchange messages between different codes. This means it can be used as a means of IPC for CoSimIO, for communicating between the coupled codes. A performance evaluation, as well as implementational details, are provided in Section 3.8.

The communication is done 1:1, meaning that both programs are running with the same number of processes. Each rank of a solver has a dedicated partner rank of the coupling tool. This design choice reduces severely the complexity of the implementation, as no means for gathering and distributing data when different numbers of processes are used on both sides are involved. What appears to be a limitation of the presented approach is its strength, namely the distribution of responsibilities as introduced in Section 2.2. CoSimIO as a detached interface is responsible only for the data exchange between the tools, which keeps it small and lightweight compared to fully featured coupling tools. Bringing together codes that run with different numbers of processes is handled in the CoSimulationApplication. In particular, the mapping is designed in a way that it can handle any number of processes on either side of the interface. Each code can thus use the number of processes that it works best with. Therefore, the proposed solution offers maximum flexibility for choosing the best suitable amount of computational resources, while at the same time limiting complexity and delegating responsibilities to separate parts of CoSimulation.

Lastly, support for distributed interfaces (meshes) is provided. Nodes can be present in multiple ranks, local in one and ghost in many. The interface of the *ModelPart* is enhanced with an interface for constructing ghost nodes. These are essential for ensuring correct communication among the ranks, see [13].

3.7.6 Initial handshake

The initial connection between two programs needs to be made in order to initialize IPC and to perform some compatibility checks e.g. for the versions of CoSimIO used by the codes. This is called a handshake. Afterward, the connection is fully initialized and can be used to exchange data.

The handshake is a crucial part of CoSimulation and follows an elaborate algorithm to avoid deadlocking and other issues. First, a distinction is made between the coupling partners, regarding which of them is the primary and which is the secondary connection. This is required as some operations need to be made by one side only. Examples are the removal of leftover files, or establishing a connection for some IPC methods such as sockets, where one side opens a port, and the other one connects to it.

Algorithm 2 shows the procedure of the handshake, the main steps are as follows: *Step one* is the removal of any leftovers from previous runs such as leftover communication files, which could be misinterpreted and lead to errors. CoSimIO uses a dedicated folder inside the working directory for the file exchange. This folder gets deleted only by the primary side to avoid blocking. *Step two* is to synchronize the partners, which is done with algorithm 3.

Afterward, both sides are ready to establish the connection. *Step three* is done only on the primary side, the IPC connection is prepared (e.g. by opening ports). In *step four* some configuration information is exchanged between the partners to ensure compatibility. This includes version checks and other settings. Additionally, IPC specific information such as port numbers are exchanged, so that the secondary connection knows which port to connect to. Files are used for this data exchange, as they are a simple and suitable means that does not require any connection to be established upfront. This communication happens only on one rank, to reduce the load on the filesystem. In a distributed environment the information is broadcasted to the other ranks, which happens in *step five*. *Step six* is the final step, it establishes the IPC connection. Afterward, the initialization is completed and CoSimIO is ready for the exchange of data.

Algorithm 2: handshake/initial connection

```
1  if rank == 0 and IsPrimary then
2      remove communication folder (if existing)
3      create communication folder
4  end
5  Synchronize partners (ensure both are ready, see algorithm 3)
6  if IsPrimary then
7      prepare connection (e.g. open ports)
8  end
9  if rank == 0 then
10     get my info
11     write my info to file
12     wait until file with info of partner becomes available
13     read info file of partner (also contains connection info such as port
         numbers)
14     perform compatibility checks
15 end
16 Broadcast partner info to all ranks
17 Perform IPC connection (e.g. connect to ports)
```

In order to make sure that both partners are ready, they are synchronized using algorithm 3 with files. First, each partner synchronizes its ranks if it is running distributed in MPI. Afterward, each partner writes, reads and deletes a sync file in an order that depends on whether it is the primary or the secondary connection. It is crucial that the secondary only writes its sync file after it read the sync file of the primary. This is to ensure that the communication folder was created by the primary. Writing the sync files happens only on one rank as writing many files in a distributed simulation can severely affect the performance. Finally, again all the ranks are synchronized.

Algorithm 3: synchronization of partners

```
partner-local Barrier (only in MPI)
if $rank == 0$ then
    if $IsPrimary$ then
        create primary sync file
        wait for secondary sync file
        remove secondary sync file
        wait until primary sync file was removed (by secondary)
    else
        wait for primary sync file
        remove primary sync file
        create secondary sync file
        wait until secondary sync file was removed (by primary)
    end
else
end
partner-local Barrier (only in MPI)
```

3.8 Interprocess communication for CoSimulation

The data exchange between tools in CoSimulation is realized with IPC. This section complements the explanations of the concepts from Section 2.2.2. The technical aspects and crucial details of the implementation in CoSimIO are presented, with a particular focus on the differences between operating systems. Performance tests are conducted to compare the methods.

CoSimIO implements various methods for data exchange. They are abstracted with a common interface (see Section 3.7.3), which means they can be selected at runtime. This concept consists of two main functionalities, namely the exchange of floating point (*double*) values and strings (character arrays). Technically everything can be represented by character arrays, the distinction is made for performance reasons. In the case of complex objects such as meshes, serialization as previously explained is used. The data exchange happens synchronously, since the dependencies are strong, meaning that the data exchange usually happens multiple times per time step. Even in a one-way coupling, the benefits of asynchronous communication are small, as running solvers in parallel is also often restricted by the hardware that is used. Thus, asynchronous communication is not considered, also due to the increased complexity.

As presented in Section 3.7.5, in a distributed environment every rank has one partner rank, and the communication happens 1:1. This simplifies the organization of the data exchange.

Several methods for IPC are implemented and compared against each other, using the criteria listed in Section 2.2.2. The focus is on CoSimulation, therefore only a subset of methods was chosen. The main criteria for the selection were performance, robustness and usability. Also, the solutions used by other coupling tools (see Section 2.5 were taken into account. The implementation was done with as little dependencies as possible, to maximize the portability. Libraries like *boost interprocess*[16] provide high-level interfaces for IPC, but can introduce critical dependencies, and were therefore avoided.

3.8.1 File

Using files for the data exchange is perhaps the most basic version of IPC. One process writes data to a file, which is then read by another process. After reading, the file gets deleted. This makes it very suitable for debugging and initial developments, as the files (and thus the flow of data) can be observed in the filesystem.

While the concept of this approach is straightforward, there are several crucial details necessary for a successful transfer. In particular, the synchronization of file accesses to avoid race conditions, where reading happens before the writing is completed. Three different ways can be used to prevent this situation:

- File renaming: The data is written to a file with a different name, and renamed to its final name after the writing is completed. This way the file will only become visible for the reading process once it is ready. No race conditions can occur during writing. However, the critical operation is now the renaming, which needs to happen atomically. The OS/filesystem needs to ensure this. Linux ensures this (see documentation of *rename*[17]), while Windows does not guarantee it. This way of synchronization is used by [29], with the intention of being used on Linux-based HPC infrastructure.
- Auxiliary file: Instead of renaming the file which contains the data, an auxiliary file is used. Its existence signals that the file is ready to be read. This avoids the requirements for the atomic rename, but it requires creating an additional file. This can reduce the performance in distributed environments, due to the creation of many files. [45] used this technique for the communication between a solver running on a local Windows machine and a solver running on a supercomputer.
- File locks: Most OSs provide ways to lock files (e.g. *flock*[18] in Linux). However, those are not portable and thus cannot be used when exchanging data among processes running in different OSs. This is not a common use case for CoSimulation, but still can be a limiting factor and was therefore not considered in CoSimIO.

Remark: A race-condition-free implementation of a file system in C++ is proposed with the low-level file I/O library *std::llfio*[19]. This could potentially be used as a better alternative solving the synchronization problems.

[16]`https://www.boost.org/doc/libs/1_81_0/doc/html/interprocess.html`
[17]`https://man7.org/linux/man-pages/man2/rename.2.html`
[18]`https://linux.die.net/man/2/flock`
[19]`https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1031r2.pdf`

The process that is waiting for the file inquires repeatedly if the file has become available. To avoid too many checks to the file system, a wait time of 0.05 s has proven to be a good compromise between speed and avoiding too many calls, which can slow down the system.

Another important aspect that needs to be considered for file-based data exchange is file permissions. They need to be set so that both sides can read and write them. This is especially relevant when one process uses elevated permissions (e.g. administrator mode). Furthermore, using a dedicated folder in which the files are written helps to not only keep the working directory clean but also simplifies the removal of leftover files from previous runs. In C++, all the necessary functionalities for implementing a file-based data exchange are part of the standard library (*std::filesystem*).

Following are the advantages and disadvantages of file-based data exchange, compared to other IPC methods:

Advantages

- Little implementational effort, and very few requirements
- Fall-back solution, in case the other methods do not work
- Easy to understand and debug as the flow of execution can be observed by how the files are written.
- Sufficiently fast and efficient for many use cases
- Transfer of data across computers and OSs. Requires that the same filesystem is mounted in all systems. This can be crucial if the tools that need to be coupled work in different systems, e.g. one in Windows and one in Linux, or one on a local computer, and one on a cluster. Another application is Windows Subsystem for Linux (WSL)[20]. One solver is executed on the Windows host, and the other one runs on the Linux guest system.
- Independent of what version and technology of tools are used (e.g. compiler)
- The data transfer is natively buffered and asynchronous
- Files can be written in a format that can be easily debugged, e.g. the VTK format (which can be read by Paraview[21]) for the mesh.

Disadvantages

- Slow for large amounts of data
- Filesystems have a limited number of writes, hence exchanging large amounts of data can decrease the lifetime. See the following paragraph for an evaluation of this statement.

[20]`https://ubuntu.com/wsl`
[21]`https://www.paraview.org/`

- Leftover files from previous executions can lead to problems if not removed upfront.
- Special synchronization techniques need to be used to avoid race conditions while reading and writing the files.
- Windows does not provide a way to perform file operations in an atomic way. This means that if several programs are trying to operate on the same file, race conditions are introduced that can cause problems.

Lifetime of hard disk
Frequent writing to a hard disk can decrease its lifetime. The following example evaluates the relevance of this problem in the context of coupled simulations. The FSI simulations with the Munich Olympic Tower are chosen as they are a large real-world simulation. For details on this example, it is referred to Section 6.5.

The memory consumption for the files per TS is computed based on the number of nodes on the interface, multiplied by the number of Degrees of Freedom (DOFs) per node (3, displacements or loads) and the memory required per DOF (datatype *double*, 64 bit / 8 Byte):

$$\begin{aligned} mem_n &= 3\frac{dofs}{n} * 8\frac{B}{dof} = 24\frac{B}{n} \\ mem_s &= 128,817n * mem_n = 2.95MB \\ mem_f &= 244,163n * mem_n = 5.59MB \\ mem_{ts} &= mem_s + mem_f = 8.54MB \end{aligned}$$

In the example, 30,000 TSs are computed (which in combination with a time-step size ΔT of 0.02 s results in a simulation time of 600 s/10 min real-time). The data exchange happens once per TS due to the explicit coupling algorithm. Hence, the memory per simulation computes as:

$$mem_{sim} = mem_{ts} * 30,000/sim = 250GB/sim$$

An average modern SSD hard disk has 500 TB of bytes that can be written[22]. This means, that 2000 simulations can be run before the disk is broken.

$$n_{sims} = 500TB/mem_{sim} = 2000$$

Of course, this is only a rough estimate that highly depends on the size of the model, the number of time steps, the amount of data exchanges per time step (which can be larger than 1 when an implicit coupling is used), and compression/file-representation on the drive. Furthermore, it depends on the specification of the disk to how much data can be written to them.

These calculations show that many simulations can be run without risking the failure of the hard disk. However, it is important to keep in mind that

[22]Example: Samsung 980 PRO or 870 EVO, see `www.samsung.com`

also other data such as field data is usually written during a simulation, which further decreases the lifetime. Therefore, it is recommended to limit the usage of file-based data exchange to the development, debugging and initial setup of coupling codes. Afterward, other methods should be used. In CoSimIO this can be easily achieved thanks to the common interfaces of IPC methods.

3.8.2 Pipe

Pipes are low-level data channels for exchanging information. One side writes, and the other side reads the data. In Linux, they are commonly used to redirect (also called *piping*) the output of one command to another. One use case is capturing the output of a command, and simultaneously writing it to a file: *./some_ command* | *tee command.log*. The | character specifies the pipe.

Two types are distinguished, anonymous and named pipes. The first one is used to communicate between parent and child processes, whereas the latter is used to exchange data between unrelated processes. In CoSimulation usually separate processes are used by the solvers, meaning that only named pipes can be used. Pipes offer a high performance since the transfer is done on the kernel level, with neither the filesystem nor the network involved. However, they are very specific to the platform. The interface in Unix and Windows is entirely different, as well as the underlying implementation, making them incompatible with each other. Furthermore, they only work locally on one machine. With some exceptions, pipes offer one-directional synchronous data transfer. Thus, a two-way coupling requires two pipes to function. Each pipe has a specific buffer size, which can be up to several MB in modern systems. If more data is to be exchanged, then it needs to be split into chunks and exchanged one after the other. The synchronous transfer means that the exchange of data can only be done if both the reading and the writing side are ready. Otherwise, the operation is blocked. This means the size of the data must be known upfront, which is realized by exchanging the size of the data upfront (with a known size to avoid blocking). With larger amounts of data, some robustness issues could be observed. These were more pronounced when the entire pipe buffer was used.

Following are the advantages and disadvantages of pipe-based data exchange, compared to other IPC methods:

Advantages

- Basic and mature method with high performance
- Little effort required for synchronization
- Fundamental part of the operating system, thus no external libraries needed

Disadvantages

- Interface and underlying implementation/concept differ between Unix and Windows

- Unix and Windows are incompatible
- Does not work in distributed environments or across machines
- Not the most robust solution, in particular for large amounts of data, prone to deadlocking

3.8.3 Socket (Network/TCP)

Programs can use the network to communicate with each other. They can be located on the same machine, or on different machines, making them suitable for distributed memory architectures. A peer-to-peer connection is established for the data exchange. During the connection phase, one side opens a port (on every rank in MPI), and then the other side connects to it, again rank-by-rank. The Transmission Control Protocol (TCP) is used to ensure that no data is lost during the transmission. Sockets are a commonly used way to exchange data over the internet. The previously mentioned *ASIO* library is used as a high-level interface, which unifies the implementation for Linux and Windows.

The only input required by the user is the IP address or the name of the network to be used. This is only required in distributed environments, where the network that connects the different compute nodes (e.g. InfiniBand) needs to be selected.

Following are the advantages and disadvantages of network-socket-based data exchange, compared to other IPC methods:

Advantages

- Fast
- Robust & mature technology
- Works on small machines as well as clusters with distributed environments

Disadvantages

- External library (ASIO) is needed (until it is integrated into the C++ standard library)

3.8.4 Socket (Unix/local)

Unix domain/local sockets are similar to network sockets, but differ in the following two aspects: a) they can only be used for communication on the same machine, and b) they use the kernel memory for the communication instead of the network, giving them a better performance. While they were originally only available in Unix OSs, Windows added initial support for them end of 2017[23]. The *ASIO* library supports both types of sockets, although

[23]https://devblogs.microsoft.com/commandline/af_unix-comes-to-windows/

Windows support for Unix domain sockets was added only recently and was not yet robust to use at the time of the developments of this work.

Following are the advantages and disadvantages of Unix-socket-based data exchange, compared to other IPC methods:

Advantages

- Fast
- Robust & mature technology (only on Unix)

Disadvantages

- Does not work in distributed environments or across machines.
- Does not yet work stably in Windows with ASIO (still under development at the time of this writing).

3.8.5 MPI

As the name suggests, the **M**essage **P**assing **I**nterface sends messages between different processes (typically of one program). It is extensively used in HPC for communication in distributed algorithms, see also Section 2.4.2. The high performance and availability of clusters make it a suitable candidate for IPC in CoSimulation. It works similarly to socket-based communication, after a connection is established, data can be exchanged between the codes.

This type of communication has several drawbacks, which limit its applicability compared to socket and file-based data exchange. A main restriction is a dependency on MPI, as the codes that use CoSimIO need to be compiled and launched with it. This can be problematic, especially for applications that are usually not deployed on HPC infrastructures. The same applies to Windows, which technically provides an MPI implementation (*MS-MPI*[24]), but it is not often used in practice. Furthermore, some implementations of MPI (*OpenMPI*[25] in particular) are known to have robustness issues, as reported in [53].

Finally, also the way the execution is launched plays a role in how the communication is established. If the two codes are launched independently with *mpiexec*, which is usually the case, then they each have their own global and application-specific communicator *MPI_COMM_WORLD*. The communication between the codes can only be done after a connection via *MPI_Ports* is performed. Same as for sockets, one side opens a port, and the other side connects to it. This is referred to as intercommunication since it connects to independent MPI communicators. The other alternative would be intracommunication, which would require launching both codes together with one *mpiexec* call. They would then share one MPI_COMM_WORLD. This option is not considered, as it is intrusive and alters how the codes use MPI.

[24] `https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi`
[25] `https://www.open-mpi.org/`

Following are the advantages and disadvantages of MPI-based data exchange, compared to other IPC methods:

Advantages

- Fast
- Already used by most HPC codes

Disadvantages

- Some implementations are not robust (in particular OpenMPI)
- Adds a layer of complexity for non-MPI codes & executions
- Windows support of MPI
- Features required for CoSimulation are not available in old standards
- Requires both programs to use the same MPI implementation

3.8.6 Performance evaluation

Numerical simulations are computationally expensive. Therefore, it is crucial to assess and evaluate the performance of every component in the simulation chain. This includes also the IPC methods, as they are heavily used for communication among the tools in CoSimulation. For this purpose, a set of relevant test scenarios is designed. They are conducted for the different methods used in this work.

The first case represents the exchange of interface data, which happens during each coupling iteration. The data is considered as a memory-contiguous vector of 64-bit floating point numbers (*double*), which can be transferred without additional treatment such as serialization. It is sent back and forth multiple times between two processes in a ping-pong fashion. The size of this vector varies and is inspired by the examples (see Chapter 6) used in this work to provide realistic and practical values. The result of these tests is the throughput in *Gbit/s*, indicating how much data can be exchanged in a given time interval.

The second case is identical to the first one, except that the data is exchanged in its serialized form. This adds two additional steps, namely the serialization before the transfer and the deserialization afterward. As previously explained, complex objects such as meshes require these additional steps due to their layout in memory. In CoSimulation, this is typically used for communicating the meshes at the beginning of the coupling, e.g. for establishing the mapping relations. Therefore, in many cases this happens much less than the exchange of interface data. However, still it is a very relevant scenario, e.g. more complex simulations that include adaptive/dynamic meshes might require a continuous update of the meshes.

The two cases are tested on different soft- and hardware, which are commonly used to conduct numerical simulations. This includes the three main operating systems Linux, Windows and macOS, as well as a regular personal

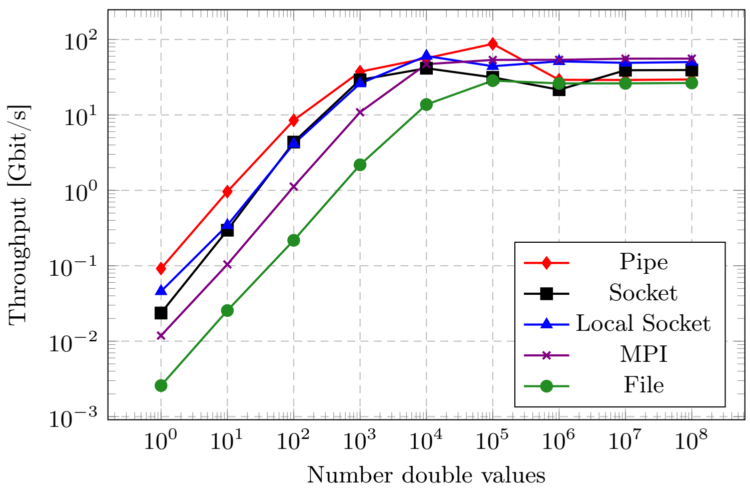


Figure 3.9: [Linux] Data exchange (without serialization)

computer and a supercomputer. The specifications of the hard- and software can be found in Appendix A.

Before going into detail for the individual cases, the following general observations are made, independent of technology, soft- and hardware:

- The overhead for transferring small amounts of data is clearly visible. A minimum size of data is required to achieve the highest throughput, it is in the range of 1×10^3 to 1×10^5 values.
- Serialization reduces the performance, by one order of magnitude. This holds for binary serialization. Using ASCII reduces it by another order.
- The speed of the underlying technology used is reflected in the throughput in most cases. This is expected and indicates that the proposed workflow enables the full capabilities of the hardware.

Figure 3.9 shows the throughput for different amounts of data, on a Linux system. As per the general observation, a minimum required size of data is necessary to reach the highest performance. The slower methods like the file-based data transfer need more data to reach the maximum, compared to the faster methods. The difference is roughly a factor of 100 (1×10^3 to 1×10^5). This can be explained by the overall larger overhead for the slow methods.

Furthermore, the graph can be divided into two parts, depending on the number of values to be exchanged. Below $\sim 1 \times 10^4$ values, the technologies that are closest to the kernel like pipes or local sockets are the fastest. It can be clearly seen that they are optimized for the fast transfer of little to intermediate-sized data. They do not involve additional hardware like the

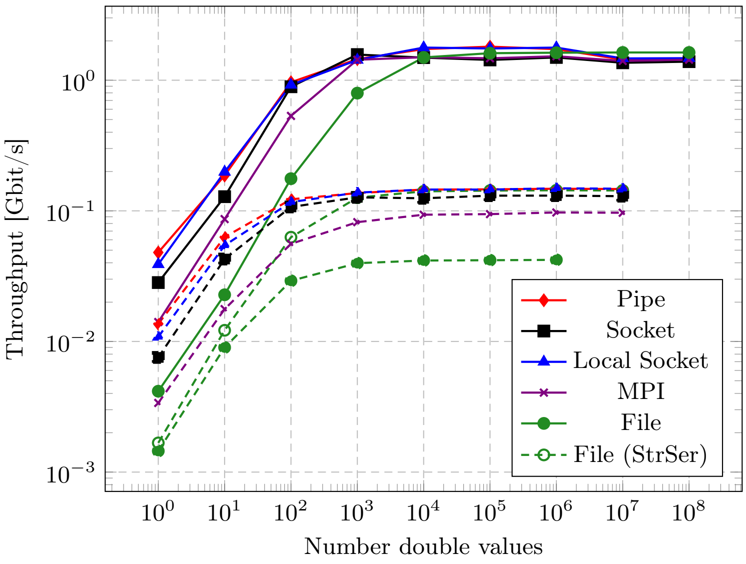


Figure 3.10: [Linux] Data exchange (with serialization). Solid lines represent binary serialization, dashed lines represent ASCII serialization. *StrSer* means first serialization to a stream and not directly to the file.

network or even the filesystem, resulting in very low latencies. However, due to their limited cache and buffer sizes, the situation changes above the threshold of $\sim 1 \times 10^4$ values. For large amounts of data, MPI yields the highest throughput. This is consistent with the general recommendation for distributed applications to communicate a few times with grouped data, as opposed to communicating many times with little data. The local sockets always outperform the TCP sockets, but only by a small margin. Lastly, it is important to mention that file-based data transfer is only 5-15 times slower than the fastest method. This can be largely attributed to the modern Solid State Drive (SSD) hard disk, which is characterized by its high performance due to the lack of mechanical moving parts. Furthermore, the filesystem acts as a huge buffer, which is an advantage for the exchange of large amounts of data. Only for small amounts of data, the large overhead is a drawback. A comparison between the different methods for synchronization with file-based data exchange (file renaming and auxiliary file) as explained in Section 3.8.1 did not yield relevant differences.

The performance results for serialized data exchange on a Linux system (see Appendix A) are shown in Figure 3.10. The overall behavior of the measured throughput is similar to the direct version. Smaller data sizes that are exchanged yield a lower throughput due to overhead. However, the

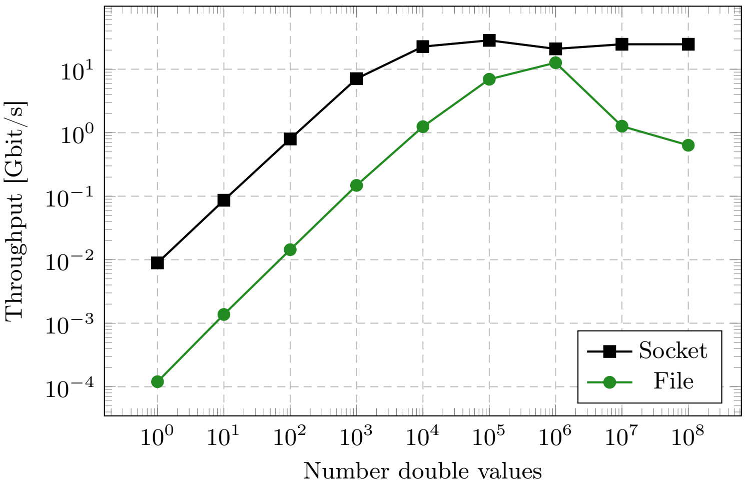


Figure 3.11: [Windows] Data exchange (without serialization)

maximum achieved performance is considerably lower. Around one order of magnitude with binary serialization, and two orders with ASCII. This shows that the serialization procedure is the bottleneck in this variant of the data exchange. Therefore, also the differences between the IPC techniques are smaller, in particular with the binary format. With ASCII, a notable exception needs to be considered for file-based data exchange. Serializing the data first to a stream and then writing it to a file is roughly one order of magnitude faster than writing directly to the file. For binary data, there is no notable difference. This is because of the representation of the data in memory: Binary data is faster to write to a file as it only consists of ones and zeros. ASCII data is stored as individual characters, which have to be encoded before writing them. Doing this at once (when using a stream) is faster than doing it one by one (when writing directly to the file).

The results for measuring the performance of file-and socket-based data exchange on a Windows system (see Appendix A) are presented in Figure 3.11 and Figure 3.12. As before, direct and serialized data transfer is measured. The results are similar to the Linux system, which used the same hardware. Overall the performance is a bit lower, but the same general behavior can be observed. The decreased performance can be attributed to the underlying implementations of the Windows OS.

Conducting the performance tests on a macOS system (different hardware, see Appendix A) yields again similar results, see Figure 3.13 and Figure 3.14. Direct data transfer without serialization is consistently and considerably faster than using serialization. With local sockets, the maximum performance is reached with $\sim 1 \times 10^{3}$ values, afterward the throughput is again lower. This can on the one hand stem from the not mature implementation in ASIO,

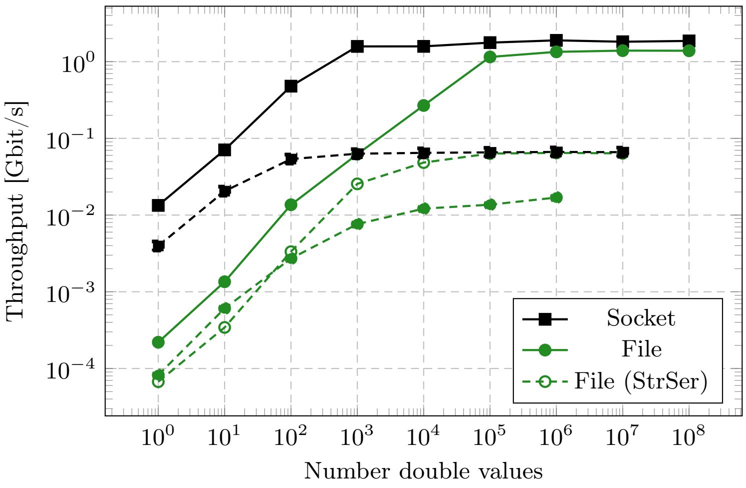


Figure 3.12: [Windows] Data exchange (with serialization). Solid lines represent binary serialization, dashed lines represent ASCII serialization. *StrSer* means first serialization to a stream and not directly to the file.

and on the other hand from limited buffer sizes.

Finally, the tests were also executed on a supercomputer (see Appendix A), the results can be seen in Figure 3.15 and Figure 3.16, respectively for direct and serialized transfer of data. 48 cores/MPI processes (1 compute node) were used on each side, making communication across distributed memory. Not all IPC technologies support this, among others files, TCP-sockets and MPI. Pipes and local/Unix sockets only work in shared memory.

Even though some similarities with the results obtained on the other systems can be observed, also some significant differences are present. Using MPI is considerably faster than sockets and files, in particular for small sizes of data. It is also notable that it performs very homogenous across all sizes of data, the previously observed reduced performance with little data can hardly be seen. These results are plausible since MPI was designed for highly performant communication in distributed memory architectures. The performance of sockets is higher than with files until a threshold of 1×10^6 values, after which the file-based data exchange gives higher performance. This can be related to overall network traffic on the system, and only holds for data exchange on one compute node.

Large-scale simulations can easily exceed 48 cores/1 compute node, depending on the size of the model and the numerical complexity. Therefore, the tests were conducted with up to 960 cores/20 compute nodes on either side. The performance with MPI and sockets hardly changes with a larger number of cores. This is expected since the communication happens peer-to-peer.

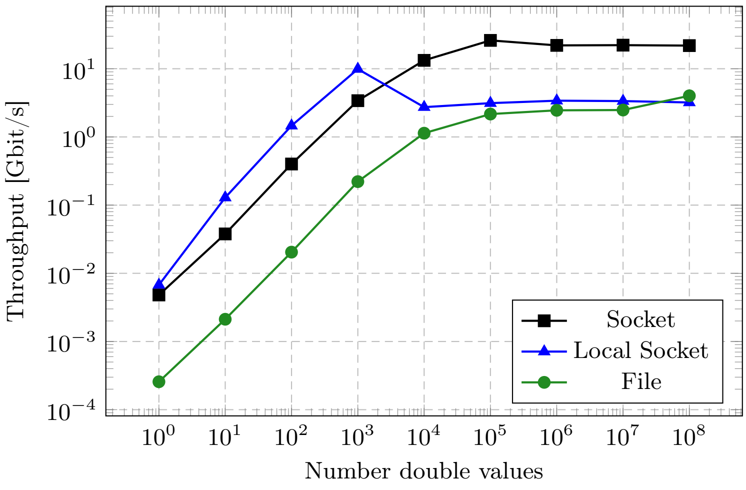


Figure 3.13: [macOS] Data exchange (without serialization)

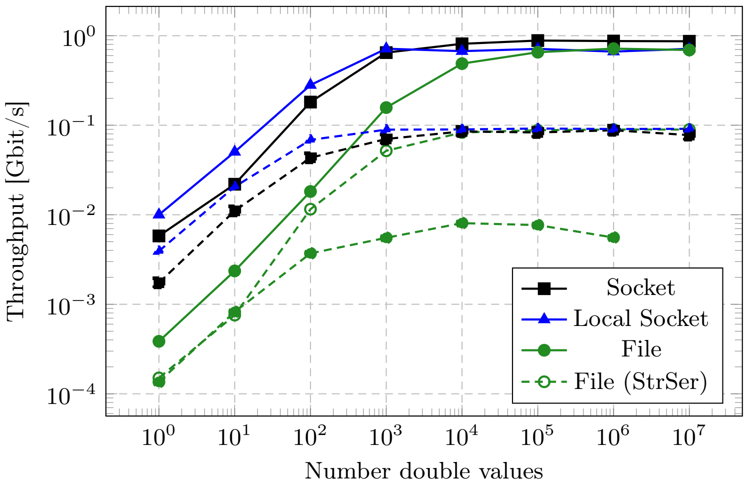


Figure 3.14: [macOS] Data exchange (with serialization). Solid lines represent binary serialization, dashed lines represent ASCII serialization. *StrSer* means first serialization to a stream and not directly to the file.

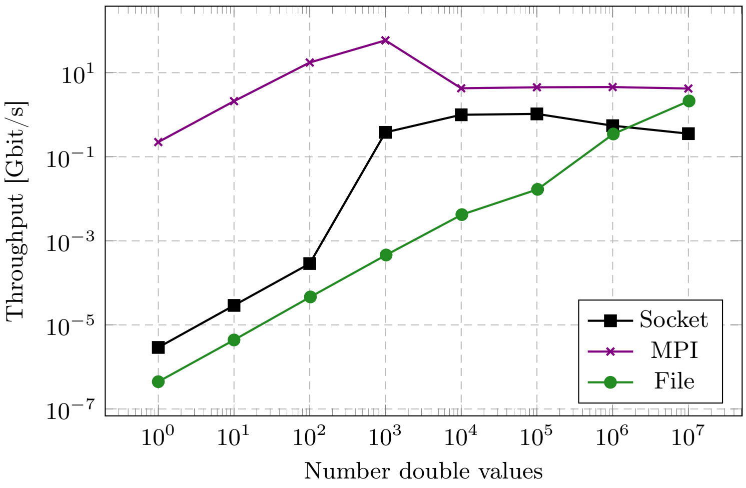


Figure 3.15: [SuperMUC] Data exchange (without serialization)

Furthermore, at no point do any collective operations occur which could potentially result in bottlenecks.

With files, however, the performance deteriorates drastically, already with 96 cores/ 2 compute nodes. The results obtained over several executions were very inconsistent, therefore no meaningful plots could be generated. The amount of files that are exchanged was too much for the filesystem to handle, resulting in very or even extremely low throughput. This was also affected by other simulations running on the same hardware. While this may appear as a disadvantage for conducting benchmarks, it represents the real situation on clusters and supercomputers. Due to their size, almost no single job can make efficient use of the entire system. Usually, many jobs are running at the same time, potentially using (different parts of) the same filesystem. Using renaming of files for the synchronization is hereby still a bit better, since using an auxiliary file doubles the amount of files, resulting in an even lower performance.

These studies lead to the conclusion that file-based data exchange is not suitable for HPC simulations. Especially when one considers that the supercomputer used in this work is using state-of-the-art hardware, including a highly performant file system. [53, chapter 5.3.2] proposes an algorithm, in which the files are written in a special directory structure, to limit the number of files in one directory. This could potentially improve the throughput. Alternatively, a scratch directory for temporary files that is often available on large computing systems could be used for writing the data exchange files. However, with both alternatives, the number of files exchanged remains the same and therefore still a lower performance compared to MPI or sockets is expected.

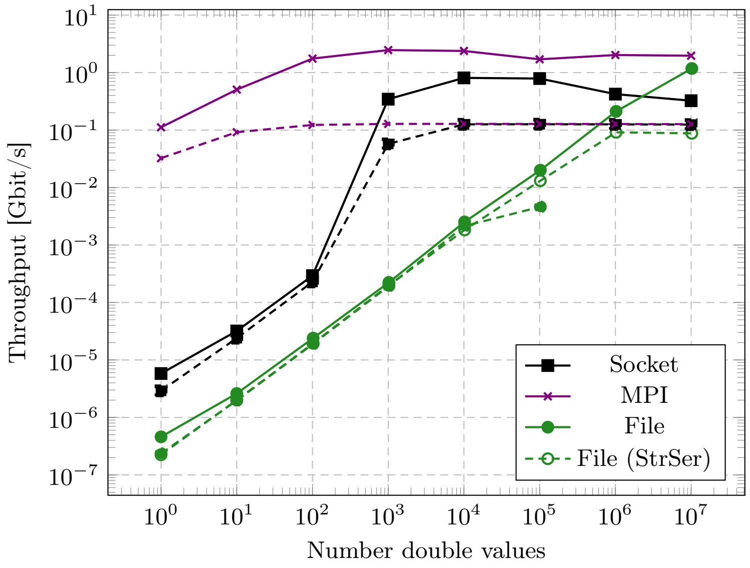


Figure 3.16: [SuperMUC] Data exchange (with serialization). Solid lines represent binary serialization, dashed lines represent ASCII serialization. *StrSer* means first serialization to a stream and not directly to the file.

Independent of hard- and software it was found that the performance can deteriorate significantly if the system comes under load. In particular, if the components that are involved in the IPC are used by other tasks. For example, accessing the filesystem with a different process while it is used for the data exchange in CoSimulation affects the performance. This was seen clearly for the supercomputer, thus the performance tests were conducted (as much as possible) with an idle system.

Furthermore, the results showed the large influence that the serialization technique has on the performance, it could be identified as the bottleneck. Further reducing the size of the data to be exchanged by using compression algorithms would most likely further decrease the performance. It is instead recommended to optimize the serialization implementation as much as possible. Only when its performance reaches levels of direct data transfer, then compressing the data could be advantageous. In general, binary serialization performed significantly better than ASCII serialization and should therefore be used.

	File	Pipe	S-TCP	S-Unix	MPI
Performance	**0**	+	+	+	+
Robustness	+	-	+	**0**	**0**
Portability	+	-	+	-	-
Usability (local)	**0**	**0**	+	+	-
Usability (cluster)	-	**x**	+	**x**	+
Implementation	+	-	**0**	**0**	**0**
Dependencies	+	+	**0**	**0**	-

Table 3.1: Summary of comparison of different methods for IPC in CoSimulation. **TCP-Sockets are the best compromise** between performance & robustness, usability & portability and implementation & development effort.
The symbols represent the following:
Good: +
Neutral: **0**
Bad: -
Unavailable: **x**

3.8.7 Summary IPC for CoSimulation

The data exchange between processes is one of the most important components required for CoSimulation with different tools. A large effort was therefore spent in this work to compare different methods. The metrics relevant for CoSimulation were used, namely performance & robustness, usability & portability and implementation & development effort (see Section 2.2.2). Table 3.1 summarizes the results. It was found that TCP-sockets are the best method for IPC in CoSimulation, as they provide the best compromise of the previously mentioned metrics.

An example of running large-scale FSI simulations with the Olympic Tower in Munich is presented in Section 6.5. Some of the studies conducted with this case compare and test the different IPC methods. The findings of the previous sections are complemented and confirmed by the results obtained with the Olympic Tower simulations.

Chapter 4

Mapping

Mapping is the process of transferring data from one mesh to another. The discretizations of the meshes are not matching in many practical cases, therefore interpolation or other techniques are required. Practical applications are coupled simulations, where the interface data of one solver needs to be transferred to another solver, or remeshing, where the data on the old mesh needs to be mapped onto the new mesh. Figure 4.1 visualizes the mapping between non-matching meshes.

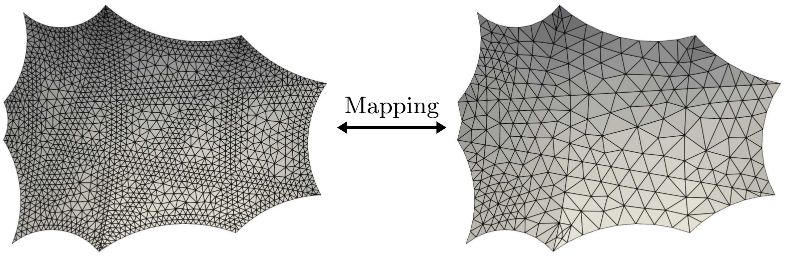


Figure 4.1: Mapping between non-matching meshes.

As one of the main three building blocks (see Figure 2.1), mapping (as a subset of data transfer methods) is a crucial component of CoSimulation. Usually, numerical solvers use the discretization that suits them best for solving a particular problem. It is therefore imperative not to impose any restrictions in terms of discretization if they are used in a coupled simulation. The mapping techniques need to adapt and handle the different mesh setups.

This chapter revisits the most commonly used mapping techniques and carefully evaluates their suitability for large-scale mapping in distributed memory environments. Additionally, efficient strategies and solutions for mapping in those situations are presented, and their performance is tested with several configurations on a supercomputer. The implementations are realized within the *MappingApplication* and the Core of Kratos and are an integral part of the CoSimulation capabilities of Kratos.

The terminology used in this work was introduced in Section 2.2.3. Data is mapped from the *origin* to the *destination*. Preliminary developments and results were already presented in [8].

4.1 Mapping algorithms

A large variety of mapping methods have been developed, differing in many aspects such as accuracy, speed and robustness. [18] gives an overview of the state-of-the-art methods. In the following, the most commonly used techniques are introduced, highlighting their strengths and weaknesses. Additionally, their suitability for large distributed systems is evaluated, as this is the scope of this work.

It is important to recall the equation for mapping from 2.2. The transfer matrix **H** represents the relation between origin and destination. Its construction (implicitly or explicitly) is the aim of each mapping method.

4.1.1 Nearest neighbor

This mapper is probably the simplest mapping technique: Each point in the destination searches its geometrically closest point in the origin, and takes its value. Due to these properties, it is also referred to as closest point in some publications. The workflow is described in 4.

Algorithm 4: Workflow nearest neighbor mapper

```
foreach point in destination do
    initialize $d_{min}$ to large value
    foreach point in origin do
        compute distance $d$ between points
        if $d < d_{min}$ then
            $d_{min} = d$
        end
    end
    closest point is the one associated with $d_{min}$
end
```

The advantages of this mapper are the simple implementation and the robustness. It only requires point clouds as input and not full meshes. Furthermore, the assembled transfer matrix **H** consists of only ones and zeros, making it the mapper with the smallest memory footprint. This mapper is very suitable for distributed environments since it requires only the searching of neighbors in different partitions.

The low accuracy, in particular for very heterogeneous meshes, is the main disadvantage of this mapper. It can cause a step-like shape on the destination when the fineness of the discretizations is too different.

Mapping on matching meshes can be considered a subcase of the nearest neighbor mapper. This allows for several optimizations, such as specific search settings.

4.1.2 Nearest element

This mapper makes use of the geometry of the origin interface, by interpolating the values using its shape functions, see [81]. For this, each point in the destination searches the geometrically closest elements in the origin, and projects to them. It then takes the element with a valid projection that has the smallest projected distance (it is therefore also referred to as closest projection mapper). The value assigned to the destination is interpolated using the shape functions of the element. Therefore, the transfer matrix **H** contains the values of the shape functions evaluated at the projection. The workflow is described in 5.

Algorithm 5: Workflow nearest element mapper

```
1  foreach point in destination do
2      initialize $d_{proj\_min}$ to large value
3      foreach element in origin do
4          compute projection and projected distance $d_{proj}$ between point
             and element
5          if projection is valid and $d_{proj} < d_{proj\_min}$ then
6              $d_{proj\_min} = d_{proj}$
7          end
8      end
9      closest point is the one associated with $d_{proj\_min}$
10     evaluate shape functions at projection
11 end
```

The advantages of this mapper are the rather simple implementation and interpolative properties. The disadvantage is that it requires elements and their shape functions as input. It involves projections that can fail in practical applications and hence require further treatment to increase the robustness. For the usage in distributed systems, the projections need to happen across partitions, similar to the nearest neighbor mapper. The memory footprint is larger compared with the nearest neighbor mapper, due to more entries from the shape function values used for the interpolation.

4.1.3 Barycentric

This mapper is similar to the nearest element mapper, but it does not require elements/shape functions as input, see [46]. Instead, it reconstructs elements from the closest points and projects to them. For 1D mapping it constructs a line i.e. needs to find the two closest points, for 2D (surface) mapping it needs the 3 closest points to construct a triangle and for 3D (volume) mapping it needs the 4 closest points to construct a tetrahedron. Note that it is possible to reconstruct other topologies as well (e.g. quadrilaterals in 2D) but this requires more computational effort and makes the projections more complicated. Due to the used projections, the transfer matrix **H** contains the values of the shape functions, evaluated at the projection, same as for the nearest element mapper. The workflow is described in 6.

Algorithm 6: Workflow barycentric mapper

```
1 foreach point in destination do
2     find closest points (making sure they are not collinear)
3     foreach point in origin do
4         check if point is one of the closest
5     end
6     reconstruct geometry to project to
7     project on reconstructed geometry
8     evaluate shape functions at projection
9 end
```

This mapper has the same advantages/disadvantages as the nearest element mapper, mainly its good interpolative properties and the projections that can cause problems for practical applications. One advantage is that it works with only point clouds, but is slightly more complicated to implement due to the reconstruction of the geometry to project on. Furthermore, the reconstructed elements require some attention to avoid distorted shapes. The points for reconstructing the geometry need to be collected across partitions when this mapper is used in distributed memory, the projection then happens locally.

4.1.4 Other mapping techniques

Until now, only a subset of commonly used mapping techniques was introduced. Mapping based on Radial Basis Functions (RBFs) and mortar/weighted-residuals method are also used, but their applicability for distributed systems is limited, due to several complexities. Interpolation with RBFs as presented in [2], leads to a rather densely populated transfer matrix, which can result in memory issues, in particular for volumetric mapping. Additionally, a linear system needs to be solved, which is challenging in distributed environments, as shown in [53]. Mortar-type methods (see [25] and [24]) use an elaborate projection and integration scheme, but they can suffer from robustness issues in particular for complex geometries. Furthermore, the numerical integration used to compute the transfer matrix requires a considerable amount of data to be exchanged in distributed systems, and can also lead to more entries in the transfer matrix. A linear system needs to be solved during each mapping step, thus introducing further difficulties.

Given the limitations of these techniques, they will not be further considered in this work.

4.2 Conservative Mapping

Two different ways of mapping quantities exist, consistent/direct and conservative mapping. Consistent mapping is used with distributed fields such as displacements or tractions, and conservative mapping for concentrated/integrated quantities such as nodal loads. As the name implies, conservative mapping preserves the force/energy on the interface. The nodal sum of quantities on either side of the interface is the same. It is typically used to map nodal loads in FSI simulations from the fluid to the structural domain, see also equation 2.11.

The relation between consistent and conservative mapping is shown in equation 4.1:

$$\mathbf{x}_o = \mathbf{H}_{od}^T \mathbf{x}_d \tag{4.1}$$

Consistent mapping from origin to destination is done with $\mathbf{H}_{od}$, conservative mapping in the other direction with $\mathbf{H}_{od}^T$. This relation has the significant advantage, that only one mapping matrix needs to be computed, which can be used for mapping in both directions.

Comparisons between consistent and conservative mapping were conducted by [17] and [81]. In some setups, conservative mapping can lead to unphysical oscillations, depending on mesh sizes on both sides of the interface, the employed mapping technique and the distribution of the mapped quantities. More studies regarding this can be found in [81].

4.3 Mapping in distributed environments

As explained in previous chapters, the support of distributed memory machines such as clusters or supercomputers is crucial for CoSimulation, as it enables the simulation of large-scale engineering problems. The same applies to mapping since it is at the interface of the solvers. Without parallel algorithms, the data to be mapped would need to be gathered on one rank before mapping, and scattered back afterward. This introduces a significant bottleneck for the performance, which becomes more severe with larger problems and more cores used.

To avoid any bottlenecks, the algorithms developed in this work use a peer-to-peer approach, in which the computing ranks can directly communicate with each other if required. This avoids any gathering and scattering of data, both during the initialization and mapping of quantities. Furthermore, the presented algorithms work with any distribution and number of the computing domains on either side of the interface, no restrictions are placed on the solvers. All the scenarios of possible scenarios shown in Figure 3.2 are supported. As explained in Section 3.3.1, each solver has its own *MPI_Comm* MPI-Communicator. Mapping uses an MPI-Communicator, which includes the ranks of both solvers and thereby enables peer-to-peer communication.

4.4 Searching on interface

Mapping geometrically relates two interfaces to each other, which is expressed by the transfer matrix **H**. To establish the relations, first, a geometrical search on the interface is required. The results of this search are then used by the mapping technique to construct **H**. An efficient and robust search, in particular in a distributed domain, is crucial for stable and accurate mapping.

The information to be searched for depends on the mapping algorithm. The nearest-neighbor technique requires the geometrically closest points, the nearest element mapper the element with the smallest projection distance, and the barycentric mapper multiple closest nodes for the reconstruction of the element to be projected onto. RBF-based mapping methods need several points in the vicinity, depending on the chosen support radius. Mortar methods need the element information of each overlapping element, for the numerical integration. Generally, the amount of required information increases with the complexity of the method.

In distributed simulations, the searched data can be in different ranks, which must be taken into account for partitioning-independent mapping. Here, two-level search algorithms can be used, combining a global search across the ranks with a partition-local search. Furthermore, the least amount of search information should be exchanged, to avoid communication overhead. Thus, a preselection of possible search results needs to happen before the communication.

4.5 Implementational aspects

This section presents the implementation of the newly developed mapping in Kratos. The nearest-neighbor, nearest-element and barycentric mapping methods are realized since they cover a large range of applications, and have a very good performance in large distributed simulations. Additionally, they can be used for 1D (line-line), 2D (surface-surface) and 3D (volume-volume) domains.

The highlights of the developments are the unified workflows for shared and distributed memory architectures, and the efficient and robust mapping strategies and procedures with minimal memory footprint and very good scaling behavior. Furthermore, an intuitive and computationally efficient interface is proposed for a mapper, consisting of three functions: *Map* is used for mapping from origin to destination. In the other direction, *InverseMap* is used. Combining the ability to map in both directions within one mapper allows for several optimizations, such as the reuse of the transfer matrix. *UpdateInterface* completes the functionalities, it is used to recompute the transfer matrix after an update of the interfaces, e.g. after remeshing.

The workflow for constructing the (sparse) transfer matrix **H** is done in two steps. First, the search on the interface is conducted, which involves communication among the ranks in distributed simulations. Second, the matrix is constructed, similarly to a FEM assembly. Mapping of quantities can then be executed by a (sparse) matrix-vector multiplication (SpMV), see also [38]. This method has the significant advantage, that the underlying linear algebra library can be employed for computationally expensive parts to achieve maximum performance.

4.5.1 Searching on interface

The search is divided into two parts in distributed simulations, a global and a local search. In serial, only the local search is required. This division allows for a unified treatment of the local search.

The local search is performed with a bin search-structure (see [16]). After the search, the best results are filtered, and assigned to their respective partners. The workflow for the local search is shown in algorithm 7.

Algorithm 7: Local search for mapping

```
1 Create local search structure (bins)
2 Perform search
3 foreach search result do
4     Filter potential partners from search results
5 end
6 Assign filtered search results
```

The search in distributed environments is built on top of the local search. First, a global search among the partitions is performed to determine where possible partners exist. This is done with axis-aligned bounding boxes (see [5]). The search-related information that is required for the local search is then sent to the candidate partitions. Following is the local search, as shown above. Afterward, the search results are sent back to their respective origin partitions for constructing the transfer matrix.

An appropriate search radius is vital for a performant search, as it can influence the number of search results to process significantly, leading to long construction times for the mapper. Still, it is necessary to find all correct pairings, therefore the radius needs to be chosen proportional to the mesh size. This work proposes the following procedure to compute the search radius. The idea is to start with a small search radius and increase it over several iterations until all search results are found. This works particularly well for heterogeneous meshes, which happen very often in practical cases. For this, the following four parameters with suitable defaults are introduced:

- *search radius* r: The initial search radius. It is typically chosen to be rather small, such that in highly refined areas of the mesh not too many results are found. By default, it is chosen to be the largest dimension of the local bins search-structure.
- *maximum search radius* r_{max}: The maximum search radius is by default chosen as the largest edge length in the mesh.
- *search radius increase factor* r_{incr}: The factor by which the search radius increases in each search iteration. Smaller values lead to more search iterations, but fewer search results in each iteration and thus fewer search results to process. The default is 2.0, meaning that the search radius is doubled in every search iteration.

- *maximum number of search iterations n*: Computed based on the previous three settings according to equation 4.2.

$$n = \lceil (ln_{r_{incr}}(r_{max}) - ln_{r_{incr}}(r)) \rceil \tag{4.2}$$

4.5.2 Construction of transfer matrix

A new development of this work is to perform the construction of the transfer matrix **H** in the same way as the assembly of the stiffness matrix in the FEM. This way, many existing procedures and developments can be reused. The equivalent to an element in the FEM is the *local system*, which provides the local contribution for the transfer matrix. Table 4.1 compares regular FEM with the approach proposed in this work.

FEM	**Mapping**
Local element stiffness matrix	Local system
Global stiffness matrix	Transfer matrix

Table 4.1: Comparing FEM assembly and assembly of transfer matrix

In order to construct the local system, it needs the partner's info, which is why the search has to be executed beforehand. Its contribution is then assembled into the transfer matrix. The assembly is done in a generic and parallelism-agnostic way. This means, that the local system is unaware of the employed method of parallelization, which simplifies the development of new mapping techniques significantly. Assembly routines for shared and distributed memory parallelization are developed, using the same underlying linear algebra as for the Kratos solvers, namely ublas[1] for shared memory and trilinos [76] for distributed memory.

4.6 Performance tests

This section evaluates the performance of the developed algorithms and procedures in distributed environments through strong scaling tests. Surface-to-surface and volume-to-volume mapping are considered.

The Olympic Roof model (see Section 6.7) is used for the tests, as it represents a large real-world application. The surface-to-surface mapping is done between the FSI interfaces. For the volume-to-volume mapping, an even finer mesh was created in addition to the one presented in Section 6.7. The sizes of the meshes are listed in Table 4.2.

The results for the time required for mapper construction (figures 4.2 and 4.3) and mapping of vector quantities with 3 components (figures 4.4 and 4.5) are presented in the following. The solid lines represent the mapping

[1] `https://boostorg.github.io/ublas/`

Mesh	Surface		Volume	
	Nodes	Elements	Nodes	Elements
coarse	13,930	26,148	3,442,468	19,632,138
fine	259,541	519,110	7,433,611	46,265,844

Table 4.2: Mesh configurations for the strong scaling tests of the mapper performance in distributed environments.

from coarse to fine mesh, and the dashed lines are in the other direction. The number of MPI cores ranges from 48 to 3,072 for surface-to-surface and 48 to 12,288/24,576 for volume-to-volume mapping.

Common for both types of mapping it can be observed that the time required for initialization of the mapper increases with more ranks. This is attributed to the searching that is required on the interfaces. On one hand, having more ranks makes the search on each core faster, as fewer entities per core are participating in the search. On the other hand, more communication is required between the ranks increases with a larger number of ranks. This leads to the observed results. Communication taking precedence can be nicely observed when the time used by the nearest neighbor mapper reaches almost the time of the nearest element mapper, despite its less expensive algorithm. Aside from the search, the second computationally expensive operation is the assembly of the transfer matrix. It is almost negligible for large meshes compared to the search. This can be attributed to the employed linear algebra libraries, which have reached a very mature and optimized state. These results demonstrate the importance of an efficient and performant search. The absolute time for the initialization is still small, particularly when compared to the solution times of the solvers. In most simulations, this step only needs to be executed once.

Unlike the initialization, the mapping of quantities scales very well with an increasing number of MPI ranks. Beyond a certain number of ranks, no further speedup is achieved, which is because the problem size per rank becomes very small. The absolute times are very small, in particular when compared to typical solution times of solvers. The performance is mainly driven by the matrix-vector multiplication, which is very efficiently realized by the trilinos framework.

The presented tests clearly show the high performance of the developed mapping framework in distributed environments, even with very large meshes and a high number of MPI ranks. The proposed strategies and procedures can thus be used even with very large coupled simulations.

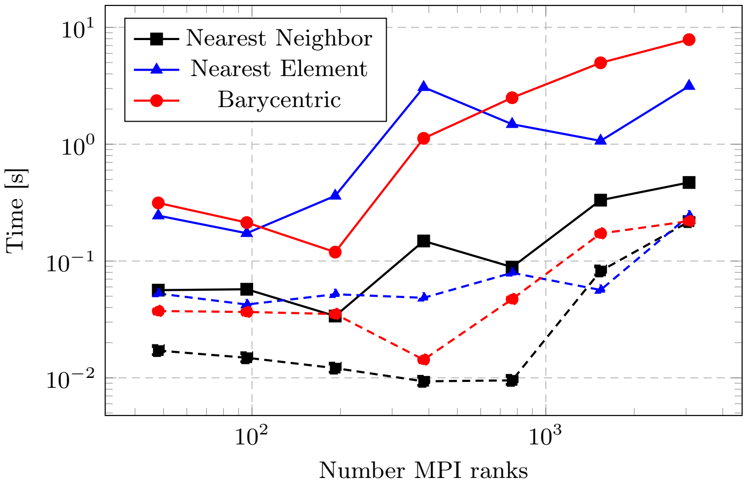


Figure 4.2: Mapper initialization times for different numbers of MPI ranks in surface mapping. Solid lines represent mapping coarse → fine mesh, dashed lines represent mapping fine → coarse mesh.

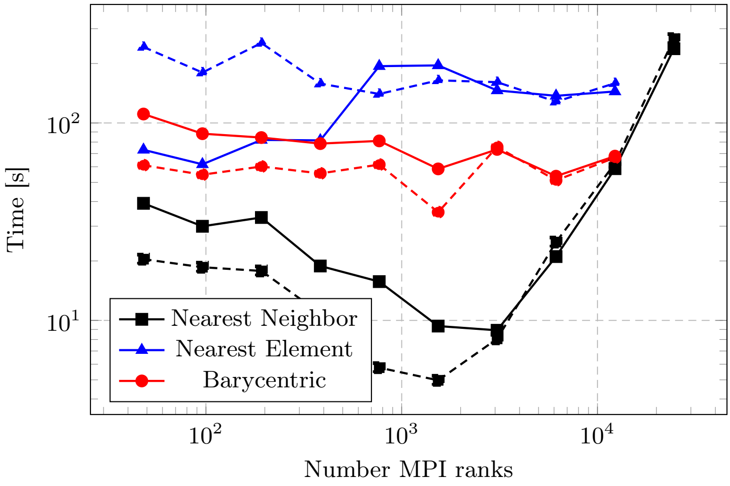


Figure 4.3: Mapper initialization times for different numbers of MPI ranks in volume mapping. Solid lines represent mapping coarse → fine mesh, dashed lines represent mapping fine → coarse mesh.

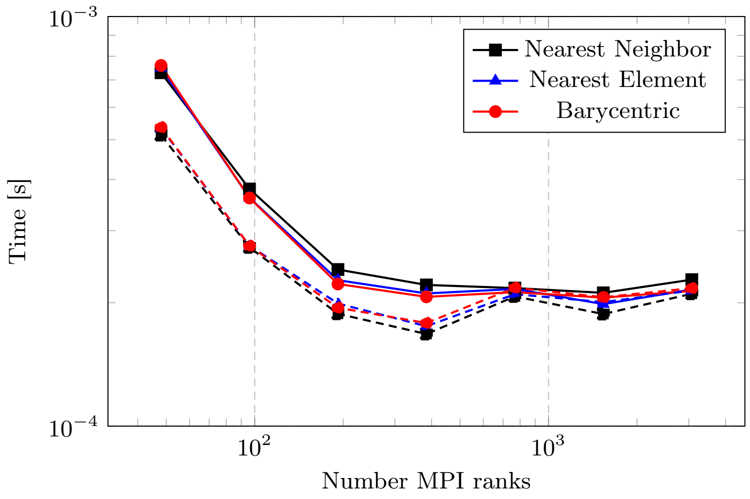


Figure 4.4: Mapping times for different numbers of MPI ranks in surface mapping. Solid lines represent mapping coarse → fine mesh, dashed lines represent mapping fine → coarse mesh.

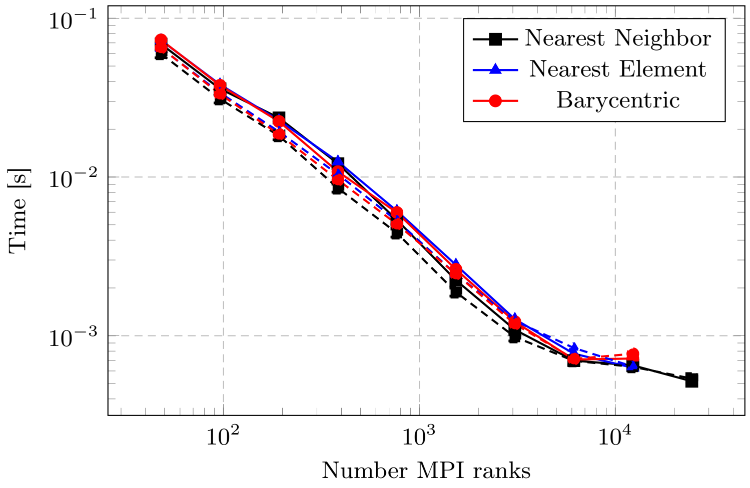


Figure 4.5: Mapping times for different numbers of MPI ranks in volume mapping. Solid lines represent mapping coarse → fine mesh, dashed lines represent mapping fine → coarse mesh.

Chapter 5

Realizing coupled simulations

This chapter contains a collection of experiences and recommendations for approaching, setting up and running coupled simulations. The first section details the different aspects and components of coupled simulations and their interaction and importance. The following sections provide information on CoSimulation on HPC systems, which is helpful due to the often large computational requirements of coupled simulations.

The findings presented in the following are gathered and developed while conducting this work, and successfully applied to the examples presented in Chapter 6.

5.1 Practical aspects and experiences

Coupled simulations are a challenging endeavor to embark on. Not only do they contain and combine the difficulties of standalone simulations, but furthermore add coupling-specific problems on top. Therefore, this section lists and details important aspects and experiences that can be crucial for successfully conducting coupled simulations. Additional information can be found in [27], which provides a comprehensive overview of the partitioned analysis of coupled mechanical systems.

A general recommendation in the context of FEM by [26, Chapter 7.2] is to *keep it simple*. This is even more vital for coupled simulations, which contain the complexity of the coupling in addition to the challenges of their models. Therefore, this can be considered a principle for conducting CoSimulation.

See also Section 2.2 for the theory related to coupled simulations.

5.1.1 Bottom up, from individual models

Before any coupled simulation is set up, each model should be thoroughly tested standalone. Mistakes and other shortcomings would undoubtedly be revealed during the coupling and complicate the simulation, particularly when a strong interaction occurs.

Coupling relevant tests can be conducted in addition to the tests with which a model is checked standalone. Representative scenarios which are expected to occur during the coupling can be used to check the model and understand its behavior. A way to obtain relevant data is to run the standalone simulations and use their output as input or boundary conditions for the other model. This one-way coupling might not contain and show all the phenomena that can happen during a fully two-way coupled simulation, but it can certainly assist in debugging a model. If applicable, this data can be modified, e.g. by adding more fluctuations to represent a stronger interaction.

Such simplified couplings are particularly useful when simulations with large computational requirements are involved. Then the coupling can be prepared, thus potentially saving many resources.

A dedicated *CouplingOperation* (see Section 3.2.10) can be used to read the information for the one-way coupling.

5.1.2 Choosing the coupling algorithm

An early choice to be made is the selection of the coupling algorithm. Due to the computational cost, starting with a weak coupling (see Figure 2.3a) and a Gauss-Seidel pattern (see Figure 2.2b) is recommended. If it is known upfront that the interaction between the fields is strong, then a strong coupling is suggested instead, again with a Gauss-Seidel pattern.

If the simulation diverges because the interaction is too strong for a weak coupling, then a strong coupling (see Figure 2.3b) should be used instead. Some boundary conditions setups might also require to use a Jacobi pattern (see Figure 2.2a) instead.

5.1.3 Prediction techniques

It is recommended to use prediction techniques to improve the convergence behavior of the coupled solution. In particular, for a weak coupling, using prediction can make the difference to achieve a converged solution. For FSI problems, the prediction technique presented by [21] has worked well for practical problems. Note that some prediction techniques require a certain coupling sequence.

5.1.4 Relaxation and convergence acceleration techniques

Strongly coupled simulations often require (under-)relaxation to achieve stable solutions. Convergence acceleration techniques should then be used to speed up the convergence of the coupled solution. This decreases the number of coupling iterations and, therefore, results in a faster overall simulation time.

As before, the stronger the interaction becomes, the more elaborate the acceleration techniques should be. The Aitken acceleration [50] works well until a certain degree, after which it is beneficial to use methods like MVQN [7] or IQN-ILS [20]. [85] presents an advanced accelerator focusing on memory efficiency.

5.1.5 Mapping

If mapping on meshes/geometries is required for the coupling, then it is recommended to use matching meshes if possible for setting up the coupling. This way, no mapping-related problems can occur, and the focus can be set to the other components of the coupling. Once those are done, the mapping can be set up using the actual meshes. As explained in Chapter 4, more complex mapping techniques introduce more sources for errors. Hence, the usage of the nearest neighbor mapper 4.1.1 can simplify the initial setting up of the mapping. Once this is working, then more elaborate mappers can be used. The nearest element mapper 4.1.2 is generally recommended since it provides good results for low computational cost.

Mapping involves a geometrical search on the interfaces to determine the partners among which the data is exchanged, see Section 4.4. Depending on the mapping algorithm used, the searched entities vary, most commonly are closest points (nearest neighbor/barycentric) or closest projections (nearest element). Wrong search results can happen in practical examples for many reasons, such as corner cases or very small entities. Many of these problems can be avoided by splitting the geometry into appropriate inputs for the mapper. Instead of passing the entire interface all at once, multiple mappers can be created, each with a conforming part of the interface. This not only makes the search more computationally efficient, but it also helps to avoid wrong pairings.

Finally, the choice of mapping technique also affects the data that is required by an (external) solver. If the mapper does not use the underlying geometry or shape functions (such as nearest neighbor or barycentric mapper), then it is sufficient to provide only the nodes with the data.

5.1.6 Restart in CoSimulation

Some solution techniques such as CFD require, depending on the problem setup, some time until they reach a converged/steady-state behavior. Unsteady effects during the initial phase can lead to numerical instabilities for the coupling and should therefore be avoided. One way is to run the solvers standalone before the coupling until a stable state is reached. A practical realization is to save this state in a restart file and start the coupling from there.

Note that the mapping might need some modifications if the current configuration after the restart is different from the initial configuration, on which the search for the coupling partners is usually conducted. This is used in example 6.7.

Restarts are also a commonly used technique on HPC systems, where the runtime per simulation is typically limited. Then the total simulation time needs to be split into several sub-intervals.

Lastly, they can be used for debugging. With restarts, it is possible to start from a simulation step close to the problematic one, without having to rerun a long simulation entirely.

5.1.7 Gradual start of coupling with ramping

Applying loads or other boundary conditions instantaneously on a model can lead to numerical instabilities, in particular for complex real-world examples. It is therefore advised to apply them gradually, over a short time. The more sensitive the model is to a particular load, the longer the ramping time should be. This has been shown to improve the convergence behavior significantly.

Different versions of ramping functions can be used, a linear one being the most basic and straightforward to realize. However, it has two kinks, at the beginning and the end. Those can introduce problems with sensitive models. A better solution is to use a smooth ramping function as shown by [60] and displayed in Figure 5.1. This function does not have any kinks and provides a very smooth ramping.

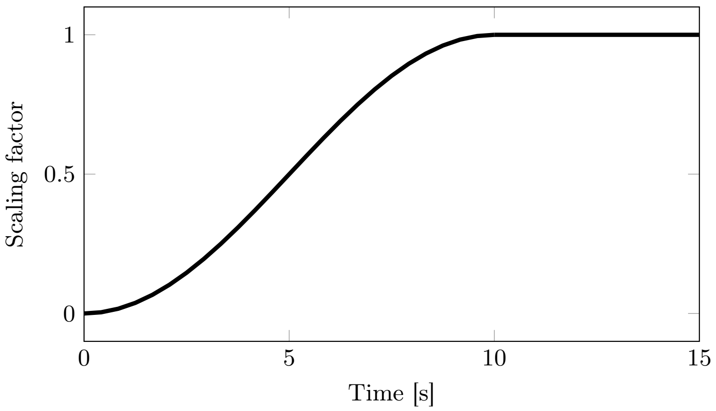


Figure 5.1: Smooth ramping up over 10 s.

The displayed function can be realized with *cos* or *sin* function (with 10 s as ramping interval):

$$f(t) = 0.5\left(1 - \cos\left(\pi \frac{t}{10}\right)\right)$$

$$f(t) = \sin^2\left(0.5\pi \frac{t}{10}\right)$$

This function has been applied successfully for different coupling quantities and simulations, such as shown in Figure 6.41. The integration into the CoSimulationApplication is done with a dedicated *CouplingOperation*, see Section 3.2.10.

5.1.8 Coupling with external solvers/tools

Coupling with external tools increases the complexity of the coupling since several programs need to be combined. Following are some pieces of advice on setting up the coupling in smaller steps, which helps to isolate potential problems.

Using remote-controlled CoSimulation

Problems with the coupling sequence are common issues faced during the setting up. In particular, if the classical approach (see Section 3.4.2) for CoSimulation is employed, because the orchestration is not handled centralized but individually by each tool. One way of overcoming this is to use the remote-controlled approach, as explained in Section 3.4.3. Here the orchestration is handled by one component, thus avoiding deadlocking due to inconsistencies in the coupling sequence among the tools.

Using internal solvers to set up a coupling

Using a multiphysics tool for the coupling has another inherent advantage over traditional coupling tools: The coupling can be first developed with internal solvers (given that the required solver is available), thus avoiding problems with synchronization and data exchange between tools. This can be done with simplified models if appropriate. Once the coupling is working, the internal solvers can be replaced with the real (external) ones, which limits the complexity and helps to narrow down potential problems.

Choice of IPC method for communication

In most cases, IPC is required for CoSimulation with an external solver. The interface data needs to be exchanged, which can be done with different methods. While TCP-sockets have been found to be the best solution for running coupled simulations (see Section 3.8), file-based communication can be a better choice for the development and initial setting up of a coupling. The flow of information can be easily followed and observed, which helps to understand the coupling sequence.

CoSimIO (see Section 3.7) unifies the interfaces of different IPC methods, such that they can be changed at runtime with a configuration parameter.

5.1.9 Example setup of strongly coupled FSI

The example configuration in 5.1 shows some components of a strongly coupled FSI simulation. It is crucial that the fields on which prediction or acceleration act (field *disp* of solver *fluid* in this example) are not overwritten by mapping! The mapping of the displacements from the fluid solver to the structural solver happens at the end of the coupling sequence because it is specified in the *output_data_list* of the structural solver. The acceleration (in case the solution did not converge in this timestep) happens at the end of the coupling iteration after the mapping. Similar rules apply to the prediction it happens at the beginning of the timestep, before the coupling sequence. This means that the prediction or the relaxed solution would be overwritten if the mapping of the displacements would happen in the *input_data_list* of the fluid solver. The same rules applies also to a weak coupling when using a predictor.

```
...
"predictors" : [{
    "type"       : "linear",
    "solver"     : "fluid",
    "data_name"  : "disp"
}],
"convergence_accelerators" : [{
    "type"       : "aitken",
    "solver"     : "fluid",
    "data_name"  : "disp"
}],
"coupling_sequence": [{
    "name": "fluid",
    "input_data_list"   : [],
    "output_data_list" : []
},{
    "name": "structure",
    "input_data_list": [{
        "data"                 : "load",
        "from_solver"          : "fluid",
        "from_solver_data" : "load",
        "data_transfer_operator" : "mapper"
    }],
    "output_data_list": [{
        "data"               : "disp",
        "to_solver"          : "fluid",
        "to_solver_data" : "disp",
        "data_transfer_operator" : "mapper"
    }]
}]
...
```

Listing 5.1: Snippet of configuration of strong coupling

5.2 CoSimulation on HPC systems

Most HPC systems like SuperMUC-NG (see Appendix A) use batch systems for the processing of jobs. This means that simulations cannot be executed right after logging into the system, instead, they must be submitted to a job scheduler. Depending on the resource requirements, runtime, and priority, they are then executed. SuperMUC uses the workload manager SLURM [84].

The submission of jobs with those schedulers is different from the regular execution of programs. In particular for CoSimulation, where multiple programs are involved. Launching the simulations with a different number of tools and types of parallelizations is explained in the following. While the descriptions are made for the SuperMUC-NG system with the SLURM workload scheduler, the concepts can be transferred to other systems.

One software, with or without MPI

Running single-physics simulations with one solver or tool means that only one executable is used. The same applies to conducting coupled simulations with a multiphysics tool like Kratos. The executable can directly be executed when MPI is not used. With MPI, *mpiexec* can be employed directly to launch the execution, see 5.2.

```
# without MPI:
./executable > std.out 2> err.out

# with MPI (100 processes):
mpiexec -n 100 ./executable > std.out 2> err.out
```

Listing 5.2: Running a single executable with or without MPI. Output and errors are redirected to dedicated files.

Multiple software, without MPI

Conducting multiphysics simulations with more than one tool can be done straightforwardly if MPI is not used. The executables can be launched one after the other. The & at the end is required to return right away and not wait until the command is finished, otherwise the simulation would hang. This needs to be combined with a *wait* at the end to prevent premature termination before all the programs have finished. A short wait between the launches of the programs ensures stability by avoiding concurrent starts, which can be problematic on some systems. An example with three programs can be seen in 5.3.

```
./executable1 > std1.out 2> err1.out &

sleep 5s # helps to avoid problems when starting the job

./executable2 > std2.out 2> err2.out &

sleep 5s # helps to avoid problems when starting the job

./executable3 > std3.out 2> err3.out &

```

```
wait # prevent premature termination before individual programs have
     completed
```

Listing 5.3: Running three executables without MPI. Output and errors are redirected to dedicated files.

Multiple software, one uses MPI

Multiphysics simulation with multiple tools can be done in a similar way as before when only one of them uses MPI. *mpiexec* can still be used in this case, an example can be seen in 5.4. Note that process-pinning as described in Section 5.3 needs to be used when the other tools employ shared memory parallelization.

```
./executable1 > std1.out 2> err1.out &

sleep 5s

./executable2 > std2.out 2> err2.out &

sleep 5s

mpiexec -n 100 ./executable3 > std3.out 2> err3.out &

wait
```

Listing 5.4: Running three executables, one of them with MPI. The order in which they are called does not matter. Output and errors are redirected to dedicated files.

Multiple software, all use MPI

Using *mpiexec* is no longer possible when more than one tool uses MPI. The low-level command *srun*[1] (which is also called by *mpiexec*) provided by the SLURM workload manager needs to be used then. This is because several options such as the distribution of compute cores need to be specified, which is not possible with *mpiexec*. 5.5 shows an example of running three programs with 240 cores/ 5 compute nodes each. In total 720 cores/15 nodes need to be allocated for this job. The distribution of cores for the programs is blocked, and the type of MPI is specified as pmi2. Each program uses its resources exclusively, without sharing them. Note that those settings might be different on other systems with different schedulers.

```
# starting executable 1 on 5 compute nodes (240 compute cores)
srun -N 5 --ntasks-per-node=48 -m block:block --export=ALL --
     exclusive --mpi=pmi2 ./executable1 > std1.out 2> err1.out &

sleep 5s

# starting executable 2 on 5 compute nodes (240 compute cores)
srun -N 5 --ntasks-per-node=48 -m block:block --export=ALL --
     exclusive --mpi=pmi2 ./executable1 > std2.out 2> err2.out &

```

[1] https://slurm.schedmd.com/srun.html

```
sleep 5s

# starting executable 3 on 5 compute nodes (240 compute cores)
srun -N 5 --ntasks-per-node=48 -m block:block --export=ALL --
     exclusive --mpi=pmi2 ./executable1 > std3.out 2> err3.out &

wait
```

Listing 5.5: Running three executables, all of them with distributed memory. The srun command together with other options is used as mpiexec does not offer to specify these options. Output and errors are redirected to dedicated files.

Finally, comparing the different ways of launching coupled simulations on a HPC system highlights the advantage of a multiphysics framework. The handling is significantly easier compared to using multiple tools, particularly when using MPI.

5.3 Mixing MPI and OpenMP with process pinning

Mixing different methods of parallelism is often done with hybrid parallelization. Shared and distributed memory parallelization (see Section 2.4) are combined by employing shared within one computing node, and using distributed in-between the nodes. This type of parallelization requires the solver to support it. Some solvers however implement only one of the technologies. Combining solvers that support different means of parallelization for CoSimulation might require special solutions to enable both solvers to employ their preferred method of parallelization. This solution is hardware-dependent. A solution on Intel systems to enable both MPI and OpenMP is to use *process pinning* as presented in the following.

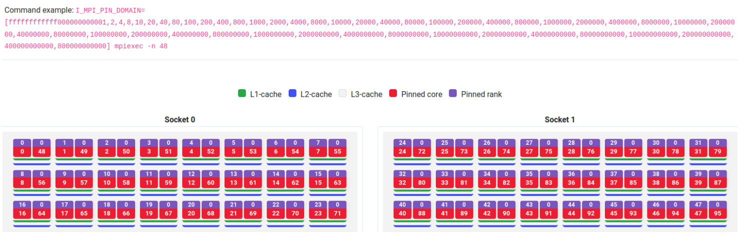
The system used in this work for the large simulations, SuperMUC, uses Intel CPUs, see Appendix A. The Intel-MPI library provides a way to pin processes to specific CPUs[2]. Various options are available for the pinning which can be tested with a simulator[3]. The output is a pinning mask, which is used together through the environment variable *I_MPI_PIN_DOMAIN*.

Mixing OpenMP and MPI parallelization can be achieved by a special pinning setting. Then each tool can use its preferred mechanism. For the first rank, thread one on the first core and the second thread of every other core is chosen. For every other rank, the first thread of one core is chosen. The shared memory parallel tool runs on rank zero, where it can use all threads of one compute core, 48 in this case. This pinning is visualized in Figure 5.2.

It was observed that enforcing the pinning harms the performance of the code that uses shared memory parallelization, compared to running it standalone. 6.7.5.2 demonstrates this for the structural model of the Olympic

[2] www.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-windows/top/environment-variable-reference/main-thread-pinning/interoperability-with-openmp-api.html

[3] www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library-pinning-simulator.html



```
Command example: I_MPI_PIN_DOMAIN=
[ffffffffffff000000000001,2,4,8,10,20,40,80,100,200,400,800,1000,2000,4000,8000,10000,20000,40000,80000,100000,200000,400000,800000,1000000,2000000,4000000,8000000,10000000,200000
00,40000000,80000000,100000000,200000000,400000000,800000000,1000000000,2000000000,4000000000,8000000000,10000000000,20000000000,40000000000,80000000000,100000000000,200000000000,
400000000000,800000000000] mpiexec -n 48
```

Figure 5.2: Screenshot from the Intel pinning simulator. It shows the pinning used in this work. A computing node on SuperMUC has 48 physical cores and 96 threads (with Hyper-Threading enabled). The first rank has the first thread of core one and thread two of every other core. The other ranks have one thread each. The OpenMP parallel code runs on rank zero, where it can use up to 48 threads.

Roof. A slowdown of ~50% occurred when using process pinning. Still, this is much faster than not using parallelization.

Since the pinning is different for each CPU, the information about the used hardware needs to be provided to the simulator. This information can be obtained with the *cpuinfo*[4] tool.

The full pinning mask can also be computed by using bitshift operations as shown in 5.6. The input here is the pinning info for the first rank, the others are computed.

```bash
# from the pinning simulator for rank 0
mask=ffffffffffff000000000001

for i in $(seq 1 47); do # one CPU has 48 physical cores
  mask=$mask,$(printf "%x\n" $((1<<$i)));
done

export I_MPI_PIN_DOMAIN=[$mask]

# this results in:
# [ffffffffffff000000000001, 2, 4, 8, 10, 20, 40, 80, 100, 200, 400,
     800, 1000, 2000, 4000, 8000, 10000, 20000, 40000, 80000,
    100000, 200000, 400000, 800000, 1000000, 2000000, 4000000,
    8000000, 10000000, 20000000, 40000000, 80000000, 100000000,
    200000000, 400000000, 800000000, 1000000000, 2000000000,
    4000000000, 8000000000, 10000000000, 20000000000, 40000000000,
    80000000000, 100000000000, 200000000000, 400000000000,
    800000000000]
```

Listing 5.6: Compute the process pinning mask

A job with one executable can be launched as shown in 5.7. The important part is to specify the pinning by using the *I_MPI_PIN_DOMAIN* environment variable. Here the code runs MPI-parallel but has part of it executed in

[4] https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/command-reference/cpuinfo.html

shared memory parallel. An example of this case is shown in Section 6.5.3, Olympic tower FSI. In setup 2, the solvers run in the same process, but use different means of parallelization. The fluid solver runs in MPI, whereas the structural solver uses OpenMP.

```
# set number of threads for shared memory parallelization
export OMP_NUM_THREADS=10

I_MPI_PIN_DOMAIN=[$mask] mpiexec -n 100 ./executable
```

Listing 5.7: Using different means of parallelization within one executable.

Starting multiple executables with process pinning is done similarly, as can be seen in 5.8. A use case for this scenario is setup 3-6 of the Olympic Tower FSI. Unlike in setup 2, the structural solver runs in a separate process. This is a very common FSI scenario, the fluid solver running in MPI due to the typically higher computational cost, and the structural solver using OpenMP.

```
# set number of threads for shared memory parallelization
export OMP_NUM_THREADS=10

# starting code that uses MPI (100 processes)
I_MPI_PIN_DOMAIN=[$mask] mpiexec -n 100 ./executable1 &

# starting code that uses OpenMP (10 threads)
./executable2 &

wait
```

Listing 5.8: Using different means of parallelization within two executables.

Chapter 6

Numerical Examples

This chapter contains examples that present and show the capabilities and features explained in the previous chapters. Each example focuses on one or more aspects of CoSimulation.

The first example (Section 6.1) presents the results of a widely used FSI benchmark conducted with the presented work. Section 6.2 uses the same benchmark but replaces the structural solver with a NNet. In Section 6.3, FSI simulations of a bridge deck are conducted with a simplified structural solver, which consists of only one DOF. The simulation of rock-fall protection structures is shown in Section 6.4 by coupling a DEM and a FEM solver. A comparison of different scenarios of CoSimulation is presented in Section 6.5, where Wind-Structure Interaction (WSI) simulations of the Munich Olympic Tower are used as the reference model. The coupling to external solvers is presented in Section 6.6, where the FSI of an entire wind turbine is simulated. Finally, the culmination of this work is presented in Section 6.7 by conducting large-scale FSI simulations of the Olympic Stadium Roof in Munich.

6.1 Flexible flap in channel

This example consists of a 2D flow in a narrowing channel and a flexible flap restricting the flow. The densities of the fluid and the structural domain are of similar magnitudes. This leads to a strongly coupled problem, which is one of the main challenges of this example.

It was initially proposed by [60] and has often been used as a benchmark for FSI applications, e.g. by [78]. Therefore, it is also used to evaluate this work's capabilities for solving strongly coupled problems. The original setup is recreated, with geometry and boundary conditions shown in Figure 6.1, and material parameters listed in Table 6.1.

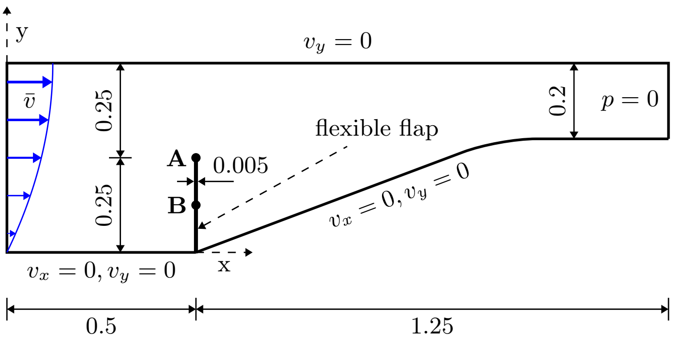


Figure 6.1: Geometry, boundary conditions and setup for Mok benchmark. Dimensions are in meters.

Structure		Fluid	
ρ^S	$1500\ Kg/m^3$	ρ^F	$956\ Kg/m^3$
E^S	$2.3e10^6 N/m^2$	μ^F	$0.145\ Pa \cdot s$
ν^S	0.45		

Table 6.1: Material parameters for Mok benchmark. The densities are of similar magnitude.

$$v(y,t) = 4\bar{v}y(1-y) \tag{6.1}$$

Kratos was used both for the fluid and the structural solution, with the *FluidDynamicsApplication* and the *StructuralMechanicsApplication*, respectively. The fluid mesh consists of around 6000 triangular elements. The inflow profile is parabolic (see equation 6.1), with $\bar{v} = 0.06067$. The velocity is ramped up over the first 10 seconds, see Section 5.1.7. The mesh is moved using a structural similarity technique.

The structural mesh is composed of 200 large displacement quadrilateral elements. A plane stress material law with unit thickness is employed.

An iterative Gauss-Seidel solution technique (see Figure 2.2b) is used in combination with the MVQN relaxation technique, see [7]. The meshes on the interface are matching in order to eliminate mapping errors. The total simulation time is 25 s, with a fixed time-step size ΔT of 0.1 s.

The results of the displacements at the reference points **A** and **B** are shown in Figure 6.2. The results of this work are in very good agreement with the reference solutions from [60] and [78]. This showcases the ability of the presented implementation to handle strongly coupled problems.

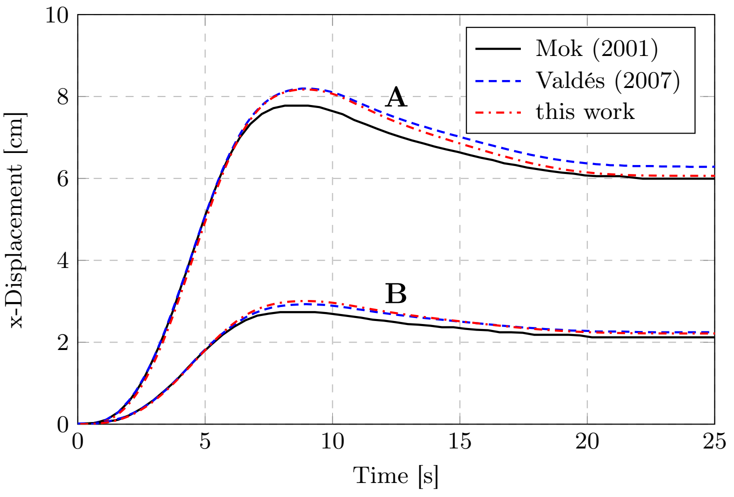


Figure 6.2: Results for displacement in X-direction at reference points **A** and **B**

6.2 Flexible flap in channel; with neural network

Artificial Intelligence (AI) and Machine Learning (ML) have seen a phenomenal rise in recent years, with their potential use cases seemingly endless. One of them is the replacement of numerical solution techniques with a predicted solution through a surrogate model. The main motivation for this is to obtain solutions much faster and with little computational effort. The downside is the accuracy, which highly depends on the data the model was trained with. In the following, the integration of a NNet based tool into CoSimulation is presented.

This example uses the same FSI benchmark as in Section 6.1, but the structural solver is replaced with a NNet. These developments are initially proposed by [23]. The NNet related functionalities are realized in Kratos within the *NeuralNetworkApplication*. The contribution of this work is the initial design for the integration into the CoSimulationApplication, as well as assisting the implementation and realization in Kratos. In the following, only the CoSimulation relevant aspects are presented. For more information, it is referred to [23].

The *NeuralNetworkApplication* is implemented in Kratos, which simplifies the integration into CoSimulation, since no communication with an external tool via IPC is required. Same as for other solvers/tools, the integration into CoSimulation was done based on the common solver interface *AnalysisStage*, see Section 3.1.3. The *ModelPart* was employed as the container for the data (see Section 3.1.1), consistent with the other Kratos internal solvers.

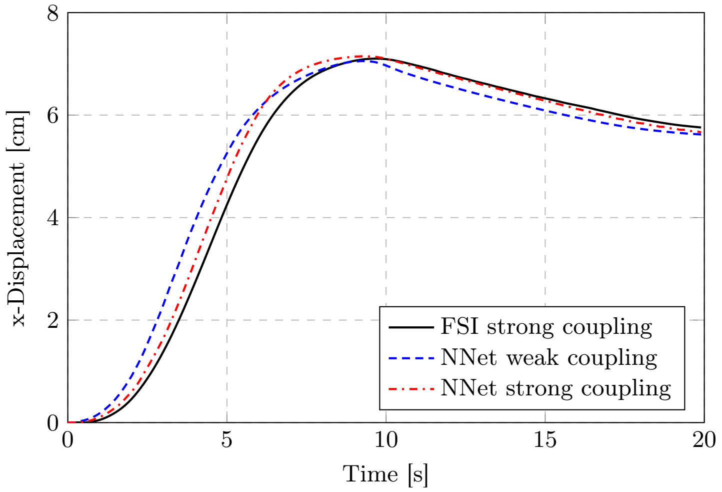


Figure 6.3: Results for displacement in X-direction at reference point **A** (top of the flap).

The coupling is set up in the same way as when using a conventional solver. The fluid solver computes the loads, which are being mapped to the structural domain. The loads are used as input to compute new displacements. Instead of using a numerical solver, a NNet model takes the loads at the nodes as input to predict the new displacements, which are then mapped back to the fluid domain. This surrogate model needs to be trained with data for an accurate prediction. The required data is generated from coupled simulation runs with varying inputs. These data generation and training procedures are part of the *NeuralNetworkApplication*. After training the model, it can be employed in the coupled simulation. Figure 6.3 shows the results for the displacements for different configurations. The solutions computed with the NNet are in very good agreement with the reference solution. Note that the strong FSI coupling results differ from the results shown in Figure 6.2 because a much coarser fluid mesh was used.

6.3 FSI with a SDOF solver

CoSimulation is often understood as the coupling of solvers that use mesh-based discretization methods. Mapping techniques as described in Chapter 4 are then employed for the data transfer between the interfaces. However, not all solvers require an approach based on FEM or FVM (including discretization and meshing). Typically, reduced physical phenomena can also be modeled with more basic solvers.

An example of such a solver is a concentrated spring-mass-damper system, which only has one DOF (in the following referred to as SDOF solver), representing either translation or rotation. It has no geometry associated and thus no mesh. This means that the integration into CoSimulation requires different procedures for the data transfer between interfaces. Regular mesh-based mapping techniques cannot be used. Coupling two SDOF solvers can be done by copying the value from one interface to the other.

Coupling solvers, where only one uses a geometry, require some treatment similar to mapping. The transfer of data depends on the direction of the exchange, as well as the physical quantities. To illustrate this, a FSI coupling between a CFD solver and a structural solver with one DOF is considered. The schematics of the setup for translational movement are shown in Figure 6.4. The displacements computed by the structural solver are communicated to the fluid solver, where they are applied to all the nodes on the interface. In the other direction, the nodal forces need to be summed up to apply them to the structural solver. This is the equivalent of conservative mapping, see Section 4.2. If the fluid solver is using distributed memory parallelization, additionally gather and scatter operations are required on the interface, since the structural solver only uses one rank (scenario 3 in Figure 3.2).

For a rotational movement, the data transfer needs to be treated differently. A reference point is required, from which the displacements of the fluid interface can be calculated based on the rotation. The same applies to the transfer of loads, they are transformed into a moment utilizing the reference point.

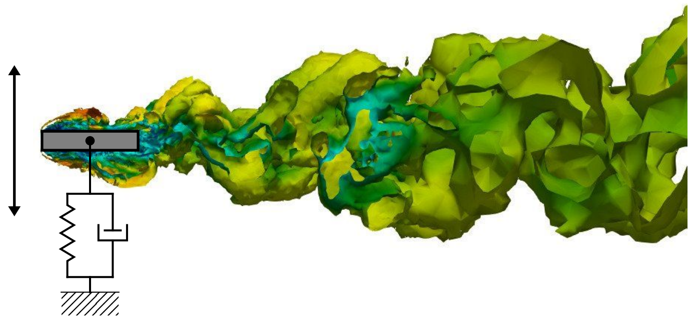


Figure 6.4: SDOF FSI setup, for translational movement. The mechanical spring-mass-damper system is displayed. The vortices that develop behind the obstacle are visualized by means of an isosurface of the Q-criterion, with a Q-value of 0.025 s^{-2}.

This type of data transfer is implemented with a dedicated *DataTransferOperator*, see Section 3.2.6. The flexible software architecture of the CoSimulationApplication allows for the realization of different techniques of data transfer, which are adapted to the physics of the problem to be solved.

One application of the described procedures is studies of the flow around bridge decks. Unsteady CFD simulations are used to enable the capturing of transient effects such as flow-induced vibrations. The structural behavior is modeled with one DOF, as shown in Figure 6.4.

[61] studies the vortex-induced vibrations over a 5:1 rectangle by means of numerical simulations as well as wind-tunnel experiments. Their simulations were recreated within a larger campaign on bridge simulations. These studies were performed together with Máté Péntek and Guillermo Martínez-López, as part of common developments of mapping and coupling methods for the validation of bridge deck simulations. The results obtained for translational movement (referred to as heaving mode given the movement of the bridge deck) with this work are in good agreement. Figure 6.5 compares the amplitude, and Figure 6.6 the frequency of the structural response.

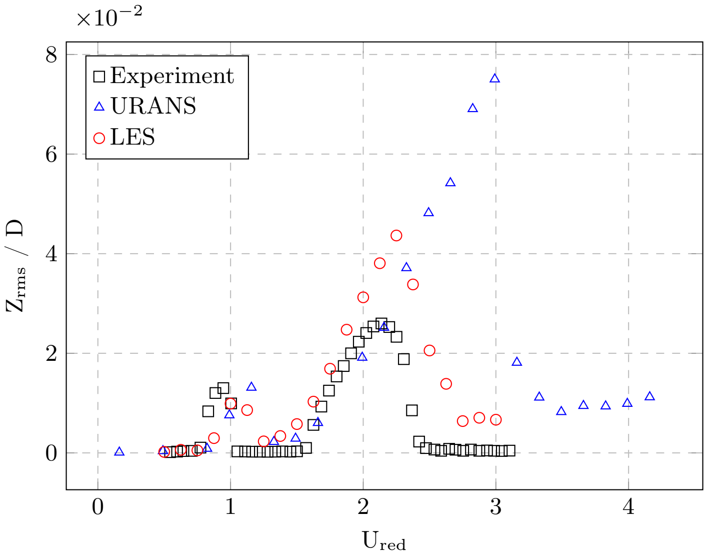


Figure 6.5: Amplitude of the structural response, for the heaving mode. *Experiment* and *URANS* refer to [61], *LES* refers to this work.

6.4 Rock fall protection structures

Roads through mountainous regions, such as in the Alps, are in danger of rock fall. Rocks falling onto the roads and hitting vehicles can have devastating results. Therefore, protection structures such as cable nets are used to protect the roads in such terrain.

As with other engineering disciplines, numerical simulations can assist in the design of these structures. The interaction between the falling rocks and the protection structures makes it a problem involving different physical

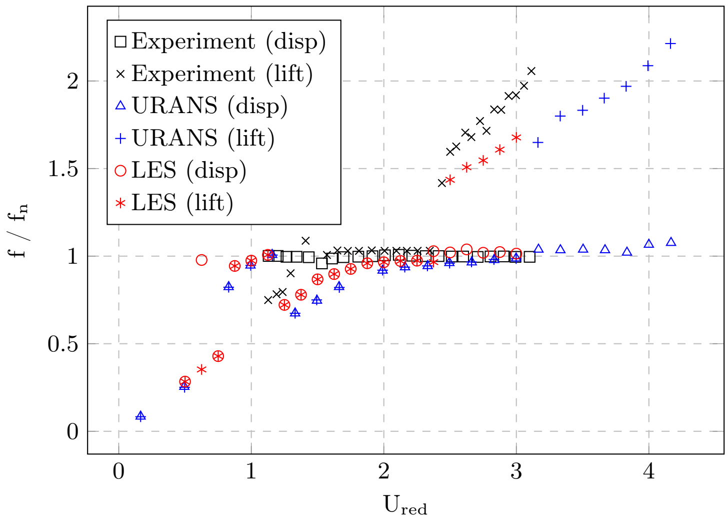


Figure 6.6: Frequency of the structural response, for the heaving mode. *Experiment* and *URANS* refer to [61], *LES* refers to this work.

phenomena. On the one hand, the behavior of the structure under the rock impact needs to be simulated. On the other hand, the rock and its dynamic behavior while falling down the mountain and then hitting the structure leads to high dynamics. The large kinetic energy is dissipated through plasticity and friction, while the large deformation capabilities of the cable net allow a relatively slow energy transfer.

[67] investigates the numerical simulation for the behavior of different structures under different conditions in detail, including validation against experimental results. [69] presents initial findings of this work.

The presented application case is taken from [67] / [69] and was prepared by Klaus Sautter. The contribution of this work is assisting in the development of the coupling, as well as the integration into the coupling framework *CoSimulationApplication*. These thoughts and considerations are presented in the following.

The cable net is modeled using the FEM, the rock with the DEM. This means that the application differs from the other cases in this work, which focus on FSI.

The DEM models the particles as discrete elements, which interact with each other and surrounding entities such as walls. Following [66], three basic steps are part of the simulation. Firstly, the contact between the particles and the surrounding entities is computed. This information is then used to compute contact forces in a second step. The third step consists of updating the positions of the particles by means of numerical integration. The contact and inertial forces are taken into account for this step.

6.4.1 Coupling DEM - FEM

The partitioned solution of a coupled problem requires establishing interface conditions, which need to be fulfilled to achieve equilibrium between the domains. For the DEM-FEM coupling, they can be formulated as the balance of contact forces $\mathbf{F}_C$ from the particles with the internal forces of the structure $\mathbf{F}_{int}$, see equation 6.2 (taken from [69]).

$$\begin{aligned} &\mathbf{F}_C\left(\mathbf{u}^{\Omega_{S,\Gamma_C}}, \dot{\mathbf{u}}^{\Omega_{S,\Gamma_C}}, \mathbf{u}^P, \dot{\mathbf{u}}^P\right) \\ &-\mathbf{F}_{int}^{\Omega_{S,\Gamma_C}}\left(\mathbf{u}^{\Omega_S}, \dot{\mathbf{u}}^{\Omega_S}, \ddot{\mathbf{u}}^{\Omega_S}\right) = 0 \end{aligned} \tag{6.2}$$

Ω_S denotes the structural domain, Ω_{S,Γ_C}, the coupling interface. The contact forces depend on position and velocity of the particles ($\mathbf{u}^P, \dot{\mathbf{u}}^P$) and the wall ($\mathbf{u}^{\Omega_{S,\Gamma_C}}, \dot{\mathbf{u}}^{\Omega_{S,\Gamma_C}}$).

The displacements and velocities of the cable net as computed by the structural solver are transferred to the wall, where they are imposed as boundary conditions for the particles. The structure undergoes large deformations during the impact of the rock, which in turn significantly changes the boundary of the particle simulation. This strong interaction requires a two-way coupling of the solvers to achieve stable and accurate numerical solutions. Weak and strong coupling algorithms, as described in Section 2.2.1, can be employed. Details regarding the selection of convergence criteria and relaxation techniques for the strong coupling are described in detail in [69]. Explicit time integration is used to resolve the highly dynamic impact process.

The mapping happens between the FEM structure (cable net) and the DEM wall. The wall as the boundary condition for the particles models the cable net exactly, using the same geometry and discretization as for the structural domain. This leads to edge-to-edge mapping on matching meshes. Figure 6.7 illustrates the exchange of data between the domains.

The solution procedures for the particle simulation are implemented in the *DEMApplication* within Kratos. The *StructuralMechanicsApplication* was used for the structural analysis. The coupling could therefore be conducted entirely within Kratos, without the need for any explicit communication via IPC between the solvers. Both solvers use the *ModelPart* as data structure, see Section 3.1.1. Therefore, CoSimulation can directly access their data. Additionally, the common solver interface as described in Section 3.1.3 is implemented for both solvers, which enables CoSimulation to orchestrate their execution and synchronization.

6.4.2 Angled protection net

This example was first presented in [69]. It simulates a protection net that spans over a street, used to redirect falling rocks to avoid impact with vehicles. Figure 6.8a shows the application of this kind of protection structure, whereas the corresponding numerical model is displayed in Figure 6.8b.

The rocks are modeled as (spherical) particles for simplicity. More complex shapes can be realized by combining many particles into clusters, as presented

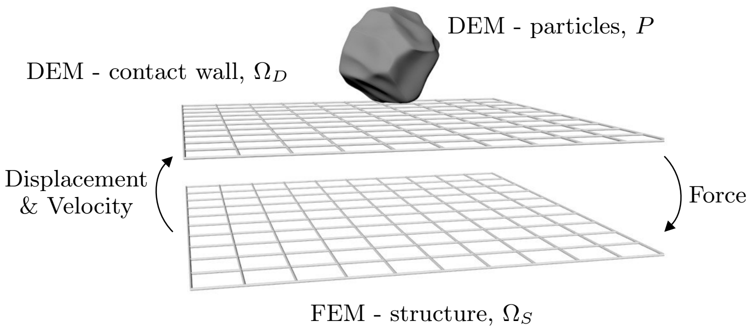


Figure 6.7: Concept of DEM-FEM coupling. The particles impact the DEM wall, causing contact forces. Those are then mapped to the structural domain, where they are taken into account as boundary conditions. Based on the external loads from the DEM, the structure computes a new solution. The newly computed displacements and velocities are then mapped to the DEM wall, where they are used to update the boundary.

(a) Angled rock fall protection structure, spanning over a road (Route Chalais-Vercorin, Valais). Source: Geobrugg `www.geobrugg.com`

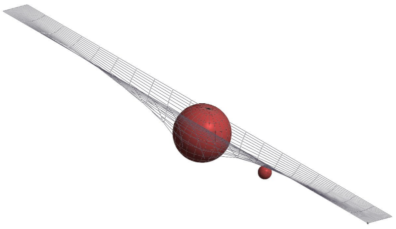

(b) Computational model for protection net. The stones are modeled as individual particles.

Figure 6.8: Rock fall protection net and computational model.

in [68]. A large and a small particle are used to verify that small objects are not caught by the net. To test the limits of the coupling, a challenging setup is used. The cable net is not prestressed, and the rock has a high impact velocity.

Figure 6.9 shows the results for weak and strong coupling for time-step size ΔT of 2×10^{-4} s. The weak coupling computes too large contact forces and loses contact between the net and the rock. Strong coupling correctly computes the contact, however, the number of coupling iterations is $\sim$10 on average with a maximum of 25. This reflects the complexity of this case.

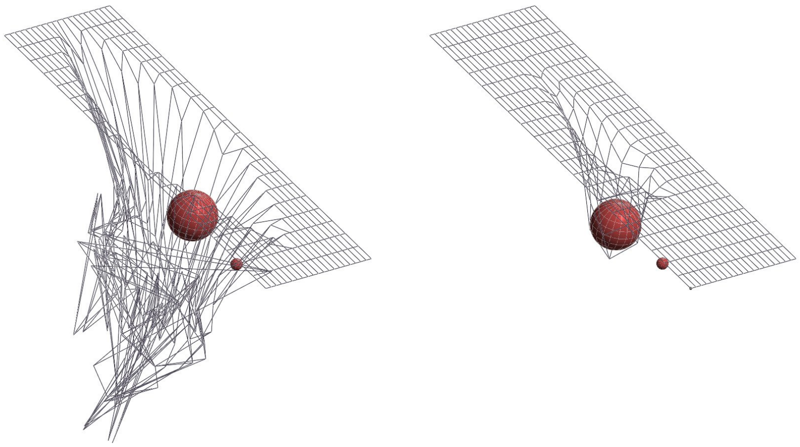

(a) Weak coupling (b) Strong coupling

Figure 6.9: Comparison of weak and strong coupling methods with ΔT of 2×10^{-4} s. Strong coupling is required to correctly resolve the contact between the rock and the cable net.

The displacements of the cable net are shown in Figure 6.10 for different configurations.

6.5 Olympic Tower

The Olympic Tower is part of the Olympic Park in Munich, which was built for the Olympic Summer Games in 1972, see Appendix B. It has a height of 291 m and is used as television tower, viewing platform and also accommodates a rotating restaurant, see Figure 6.11a.

[83] and [82] performed elaborate WSI studies on this structure, for which detailed CSD and CFD models were created using the FEM and Kratos as solver. The models were also validated against real-world measurements.

One goal of numerical simulations in engineering is to model and simulate real-world phenomena, to better understand the underlying problem. Hence,

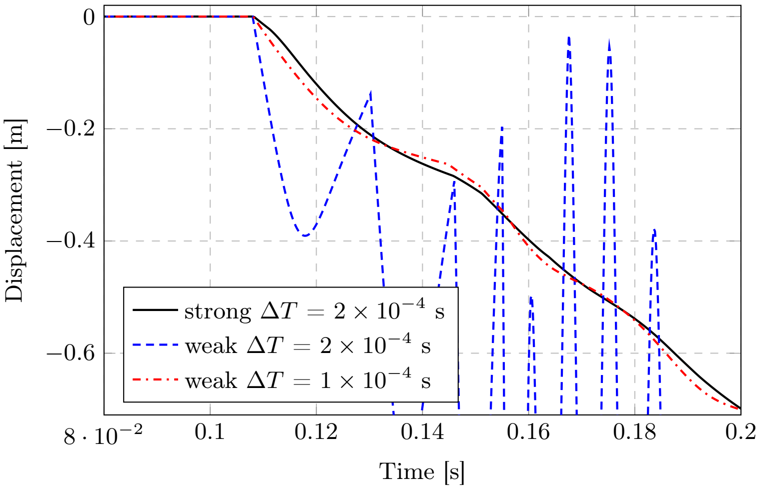


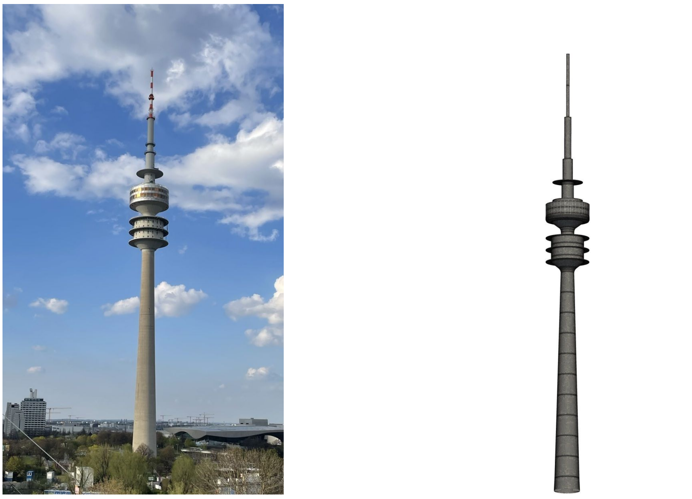
Figure 6.10: Deformation over time, for the center node of the cable net for different coupling methods. The weak coupling exhibits an oscillatory solution with a larger time step.

this work puts an emphasis on using real-world applications for investigations and method development. To satisfy this ambition, the simulation model and setup were taken from [83] to perform studies with different versions/realizations of CoSimulation and compare them. The investigations focus on performing CoSimulation inside a multiphysics framework versus performing CoSimulation using the classical approach via a coupling tool. Various metrics, such as computational requirements, memory consumption, and simulation times, are considered. CoSimIO (see Section 3.7) is used for the data exchange between the solvers.

6.5.1 Reference computational setup

The reference computational setup used by [83] is briefly summarized in the following:

As for the rest of this work, Kratos is used as the solver, both for the CSD and CFD models and the CoSimulation. The coupling is done in a two-way fashion, where both solvers influence each other. An explicit coupling with the Gauss-Seidel communication pattern (see Figure 2.2b) is used, combined with the prediction technique presented by [21]. The displacements computed by the CSD solver are mapped to the CFD solver using a nearest-element mapping technique, see Section 4.1.2. In the other direction, the loads computed by the CFD solver are mapped to the structural solver using the same mapper in a conservative manner, see Section 4.2. The CFD solver considers the structural deformations using an ALE approach, see [59]. Here a structural



(a) Olympic tower (b) Structural FEM model

Figure 6.11: Olympic tower in Munich. Real structure and FEM model.

similarity approach is used for deforming the mesh in the domain and thus preventing the collapse of small elements near the coupling interface.

The tower is built from concrete and steel and consists of inner and outer load-carrying parts. A structural model was created with shell and beam elements, to model the real-world behavior of the tower accurately. Various validation studies were conducted by [83]. A special focus was set on the dynamic behavior of the structure, which is crucial for accurate FSI simulations. The model is meshed with 159,334 nodes and 322,995 elements (see Table 6.2), and is shown in Figure 6.11b. Geometrical nonlinearities are considered by using corotational elements and a Newton-Raphson solution strategy. The Bossak method was used for the time integration, and a time-step size ΔT of 0.02 s was employed. The materials are modeled as isotropic, linear-elastic. Dirichlet boundary conditions are applied to the bottom of the model. This represents the foundation of the real structure.

Entire model			FSI-Interface	
Nodes	Elements	DOFs	Nodes	Elements
159,334	322,995	956,004	128,817	257,220

Table 6.2: Mesh configuration for the structural model. Each node has 6 DOFs, one for each component of displacement and rotation.

The fluid model consists of a numerical wind tunnel with the tower inside, as depicted in Figure 6.12. The terrain surrounding the tower is not taken into consideration. The mesh has 1,411,671 nodes and 7,900,429 tetrahedral elements (see Table 6.3), with several refinement boxes towards the tower to resolve the surrounding flow accurately. The numerical wind is generated using the model from [57] and imposed on the inlet. The wind parameters are determined based on the location of the tower, which is an urban/suburban area. No-slip boundary conditions are applied on the structure and the bottom, slip conditions on the sides and the top, and zero pressure on the outlet. The fractional step solution technique is used, which solves the velocity and the pressure separately, using the LES method, see [12]. The time-step size is 0.02 s, which matches the one used for the structural model. Mesh refinement studies were conducted by [83].

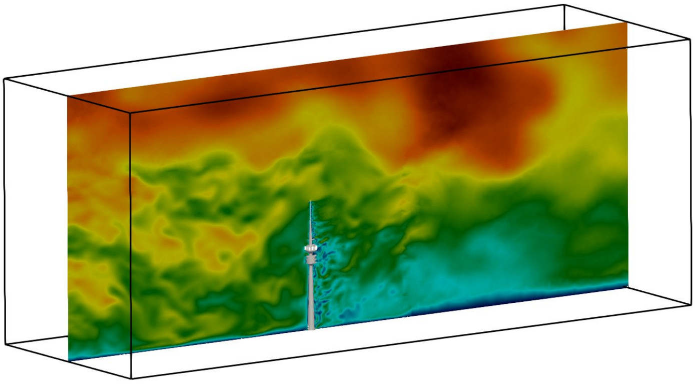


Figure 6.12: Snapshot slice through CFD flow field, with outline of flow domain. The magnitude of the velocity is displayed, which illustrates the fluctuating wind.

Entire model			FSI-Interface	
Nodes	Elements	DOFs	Nodes	Elements
1,411,671	7,900,429	5,646,684	244,163	488,116

Table 6.3: Mesh configuration for the fluid model. Each node has 4 DOFs, three for velocity, and one for pressure.

Numerical simulations involving wind need a long simulation time to satisfy statistical requirements. A typical time is 600 s / 10 min of real-time. Together with a ΔT of 0.02 s, this leads to 30,000 TSs. The large mesh required by both solvers combined with the high number of TSs means that simulations of this type have high computational requirements. Both [83] and the studies performed in this work used the supercomputer described in Appendix A to conduct the simulations. 240 MPI ranks were used by each solver and the CoSimulation. Every component ran in the same memory space within the same Kratos process, thus the total number of ranks was still 240.

6.5.2 Comparing different versions of CoSimulation

One of the main contributions of this work is the comparison of different versions and realizations of CoSimulation, see Section 3.4. The focus of this example is on comparing CoSimulation with a coupling tool (see Section 3.4.2) versus CoSimulation within one framework (see Section 3.4.4). Furthermore, it is explored how different methods of IPC influence the simulation. Lastly, also different methods of parallelization are considered, especially mixing shared (using OpenMP) and distributed (MPI) memory parallelization.

Several simulations with different setups are created to compare the differences. The considered setups are listed in the following, with deviations from the original setup pointed out. They are each focussing on a particular aspect of CoSimulation.

1. Original setup, as described in Section 6.5.1

2. The structural solver runs shared memory parallel (OpenMP, 20 threads) instead of distributed (MPI, 240 ranks). Still, it runs in the same process as the fluid solver and the CoSimulation, meaning that no IPC is required.

3. The structural solver runs shared memory parallel (OpenMP, 20 threads). It runs in a separate process and communicates via **file** with CoSimulation.

4. The structural solver runs shared memory parallel (OpenMP, 20 threads). It runs in a separate process and communicates via **pipe** with CoSimulation.

5. The structural solver runs shared memory parallel (OpenMP, 20 threads). It runs in a separate process and communicates via **TCP socket** with CoSimulation.

6. The structural solver runs shared memory parallel (OpenMP, 20 threads). It runs in a separate process and communicates via **local socket** with CoSimulation.

7. The structural solver runs shared memory parallel (OpenMP, 20 threads). It runs in a separate process and communicates via **MPI** with CoSimulation. This requires the structural solver to be launched with MPI, which implies that it runs on a separate compute node.

8. The structural solver runs shared memory parallel (OpenMP, 20 threads). It runs in a separate process and communicates via **TCP socket** with CoSimulation. Same as for communication via MPI, the structural solver is launched with MPI and runs on a separate compute node.

9. The structural solver runs shared memory parallel (OpenMP, 15 threads). It runs in a separate process **on a separate machine** (see Appendix A) and communicates via **file** with CoSimulation. This is possible by mounting the filesystem of the cluster on the separate machine.
10. Both the structural and the fluid solver as well as CoSimulation run in separate processes, each in MPI on 240 ranks. They communicate via **file** with CoSimulation
11. Both the structural and the fluid solver as well as CoSimulation run in separate processes, each in MPI on 240 ranks. They communicate via **TCP socket** with CoSimulation
12. Both the structural and the fluid solver as well as CoSimulation run in separate processes, each in MPI on 240 ranks. They communicate via **MPI** with CoSimulation

Running the structural solver in shared memory is considered as this is a commonly encountered scenario when coupling existing solvers. Here the processes of the (compute) nodes are pinned so that the solver can use them for shared memory parallelization. More details and explanations on mixing shared and distributed memory parallelization and the pinning of processes can be found in Section 5.3. Also, more methods for IPC are available in shared memory.

Different metrics are used to assess and compare the different setups. Table 6.4 compares general performance metrics of the setups, including the time required for computing one TS, the total memory consumption, and the required compute resources. Mapping is the focus of Table 6.5, here the initialization of the mapper and the timings for mapping between the different computational domains are compared. Lastly, Table 6.6 compares different methods for IPC, thereby extending and complementing the studies conducted in Section 3.8.

Remarks

If the structure is running externally on one rank, then the connecting rank on the CoSimulation side requires almost twice the memory because the entire structural simulation runs on this rank. If the structure is also running distributed, then the memory consumption is very well-balanced between the ranks. This can largely be attributed to the METIS library [43], which is used to partition the mesh among the computing ranks.

The number of threads the structural solver uses in shared memory was determined with scaling studies. On the cluster, the fastest structural solution was be achieved with 20 threads. When using an external machine for the structural solver, only 15 threads were available due to the smaller size of the machine.

The timing measurements are averaged over the ranks, as well over the TSs.

An iterative linear solver is used for the structural solution when running distributed. In shared memory parallel runs, a direct solver is used.

Case	Time TS [s]	Average total memory consumption [GB]	Allocated resources compute cores / nodes[a]
1	20.0	**109.2** (**CFS**)	240 / 5
2	34.63	**115.5** (**CFS**)	240 / 5
3	44.31	**116.0** (105.9 (**CF**) + 10.1 (**S**))	240 / 5
4	43.31	**115.9** (105.8 (**CF**) + 10.1 (**S**))	240 / 5
5	43.32	**116.0** (105.9 (**CF**) + 10.1 (**S**))	240 / 5
6	46.04	**115.9** (105.8 (**CF**) + 10.1 (**S**))	240 / 5
7	32.75	**107.9** (97.8 (**CF**) + 10.1 (**S**))	260 / 6
8	32.69	**108.1** (98.0 (**CF**) + 10.1 (**S**))	260 / 6
9	38.29	**115.5** (105.8 (**CF**) + 9.7 (**S**))	240 / 5 [b]
10	26.96	**191.4** (42.8 (**C**) + 91.9 (**F**) + 56.7 (**S**))	720 / 15
11	7.89	**190.9** (42.7 (**C**) + 91.7 (**F**) + 56.5 (**S**))	720 / 15
12	7.89	**191.5** (42.8 (**C**) + 91.9 (**F**) + 56.8 (**S**))	720 / 15

Table 6.4: Resources required for the different simulation scenarios. The memory consumption is shown for **C**oSimulation, the **F**luid, and the **S**tructural solver.

[a] one compute node has 48 cores
[b] plus the external machine

Case	Initialization		Mapping [s]	
	Time [s]	Memory [GB]	$\mathbf{F} \to \mathbf{S}$ (loads)	$\mathbf{S} \to \mathbf{F}$ (displacements)
1	0.51	3.21	7.638×10^{-4}	8.638×10^{-4}
2	6.76	3.35	2.000×10^{-2}	2.047×10^{-2}
3	14.78	3.55	1.171	1.885
4	17.16	3.51	1.206	1.887
5	16.74	3.56	1.181	1.903
6	16.18	3.56	1.153	1.882
7	7.28	1.89	1.514×10^{-2}	1.620×10^{-2}
8	7.52	1.93	1.532×10^{-2}	1.678×10^{-2}
9	14.43	3.56	1.192	1.379
10	1.13	5.69	1.077×10^{-3}	9.867×10^{-4}
11	1.06	5.66	7.698×10^{-4}	6.992×10^{-4}
12	1.13	5.25	7.531×10^{-4}	5.821×10^{-4}

Table 6.5: Performance information for mapping.

Case	Mesh		Data	
	Fluid	**S**tructure	**F**luid	**S**tructure
3	-	8.478×10^{-1}	-	5.276×10^{-3}
4	-	6.491×10^{-1}	-	4.975×10^{-3}
5	-	6.291×10^{-1}	-	3.780×10^{-3}
6	-	7.115×10^{-1}	-	5.086×10^{-3}
7	-	4.535×10^{-1}	-	3.299×10^{-4}
8	-	4.699×10^{-1}	-	1.997×10^{-3}
9	-	8.189×10^{-1}	-	3.693×10^{-2}
10	1.067	4.221×10^{-1}	1.406	1.434
11	2.097×10^{-2}	1.952×10^{-3}	5.702×10^{-4}	5.269×10^{-4}
12	3.595×10^{-3}	2.026×10^{-3}	7.131×10^{-5}	4.439×10^{-5}

Table 6.6: Timing information for IPC for different setups. Everything is in seconds.

6.5.3 Results interpretation

Several setups to investigate different versions of CoSimulation were introduced in Section 6.5.2. This section provides an insight into the obtained results and observations and puts them in perspective to the overall goal, which is to perform coupled simulations of real-world engineering problems efficiently.

The most important metric for comparing the different setups is the time required for computing one TS. This is important as it dictates the overall simulation time. Simulations with transient effects such as FSI oftentimes require many TSs to be computed, thus faster TSs can make a significant difference.

Solvers running in the same process but employ different parallelization

In the original setup, both solvers and the CoSimulation are using the same compute resources and run in the same process/memory space. The computation of one TS requires 20 s. **Setup 2** has the structural solver using shared instead of distributed parallelization, here it can only use 20 threads (on one compute node) instead of 240 processes (on five compute nodes). A significant runtime increase can be observed, the time per TS increases by ~73%. Furthermore, memory consumption slightly increases, which can be attributed to the linear solver used (iterative in distributed, direct in shared). This gives a glimpse of how vital parallelization strategies are when solving engineering problems. The mapping also becomes more expensive, as now fewer compute resources are available on the structural side. Over 80% of available compute resources are idle in this setup when the structural solver is running and the fluid is waiting. Therefore, both the initialization and the mapping itself get significantly slower. Since also in this setup, all components run in the same memory space, there is no IPC needed.

Solvers running externally on same compute cores

In **setups 3 to 6**, the structural solver also uses shared memory parallelization but now runs in a different process. The fluid solver and the CoSimulation still run in the same process. This showcases how using a dedicated coupling tool would handle the combination of solvers that employ different methods of parallelization. Tools running in different memory spaces require the exchange of data between them using IPC. Different methods as introduced in Section 2.2.2 and evaluated in Section 3.8 are assessed to compare their performance and applicability in large-scale engineering applications. When using files, pipes, TCP sockets, or local sockets for the data exchange, it is possible to launch both codes on the same compute nodes, and no additional resources compared to the original setup are required. The data exchange happens on one rank as the structural solver is not running distributed. One downside of such setups is the further increased computing time per TS, which stems from the process pinning (see Section 5.3) that is required to make use of shared memory parallelization for the structural solver. While the memory consumption does not increase compared to setup 2, the solution becomes slower. This is partly because of the data exchange but mainly because the mapper becomes less efficient due to the parallelization. The data exchange via IPC for the presented problem is negligible since it is several orders of magnitudes faster than the solution time of the solvers. It is particularly interesting to compare the file-based data exchange to the network (TCP socket) or kernel (pipe and local-socket) based methods. It is only marginally slower, both for mesh and data exchange. Hence, it can be a reasonable choice even for larger problems when running shared memory parallel and the data exchange happens only on one core.

Solvers running externally on different compute cores
MPI is oftentimes regarded as one of the fastest ways to exchange data between tools. **Setup 7** therefore uses it as the method for the data exchange between CoSimulation and structural solver. The rest of the setup is the same as setups 3 to 6. A major difference is that instead of running on the same compute node, the structural solver now runs on a different one but again shared memory parallel. This is necessary due to how the job scheduler of the cluster is working, namely that each MPI job must be running on its distinct cores. Therefore, one additional compute node has to be allocated, which increases the overall cost of the simulation. The advantage is that now the pinning of processes can be removed and no longer interferes with the performance, which leads to a significant increase in computational efficiency. One TS is now ∼25% faster compared to setups 3-6, which is a significant difference. Also, the memory consumption is slightly reduced. Mapping is now comparable to setup 2 in terms of speed and even requires less memory. Overall this setup is even faster than setup 2, where all components run in the same memory space. This highlights that the data exchange via IPC is not the most performance-critical component for (surface) coupled simulations, given that the exchange of data happens on one rank.

Setup 8 is done in the same way as setup 7, in that the structural solver runs shared memory parallel on a separate compute node. Instead of using MPI it uses sockets for the data exchange. The purpose of this setup is to investigate if using MPI performs better than other methods of IPC. It is furthermore important to consider that in setups such as this, when the data exchange happens not within one but between different compute nodes, only a subset of methods can be used. The data can be exchanged either via files on a shared filesystem or through the network (e.g. with TCP-sockets or MPI). The results show that setups 7 and 8 behave very similarly and have similar performance metrics aside from measuring fluctuations. This means that the choice of IPC method is not very crucial, but using a separate compute node for the solvers is. Regarding timing only the data exchange, communication via MPI still outperforms the socket-based communication by ∼one order of magnitude. It is not noticeable in the overall performance of the coupled simulation as the solving time of the solvers is much more prominent in comparison.

Solvers running on different hardware
In some cases, it is not possible to deploy all solvers involved in the CoSimulation on the same hardware. Reasons can be that they do not support the

same OS, or that a solver (e.g. CFD) needs to run on a HPC system due to their computational requirements, while another one cannot be deployed there for licensing reasons. To enable such cases, the data exchange needs to happen not only between different processes/memory spaces but also different hardware and OSs. Such scenarios can be considered as extensions to communicating between solvers that run on different compute nodes, as shown in setups 7 and 8. Hence, only a subset of IPC methods can be used. File-based communication requires a shared filesystem, which can be achieved by mounting a remote drive locally. Another option is to communicate via the network, which requires access to the local network or even the internet to be very generic. For security reasons, network access might be disabled or restricted on some HPC systems. Resorting to file-based data exchange could be the only possibility to establish communication and enable CoSimulation. **Setup 9** shows such a scenario. Same as in setups 3-8, the structural solver runs separately and uses shared memory parallelization. The difference is now that the structural solver additionally runs on a different machine, a workstation with the remote (cluster) drive mounted locally. The data exchange happens via files in this drive. The measurements show that CoSimulation can be performed efficiently, even if the involved tools run on different hardware, given a fast enough network speed to synchronize the remote drive. A direct comparison with setup 3 shows even a reduction in time per timestep. This can, as before, be attributed to the deployment on different hardware, which lifts the process pinning restrictions. Notably, the data exchange is one order of magnitude slower compared to setup 3, where both solvers run on the same machine. This is expected since the remote drive introduces some latency due to synchronization.

All tools run in MPI and on separate compute cores
Finally, **setups 10-12** investigate the behavior of CoSimulation when both solvers and the CoSimulation run distributed and do not share resources. This is a modern coupling approach, which is only possible if all tools involved support distributed parallelization. Significant differences are observable, starting with the time per TS. Running on different compute cores outperforms the original setup by a factor of 2.5 when using sockets or MPI for the data exchange between the solvers. This is an enormous advantage over integrated CoSimulation. Mainly two reasons are the cause, one being the context switch between the different solvers and the coupling tool. Several times per TS, different parts of memory (e.g. the system matrix from the solvers or the mapping matrix from the coupling) must be loaded into cache. With modern computers and HPC systems being rather limited by memory speed than CPU speed, this can have a significant impact. If the tools run in their separate compute cores and memory spaces, these context switches occur much less often. Also, the solvers run sequential within one rank in the integrated CoSimulation. In contrast, if they are running separately, they can perform different tasks like writing output in parallel, which further decreases the time per TS. Considering that a Gauss-Seidel communication pattern was used, which has data dependencies, the difference is most probably even larger for a Jacobi communication pattern (see Figure 2.2a) which does not have these

dependencies and hence can make better use of the non-shared resources.

On the other hand, memory consumption increases by ∼75%, which is a significant overhead. It results mainly from duplicating the data structures of the solvers, particularly interface-mesh and -data in the coupling tool. The coupling tool needs access to this data to perform coupling-related operations such as mapping or relaxation. If the coupling tool is not running in the same process as the solvers and hence does not have direct access to their data structures, duplication is required. To put the impact of this increased memory consumption into perspective, the available memory on the cluster is considered: Each compute node has 96 GB of memory, of which 90 GB are available. With integrated CoSimulation, five compute nodes are used, which means that, in total, 450 GB of memory is available. In total, ∼24% of the available memory is used. When the tools run on separate compute nodes, then each of them uses five nodes, which increases the total available memory to 1350 GB. Here then ∼14% of the available memory is used. At first sight, this looks like an improvement, but one must keep in mind that it is not always possible to use more compute nodes, e.g. if the available HPC system is smaller than the one used in this work. Therefore, if the same number of nodes as in the original setup would have to be used, then 42% of the available memory would be used, which is considerably more. The coupling for FSI happens from surface to surface in this example, which is one dimension smaller than the domain used for the field solution. In other applications (such as Conjugate Heat Transfer (CHT)) however, it might be required to perform a volume-to-volume coupling. Here the interface is of the same dimensionality as the solution domain, which can increase memory consumption significantly. Lastly, it is important to note that Kratos has been optimized toward a small memory footprint. This means that if other tools are used in the CoSimulation which are not as optimized regarding memory consumption, then the available memory might not be enough.

Duplicating the data structure also introduces the additional challenge that the data in the coupling tool needs to be kept up to date through synchronization via IPC. CoSimulation within the same tool neither needs synchronization nor data exchange, as the coupling tool has direct access to the data structures. Lastly, the computing resources required to run the tools individually are exactly three times more compared to integrated CoSimulation since the solvers do not share resources but still need the same amount of cores.

Mapping shows only little differences, except for a longer initialization time which is because the (duplicated) meshes in the coupling tool require some initialization. This mesh initialization is done by the solvers in the integrated CoSimulation. Since the exchange of meshes happens only once, this can be considered negligible.

Lastly, comparing the different IPC methods yields interesting findings. The network-based methods are several orders of magnitude faster than the solvers and can hence be neglected time per TS. MPI outperforms sockets by roughly one order of magnitude, which confirms the results obtained in Section 3.8. File-based data exchange however has a significant impact on the overall performance of the simulation. The time per TS is almost three times higher compared to using the network-based methods. This is also consistent with the findings from Section 3.8. It confirms that using files for the data exchange for distributed tools is not suitable and highly depends on the current workload of the filesystem. Stressing the filesystem affects not only the data exchange but also the writing of result files, which leads to times per TS that are even more than the increase from the data exchange. A final remark is made that pipes and local sockets cannot be used when communication across compute nodes is required, as they can only be used within one CPU.

6.5.4 Conclusions

The main conclusions and findings from the studies with the Olympic Tower coupled WSI simulations are briefly summarized in the following.
A crucial practical observation is that mixing different methods of parallelization leads to performance degradation of both methods but might still lead to an overall faster solution.
Performing CoSimulation within one framework is a very efficient solution, particularly regarding memory consumption and required compute resources. No IPC is required, which increases the stability of the coupling. Additionally, the deployment, handling, and debugging are considerably simpler when using the framework since only one software is used, see Section 5.2. This can make a large difference in successfully conducting coupled simulations, particularly on HPC systems.
However, if the computing resources are available, having tools run on separate compute cores can be faster at the cost of larger memory consumption and compute resource requirements. Network-based data exchange needs to be used, as file-based data exchange is not suitable for distributed coupling.
The methods and implementations for mapping as presented in Chapter 4 are highly efficient and are negligible in terms of required time and memory consumption compared to the solvers.
Evaluating different methods for exchanging data between tools via IPC has shown that network TCP socket-based communication is a very efficient and versatile method. It combines high speed with robustness and applicability, both for coupling on one rank and in distributed setups.
Pipes and local sockets can in theory be faster than network-based sockets, as they directly use the kernel and do not use the network. However, these studies have shown that the differences to network sockets are negligible for coupled simulations.
Lastly, the file-based data exchange is only slightly less efficient compared with sockets, but only when coupling on one rank. It can be especially useful when communicating with tools that run on different hardware, as only a shared filesystem is required to exchange data.

6.6 Coupling to external codes - Full Wind Turbine

This example presents high-fidelity aeroelastic FSI simulations of a wind turbine. They were conducted as part of the research project *Wind Energy Science and Engineering in Complex Terrain* (WINSENT), in a collaboration between the Chair of Structural Analysis at the Technical University of Munich and the Institute of Aerodynamics and Gas Dynamics at the University of Stuttgart.

The CFD model was prepared by Giorgia Guma, the CSD models by the author of this work, and the coupling between the solvers was developed together, based on the work of [70]. A summary of previous works, including [33], [34] and [71] is presented. The focus is set to complement and highlight the developments and findings of this work.

The WINSENT project investigates the operation of wind turbines in complex terrain. Throughout the project, two turbines were built in the Swabian Alps near Stuttgart in southern Germany, see Figures 6.13 and 6.14. At the same time, a numerical toolchain was created to simulate the behavior of the turbines. For this, low- and high-fidelity approaches are used. The high-fidelity FSI simulations were conducted by coupling the CFD solver FLOWer [64] to the CSD solver of Kratos (*StructuralMechanicsApplication*) through the *CoSimulationApplication*.

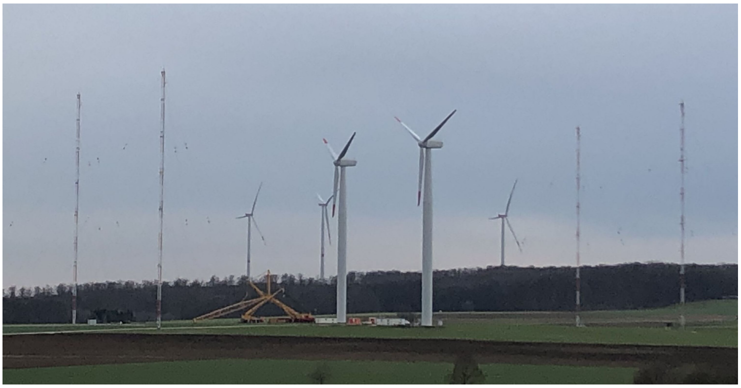


Figure 6.13: Test site with the two turbines and four met masts. Source: ZSW `www.zsw-bw.de`

The turbine has a rotor diameter of ∼50 m, a blade length of ∼25 m, a hub height of ∼70 m hub height and a rated power of 750 KW. Two structural models were created, one based on beam elements and one based on shell elements, to investigate the differences between the two modeling approaches. Furthermore, the differences between flat and complex terrain are assessed by creating and comparing two fluid models.

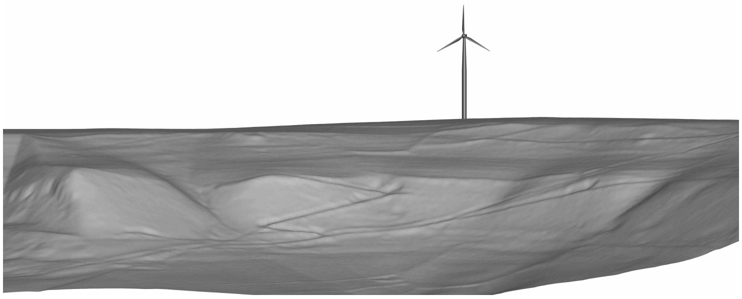


Figure 6.14: Turbine in complex terrain, as represented in the CFD model. It is located behind a slope edge, which affects the turbulence of the wind. These conditions influence the power output as well as the lifespan.

6.6.1 Structural Model

Two structural models were built, one with beam and one with shell elements. The two models are shown in Figure 6.15. Models created with beam elements are commonly used to simulate the turbines' structural behavior in industrial practice. While the beam theory is very applicable for the simulation of slender structures such as wind turbine blades or towers, it does not account for deformations of the cross-section. Modern wind turbines with blade lengths of over 100 m are very flexible, therefore an accurate simulation of the structural behavior should include those effects. Additionally, the deformations of the cross-section alter the aerodynamic properties of the blade, leading to a two-way coupled problem. In contrast, shell models directly model the geometry and can represent the deformation of the cross-section. This comes at the cost of higher computational requirements and more preprocessing effort.

Both models consider shear deformations, the beam model by employing the Timoshenko theory, and the shell model by means of the Reissner-Mindlin theory. Rayleigh damping (see [3] and equation 2.8) is used with a damping ratio of 0.03.

The main components of the turbine are modeled flexible, to achieve a highly accurate structural response in the coupled simulations. This includes the blades and tower for the beam model, and additionally the nacelle for the shell model. The hub is modeled rigid due to its very high stiffness compared to the other components. The connections between the component are also considered rigid, and modeled with stiff beam elements. Figure 6.16 shows the details of the connection between the components.

The rotation of the rotor is considered in both models, which is required to correctly include gravitational and centrifugal effects. This rotation however poses significant challenges for the solver. Adequate nonlinear elements and solution techniques were employed to overcome these numerical instabilities.

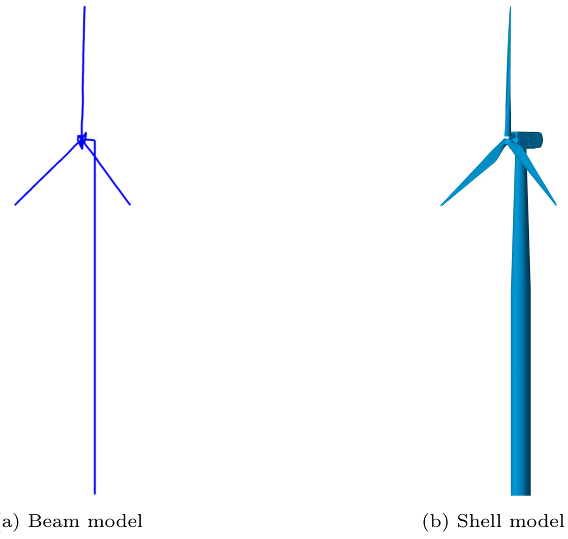

(a) Beam model (b) Shell model

Figure 6.15: Structural models of the turbine. The beam model represents the geometry by lines, whereas the shell model accurately takes all geometrical features into account.

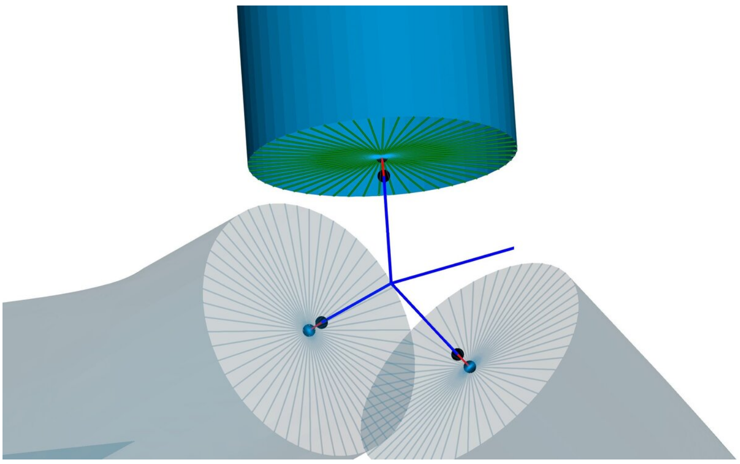


Figure 6.16: Detail of how the components of the turbine are connected, here shown for the connection between blades and hub. The rigid connections (red) connect the endpoints of the components (black). Additionally, for the shell model, a spiderweb-looking support structure (green) was introduced to bridge the different dimensionalities.

Great effort was spent validating the models since errors in the individual problems would propagate to the coupled simulation, as explained in Section 5.1.1. The manufacturer of the turbines provided the CAD files of the geometry as well as the material specifications of the composite blades and further information, which was used to create the models. Afterward, they were compared against the reference by means of different metrics such as the eigenfrequencies, see Table 6.7. A very good agreement could be achieved, especially when considering the complexity of the models.

Eigenfrequency [Hz]	Manufacturer	Beam	Shell
1st flap (blade)	1.3	1.34	1.38
1st edge (blade)	2.2	2.25	2.18
2nd flap (blade)	3.8	3.91	3.99
2nd edge (blade)	6.8	6.98	6.53
1st bending (tower)	0.5	0.52	0.55
2nd bending (tower)	3.4	3.63	3.64

Table 6.7: First eigenfrequencies of the blade and tower for both models, compared to the reference values provided by the manufacturer.

The validation of the models also included extensive mesh studies to eliminate any mesh-dependent behavior. The final meshes are listed in table 6.8. It is evident from the sizes of the meshes that the shell model has a much larger computational effort than the beam model.

Component	Beam	Shell
Blade	139 N / 139 E | flexible	15913 N / 30892 E | flexible
Hub	5 N / 4 E | rigid	5 N / 4 E | rigid
Nacelle	3 N / 2 E | rigid	13524 N / 26888 E | flexible
Tower	37 N / 48 E | flexible	8906 N / 10141 E | flexible
Turbine	462 N / 478 E	70174 N / 129715 E

Table 6.8: Mesh configuration of individual components (***N***odes, ***E***lements). Furthermore, it is specified which components are deformable/flexible.

Figure 6.17 provides insight into some details of the mesh of the shell model. Because of the fine mesh and therefore large computing requirements, different parallelization methods were used for the shell model, to reduce the time to solution. Shared and distributed memory parallelization were employed and compared.

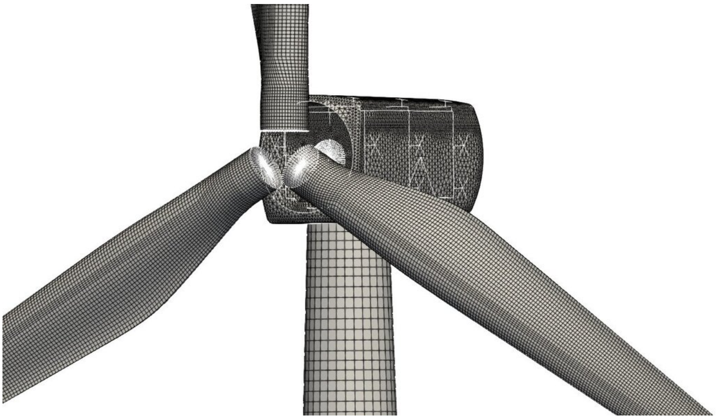


Figure 6.17: Detail of the mesh of the shell model. Each component is meshed according to its accuracy requirements and geometrical features. Stiff beam elements are used to connect the components, see Figure 6.16.

6.6.2 Fluid Model

The fluid flow is simulated using the compressible CFD URANS/DES solver FLOWer [64] to accurately resolve the highly turbulent flow conditions of the test site. The details of the model can be found in [33] and [34]. The chimera technique [4] is used to handle the rigid body motion of the rotor, as well as embedding the model of the turbine in flat and complex terrain.

The wind conditions of the test site are applied as boundary conditions in the numerical model to ensure consistency between the real world and the simulation results. Their application is presented in [52].

The influence of the complex terrain was evaluated by creating and comparing two models. Figure 6.18 shows the flow field around the turbine in flat terrain, Figure 6.19 shows the same, but in complex terrain. The slope edge and its influence on the flow field are clearly visible.

FLOWer uses MPI parallelization with over 1,000 cores due to the very high computational effort of full-scale wind turbine simulations.

6.6.3 Coupling

The coupling between FLOWer and Kratos was realized by using the developments presented in this work, see Chapter 3. Partitioned two-way CoSimulation with explicit and implicit strategies and Dirichlet-Neumann coupling was employed. FLOWer computes the wind loads on the structure based on the displacements from Kratos. Kratos, on the other hand, uses the loads from FLOWer as input to compute the structural response of the turbine. [70] developed a coupling between FLOWer and Carat++ by using

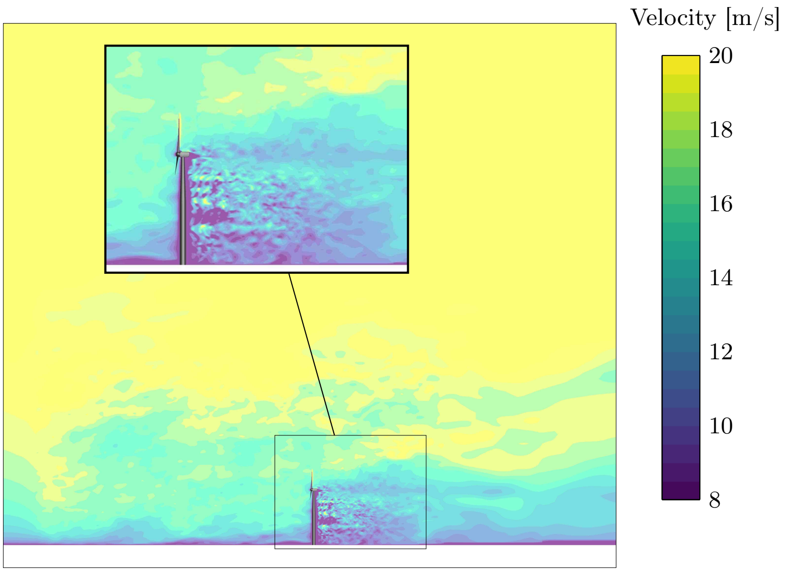


Figure 6.18: Vertical slice through the computed flow field in flat terrain.

the coupling tool *EMPIRE* [80], which served as the basis for the presented coupling.

FLOWer is an external solver from the point of view of Kratos. It uses its own address/memory space, therefore it is necessary to exchange the coupling data via IPC. The detached interface technique (see Section 3.6.1) is used, with files as means to communicate the data. The coupling between the solvers happens on one rank and thus requires gathering and scattering of data. Figure 6.20 shows the schematics of the coupling between the solver, including the treatment of the distributed simulations.

The coupling sequence for such detailed and complex cases requires special attention. Many of the best practices as presented in Chapter 5, were employed to achieve stable, accurate, and robust coupled simulations. In particular, the starting phase of the coupling is prone to numerical instabilities. Each solver was first run for some time until they reached a stable solution. Afterward, the coupling was gradually started. Figure 6.21 shows the procedure that was used.

Due to the large computational effort, the simulations were conducted on the supercomputer SuperMUC-NG (see Appendix A).

6.6.4 Results

The wind energy is converted into a rotating motion by the blades of the turbine, making them the most crucial component of the turbine. This

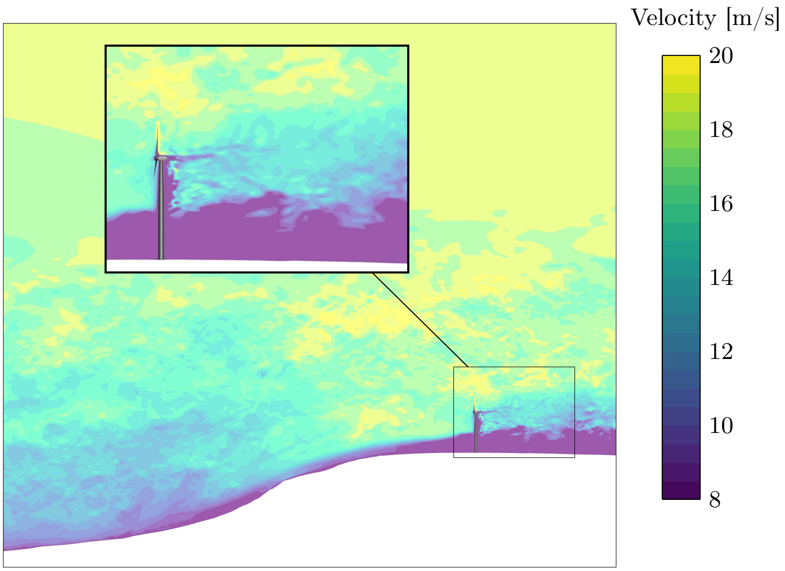


Figure 6.19: Vertical slice through the computed flow field in complex terrain. The slope edge behind which the turbine is located can be seen.

makes them the main focus of comparing the beam and the shell model in flat and complex terrain. The long and slender shape of the blade can lead to considerable deformations under loading. This modified external geometry alters the aerodynamic properties, and strong motions can reduce the lifetime due to fatigue. Figure 6.22 shows the deformation of the turbine in complex terrain during operation. The corresponding time history of the tip deformations is presented in Figure 6.23, starting with the application of the aerodynamic loads (after 3.14 revolutions, see Figure 6.21).

Considerable deformations can be observed for both models and in both terrains. While the overall course of the deformations over several revolutions is similar for the two structural models, the shell model exhibits higher peaks in all but one configuration. Its higher fidelity allows it to capture more physical effects, particularly in highly turbulent flow conditions in complex terrain. This observation is also supported by the statistical analysis of the deformations as shown in Table 6.9. Here, the shell model shows a higher variance of the deformations.

Changes in the cross-section can only be represented with the shell model, as the beam theory assumes it to be rigid. This type of deformation leads to fatigue in the blades and changes their aerodynamic properties. Therefore, they need to be considered for highly accurate simulations that predict their operational behavior. Figure 6.24 shows the deformations of the cross-section, obtained with the shell model. The maximum deformation was 3 mm, which

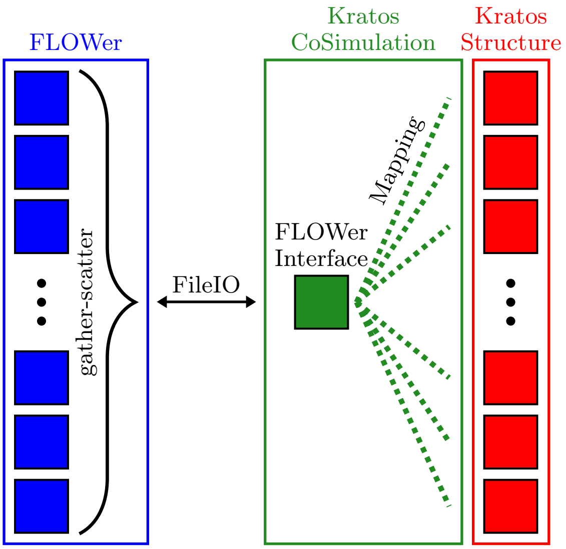


Figure 6.20: Coupling schematics when using a distributed memory treatment of the structural model. Each square refers to one MPI rank. On the FLOWer side, the data is gathered on one rank before sending it via files to Kratos. Kratos maps the data then to the computing ranks. In the other direction, Kratos first maps the data to the FLOWer interface (which exists on one rank) before sending it to FLOWer. After receiving the data, FLOWer scatters it to its computing ranks. The detached interface technique (see Section 3.6.1) is used.

might seem small compared to the other dimensions of the blade. Nevertheless, it is not negligible, even for this short and stiff blade. It is also important to consider that modern wind turbines have much longer and more flexible blades, where these deformations are expected to be much more significant. Additionally, shell models are required for an accurate and efficient design of the internal support structures such as webs and spar caps.

The differences between implicit and explicit coupling were found to be negligible. This is expected given the relatively short and stiff blades. Therefore, the same setup as shown in Section 6.5 was used, an explicit coupling strategy combined the prediction technique from [21].

Finally, the computational effort is assessed and compared. One time step of the CFD solver took ∼100 s to compute. It also used the most computational resources, 4,224 cores for the simulations in flat and 7,460 in complex terrain. On the structural side, significant differences can be observed, see Table 6.10. The time to compute one TS with the beam model is 0.1 s, which is negligible compared to the fluid solver. For the shell model,

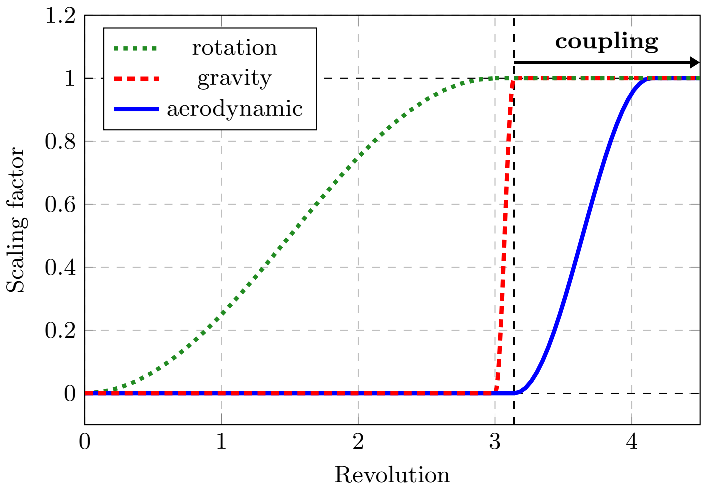
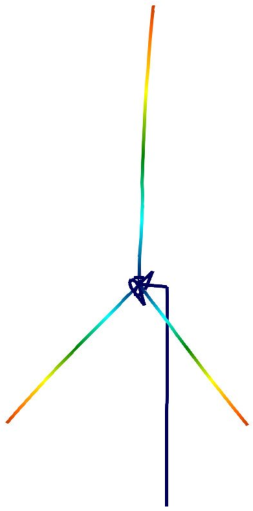
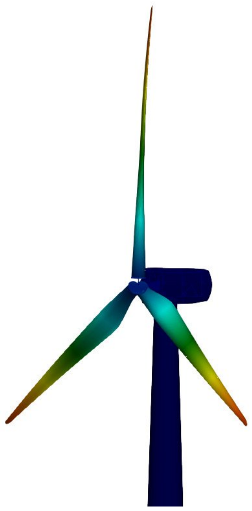


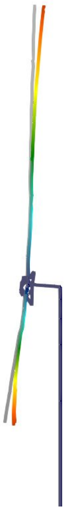
Figure 6.21: Loads on the structural model and their gradual application (see Section 5.1.7): First, the rotation is ramped up over three revolutions, then the gravitational loads are applied over 50 TSs, equivalent to 0.14 revolutions. The coupling is started after the structural loads (rotation and gravity) are applied, i.e., after 3.14 revolutions. Then the aerodynamic wind loads are applied over one revolution. Finally, after 4.14 revolutions, all loads are fully active. This procedure is necessary to ensure numerical stability in the coupled aeroelastic simulation.

	flat			complex		
	Beam	Shell	diff (%)	Beam	Shell	diff (%)
mean (flap)	0.621	0.564	9.7 (b)	0.523	0.535	2.4 (s)
mean (edge)	-0.023	-0.016	34.9 (b)	-0.014	0.015	7.3 (s)
std-dev (flap)	0.132	0.137	4.1 (s)	0.149	0.151	1.5 (s)
std-dev (edge)	0.093	0.105	12.8 (s)	0.096	0.107	10.5 (s)
RMS (flap)	1.016	0.975	4.1 (b)	0.970	0.990	2.0 (s)
RMS (edge)	0.255	0.300	16.2 (s)	0.276	0.336	19.9 (s)

Table 6.9: Statistics of blade tip deformations (in meters), for beam and shell model in flat and complex terrain, flapwise and edgewise direction. Revolutions 3-9 are used to skip the ramp-up phase. The identifiers behind the differences indicate which model has the larger value, **b**eam, or **s**hell.


(a) Beam model deformation.


(b) Shell model deformation.


(c) Beam model deformation, compared to undeformed configuration.

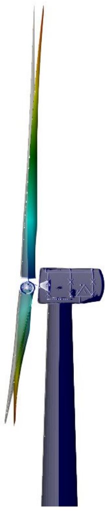
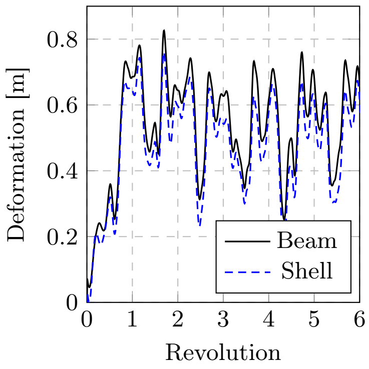
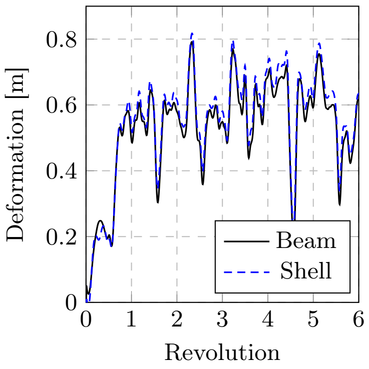

(d) Shell model deformation, compared to undeformed configuration.

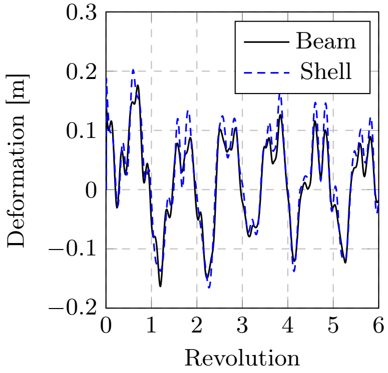
Figure 6.22: Deformations of the turbine in operation in complex terrain, for beam and shell model. The corresponding time history of the blade tip deformations is shown in Figure 6.23.



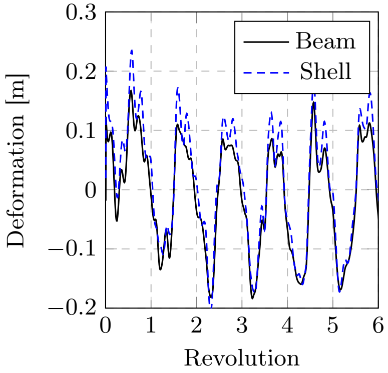
(a) flat terrain, flapwise direction



(b) complex terrain, flapwise direction



(c) flat terrain, edgewise direction



(d) complex terrain, edgewise direction

Figure 6.23: Blade tip deformations for different models and terrains (relative to the hub), starting with the application of the aerodynamic loads.

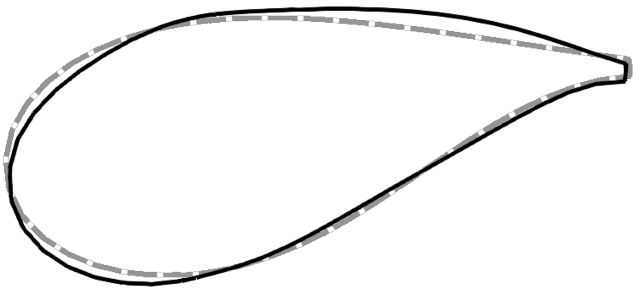


Figure 6.24: Exemplary cross-section deformation of a blade (qualitative), 25 times magnified. The internal support structure (webs and spar caps) is not shown. The dashed line represents the undeformed, and the solid line the deformed blade. Considerable changes in the profile shape due to deformations can be observed at the leading edge as well as towards the trailing edge.

however, the parallelization has a major impact on the overall solution time. Using shared memory parallelization with OpenMP and 20 threads takes ten times longer compared to using distributed memory with MPI and 144 processes. This also has a large impact on the total solution time of the coupling. Employing MPI for the structural model increases the total solution time by only 10%, whereas when OpenMP is used, it is twice as high. The time spent on coupling functionalities such as mapping and data exchange is negligible compared to the solution times of the solver. Consistent with the investigations presented in Section 3.8, these results show that using file-based data exchange can be efficiently used also for large-scale applications, given that the exchange happens only on one rank. Overall, these results are in agreement with the findings presented in Section 6.5.3.

Setup	Structure	FSI
Beam model (serial)	0.1	100.1
Shell Model (OpenMP, 20 threads)	100	201
Shell Model (MPI, 144 processes)	10	111

Table 6.10: Solving times (in seconds) for one time step for different structural models. FSI includes the time for the fluid and the structural solver as well as the coupling.

Concluding, the benefits and necessity of distributed algorithms are clearly shown with this example. It highlights once more, how real-world engineering applications can benefit from modern computing capabilities.

Acknowledgment

The author gratefully acknowledges the financial support of the project WIN-SENT (code number 0324129): the Federal Ministry for Economic Affairs and Climate Action (BMWK) and the Ministry of the Environment, Climate Protection and the Energy Sector Baden-Württemberg under the funding number L75 16012.

6.7 Olympic Stadium Roof

For my grandparents Rosemarie and Arnold Gillich, whose stories from the Games in 1972 inspired this example. You are dearly missed.

The goal of this example is a detailed modeling and study of the structural response of the Olympic Stadium roof in Munich (see Figures 6.25 and 6.26) under wind loading. This structure was constructed for the 1972 Olympic summer games, see Appendix B for more information. Detailed geometric modeling of the roof is performed, including terrain and the stadium as the analysis-relevant surroundings. Different sources such as the original construction plans as well as on-site measurements are used to construct suitable geometrical models.

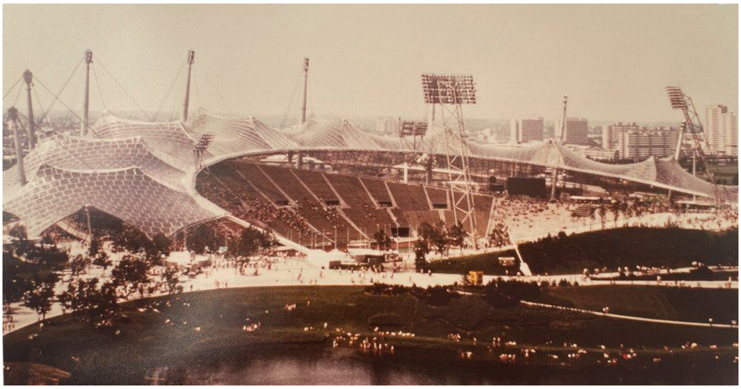


Figure 6.25: Olympic Stadium during the Summer Olympic Games in 1972. Source: A. Gillich

The roof is a lightweight structure built from cable nets, covered with acrylic plates and metallic rubber connectors. It consists of nine large patches and covers half of the spectator stands. The entire structure hangs from large support pylons and is constrained by cables. Only the load-bearing structure is considered, consisting mainly of cable-net, pylons and support cables. The cable net structure is simplified with plates in membrane action (later referred to as membrane elements) for the structural analysis, due to its

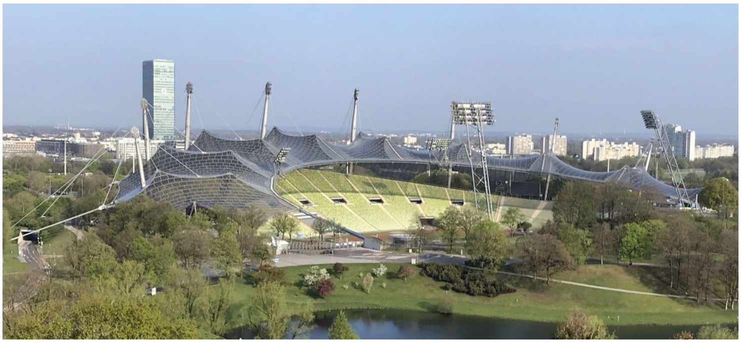


Figure 6.26: Olympic Stadium, half a century after the Olympic Games of 1972. Source: P. Bucher

complexity. The prestresses in the cable net are converted for the membranes, and improved by use of an optimization algorithm.

The wind loading analysis is conducted with the dominant wind direction. A power-law inlet is used to model the wind speed varying over height. The gust wind profile was used, without taking the natural turbulence into account. The surroundings and topographic details that influence the flow behavior are taken into account. Combined, they enable a realistic and complex load case.

The structural response under wind loading is simulated by combining the structural model with the wind loading, in an FSI analysis. This loading leads to considerable deformations on the lightweight roof structure. Strong coupling in combination with convergence acceleration was employed. Due to the high level of detail of the model, the mesh sizes and thus the computational requirements are large. Therefore, the simulations were conducted on a supercomputer, leveraging HPC to keep the solution times within practicable bounds.

This case was prepared as a collaborative work by multiple people at the institute. Previous works include [10] and [63]. The contributions of the author are the setting up of coupling and mapping as preparation for the FSI, testing the fluid solver, setting up the structural solver settings, scaling studies, and conducting the simulations on the supercomputer. Máté Péntek created the geometrical models and meshes for both the fluid and the structural solver, as well as some initial setup of the structural properties and the simplified wind setup. The optimization procedure for the prestresses as well as on-site measurements were done together. Klaus Sautter worked on the particularities of the FEM formulation related to the structural model, insights on and decisions for the equivalent membrane and the prestresses, as well as various tests of formfinding. Kai-Uwe Bletzinger is recognized as a consultant, Roland Wüchner for preliminary discussions. Furthermore, the previous works of colleagues Andreas Winterstein, Benedikt Philipp and Andreas Apostolatos, including some advised student works, are acknowledged.

6.7.1 Recreation of the geometry

The design and construction of the Olympic Park and its buildings were done long before the emergence of CAD software. Therefore, no computational geometry was available, it needed to be recreated from several sources, together with the terrain of the stadium. These included the original construction plans, on-site measurements, geodetic data, preliminary studies conducted at the institute, and online available material from 3D Warehouse[1] and Google Maps[2]. The latter one was used within the workflow developed by [40] to generate 3D models.

The basis for all efforts was the special research report [51] about the construction of the roof. The architects and engineers went into great detail to explain relevant information. They provide sketches and plans, dimensions, and prestresses as well as corner and anchor points of the structure. However, this information proved to be insufficient for modeling the structure in a CAD environment, which is a requirement for creating analysis-suitable models. Therefore, additional information was used from different sources. While the main corner points of the structure could be confirmed and validated from several sources, such as [41] and [56], modeling the curved shapes proved a challenge. For this, 3D data was consulted such as from Google Maps and 3D Warehouse. The patches were modeled as continuous surfaces, without the cable net details since no geometrical information for them was available. The support pylons and cables of the structure were considered based on the original construction plans as well as the corner points and edges of the patches. Several iterations of improvement were necessary, always taking into account and balancing all available information to achieve the most accurate result.

The reconstruction of the geometry introduced an additional challenge for the structural analysis. Whereas in the original design of the structure, the prestresses were known, and the geometry was a result of them, in this case, it was the other way round. The target (reconstructed) geometry was known, and a suitable prestress state with which the structure is in equilibrium needed to be determined. This procedure is explained in detail in Section 6.7.3. Due to the strong interaction of form and force in such as lightweight structure, also the prestresses and their state of equilibrium was taken into account for the geometrical model.

The final results of these tasks were geometries for the computational models, as shown in Figure 6.27. The small eye-shaped patches between the large membrane patches are neglected for simplicity. Furthermore, only the structural model considers the pylons and cables.

6.7.2 CAD-Geometry to structural FE-model

The reconstructed geometry was used as input for the structural model of the roof. The goal was to model the main behavior under load with as much detail as possible. This includes in particular the patches hanging

[1] `https://3dwarehouse.sketchup.com/`
[2] `https://www.google.com/maps`

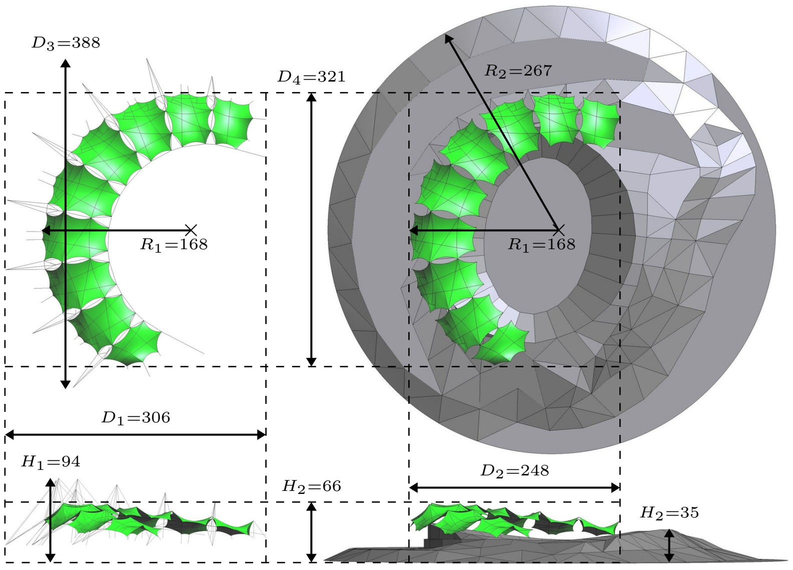


Figure 6.27: Constructed computational models. Structural model on the left side, the fluid model with terrain on the right side. The small eye-shaped patches between the large membrane patches are neglected. Dimensions are in meters.

free from the pylons, without any additional constraints. Only the load-bearing structure was considered. As mentioned previously, no detailed geometrical information was available for the cable net. It was thus modeled with membrane elements, see Figure 6.28. While this introduces additional challenges when it comes to the selection of suitable material properties, it reduces the effort for the geometrical modeling significantly. The patches could be modeled as continuous surfaces in CAD instead of the cable net. Additionally, the membrane elements can directly be used in the coupled simulation for mapping. Modeling the cable net would have required non-load-bearing elements for the surface mapping with the fluid interface.

The patches were meshed with triangular surface elements, whereas the cables and pylons are modeled with one-dimensional truss elements. The created mesh can be seen in Figure 6.29, and details are shown in Figure 6.42. The mesh configuration is listed in Table 6.11. The mesh sizes are a result of experience, the objectives of this work, as well as mesh refinement studies.

The material properties are taken from [51]. For the cables and the pylons, the report lists in detail all the necessary information. Since the cable net is modeled with membrane elements, a suitable conversion of prestresses was used. They were then refined with the help of an optimization procedure, as presented in Section 6.7.3.

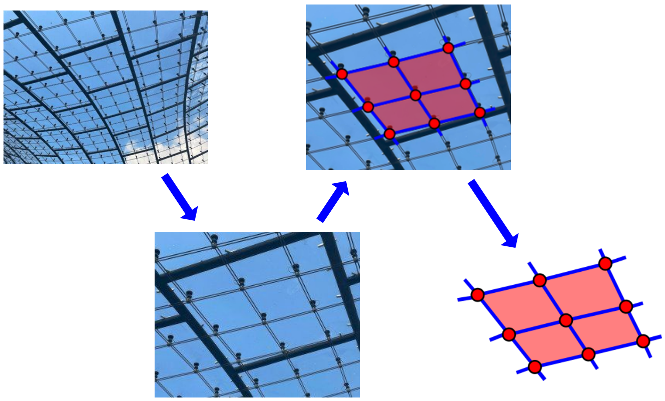


Figure 6.28: Using an equivalent membrane for the actual roof. The original cable net as well as the acrylic plates covering it can be seen. The thick black lines are the rubber connectors between the plates.

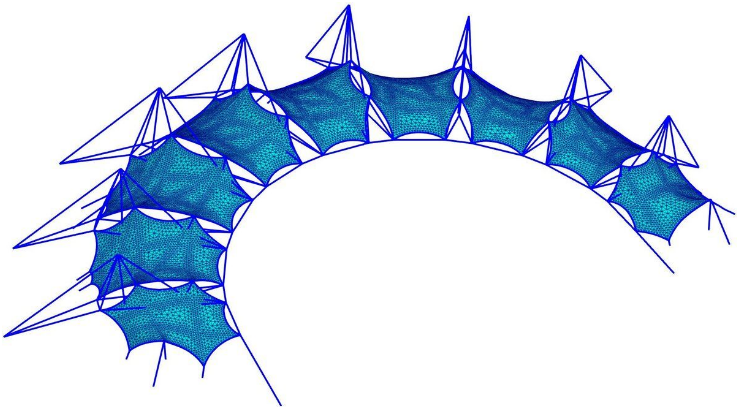


Figure 6.29: Structural mesh for the Olympic stadium roof.

Entire model			FSI-Interface	
Nodes	Elements	DOFs	Nodes	Elements
13,987	28,028	41,961	13,930	26,148

Table 6.11: Mesh configuration for the structural model. Each node has 3 DOFs, one for each component of displacement.

Structural damping was considered in the form of Rayleigh damping [3]. 3% damping (damping ratio of 0.03) was used, which was chosen because of the rubber connectors and other connecting elements in the structure. The Rayleigh damping coefficients α and β (see equation 2.8) are calculated from the first two eigenfrequencies of the structure.

6.7.3 Prestress determination

The final geometry of the roof and its surroundings was presented in Section 6.7.1. Because of the substitute modeling of the cable net with membrane elements, their prestress state to achieve the target geometry was unknown. An estimation could be derived from the information available for the cable net in [51]. The prestress field was assumed as isotropic and homogeneous for each patch. In order to reduce the discrepancies between geometry and prestresses stemming from modeling errors and approximations, an optimization algorithm was used to improve the prestresses. The goal was to reach an equilibrium of the target geometry, under prestress and self-weight. This procedure can be seen as an inversion of the formfinding process (see [6]): Instead of finding a geometry given a prestress state, the prestress state leading to a given geometry needed to be found. This is expressed with equation 6.3, where the displacements $\mathbf{d}$ are to be minimized, depending on the prestresses σ_{ps}.

$$\min \mathbf{d}(\sigma_{ps}) \tag{6.3}$$

Each optimization iteration starts with a static, nonlinear structural analysis under self-weight and with prestress. The resulting deformations need to be minimized since the given geometry represents the target one. The L_2 norm of the displacements is then passed to an optimization algorithm, with which it updates the prestresses. These are then applied to the original geometry, and another analysis is conducted. This iterative procedure leads to an improved prestress state, which can then be used in further simulations. Algorithm 8 shows the individual steps of the entire optimization. The workflow was implemented in Python. Kratos was used for the structural solution, combined with the *COBLYA* algorithm[3] for the optimization.

Due to the large number of design variables as well as their expected changes, the optimization was conducted in two stages. In the first stage only the prestresses in the nine patches were optimized, resulting in nine design variables. No limits or bounds were applied, making this stage an unconstrained optimization. Since only a rough estimate was available as the initial guess for the prestresses in the patches, large changes were expected. The results of this optimization stage are shown in Figure 6.30. Compared to the initial value of 4.9×10^8 N/m^2, the prestresses in the individual patches are reduced by up to 50%.

The improved prestresses for the nine membrane patches are used as input for the second stage of the optimization. In addition to the patch prestresses, the cable prestresses are now considered, yielding a total of 224 design variables.

[3] `https://docs.scipy.org/doc/scipy/reference/optimize.minimize-cobyla.html`

Algorithm 8: Optimization procedure for determining the prestress state, for a given geometry.

```
1 guess prestress
2 prepare/initialize solver
3 while optimization not converged do
4     reset solver (reinitialize if not possible)
5     update model with new prestresses
6     solve for displacements
7     compute $L_2$ norm of displacements
8     optimizer computes new prestresses
9 end
```

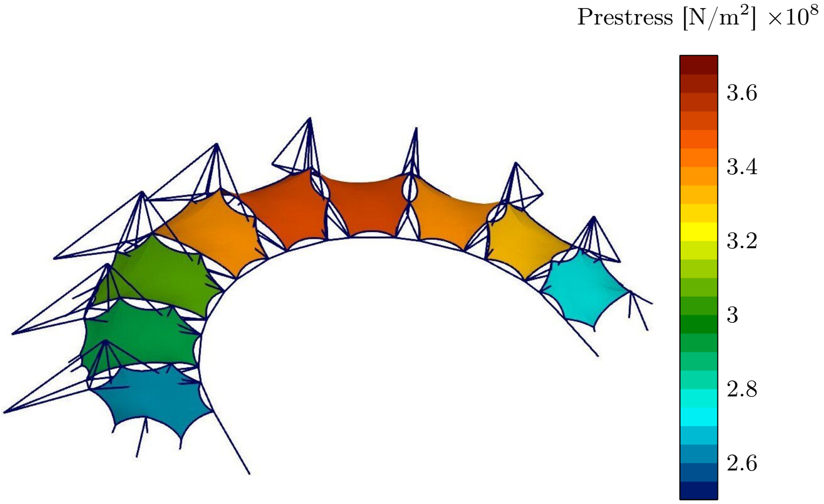


Figure 6.30: Updated prestress in the membrane patches, after the first stage of the optimization. The initial value was 4.9×10^8 N/m^2.

This is done to account for tolerances and modeling inaccuracies. As the cable prestresses are known from [51], only small deviations are expected to occur. Therefore, bounds were used to limit the deviation from the given values to a maximum of $\pm 10\%$. Figure 6.31 shows how much the prestresses deviate from the initial values. The membrane prestresses undergo almost no change anymore, indicating an overall converged state.

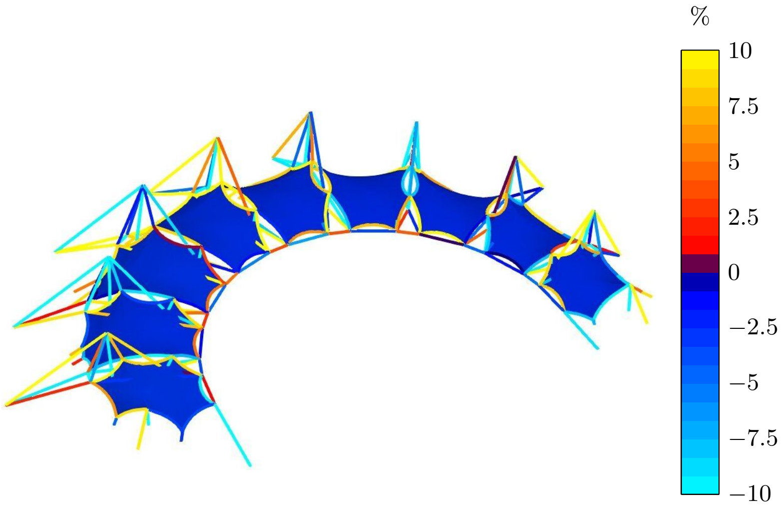


Figure 6.31: Deviations in the prestresses, from the initial values (after the first optimization stage).

Finally, after both stages of optimizations, the resulting deformations are shown in Figure 6.32. It can be seen that in most places only small deformations occur, compared to the overall dimensions of the roof (which are shown in Figure 6.27). One can derive from this, that the presented procedure for determining suitable prestresses for the entire structure succeeded. The largest deformations are in areas where the least input for the geometry was known upfront, namely the two outermost patches.

6.7.4 Wind loading analysis

The Olympic Stadium roof is a wide-spanning, lightweight structure. Its large characteristic patches make it susceptible to wind loading, therefore wind can be considered a crucial load case. This loading is simulated by means of a CFD analysis, using as input the same geometry as presented in Section 6.7.1. The roof can undergo significant deformations, which means that these can alter the fluid flow and hence need to be taken into consideration within a two-way coupled FSI simulation.

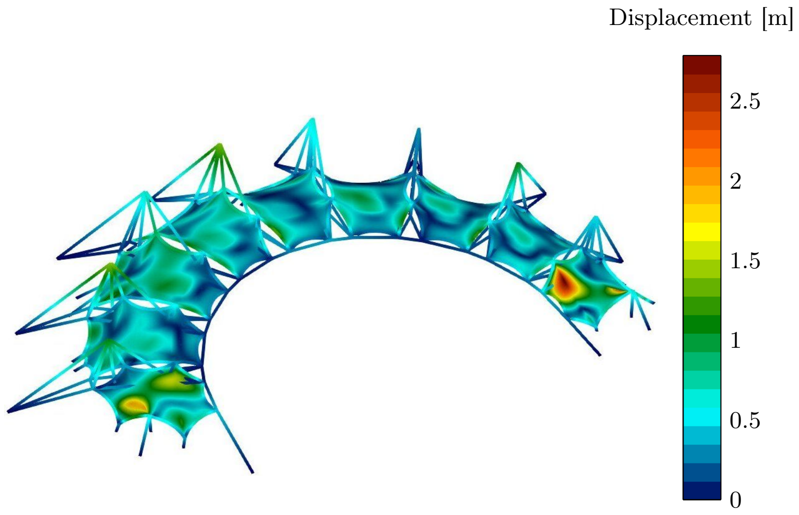


Figure 6.32: Resulting deformations due to prestresses. Shows the displacements from original to final geometry.

The terrain surrounding the roof influences the flow and was therefore considered. Only the membrane patches of the roof are modeled, as they have by far the largest influence on the loading. The pylons and columns were neglected due to their comparatively small dimensions. Furthermore, their consideration would have required a much finer resolution of the mesh, leading to increased computational effort. The full computational domain together with roof and terrain is pictured in Figure 6.33.

The boundary conditions are set to best represent reality. A simplified wind model is used for the inflow, and slip boundary conditions are applied on the sides the top and the bottom of the domain. A fixed pressure is considered at the outlet. The surface of the roof was modeled as no-slip. The flow equations are solved with a monolithic approach, using the LES method, see [12].

The computational mesh is visualized in Figure 6.34. A body-fitted mesh is utilized. Three refinement boxes are used to improve the resolution closer to the roof. The element sizes for different parts of the domain are listed in Table 6.12. They are chosen based on experience and mesh-refinement studies. The complicated and detailed geometry and terrain pose a challenge for meshing. The close proximity of the roof to the terrain below further dictated and restricted the element sizes. In order to adequately resolve the flow, a rather small element size is needed to be used. This was complemented by the requirements of the mesh moving strategy employed for the FSI analysis, as it also needs a fine mesh resolution for propagating the movement of the structure to the fluid domain. A coarse mesh would restrict the structural movement

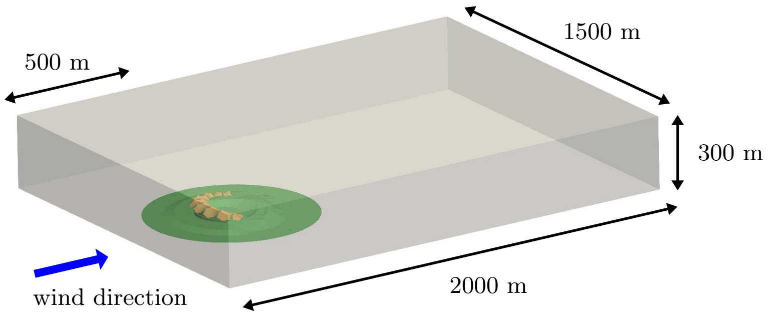


Figure 6.33: Fluid domain considered for the wind loading analysis. The roof as well as the surrounding terrain can be seen.

and could cause convergence problems during the coupled simulation. The final mesh size is listed in Table 6.13.

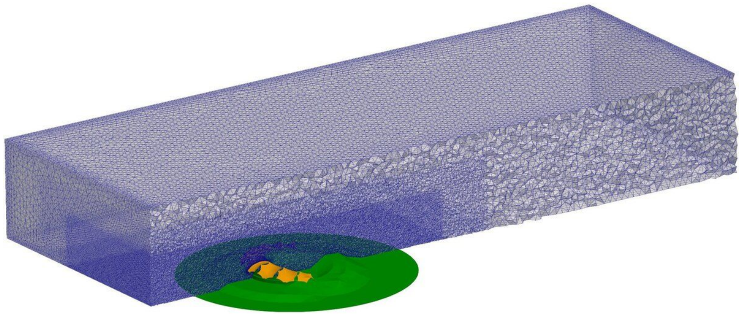


Figure 6.34: Cut through the fluid mesh of the Olympic stadium roof. The refinement boxes as well as the terrain can be seen.

As previously mentioned and shown in Figure 6.27, the eye-shaped connection patches are not modeled. They might have a non-negligible influence on the flow since they close the gaps between the large patches. Their consideration would therefore further increase the accuracy of the model, due to the improved representation of the real roof structure. This would however also lead to a larger mesh size and thus more computational requirements.

It has to be noted, that the reconstructed geometry in CAD represents the membrane patches as smooth surfaces. Once meshed numerically, these surfaces are built up from triangular elements. This means that the surfaces are no longer perfectly smooth, as due to their curvature each element has a slightly different orientation. The edges between the elements are kinks on

Mesh	Element size [m]
Patches	0.5
Terrain	3
Inner Box	5
Middle Box	10
Outer Box	20

Table 6.12: Element sizes for fluid mesh. The refinement boxes can be seen in Figure 6.34.

Entire model			FSI-Interface	
Nodes	Elements	DOFs	Nodes	Elements
3,442,468	19,632,138	13,769,872	259,541	519,110

Table 6.13: Mesh configuration for the fluid model. Each node has 4 DOFs, three for velocity, and one for pressure.

the surface. While they are typically very small, they can still influence the flow pattern and can thus be considered as surface roughness. On the other hand, also the real structure has a certain degree of surface roughness, in particular, because of the rubber connectors between the membrane patches as shown in Figure 6.28. Therefore, the numerical approach should be able to balance the different considerations when it comes to the roughness of the surface.

Modeling wind

The inflow velocity was modeled by taking into account the wind conditions in Munich. Typically, the dominant wind direction is in the 3rd quadrant. Data from Meteoblue[4] as well as previous works that focus on the wind conditions around the Olympic Tower (see [83] and [82]) suggest West-Southwest as the dominant wind direction. 250° was therefore used in the simulations, as is illustrated in Figure 6.33.

A power law profile using the mean wind profile which varies over the height was used, see equation 6.4. To limit the scope and complexity of the following studies, no additional fluctuations in space or time are considered, and the naturally occurring turbulence is not taken into account. The value of 25 m/s represents the basic wind speed in Germany for wind zone II and corresponds to a height of 10 m. It is a conservative assumption, as it overestimates the magnitude of the wind speed. The assumed distribution over height thus approximates an equivalent gust wind profile and is aligned

[4] `https://www.meteoblue.com`

with the conditions described in [51]. The power law exponent of 0.26 is chosen according to DIN EN 1991-1-4. The value was interpolated because the terrain category is in a transition zone between zones III and IV. At the elevation of the patches (roughly 60 m, see Figure 6.27), the wind speed is approximately 40 m/s.

$$v(z) = 25\frac{m}{s}\left(\frac{z}{10m}\right)^{0.26} \tag{6.4}$$

6.7.5 Scaling studies

Scaling studies were conducted for both models/solvers, in order to find the optimal computational setup. The objective was the fastest solution time.

6.7.5.1 Fluid

The scaling behavior of the solution time with different numbers of MPI ranks is shown in Figure 6.35. The strong scaling is presented in Figure 6.36, computed with equation 2.12. Due to the size of the problem, the time for computing a TS without parallelization is not available. Hence, the basis was the solving time with 240 cores, making the speedup appear larger.

The size of this model requires distributed memory, shared memory parallelization was not considered, which is confirmed by the choice of number of cores. The numbers of ranks are multiples of 48, see Appendix A. It can be seen that the smallest solving time per TS is reached with 2,160 cores. This time is composed of the solving time and other times such as writing output. As expected, the solving time takes the largest part of the time per TS. It decreases with an increasing number of cores until no more speedup can be observed with over 2,000 ranks. On the other hand, the other time increases consistently with larger numbers of cores as for example the postprocessing is done as one file per rank. This means, that more and more files need to be written with an increasing number of ranks, which increases the total amount of time spent on writing output. The filesystem of the cluster used is not suitable for writing many small files, see Appendix A.

With 1,440 ranks, a sudden drop in solving times can be observed. This can be related to superlinear speedup [65], where the model fits into the cache due to the local size. Still, with 2,160 the solving times decrease further, which is why this amount of cores is chosen. Note that the objective is to achieve the fastest solution time. Otherwise also using 1,440 ranks would be a good solution as it is only 20% slower in terms of solution time per TS, but uses 2/3 of the ranks, which results in overall less allocation of ranks.

Given the total number of nodes (3,442,468) and elements (19,632,138), this results in ~1,600 nodes and ~9,100 elements per rank.

6.7.5.2 Structure

The computational requirements of the structural model are lower than the ones of the fluid. Hence, both shared memory parallelization (using OpenMP) and distributed parallelization (using MPI) were tested and compared. For

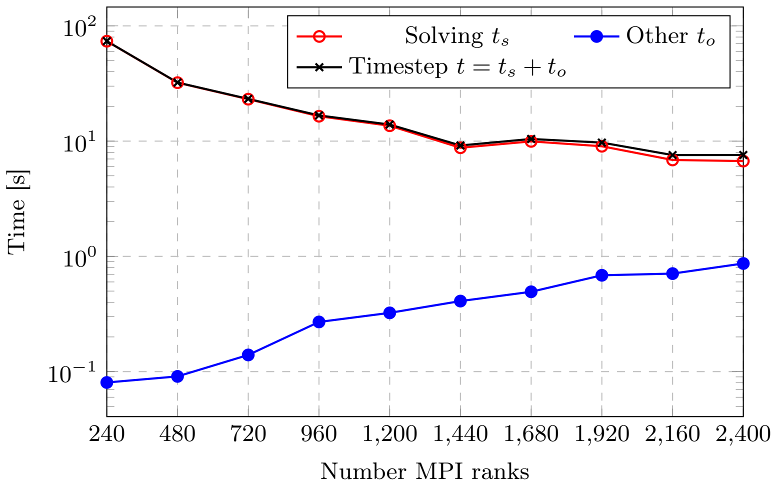


Figure 6.35: Scaling study for Olympic roof fluid model with different number of MPI ranks. The time per TS t is the sum of the solving time t_s and the time needed for other tasks t_o (such as writing output).

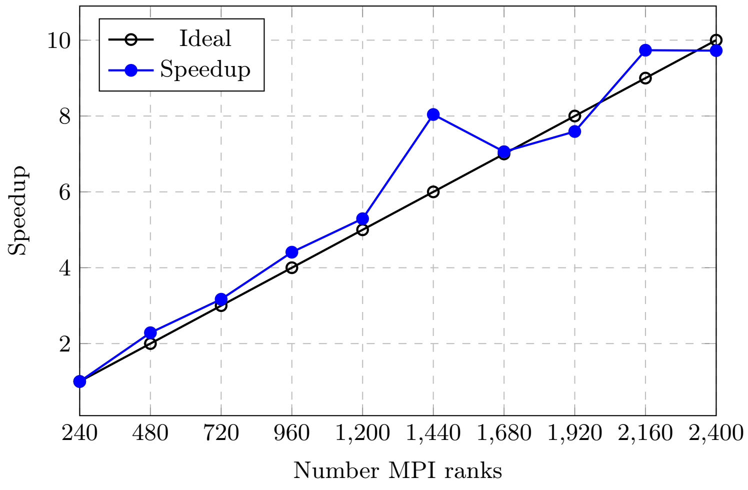


Figure 6.36: Strong scaling study for Olympic roof fluid model with different number of MPI ranks. Due to the size of the problem, the basis for computing the speedup is 240 cores. Hence, the speedup appears higher.

shared memory parallelization, a maximum of 48 threads was chosen as one compute node has 48 physical cores. A maximum of 336 cores was used in the distributed case, which is much smaller than the number of cores used for the fluid model (2,160).

As expected, and consistent with the fluid, the solving time is by far the largest contributor to the total time required for the computation of one TS. In both versions, the solving time was over 90% of the total time.

When comparing the results of shared (Figure 6.37) and distributed (Figure 6.38) memory parallelization, it can be seen that the latter outperforms the first by an order of magnitude and was hence chosen for the structural model too. The smallest time per TS was achieved with 288 cores.

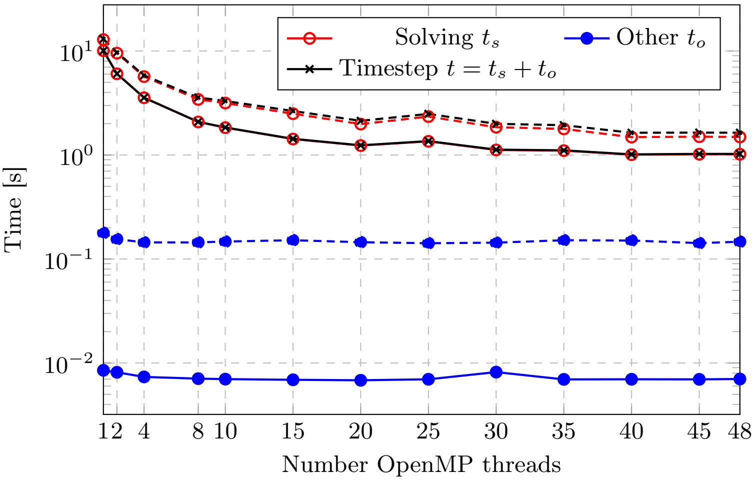


Figure 6.37: Scaling study for Olympic roof structural model with a different number of OpenMP threads. The solid lines are when the model is running standalone, the dashed lines are when the model is part of a FSI simulation.

The comparison of the speedup with shared (Figure 6.39) and distributed (Figure 6.40) memory parallelization reveals the maximum speedup to be ∼10 respectively ∼100.

Given the total number of nodes (13,987) and elements (28,028), this results in ∼49 nodes and ∼97 elements per rank. This is significantly smaller than for the fluid model, which is because of the higher computational effort per element.

For the shared memory parallel computation, some interesting observations can be made: A significant slowdown of ∼50% could be observed when running the model as part of a FSI simulation (where the fluid was using distributed memory parallelization) compared to running the model standalone. This is attributed to the process pinning which is needed for mixing different methods

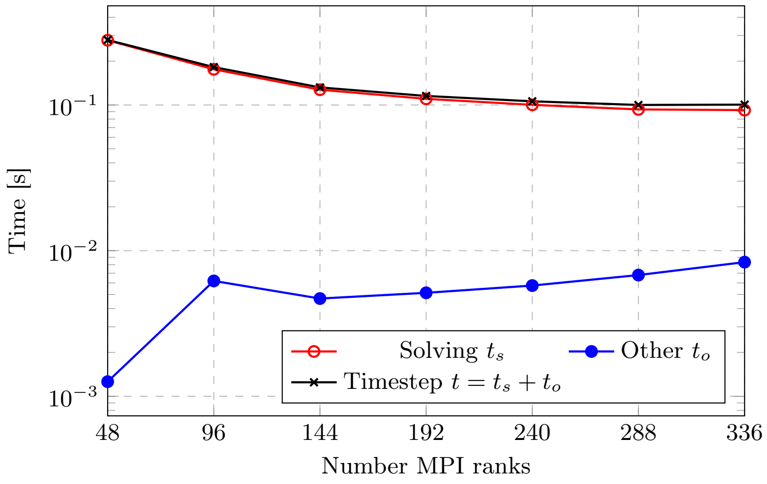


Figure 6.38: Scaling study for Olympic roof structural model with a different number of MPI ranks.

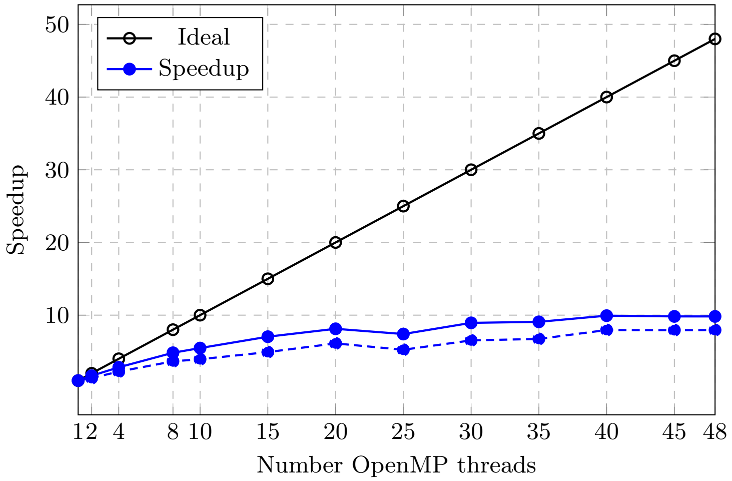


Figure 6.39: Strong scaling study for Olympic roof structural model with a different number of OpenMP threads. The solid lines are when the model is running standalone, the dashed lines are when the model is part of a FSI simulation.

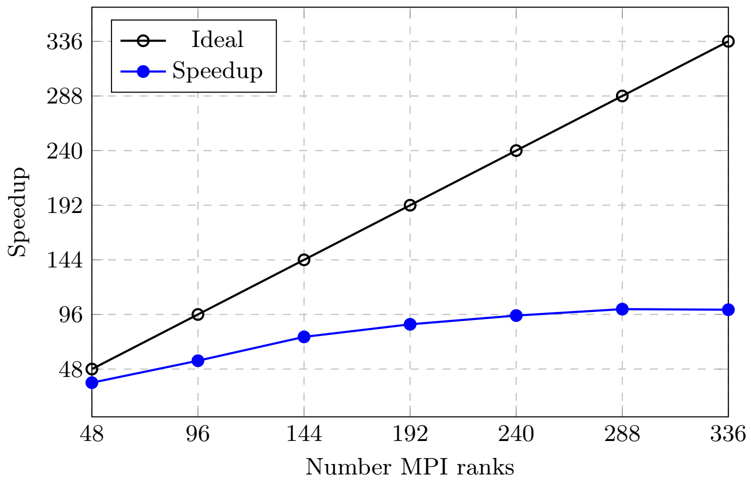


Figure 6.40: Strong scaling study for Olympic roof structural model with a different number of MPI ranks.

of parallel computing, as explained in Section 5.3. The minimal time per TS with the shared memory parallel version could be achieved with 40 threads. Using distributed memory parallelization for the structural solver did not yield a slowdown for the solvers. This is an advantage of consistently using one method of parallelization for all solvers.

6.7.6 Computational setup

The coupled FSI simulations were conducted using the multiphysics framework Kratos, see Section 3.1. Both solvers and the coupling components were part of the same software, as presented in Section 3.5. Therefore, no data exchange via IPC (with e.g. *CoSimIO*) was necessary. The fine resolution in particular on the fluid side leads to a high computational effort. The supercomputer SuperMUC-NG (see Appendix A) was used for running the simulations. Both solvers employed distributed memory parallelization with MPI. In total, 2,160 computing cores/MPI processes (45 compute nodes with 48 cores each) were used. CoSimulation and the fluid solver used all cores, within the *MPI_COMM_WORLD* MPI-Communicator. The structural solver used a subset of 288 (6 compute nodes) cores, within its own MPI-Communicator. The details of handling a different number of processes for different solvers are explained in Section 3.3. The optimal number of cores was determined with scaling studies, as presented in Section 6.7.5. Both solvers used iterative linear solvers, each with different and adapted input parameters due to the characteristic properties of the system matrices.

The total simulation time was 220 s, out of which the first 100 s were without coupling, then 20 s of gradual startup of the coupling, and finally 100 s of full coupling. The time-step size ΔT of both solvers was 0.01 s, to achieve a sufficient resolution of relevant physical phenomena and a stable coupling. Output on the interface was written every 0.25 s (i.e. every 25 steps), in the entire fluid domain every 0.5 s (i.e. every 50 steps).

The coupling was set up following the procedures and recommendations described in Chapter 5. The startup phase is visualized in Figure 6.41. First, both solvers ran for 100 s without coupling/interaction to reach a converged solution, based on which the coupled simulation works more robustly and stable. The coupling was then started by slowly ramping up the displacements over 10 s, using a smooth ramping function as introduced in Section 5.1.7. This allowed the mesh solver to gradually apply the motion, which avoids convergence problems for the fluid solver. Following was the gradual application of the loads on the structure, again over 10 s with the same ramping function.

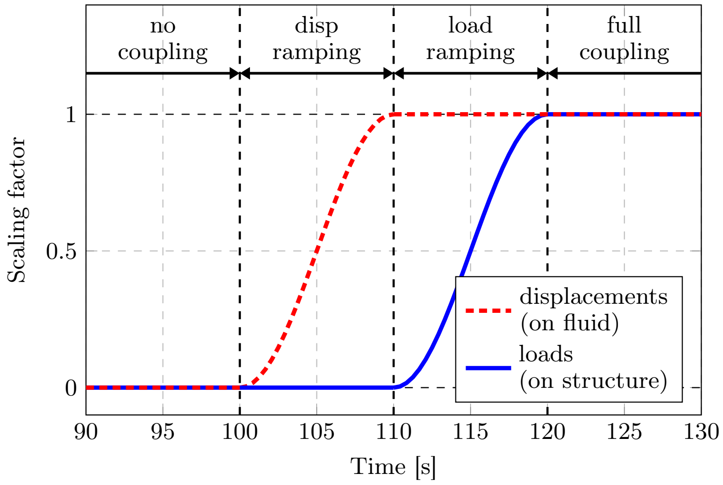


Figure 6.41: Procedure for establishing the coupling. First, the solvers are run for 100 s without interaction in order to achieve a stable solution of the individual fields. This helps to avoid coupling instabilities. Then the structural displacements are gradually applied to the fluid solution. Afterward, the loads are gradually applied to the structure.

A strongly coupled Gauss-Seidel coupling strategy (see Figure 2.2b) was used due to the strong interaction of the fields. The Aitken convergence acceleration technique (see [50]) helped to reduce the number of coupling iterations to only two per timestep, after the initial phase of the coupling with up to ten coupling iterations. The displacements on the fluid side were relaxed. The convergence check was done by comparing the difference of the displacements in the structural solver between the coupling iterations.

Convergence of the coupling was achieved as soon as the L_2 norm of this difference reached a certain threshold.

As seen in Figure 6.42, the meshes on the fluid-structure interface are not matching, and therefore the quantities needed to be mapped. The loads and displacements were mapped with a nearest-element mapper, see Section 4.1.2. The displacements as a distributed field quantity were mapped consistently. The loads however needed to be mapped conservatively as they are computed in the form of point loads (concentrated nodal quantities, and not field quantities such as the tractions in this case). The mapping was done with the initial configuration, as the deformations during the individual startup of the solvers could lead to a wrong mapper setup.

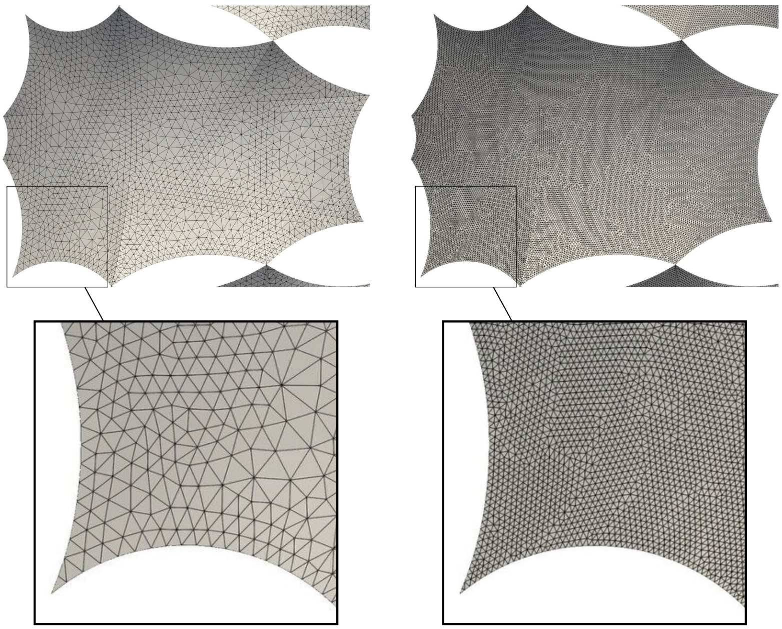


Figure 6.42: Meshes on the patches of the roof. The fluid mesh (right side) is 5 times finer than the structural mesh (left side). Therefore, mapping is required for transferring field quantities from one mesh to the other.

6.7.7 Results

The coupled simulations were conducted successfully with the setup explained in the previous sections. The results will be addressed in the following while referring to the relevant parts of this work.

Figure 6.43 gives an impression of the flow around and over the roof and the surrounding terrain. The wind reaches over 50 m/s above the roof, causing strong fluctuations.

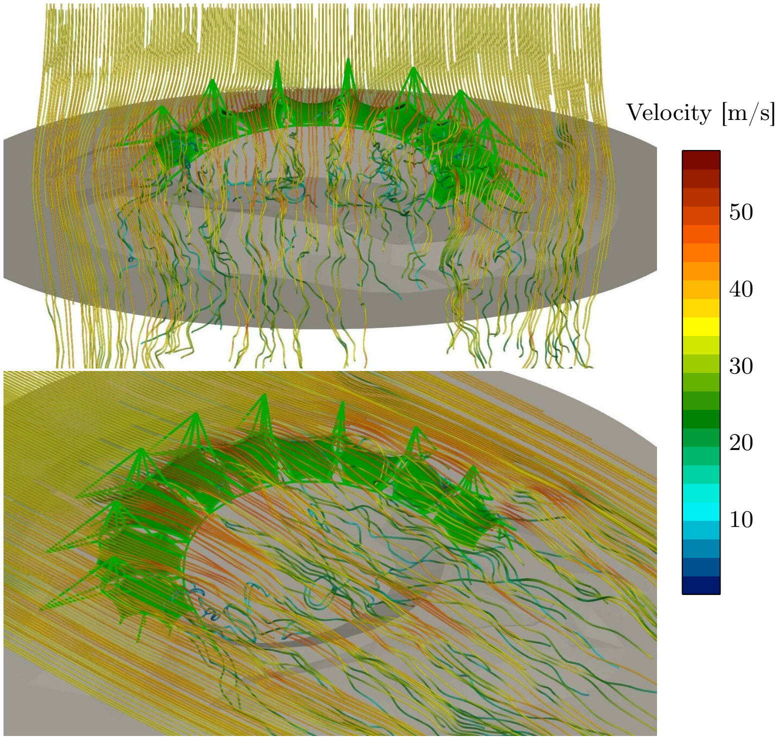


Figure 6.43: Streamlines depicting the flow over the roof. The velocity on the patches is zero due to the no-slip boundary conditions, only the streamlines are colored.

Figure 6.44 shows the maximum displacement that occurred during the simulation. Both the structural model and the interface of the fluid model show an equal distribution of the displacements, which is the expected mapping behavior. The maximum value of ∼5 m is feasible, given the large dimensions of the roof and the high wind speed. It occurs in the middle of one of the largest patches, which also has the most exposure to wind because of its orientation. Figure 6.45 shows a slice of the patch with the maximum deformation.

The loads on the roof resulting from the wind flow are shown in Figure 6.46 for the fluid model. They are computed from the pressure and friction on the patches and are mesh-dependent because they are point loads. Therefore, the mapping to the structural model must be done with conservative mapping (see Section 4.2), to preserve the virtual work on the interface. The mesh

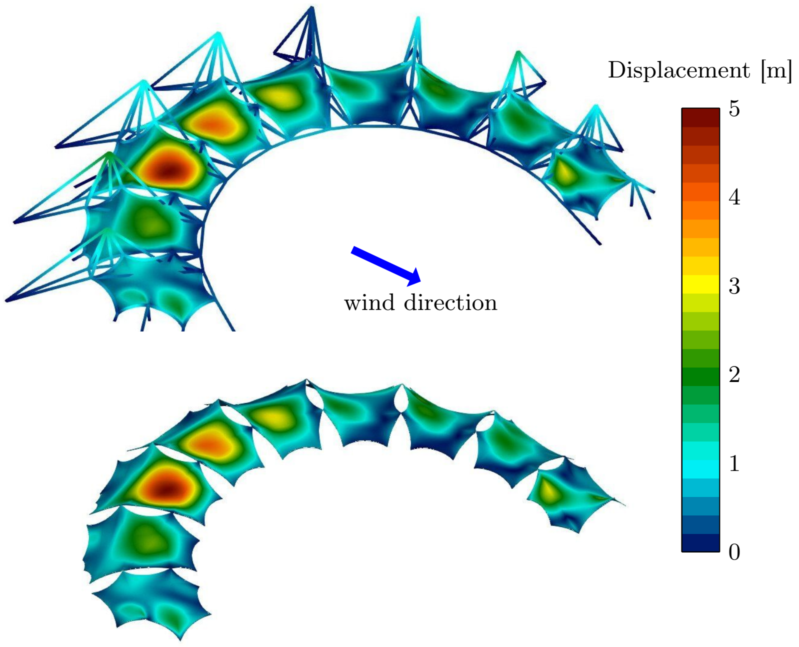


Figure 6.44: Maximum displacements during the analysis. The structural model is shown on top and the fluid interface on the bottom. The equal distribution of the displacements shows that the mapping works as expected.

dependency can be clearly observed by comparing the magnitudes of the load on each interface. The fluid interface has a much finer mesh (see Figure 6.42), which means that the loads-per-node are smaller than on the structural interface with much fewer nodes.

The mapped loads on the structure are shown in Figure 6.47. The distribution and magnitude of the loads are entirely different from the ones on the fluid interface. Once again, this is because of the mesh-dependent point loads. Furthermore, the fluid loads are mapped from the top and the bottom of the structure.

The evolution of the displacements (in the point with the highest displacement), as well as the total loads on the structure, can be seen in Figure 6.48. Significant fluctuations can be observed, originating from the turbulence caused by the roof and its surroundings.

The mapping of field quantities was done with the developments outlined in Chapter 4. Displacements were mapped from the structural to the fluid

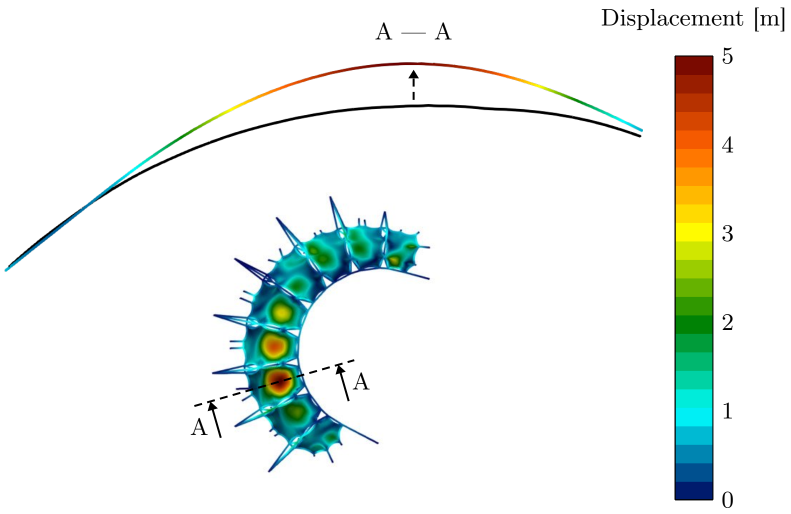


Figure 6.45: Slice through the patch with the maximum deformation. The undeformed shape is also shown.

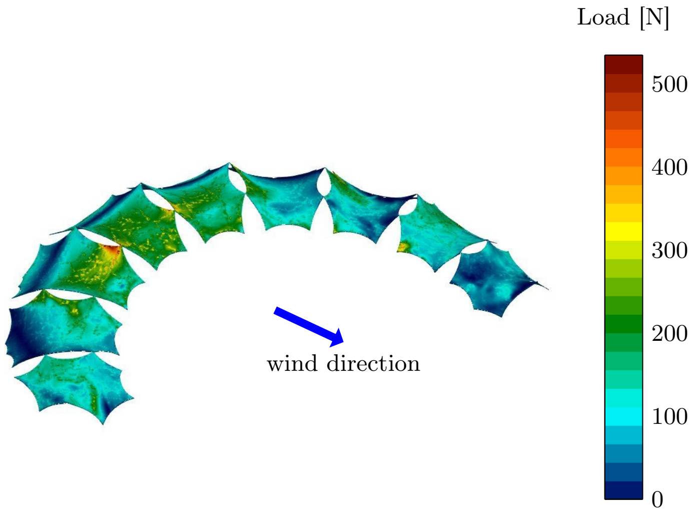


Figure 6.46: Point loads on the FSI interface of the fluid model. They are computed from the pressure on the patches.

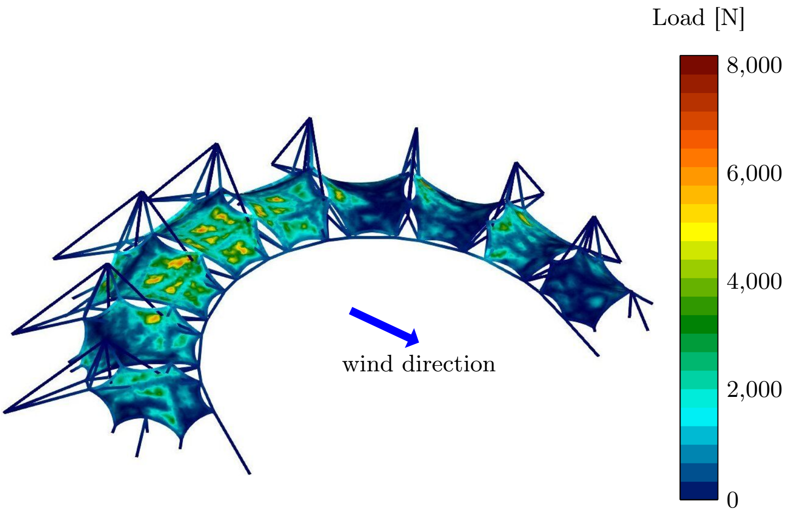


Figure 6.47: Point loads on the structural model, as mapped from the fluid model.

mesh, whereas point loads were mapped in the other direction. Table 6.14 shows the important metrics. It is evident, that the proposed algorithms and implementation are very efficient, both in terms of memory consumption and mapping time. Compared to the solving times of the solvers (see Table 6.15), the time required for mapping is negligible. This can be attributed to the fully parallel mapping procedures, which do not require any gathering of data, and therefore scale very well even with the number of cores $> 1{,}000$ as in this example.

Initialization	1.41 s
Memory usage	6.51 GB
Map Structure $\rightarrow$ Fluid	9.961×10^{-2} s
Map Fluid $\rightarrow$ Structure	1.348×10^{-3} s

Table 6.14: Mapping metrics for the mapping during the coupled FSI simulation. The initialization is only conducted once, whereas the mapping is done in every coupling iteration.

The times required by the different components are listed in Table 6.15. As expected, the fluid solver takes the most time. It uses the majority for the solution of the problem, but also a considerable amount of time is spent in the output of the solution for postprocessing. The same can be observed for

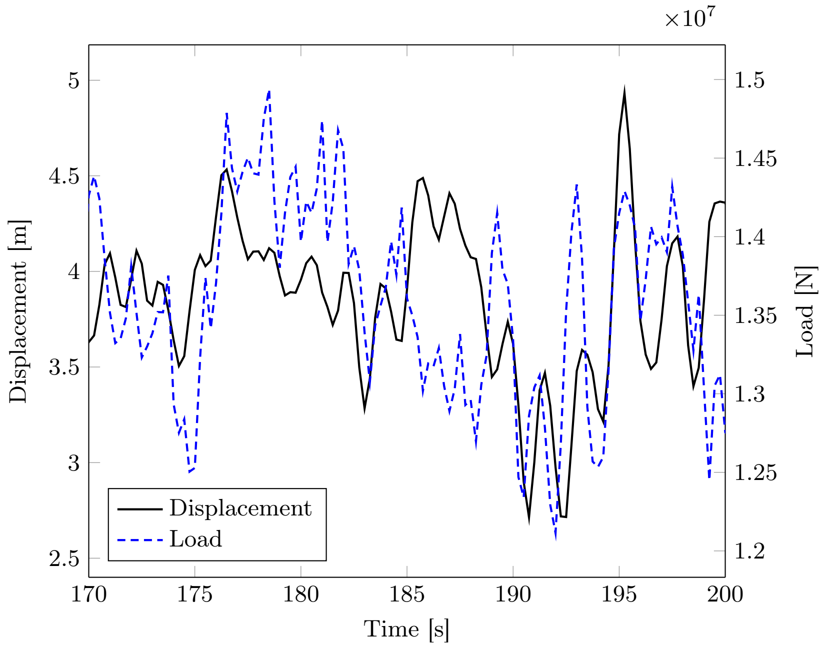


Figure 6.48: Displacement magnitude at the location with the largest deformation. Total loads, summation over all patches. Plotted over 30 s. The fluctuations originate in the wind loading.

the structural solver, even though the absolute time is less due to the smaller domain/mesh size. The most interesting finding is that a significant time is used by CoSimulation in the solution phase. The mapping is very efficient as explained before, and no IPC is needed since all solvers are executed within the same framework/memory space. The only remaining components involved are the convergence check and the convergence acceleration. These two components are not implemented for distributed parallelization. Gathering and scattering of interface data is therefore required, which presents a bottleneck when comparing it to the time used by the other components. Almost 20 % of the total simulation time is spent in CoSimulation. This result demonstrates the performance penalty arising from insufficient parallelization strategies.

The total available memory with the used computational setup was 4,320 GB, which is comprised of 96 GB of memory per compute node, for 45 compute nodes. On average, this simulation consumed 772.8 GB of memory. This is around 17.9 % of the available memory, and thus well within the limits of a modern HPC system.

Component	Solution	Other	Total
Everything	11.01	1.91	12.92
CoSimulation	2.36 (21%)	0.06 (3%)	2.42 (19%)
Fluid	8.44 (77%)	1.7 (89%)	10.14 (78%)
Structure	0.21 (2%)	0.15 (8%)	0.36 (3%)

Table 6.15: Solution times (in seconds) for the fully coupled simulation. Results are averaged over the ranks as well as the TSs, for a time of 100 s. Each timestep required 2 coupling iterations. Other is mostly output of data. The percentage refers to each column.

6.7.8 Conclusion

The fully coupled FSI simulation with the Olympic Stadium roof in Munich is the culmination of this work. The path to a successful numerical simulation required to master various problems, from many different disciplines. The challenges faced while creating computational models for the complex geometry, as well as the transition to a suitable FEM model, were presented. Many modeling decisions for both the structural and the fluid simulation are outlined. The computational setup is explained, relying heavily on the outcome of this work to achieve successful and stable simulations. The size and degree of complexity resulted in a large computational effort, which required the use of a supercomputer for conducting the simulations. The parallelization strategies developed in this work for multiphysics problems worked very well. Finally, the results are presented, highlighting both characteristics of the roof, and the computational aspects.

Chapter 7

Conclusions and outlook

This work presented the developments and advances for realizing large-scale coupled simulations for engineering applications. Several examples are used to demonstrate the capabilities of the developments, ranging from academic benchmarks to large engineering cases of real-world structures.

Firstly, the relevant theory and other fundamental information were reviewed and introduced in Chapter 2. Based on this, Chapter 3 presents the methods and algorithmic developments to achieve efficient, robust, and accurate coupled simulations. The partitioned approach was chosen such that well-established solvers can be employed. In contrast to similar works, a multiphysics framework is chosen as the basis. Conducting CoSimulation with the framework rather than a dedicated coupling tool has several advantages, as was investigated in detail. These studies were conducted with a large representative example, namely WSI simulations with the Olympic Tower in Munich. Using the multiphysics tool yielded a much better usage of the available computing resources, particularly regarding memory consumption. Employing only one software simplifies the deployment, handling, and debugging, especially on HPC systems. Furthermore, no IPC is required, which increases the stability of the coupling.

Many applications such as CFD require a considerable computational effort, therefore HPC systems such as clusters or supercomputers are often employed to keep the simulation time within practical bounds. The distributed memory architecture of those machines requires special programming techniques and algorithms, which introduce an additional layer of complexity on top of the physical problems. To fully support these systems and thus enable large-scale simulations, new procedures and methods are developed in this work. The peculiarities and other requirements are investigated and highlighted.

The coupling to external tools is an essential feature of CoSimulation, and great effort was put into realizing performant and practical methods. A newly developed detached interface was introduced, allowing for flexible and efficient integration. A key difference to existing solutions is that the interface has no dependencies on the coupling tool, meaning that the integration and deployment can be done independently. This is a clear advantage, as it simplifies the handling of the software significantly. Additionally, a new approach was developed for the coupling with external tools, the remote-controlled CoSimulation. Here the coupling sequence and execution control are handled by one central instance, and thus it avoids many problems of the existing solutions. Finally, detailed studies are conducted regarding the data exchange between the tools via IPC. The considered methods were in particular evaluated concerning their performance, robustness, and suitability for different OSs and computing systems. TCP based sockets were found to be the best compromise.

Chapter 4 revisited mapping algorithms for non-matching grids, with focus on large-scale applications. Several techniques are reviewed and some are chosen for further investigation. New ways of dealing with mapping in distributed memory environments are developed, focussing on efficiency and scalable solutions. The presented work was successfully tested on a supercomputer with over 10,000 cores, and with the examples presented in Chapter 6.

Practical experiences for setting up and running coupled simulations gathered during this work are summarized and presented in Chapter 5. These range from overall best practices on how to approach a coupled problem to choosing the coupling sequence and methods to help with convergence, such as prediction or relaxation. Furthermore, conducting CoSimulation on HPC systems is explained, in particular how to deploy the tools. An important aspect of using computing resources efficiently is to handle different methods of parallelization. An approach for coupling solvers that employ OpenMP or MPI is shown.

A wide range of examples is presented in Chapter 6. The Mok FSI benchmark was used to validate the presented work, especially for strongly coupled problems. It was also used to integrate a NNet based tool into CoSimulation, which replaced the structural solver. The results are promising, and this kind of coupling is very well-suited for future investigations. The simulation of wind flow over a bridge deck is presented, where the structural solver was realized with a SDOF. The interfaces had non-matching dimensions, therefore conventional mapping techniques could not be used, and special methods needed to be developed for the data transfer between the computational domains. The coupling of a DEM solver with a FEM solver was presented by means of simulating rock-fall protection nets. CoSimulation with a traditional coupling tool versus CoSimulation within a multiphysics framework was investigated in detail with WSI simulations of the Olympic Tower in Munich. Coupling to an external tool is shown with a FSI simulation of a full-scale wind turbine. Here, an external CFD solver was integrated into CoSimulation. A focus was also set on the structural modeling of the turbine. Finally, the culmination of this work is the coupled FSI simulation of the roof of the Olympic Stadium in Munich. The modeling of this iconic landmark is presented in detail for the geometry, fluid, and structural model. A supercomputer was employed to conduct the large simulations. This example served to evaluate the performance of the presented developments in detail.

This work showed the capabilities of a multiphysics framework for the simulation of large-scale coupled problems for engineering applications. However, still, the science is not concluded in this matter. Many promising and interesting topics can be pursued and investigated in detail. The examples and developments presented in this work employed the same time-step size for each of the solvers. This can be inefficient if the requirements of the solvers vary a lot. Using different sizes for the time-step requires interpolation in time between the solvers. It can save significant computational resources, especially if the solver with the larger time-step requirements is no longer forced to employ the smaller time-step. A use case is the integration of controllers into CoSimulation (as presented in [83]), which typically require much smaller time-step sizes compared to numerical solution techniques such as FEM-based methods.

Although mapping algorithms and coupling functionalities for volume-coupled problems were developed in this work, they are not used to simulate coupled problems. Applications such as CHT involve volume coupling and could be simulated. The integration of NNet based tools (see also [23]) was briefly introduced in this work. It has a high potential for saving computational resources and offering faster simulation times and is therefore very suitable for further investigation. The same applies to particle methods, for which only a subset of available techniques was shown in this work (an overview can be found in [67]). They are promising, particularly for the simulation of natural hazards such as landslides or avalanches, which interact with protection structures or building structures.

Finally, also the algorithmic developments for the coupling of external solvers with the detached interface and remote-controlled orchestration can be further explored and applied to more coupled problems and solvers.

Appendix A

Hardware overview

For some parts of this work, the hardware and software used are essential to consider. Therefore, this section gives a brief overview of the systems and their specifications.

SuperMUC-NG

SuperMUC-NG[1] is the supercomputer of the Leibniz-Rechenzentrum (LRZ)[2] in Garching near Munich. At the time this work was conducted, it was number 23 in the list of most powerful supercomputers worldwide[3]. It was built in 2018 and is primarily used for research applications.

SuperMUC features a total number of 311,040 compute cores, which are distributed among 6,480 compute nodes. Each compute node (Intel Xeon Platinum 8174) has 48 physical cores and 96 threads, a base frequency of 3.1 GHz, and a turbo boost frequency of 3.9 GHz. Each node has a total memory of 96 GB, of which 90 are available. In addition to its computing resources, it has a highly performant network (Intel Omni-Path) to communicate efficiently between the computing nodes in distributed computing. A highly parallel filesystem with transfer rates of up to 500 GB/s completes the system. The filesystem is optimized for handling few large files, as opposed to many small files. The OS is Suse Linux, SLURM [84] is used for the job scheduling.
The Intel compiler 2019 and Intel MPI 2019 were used to compile the software used in this work.

[1] `https://doku.lrz.de/supermuc-ng-10745965.html`
[2] `www.lrz.de`
[3] Top500 list November 2021: `www.top500.org/lists/top500/list/2021/11/`

Workstation 1

Desktop PC with the following setup:

- OS: Linux, Ubuntu 20.04
- Processor: Intel Xeon CPU E5-2660 v3 (10 cores, base/turbo boost frequency: 2.6/3.3GHz)
- RAM: 128 GB
- Hard drive: SSD 512 GB (SATA)
- Filesystem format: ext4
- C++ compiler: GNU G++ 9.4

Workstation 2

Desktop PC with the following setup:

- OS: Linux, Ubuntu 20.04 & Windows 10
- Processor: Intel Core i5-9400F (6 cores, base/turbo boost frequency: 2.9/4.1GHz)
- RAM: 16 GB
- Hard drive: SSD 1 TB (SATA)
- Filesystem format: ext4 (Linux) / NTFS (Windows)
- C++ compiler: GNU G++ 9.4 & OpenMPI 4.1.2 (Linux) / MSVC 2019 (Windows)

Virtual machine with macOS

Virtual machine for Apple-macOS with the following setup:

- OS: macOS 11.6 (Big Sur)
- Processor: virtual, (3 cores, frequency: 3.33GHz)
- RAM: 16 GB
- Hard drive: SSD 14 GB
- Filesystem format: APFS
- C++ compiler: AppleClang 13

Appendix B

Olympic Park Munich

Munich hosted the Olympic summer games in 1972 (see Figure B.1), roughly 50 years ago at the time of this writing. Many of the events were held in the Olympic Park in the northwest of Munich. It is home to the main sporting grounds, including the Olympic Stadium, Olympic Hall, Aquatic Center, and the velodrome, as shown in Figure B.2. The Olympic Tower and Olympic Mountain are also part of the park.

Figure B.1: Poster Olympic Games 1972 displaying the roof structures and the tower. Source: `www.olympiapark.de`.

The stadium and its surrounding buildings and structures have become a landmark and part of the city's image due to their iconic shape and architecture. The roofs of the stadium, aquatic center, hall, and some areas in between them

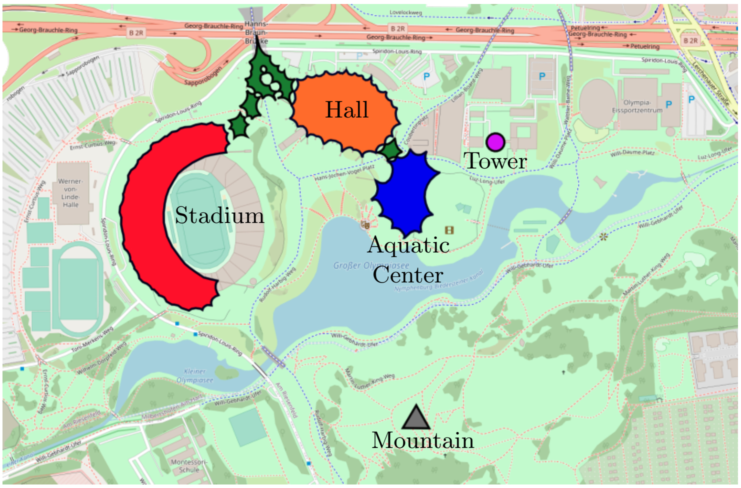


Figure B.2: Overview Olympic Park Munich. The roof structures are highlighted for the stadium (red), hall (orange), aquatic center (blue), and the areas in between the buildings (green). Additionally, the tower and the Olympic mountain are shown. Source of the base image: `www.openstreetmap.org/export#map=16/48.1716/11.5500`.

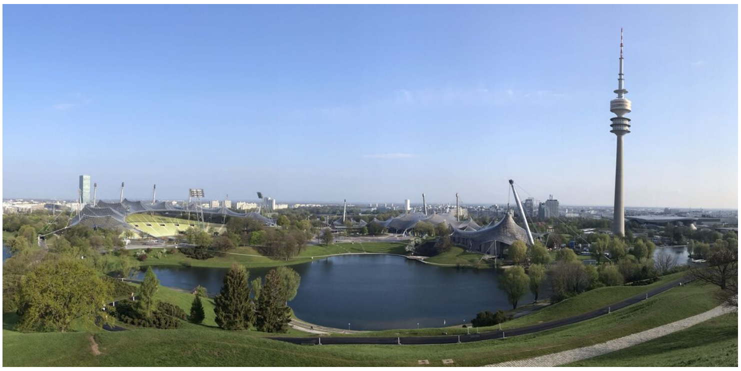


Figure B.3: View over Olympic Park, perspective from Olympic Mountain. The roof structures of the different buildings and the tower are visible. Source: P. Bucher

are covered with hanging structures made from cable nets covered with acrylic plates. These nets are held up by poles and other supporting structures. The structures were and are considered highlights of architecture and engineering and were ahead of their time when first planned and constructed. Figure B.3 gives an impression of the structures and the tower in the park.

The iconic design was done by the offices of Behnisch & Partner and Leonhardt & Andrä in collaboration with Frei Otto. It was the winner of the design competition in 1967. The research report SFB 64 *Long-Span Surface Structures* [51] describes the design and explains the thoughts of the architects and engineers.

Bibliography

[1] K.-J. Bathe. *Finite Element Procedures*. Prentice Hall, 2006. ISBN: 9780979004902.

[2] A. Beckert and H. Wendland. “Multivariate Interpolation for Fluid-Structure-Interaction Problems Using Radial Basis Functions”. In: *Aerospace Science and Technology* 5 (Mar. 2001). DOI: 10.1016/S1270-9638(00)01087-7.

[3] T. Belytschko, W. K. Liu, B. Moran, and K. Elkhodary. *Nonlinear finite elements for continua and structures*. John wiley & Sons, 2013.

[4] J. Benek, J. Steger, and F. C. Dougherty. “A flexible grid embedding technique with application to the Euler equations”. In: *6th Computational Fluid Dynamics Conference Danvers*. 1983, p. 1944.

[5] G. van den Bergen. “Efficient Collision Detection of Complex Deformable Models using AABB Trees”. In: *Journal of Graphics Tools* 2.4 (1997), pp. 1–13. DOI: 10.1080/10867651.1997.10487480.

[6] K.-U. Bletzinger and E. Ramm. “A General Finite Element Approach to the form Finding of Tensile Structures by the Updated Reference Strategy”. In: *International Journal of Space Structures* 14.2 (1999), pp. 131–145. DOI: 10.1260/0266351991494759. eprint: https://doi.org/10.1260/0266351991494759.

[7] A. Bogaers, S. Kok, B. Reddy, and T. Franz. “Quasi-Newton methods for implicit black-box FSI coupling”. In: *Computer Methods in Applied Mechanics and Engineering* 279 (2014), pp. 113–132. DOI: https://doi.org/10.1016/j.cma.2014.06.033.

[8] P. Bucher, J. Cotela-Dalmau, T. Teschemacher, and R. Wüchner. “Implementation of a general mapping framework for different discretizations in distributed memory environments”. In: *VIII International Conference on Coupled Problems in Science and Engineering*. ECCOMAS. Sitges (Barcelona), Spain, June 2019.

[9] P. Bucher, A. Ghantasala, P. Dadvand, R. Wüchner, and K. Bletzinger. “Realizing CoSimulation in and with a multiphysics framework”. In: *9th edition of the International Conference on Computational Methods for Coupled Problems in Science and Engineering (COUPLED PROBLEMS 2021)*. 2021. DOI: 10.23967/coupled.2021.048.

[10] P. Bucher, M. Péntek, K. B. Sautter, R. Wüchner, and K.-U. Bletzinger. “Detailed FSI modeling and HPC simulation of the Olympic stadium roof in Munich under wind loading”. In: *10th edition of the conference on Textile Composites and Inflatable Structures.* CIMNE, 2021. DOI: `10.23967/membranes.2021.039`.

[11] H.-J. Bungartz, F. Lindner, B. Gatzhammer, M. Mehl, K. Scheufele, A. Shukaev, and B. Uekermann. “preCICE – A fully parallel library for multi-physics surface coupling”. In: *Computers & Fluids* 141 (2016). Advances in Fluid-Structure Interaction, pp. 250–258. DOI: `https://doi.org/10.1016/j.compfluid.2016.04.003`.

[12] J. Cotela-Dalmau. “Applications of Turbulence Modeling in Civil Engineering”. Dissertation. Universidad Politécnica de Cataluña, 2016.

[13] P. Dadvand, R. Rossi, M. Gil, X. Martorell, J. Cotela, E. Juanpere, S. Idelsohn, and E. Oñate. “Migration of a generic multi-physics framework to HPC environments”. In: *Computers & Fluids* 80 (2013). Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011, pp. 301–309. DOI: `https://doi.org/10.1016/j.compfluid.2012.02.004`.

[14] P. Dadvand. “A framework for developing finite element codes for multi-disciplinary applications”. Dissertation. Universidad Politécnica de Cataluña, 2007.

[15] P. Dadvand, R. Rossi, and E. Oñate. “An Object-oriented Environment for Developing Finite Element Codes for Multi-disciplinary Applications”. In: *Archives of Computational Methods in Engineering* 17.3 (Sept. 2010), pp. 253–297. DOI: `10.1007/s11831-010-9045-2`.

[16] “Data Structures and Algorithms”. In: *Applied Computational Fluid Dynamics Techniques.* John Wiley & Sons, Ltd, 2008. Chap. 2, pp. 7–33. ISBN: 9780470989746. DOI: `https://doi.org/10.1002/9780470989746.ch2`.

[17] A. de Boer, A. van Zuijlen, and H. Bijl. “Comparison of conservative and consistent approaches for the coupling of non-matching meshes”. In: *Computer Methods in Applied Mechanics and Engineering* 197.49 (2008), pp. 4284–4297. DOI: `https://doi.org/10.1016/j.cma.2008.05.001`.

[18] A. de Boer, A. van Zuijlen, and H. Bijl. “Review of coupling methods for non-matching meshes”. In: *Computer Methods in Applied Mechanics and Engineering* 196.8 (2007). Domain Decomposition Methods: recent advances and new challenges in engineering, pp. 1515–1525. DOI: `https://doi.org/10.1016/j.cma.2006.03.017`.

[19] J. Degroote, K.-J. Bathe, and J. Vierendeels. “Performance of a new partitioned procedure versus a monolithic procedure in fluid-structure interaction”. In: *Computers & Structures* 87 (June 2009), pp. 793–801. DOI: `10.1016/j.compstruc.2008.11.013`.

[20] Degroote, Joris. “Development of algorithms for the partitioned simulation of strongly coupled fluid-structure interaction problems”. eng. PhD thesis. Ghent University, 2010, XXXVII, 267. ISBN: 9789085783442.

[21] W. G. Dettmer and D. Perić. “A new staggered scheme for fluid-structure interaction”. In: *International Journal for Numerical Methods in Engineering* 93.1 (2013), pp. 1–22. DOI: `https://doi.org/10.1002/nme.4370`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.4370`.

[22] J. Donea and A. Huerta. *Finite Element Methods for Flow Problems.* John Wiley & Sons, Ltd, 2003. ISBN: 9780470013823. DOI: `https://doi.org/10.1002/0470013826`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/0470013826`.

[23] R. Ellath Meethal. “Hybrid modelling and simulation approaches for the solution of forward and inverse problems in engineering by combining finite element methods and neural networks”. Dissertation. Technical University of Munich, 2023.

[24] P. W. Farah. “Mortar Methods for Computational Contact Mechanics Including Wear and General Volume Coupled Problems”. Dissertation. Technical University of Munich, 2018.

[25] C. Farhat, M. Lesoinne, and P. Le Tallec. “Load and motion transfer algorithms for fluid/structure interaction problems with non-matching discrete interfaces: Momentum and energy conservation, optimal discretization and application to aeroelasticity”. In: *Computer Methods in Applied Mechanics and Engineering* 157.1 (1998), pp. 95–114. DOI: `https://doi.org/10.1016/S0045-7825(97)00216-8`.

[26] C. A. Felippa. *Introduction to Finite Element Methods.* Lecture notes. 2004.

[27] C. A. Felippa, K. Park, and C. Farhat. “Partitioned analysis of coupled mechanical systems”. In: *Computer Methods in Applied Mechanics and Engineering* 190.24 (2001). Advances in Computational Methods for Fluid-Structure Interaction, pp. 3247–3270. DOI: `https://doi.org/10.1016/S0045-7825(00)00391-1`.

[28] T. G. Gallinger. “Effiziente Algorithmen zur partitionierten Lösung stark gekoppelter Probleme der Fluid-Struktur-Wechselwirkung”. Dissertation. Technical University of Munich, 2010.

[29] B. Gatzhammer. “Efficient and Flexible Partitioned Simulation of Fluid-Structure Interactions”. Dissertation. Technical University of Munich, 2014.

[30] C. Gear. “Simultaneous Numerical Solution of Differential-Algebraic Equations”. In: *IEEE Transactions on Circuit Theory* 18.1 (1971), pp. 89–95. DOI: `10.1109/TCT.1971.1083221`.

[31] A. Ghantasala. “Coupling Procedures for Fluid-Fluid and Fluid-Structure Interaction Problems Based on Domain Decomposition methods”. Dissertation. Technical University of Munich, 2021.

[32] G. Guennebaud, B. Jacob, et al. *Eigen v3.* http://eigen.tuxfamily.org. 2010.

[33] G. Guma, P. Bucher, P. Letzgus, T. Lutz, and R. Wüchner. "High-fidelity aeroelastic analyses of wind turbines in complex terrain: fluid-structure interaction and aerodynamic modeling". In: *Wind Energy Science* 7.4 (2022), pp. 1421–1439. DOI: 10.5194/wes-7-1421-2022.

[34] G. Guma. "Aeroelastic Effects of Wind Turbines in Complex Terrain". Dissertation. University of Stuttgart, 2023.

[35] F. Hecht. "New development in FreeFem++". In: *J. Numer. Math.* 20.3-4 (2012), pp. 251–265.

[36] M. Heil, A. L. Hazel, and J. Boyle. "Solvers for large-displacement fluid–structure interaction problems: segregated versus monolithic approaches". In: *Computational Mechanics* 43.1 (Dec. 2008), pp. 91–101. DOI: 10.1007/s00466-008-0270-6.

[37] G. A. Holzapfel. "Nonlinear Solid Mechanics: A Continuum Approach for Engineering Science". In: *Meccanica* 37.4 (July 2002), pp. 489–490. DOI: 10.1023/A:1020843529530.

[38] G. Houzeaux, R. Borrell, J. Cajas, and M. Vázquez. "Extension of the parallel Sparse Matrix Vector Product (SpMV) for the implicit coupling of PDEs on non-matching meshes". In: *Computers & Fluids* 173 (2018), pp. 216–225. DOI: https://doi.org/10.1016/j.compfluid.2018.03.006.

[39] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis.* Dover Publications Inc, 2000.

[40] *Importing Actual 3D Models From Google Maps.* https://blog.exppad.com/article/importing-actual-3d-models-from-google-maps. Last accessed 2023-03-14.

[41] *Inventory documents: Measurements of the main points of the stadium roof.* Carried out as part of the construction (1971 and 1972). Archives Olympiapark Munich.

[42] W. Joppich and M. Kürschner. "MpCCI - a tool for the simulation of coupled applications". In: *Concurr. Comput. Pract. Exp.* 18.2 (2006), pp. 183–192. DOI: 10.1002/cpe.913.

[43] G. Karypis and V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *Siam Journal on Scientific Computing* 20 (Jan. 1999), pp. 359–392. DOI: 10.1137/S1064827595287997.

[44] D. E. Keyes et al. "Multiphysics simulations: Challenges and opportunities". In: *The International Journal of High Performance Computing Applications* 27.1 (2013), pp. 4–83. DOI: 10.1177/1094342012468181. eprint: https://doi.org/10.1177/1094342012468181.

[45] L. Klein. “Numerische Untersuchung aerodynamischer und aeroelastischer Wechselwirkungen und deren Einfluss auf tieffrequente Emissionen von Windkraftanlagen”. Dissertation. University of Stuttgart, 2019. DOI: `http://dx.doi.org/10.18419/opus-147`.

[46] M. König. “Partitioned solution strategies for strongly-coupled fluid-structure interaction problems in maritime applications”. Dissertation. Technische Universität Hamburg, 2018. ISBN: 978-3-18-335118-3. DOI: `10.15480/882.1736`.

[47] M. König, L. Radtke, and A. Düster. “A Flexible C++ Framework for the Efficient Solution of Strongly Coupled Multifield Problems”. In: *PAMM* 16.1 (2016), pp. 455–456. DOI: `https://doi.org/10.1002/pamm.201610216`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/pamm.201610216`.

[48] D. Kuhl. “Stabile Zeitintegrationsalgorithmen in der nichtlinearen Elastodynamik dünnwandiger Tragwerke”. Dissertation. University of Stuttgart, 1996.

[49] U. Küttler. “Effiziente Lösungsverfahren für Fluid-Struktur-Interaktions-Probleme”. Dissertation. Technical University of Munich, 2009.

[50] U. Küttler and W. Wall. “Fixed-point fluid-structure interaction solvers with dynamic relaxation”. In: *Computational Mechanics* 43 (Jan. 2008), pp. 61–72. DOI: `10.1007/s00466-008-0255-5`.

[51] F. Leonhardt and J. Schlaich. “Weitgespannte Flächentragwerke”. In: *Sonderforschungsbereich 64* (1973).

[52] P. Letzgus, G. Guma, and T. Lutz. “Computational fluid dynamics studies on wind turbine interactions with the turbulent local flow field influenced by complex topography and thermal stratification”. In: *Wind Energy Science* 7.4 (2022), pp. 1551–1573. DOI: `10.5194/wes-7-1551-2022`.

[53] F. Lindner. “Data Transfer in Partitioned Multi-Physics Simulations: Interpolation & Communication”. Dissertation. University of Stuttgart, 2019. DOI: `http://dx.doi.org/10.18419/opus-10581`.

[54] I. López, J. Piquee, P. Bucher, K.-U. Bletzinger, C. Breitsamter, and R. Wüchner. “Numerical analysis of an elasto-flexible membrane blade using steady-state fluid-structure interaction simulations”. In: *Journal of Fluids and Structures* 106 (2021), p. 103355. DOI: `https://doi.org/10.1016/j.jfluidstructs.2021.103355`.

[55] S. A. Maas, B. J. Ellis, G. A. Ateshian, and J. A. Weiss. “FEBio: Finite Elements for Biomechanics”. In: *Journal of Biomechanical Engineering* 134.1 (Feb. 2012). 011005. DOI: `10.1115/1.4005694`. eprint: `https://asmedigitalcollection.asme.org/biomechanical/article-pdf/134/1/011005/5665064/011005\_1.pdf`.

[56] M. J. Mair. “Formfindung der Überdachung des Olympiastadions”. Bachelor’s Thesis. Technical University of Munich, 2010.

[57] J. Mann. “The spatial structure of neutral atmospheric surface-layer turbulence”. In: *Journal of Fluid Mechanics* 273 (1994), pp. 141–168. DOI: `10.1017/S0022112094001886`.

[58] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0.* June 2021.

[59] A. Mini, R. Wüchner, and K.-U. Bletzinger. “Mesh updating strategies for computational fluid-structure interaction with large deformations”. In: *Baustatik-Baupraxis Forschungskolloquium 2015, Döllnsee/Brandenburg.* 2015.

[60] D. P. Mok. “Partitionierte Lösungsansätze in der Strukturdynamik und der Fluid-Struktur-Interaktion”. Dissertation. University of Stuttgart, 2001. DOI: `http://dx.doi.org/10.18419/opus-147`.

[61] D. T. Nguyen, D. M. Hargreaves, and J. S. Owen. “Vortex-induced vibration of a 5:1 rectangular cylinder: A comparison of wind tunnel sectional model tests and computational simulations”. In: *Journal of Wind Engineering and Industrial Aerodynamics* 175 (2018), pp. 1–16. DOI: `https://doi.org/10.1016/j.jweia.2018.01.029`.

[62] F. Pellegrini and J. Roman. “Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs”. In: *High-Performance Computing and Networking.* Ed. by H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 493–498. ISBN: 978-3-540-49955-8.

[63] M. Péntek, P. Bucher, K. B. Sautter, and K.-U. Bletzinger. “The 50-year anniversary of the Olympic Stadium in Munich as a motivator for advances in computational wind engineering”. In: *8th European-African Conference on Wind Engineering.* 2022, pp. 223–226.

[64] J. Raddatz and J. K. Fassbender. “Block Structured Navier-Stokes Solver FLOWer”. In: *MEGAFLOW - Numerical Flow Simulation for Aircraft Design.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 27–44. ISBN: 978-3-540-32382-2.

[65] S. Ristov, R. Prodan, M. Gusev, and K. Skala. “Superlinear speedup in HPC systems: Why and when?” In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS).* 2016, pp. 889–898.

[66] M. Santasusana. “Numerical techniques for non-linear analysis of structures combining discrete element and finite element methods”. Dissertation. Universidad Politécnica de Cataluña, 2016.

[67] K. B. Sautter. “Modeling and Simulation of Flexible Protective Structures by Coupling Particle and Finite Element Methods”. Dissertation. Technical University of Munich, 2022.

[68] K. B. Sautter, H. Hofmann, C. Wendeler, R. Wüchner, and K.-U. Bletzinger. “Influence of DE-cluster refinement on numerical analysis of rockfall experiments”. In: *Computational Particle Mechanics* (Feb. 2021). DOI: `10.1007/s40571-020-00382-x`.

[69] K. B. Sautter, T. Teschemacher, M. Á. Celigueta, P. Bucher, K.-U. Bletzinger, and R. Wüchner. "Partitioned Strong Coupling of Discrete Elements with Large Deformation Structural Finite Elements to Model Impact on Highly Flexible Tension Structures". In: *Advances in Civil Engineering* 2020 (Nov. 2020), p. 5135194. DOI: `10.1155/2020/5135194`.

[70] M. Sayed, T. Lutz, E. Krämer, S. Shayegan, A. Ghantasala, R. Wüchner, and K.-U. Bletzinger. "High fidelity CFD-CSD aeroelastic analysis of slender bladed horizontal-axis wind turbine". In: *Journal of Physics: Conference Series*. Vol. 753. IOP Publishing. 2016, p. 042009.

[71] M. Sayed, P. Bucher, G. Guma, T. Lutz, and R. Wüchner. "Aeroelastic Simulations Based on High-Fidelity CFD and CSD Models". In: *Handbook of Wind Energy Aerodynamics*. Ed. by B. Stoevesandt, G. Schepers, P. Fuglsang, and S. Yuping. Cham: Springer International Publishing, 2020, pp. 1–76. ISBN: 978-3-030-05455-7. DOI: `10.1007/978-3-030-05455-7_22-1`.

[72] D. Schwamborn, T. Gerhold, and R. Heinrich. "The DLR TAU-code: Recent applications in research and industry". In: *ECCOMAS CFD 2006 CONFERENCE*. 2006.

[73] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. Wiley Publishing, 2018. ISBN: 978-1-119-32091-3.

[74] K. Stein, T. Tezduyar, and R. Benney. "Mesh Moving Techniques for Fluid-Structure Interactions With Large Displacements ". In: *Journal of Applied Mechanics* 70.1 (Jan. 2003), pp. 58–63. DOI: `10.1115/1.1530635`. eprint: `https://asmedigitalcollection.asme.org/appliedmechanics/article-pdf/70/1/58/5469200/58\_1.pdf`.

[75] J. W. Strutt. *The Theory of Sound*. Vol. 2. Cambridge Library Collection - Physical Sciences. Cambridge University Press, 2011. DOI: `10.1017/CBO9781139058094`.

[76] The Trilinos Project Team. *The Trilinos Project Website*. `https://trilinos.github.io`. Last accessed 2021-12-10.

[77] B. Uekermann. "Partitioned Fluid-Structure Interaction on Massively Parallel Systems". Dissertation. Technical University of Munich, 2016.

[78] J. G. Valdés Vázquez. "Nonlinear Analysis of Orthotropic Membrane and Shell Structures Including Fluid-Structure Interaction". Dissertation. Universidad Politécnica de Cataluña, 2007.

[79] W. Wall. "Fluid structure interaction with stabilized finite elements". Dissertation. University of Stuttgart, 1999. DOI: `http://dx.doi.org/10.18419/opus-127`.

[80] T. Wang, S. Sicklinger, R. Wüchner, and K. Bletzinger. "Concept and Realization of Coupling Software EMPIRE in Multi-Physics Co-Simulation". In: *Computational Methods in Marine Engineering*. 2013.

[81] T. Wang. “Development of Co-Simulation Environment and Mapping Algorithms”. Dissertation. Technical University of Munich, 2016.

[82] A. Winterstein, S. Warnakulasuriya, K.-U. Bletzinger, and R. Wüchner. “Computational wind-structure interaction simulations of high rise and slender structures and validation against on-site measurements — Methodology development and assessment”. In: *Journal of Wind Engineering and Industrial Aerodynamics* 232 (2023), p. 105278. DOI: `https://doi.org/10.1016/j.jweia.2022.105278`.

[83] A. Winterstein. “Modeling and Simulation of Wind-Structure Interaction of Slender Civil Engineering Structures Including Vibration Mitigation Systems”. Dissertation. Technical University of Munich, 2020.

[84] A. B. Yoo, M. A. Jette, and M. Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing.* Ed. by D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[85] R. Zorrilla and R. Rossi. “A memory-efficient MultiVector Quasi-Newton method for black-box Fluid-Structure Interaction coupling”. In: *Computers & Structures* 275 (Nov. 2022), p. 106934. DOI: `10.1016/j.compstruc.2022.106934`.

Bisherige Titel der Schriftenreihe

Band	Titel
1	Frank Koschnick, *Geometrische Lockingeffekte bei Finiten Elementen und ein allgemeines Konzept zu ihrer Vermeidung*, 2004.
2	Natalia Camprubi, *Design and Analysis in Shape Optimization of Shells*, 2004.
3	Bernhard Thomee, *Physikalisch nichtlineare Berechnung von Stahlfaserbetonkonstruktionen*, 2005.
4	Fernaß Daoud, *Formoptimierung von Freiformschalen - Mathematische Algorithmen und Filtertechniken*, 2005.
5	Manfred Bischoff, *Models and Finite Elements for Thin-walled Structures*, 2005.
6	Alexander Hörmann, *Ermittlung optimierter Stabwerkmodelle auf Basis des Kraftflusses als Anwendung plattformunabhängiger Prozesskopplung*, 2006.
7	Roland Wüchner, *Mechanik und Numerik der Formfindung und Fluid-Struktur-Interaktion von Membrantragwerken*, 2006.
8	Florian Jurecka, *Robust Design Optimization Based on Metamodeling Techniques*, 2007.
9	Johannes Linhard, *Numerisch-mechanische Betrachtung des Entwurfsprozesses von Membrantragwerken*, 2009.
10	Alexander Kupzok, *Modeling the Interaction of Wind and Membrane Structures by Numerical Simulation*, 2009.

Band	Titel
11	Bin Yang, *Modified Particle Swarm Optimizers and their Application to Robust Design and Structural Optimization*, 2009.
12	Michael Fleischer, *Absicherung der virtuellen Prozesskette für Folgeoperationen in der Umformtechnik*, 2009.
13	Amphon Jrusjrungkiat, *Nonlinear Analysis of Pneumatic Membranes - From Subgrid to Interface*, 2009.
14	Alexander Michalski, *Simulation leichter Flächentragwerke in einer numerisch generierten atmosphärischen Grenzschicht*, 2010.
15	Matthias Firl, *Optimal Shape Design of Shell Structures*, 2010.
16	Thomas Gallinger, *Effiziente Algorithmen zur partitionierten Lösung stark gekoppelter Probleme der Fluid-Struktur-Wechselwirkung*, 2011.
17	Josef Kiendl, *Isogeometric Analysis and Shape Optimal Design of Shell Structures*, 2011.
18	Joseph Jordan, *Effiziente Simulation großer Mauerwerksstrukturen mit diskreten Rissmodellen*, 2011.
19	Albrecht von Boetticher, *Flexible Hangmurenbarrieren: Eine numerische Modellierung des Tragwerks, der Hangmure und der Fluid-Struktur-Interaktion*, 2012.
20	Robert Schmidt, *Trimming, Mapping, and Optimization in Isogeometric Analysis of Shell Structures*, 2013.
21	Michael Fischer, *Finite Element Based Simulation, Design and Control of Piezoelectric and Lightweight Smart Structures*, 2013.
22	Falko Hartmut Dieringer, *Numerical Methods for the Design and Analysis for Tensile Structures*, 2014.
23	Rupert Fisch, *Code Verification of Partitioned FSI Environments for Lightweight Structures*, 2014.

Band	Titel
36	Benedikt Philipp, *Methodological Treatment of Non-linear Structural Behavior in the Design, Analysis and Verification of Lightweight Structures*, 2017.
37	Michael Andre, *Aeroelastic Modeling and Simulation for the Assessment of Wind Effects on a Parabolic Trough Solar Collector*, 2018.
38	Andreas Apostolatos, *Isogeometric Analysis of Thin-Walled Structures on Multipatch Surfaces in Fluid-Structure Interaction*, 2018.
39	Altuğ Emiroğlu, *Multiphysics Simulation and CAD-Integrated Shape Optimization in Fluid-Structure Interaction*, 2019.
40	Mehran Saeedi, *Multi-Fidelity Aeroelastic Analysis of Flexible Membrane Wind Turbine Blades*, 2017.
41	Reza Najian Asl, *Shape optimization and sensitivity analysis of fluids, structures, and their interaction using Vertex Morphing Parametrization*, 2019.
42	Ahmed Abodonya, *Verification Methodology for Computational Wind Engineering Prediction of Wind Loads on Structures*, 2020.
43	Anna Maria Bauer, *CAD-integrated Isogeometric Analysis and Design of Lightweight Structures*, 2020.
44	Andreas Winterstein, *Modeling and Simulation of Wind Structure Interaction of Slender Civil Engineering Structures Including Vibration Mitigation Systems*, 2020.
45	Franz-Josef Ertl, *Vertex Morphing for Constrained Shape Optimization of Three-dimensional Solid Structures*, 2020.
46	Daniel Baumgärtner, *On the Grid-based Shape Optimization of Structures with Internal Flow and the Feedback of Shape Changes into a CAD Model*, 2020.
47	Mohamed Khalil, *Combining Physics-based models and machine learning for an Enhanced Structural Health Monitoring*,2021.