Master’s Thesis

A Fully Parallel Process-to-Process Intercommunication Technique for preCICE

Author: Alexander K. Shukaev
1st Examiner: Prof. Dr. Hans-Joachim Bungartz
2nd Examiner: Prof. Dr. Miriam Mehl
Assistant Advisor: Dipl.-Math. Benjamin Uekermann
Submission Date: June 11, 2015
I hereby declare that this Master’s thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the “BIBLIOGRAPHY” section.

June 11, 2015

Alexander K. Shukaev
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my assistant advisor, Benjamin Uekermann, for his generous assistance and valuable suggestions throughout this project. I would also like to thank the chair of Scientific Computing and the department of Informatics at the Technische Universität München for giving me the opportunity to become a student of the Master’s program in Computational Science and Engineering and write this interesting Master’s thesis. Most of all, I am forever indebted to my family and especially my parents for their invaluable support throughout all these hard years of studies. Without them, I would never be where I am.
ABSTRACT

Large-scale numerical simulations of multiphysics scenarios have become a common daily task in the field of scientific computing. During the past decade, partitioned coupling of existing (application-specific) solvers through an intermediate piece of software, which treats those solvers as black boxes, has emerged as a well-established trend. Compared to the monolithic simulations, this provides the three main benefits: put emphasis on reuse of mature solvers, generically overcome the complexities associated with (partitioned) coupling, allow for high flexibility in choice of solvers and extensive configuration of their interaction process. The three essential facilities are necessary to constitute a (partitioned) coupling framework: equation coupling, data mapping, and data intercommunication. It is clear that for coupling framework to produce minimum overhead, it is important that all three components run efficiently and rely on parallelism where possible.

Precise Code Interaction Coupling Environment (preCICE) is a modern code coupling library for partitioned simulations of multiphysics scenarios that was initially developed by Bernhard Gatzhammer at the Technische Universität München (TUM). It is already successfully used to couple and drive (partitioned) simulations of fluid–fluid interaction (FFI), fluid–structure interaction (FSI), and some other scenarios. Previously, preCICE only offered a centralized intercommunication model (CICM) between the processes of the coupled (intraparallel) solvers, that is they had to forward all (data) intercommunication either through an intermediate server process or through their respective master processes. Performance testing has revealed that this approach indeed results in a severe data throughput bottleneck, which leads to a dramatic loss of scalability for medium- and large-sized production runs.

The major goal pursued by this work was to eliminate the intercommunication bottleneck completely and yield nearly ideal strong and weak scaling of the preCICE intercommunication routines. It was successfully accomplished by introducing a distributed intercommunication model (DICM) into preCICE, which is based upon a fully parallel process-to-process (P2P) intercommunication technique between the coupled solvers. Besides, another prime objective was that the implementation of the P2P intercommunication technique must transparently support both the TCP/IP and the MPI network communication standards. The ideal strong scaling performance achievements have been verified and confirmed at least up to 32768 processes overall by profiling various scalability benchmarks and application scenarios on the SuperMUC massively parallel system.
## CONTENTS

**ACRONYMS** vii

**INTRODUCTION** 1

### 1 Coupling Software Review

#### 1.1 Advanced Scientific Computing Development Toolkit (ASCoDT) 5

#### 1.2 ADVanced ENgineering analysis Tool for Ultra large REal world (ADVENTURE) 7

#### 1.3 Coupling With Interpolation Parallel Interface (CWIPPI) 9

#### 1.4 Enhanced Multi Physics Interface Research Engine (EMPIRE) 10

#### 1.5 Mesh-based parallel Code Coupling Interface (MpCCI) 10

#### 1.6 OASIS3-MCT 12

#### 1.7 OpenPALM 13

#### 1.8 Other Software 13

### 2 Background

#### 2.1 The Centralized Intercommunication Model (CICM) 15

##### 2.1.1 The Client/Server Intercommunication Technique 15

##### 2.1.2 The Master/Slave Intercommunication Technique 16

#### 2.2 The Distributed Intercommunication Model (DICM) 17

##### 2.2.1 The Process-to-Process (P2P) Intercommunication Technique 17

#### 2.3 Conclusion 20

### 3 Implementation

#### 3.1 Architecture 21

#### 3.2 TCP/IP Sockets Communication 23

##### 3.2.1 Rationale 23

##### 3.2.2 Basics 24

##### 3.2.3 Wildcards 24

##### 3.2.4 Linger Time 25

##### 3.2.5 Boost.Asio 25

##### 3.2.6 The precice::com::SocketCommunication Class 26

#### 3.3 MPI Ports Communication 30

##### 3.3.1 Rationale 31

##### 3.3.2 Basics 31

##### 3.3.3 The precice::com::MPIPortsCommunication Class 32

#### 3.4 Asynchronous (Non-Blocking) Communication Requests 35

#### 3.5 One-to-Many Communication Factories 35

#### 3.6 Process-to-Process (P2P) Intercommunication 37

##### 3.6.1 Rationale 37
<table>
<thead>
<tr>
<th>ACRONYMS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAI</td>
<td>Acoustics–Acoustics Interaction</td>
</tr>
<tr>
<td>ADVENTURE</td>
<td>ADVanced ENgineering analysis Tool for Ultra large REal world</td>
</tr>
<tr>
<td>APES</td>
<td>Adaptable Poly-Engineering Simulator</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASCoDT</td>
<td>Advanced Scientific Computing Development Toolkit</td>
</tr>
<tr>
<td>BEM</td>
<td>Boundary Element Method</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CEDRE</td>
<td>Calcul d’Écoulements Diphasiques Réactifs pour l’Énergétique</td>
</tr>
<tr>
<td>CERFACS</td>
<td>Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique</td>
</tr>
<tr>
<td>CFD</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>CICM</td>
<td>Centralized Intercommunication Model</td>
</tr>
<tr>
<td>CIP</td>
<td>Coupling Interface Process</td>
</tr>
<tr>
<td>CiCP</td>
<td>Communications in Computational Physics</td>
</tr>
<tr>
<td>CMD</td>
<td>Computational Mesh Dynamics</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSD</td>
<td>Computational Structural Dynamics</td>
</tr>
<tr>
<td>CSE</td>
<td>Computational Science and Engineering</td>
</tr>
<tr>
<td>CUG</td>
<td>Cray User Group</td>
</tr>
<tr>
<td>CWIPI</td>
<td>Coupling With Interpolation Parallel Interface</td>
</tr>
<tr>
<td>DG</td>
<td>Discontinuous Galerkin</td>
</tr>
<tr>
<td>DICM</td>
<td>Distributed Intercommunication Model</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic-Link Library</td>
</tr>
<tr>
<td>ECCM</td>
<td>European Conference on Computational Mechanics</td>
</tr>
<tr>
<td>ECFD</td>
<td>European Conference on Computational Fluid Dynamics</td>
</tr>
<tr>
<td>EMPIRE</td>
<td>Enhanced Multi Physics Interface Research Engine</td>
</tr>
<tr>
<td>FAI</td>
<td>Fluid–Acoustics Interaction</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite Element Method</td>
</tr>
<tr>
<td>FFI</td>
<td>Fluid–Fluid Interaction</td>
</tr>
<tr>
<td>FSAI</td>
<td>Fluid–Structure–Acoustics Interaction</td>
</tr>
<tr>
<td>FSI</td>
<td>Fluid–Structure Interaction</td>
</tr>
<tr>
<td>GCS</td>
<td>Gauss Centre for Supercomputing</td>
</tr>
<tr>
<td>GPFS</td>
<td>General Parallel File System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>GVD</td>
<td>Global Vertex Data</td>
</tr>
<tr>
<td>GVDA</td>
<td>Global Vertex Data Array</td>
</tr>
<tr>
<td>GVDI</td>
<td>Global Vertex Data Index</td>
</tr>
<tr>
<td>GVDO</td>
<td>Global Vertex Data Offset</td>
</tr>
<tr>
<td>HCA</td>
<td>Host Channel Adapter</td>
</tr>
<tr>
<td>HPC</td>
<td>High-Performance Computing</td>
</tr>
<tr>
<td>IAS</td>
<td>Institute for Advanced Study</td>
</tr>
</tbody>
</table>
IB  InfiniBand
IDE  Integrated Development Environment
IEEE  Institute of Electrical and Electronics Engineers
IP  Internet Protocol
IPC  Inter-Process Communication
IPDPS  International Parallel & Distributed Processing Symposium
IPoIB  Internet Protocol over InfiniBand
IPVS  Institut für Parallele und Verteilte Systeme
ISC  International Supercomputing Conference
ISPASS  International Symposium on Performance Analysis of Systems and Software
I/O  Input/Output
LCM  Local Communication Map
LES  Large Eddy Simulation
LVD  Local Vertex Data
LVDA  Local Vertex Data Array
LVDI  Local Vertex Data Index
LVDO  Local Vertex Data Offset
MCT  Model Coupling Toolkit
MPI  Message Passing Interface
MPP  Massively Parallel Processing
MpCCI  Mesh-based parallel Code Coupling Interface
MUSCLE  Multiscale Coupling Library and Environment
NAS  Network-Attached Storage
NI  Network Interface
NUMA  Non-Uniform Memory Access
ONELAB  Open Numerical Engineering LABoratory
ONERA  Office National d’Etudes et Recherches Aérospatiales
OS  Operating System
PDE  Partial Differential Equations
P2P  Process-to-Process
preCICE  Precise Code Interaction Coupling Environment
RDMA  Remote Direct Memory Access
RMI  Remote Method Invocation
RPC  Remote Procedure Call
SCAI  Institute for Algorithms and Scientific Computing
SDP  Sockets Direct Protocol
SIDL  Scientific Interface Definition Language
SO  Shared Object
TCP  Transmission Control Protocol
TUM  Technische Universität München
VAPI  Verbs Application Programming Interface
VD  Vertex Distribution
WCCM  World Congress on Computational Mechanics
INTRODUCTION

When a software solution becomes overly complex to maintain and develop further, the basic “divide and conquer” principle of computer science suggests that it should be decomposed into smaller manageable modules, preferably with low coupling and high cohesion. Application of this principle to numerical simulations of multiphysics scenarios naturally leads to decomposition into different domains on the basis of governing physical laws. Solution of the corresponding equations for each domain separately and further combination of these solutions into a complete solution is best known as the partitioned coupling approach. For instance, partitioned numerical simulations of fluid–structure interaction (FSI) phenomena are characterized by decomposition into fluid and solid domains according to the governing physical laws of fluid dynamics and structural mechanics respectively.

Since large-scale numerical simulations of multiphysics scenarios have become a common daily task in the field of scientific computing during the past decade, the partitioned coupling approach has emerged as a well-established trend. Its key advantages over the “traditional” monolithic coupling approach are

— flexibility in choice of actual simulation software;
— ability to reuse existing (mature) simulation software for each particular physical law;
— possibility to combine different state-of-the-art (potentially more efficient) simulation techniques which have been developed specifically to solve each particular physics problem;
— configurability potential of interaction process between chosen simulation software through a separate generic (reusable) software component;
— encapsulation of the complexities associated with (partitioned) coupling in a separate generic (reusable) software component;
— fine control over computational complexity due to separate discretizations of each domain;
— better conditioning of the resulting system matrices in general [“2.1 Coupling approach”, pp. 998–1001].

For example, in its most general form, the numerical modeling of FSI requires the concurrent application of techniques from the fields of computational fluid dynamics (CFD), computational structural dynamics (CSD), and computational mesh dynamics (CMD). In fact, FSI problems require numerical coupling of the fundamental equations from each of these three fields. The monolithic coupling approach is sometimes suitable for relatively small problems, but is usually labor intensive and entails considerable memory footprints. Besides, the complete set of algebraic equations associated with the monolithic coupling approach may be quite stiff because the inherent properties and characteristic scales of the CFD, CSD, and CMD equations often differ significantly from each other. On the other hand, the partitioned coupling approach requires relatively smaller amounts of memory and employs simulation techniques which are uniquely designed for each of the three fields — CFD, CSD, and CMD.

sometimes also called weak coupling approach or loose coupling approach [“2.1 Coupling approach”, pp. 998–1001]

sometimes also called strong coupling approach or tight coupling approach [“2.1 Coupling approach”, pp. 998–1001]
The three essential facilities are necessary to constitute a coupling software component: (1) equation coupling, (2) data mapping, and (3) data intercommunication. Thus, it is clear that for a coupling software component to produce minimum overhead during simulation, it is important that all three facilities run efficiently and rely on parallelism where possible. Additionally, in order to fully exploit the inherent flexibility of the partitioned coupling approach, a coupling software component should be able to treat the actual third-party simulation software (physics solvers) as black boxes. This also implies that minimal efforts are required to couple a new solver.

According to [2, “2.6.2 Categorization of Approaches”, pp. 95–96], coupling software can be categorized as following either
1. library approach — provides API that has to be used at appropriate places in the code to be coupled, or
2. framework approach — predefines a fixed set of methods to be implemented with proper solver functionality.

While the former yields low intrusiveness and simple integration into an existing structure of code, the latter more intelligibly establishes a certain structure and form of functionality to be implemented. Furthermore, [2, “2.6.2 Categorization of Approaches”, pp. 95–96] distinguishes between low-level and high-level application programming interfaces (APIs) of coupling software. The former implies that data intercommunication routines have to be used directly in the solver code, the coupling control might not be given as ready-to-use, and other functionality is given only in rudimentary form to be explicitly combined by users in a suitable way. The latter might completely hide data intercommunication phase and other coupling methods from the solver code; the actual coupling functionality performed within a high-level method is configured outside of the code. Thus, while a low-level API gives more freedom on how to implement the coupling, a high-level API reduces the integration efforts and improves interoperability between different codes. Finally, coupling software can be classified as application-generic or application-specific. [2, “2.6.2 Categorization of Approaches”, pp. 95–96]

Precise Code Interaction Coupling Environment (preCICE) [2–6] is a modern code coupling library for partitioned simulations of multiphysics scenarios. It was initially developed by Bernhard Gatzhammer at the Technische Universität München (TUM) and further extended by Benjamin Ueckermann at the TUM Institute for Advanced Study (IAS) and Florian Lindner at the Institut für Parallele und Verteilte Systeme (IPVS) of the Universität Stuttgart. Currently, preCICE is mostly focused on FSI [2, 5], fluid–acoustics interaction (FAI) [4], fluid–structure–acoustics interaction (FSAI) [3], fluid–fluid interaction (FFI) [3, 4], and acoustics–acoustics interaction (AAI) [6], but as it is tailored to be application-generic from scratch, practically, it has no hard limitations to couple other multiphysics scenarios.

The four main software design goals of preCICE are summarized in [2, “3.1 Software Design Goals”, pp. 97–98] as follows:
1. Minimize the effort of preparing an existing solver for partitioned coupling.
2. Maximize the coupling flexibility coming with the partitioned approach.
3. Provide a surface geometry interface with query functionality for fixed-grid solvers.
4. Provide a maintainable and extensible platform for research environments.

Besides, [2, “3.2.1 Functional Requirements”, pp. 99–100] defines the five essential functional requirements to preCICE: (1) equation coupling, (2) data mapping, (3) data intercommunication, (4) coupling control, and (5) geometry interface. Additionally, [2, “3.2.2 Non-Functional Requirements”, pp. 100–101] defines the five basic non-functional requirements to preCICE: (1) coupling flexibility, (2) minimal invasiveness, (3) maintainability, (4) expandability, and (5) portability. Finally, synthesizing [2, “2.6.2 Categorization of Approaches”, pp. 95–96], [2, “3.1 Software Design Goals”, pp. 97–98], [2, “3.2.1 Functional Requirements”, pp. 99–100], and [2, “3.2.2 Non-Functional Requirements”, pp. 100–101], the following design decisions for preCICE have been made in [2, “3.3.1 Design Decisions”, pp. 101–102]:
1. Use a library approach (in favor of a framework approach).
2. Provide a high-level coupling API.
3. Employ generic concepts as much as Design Goal 1 is not contradicted.
4. Use C++ as programming language.

This thesis focuses on improvement of the data intercommunication functionality of the preCICE library as far as flexibility, quality, reliability, portability, and (most importantly) efficiency are concerned. The survey of similar existing coupling software in Chapter 1 suggests that the state-of-the-art high-performance data intercommunications between the coupled intraparallel solvers are done in the process-to-process (P2P) manner either through an intermediate intraparallel coupling server or directly between the processes of the coupled (intraparallel) solvers, preferably using asynchronous (non-blocking) intercommunication routines.

Prior the efforts of this thesis, preCICE only offered simple and not optimized implementations of the centralized intercommunication model (CICM) between the processes of the coupled (intraparallel) solvers, that is they had to forward all the (data) intercommunication either through a single coupling server process or through their respective master processes. Furthermore, both intra-solver and inter-solver communications were based on synchronous (blocking) communication routines exclusively, what deteriorated performance capabilities of preCICE even more. Eventually, performance testing has revealed that such implementations indeed result in a severe data throughput bottleneck which leads to a dramatic loss of scalability for medium- and large-sized runs of coupled simulations. Evolution of the preCICE intercommunication techniques and issues associated with them are discussed in more detail in Chapter 2.

The major goal pursued by this work is to completely eliminate the intercommunication bottleneck and yield nearly ideal strong and weak scaling of the preCICE intercommunication routines. The respective motivation, related challenges, functional requirements, and design decisions are given in Chapter 2. The goal was successfully accomplished by introducing the distributed intercommunication model (DICM) into preCICE, which is based upon a fully parallel P2P intercommunication technique directly between the coupled (intraparallel) solvers, using the newly implemented asynchronous (non-blocking) intercommunication routines. Besides,
another prime objective that was fulfilled is that the implementation of the P2P intercommunication technique transparently supports both the TCP/IP and the MPI network communication standards. The comprehensive overview of the corresponding implementation details can be found Chapter 3.

The performance achievements have been verified and confirmed by profiling one scalability benchmark and one (real-world) application scenario, which involve coupling of the actual third-party simulation software, on the SuperMUC massively parallel system [7]. The scalability benchmark, where the “symmetric” FFI phenomenon is simulated by coupling the two Ateles [3, 4, 8–10] solvers via preCICE, has proven the ideal strong scaling capabilities of the implemented P2P intercommunication technique at least up to 32768 processes overall (16384 processes per each Ateles solver accordingly). The application scenario, where the “asymmetric” FSI phenomenon is simulated by coupling the two Alya [5, 11] solvers via preCICE, has proven that the overhead of the implemented P2P intercommunication technique scales and is infinitesimally small, compared to the rest of the simulation labor, even in case of the imposed asymmetry and throughput imbalance of the intercommunication pattern. Chapter 4 presents detailed descriptions of the test cases used for verification, the actual third-party simulation software used for coupling via preCICE, and the corresponding performance results supplemented with thorough analysis and explanations.
1 Coupling Software Review

In this chapter a more or less detailed survey of some existing modern software that provides approximately the same functionality and/or has similar goals as preCICE does is presented. It should be noted that because this work is centered around data communication techniques in the context of partitioned coupling for large-scale multiphysics simulations, the following reviews will be mainly focused on the communication capabilities of coupling tools, libraries, or frameworks under consideration. The aim of this chapter is to analyze and evaluate the design decisions and the implementation details made by other projects regarding data communication methods, especially reflecting on massively parallel processing (MPP). Please, understand that the bulk of the information given here is taken from publications, presentations, user manuals, web pages, or brief experiences with code bases of the corresponding software. For additional overview, consult [2, “2.6 Existing Coupling Software for Partitioned Fluid–Structure Interaction”, pp. 92–96].

1.1 Advanced Scientific Computing Development Toolkit (ASCoDT)

The Advanced Scientific Computing Development Toolkit (ASCoDT) [12, 13] is an open-source integrated set of tools that allows development, deployment, and execution of component-based HPC applications with the emphasis on the field of scientific computing. Some of its key features are

— continuous integration — ASCoDT is implemented as an Eclipse IDE plugin [14], what allows its seamless integration into the development cycle;

— components are defined by means of a rigorous subset of the Scientific Interface Definition Language (SIDL) [15, 16], and the Eclipse plugin generates the corresponding glue code for C, C++, Java, or Fortran programming languages;

— first-class support for Remote Method Invocation (RMI), the object-oriented equivalent of Remote Procedure Call (RPC) [17], allows for the (easier) development of parallel distributed applications;

— thanks to well-designed architecture of abstraction layers, from the developer’s perspective there is code transparency between localized (when a component is built as a shared object (SO) for Unix-based platforms and as a dynamic-link library (DLL) for Microsoft Windows-based platforms) and distributed (when a component is built as a standalone executable) execution models.

All these features and many others indeed help to deal with intensively growing complexity of HPC applications in the field of scientific computing. However, in context of partitioned coupling, ASCoDT has significant downsides, especially concerning the intercommunication between intraparallel components.

First of all, although, the many-to-many intercommunication is indeed possible within ASCoDT, it is not available in a user-friendly and ready-to-use form by default, but has to rather
be manually implemented by application programmer using the generated glue code of the process-to-process RMI facilities.

Secondly, the only underlying intercommunication implementation is the TCP/IP socket-based one (i.e. the MPI port-based one is not supported). This reduces flexibility when deploying to a massively parallel environment where highly-optimized MPI implementation is available.

Third, selection and assignment of TCP ports is done with the trivial scheme: only the “base” TCP port, which corresponds to the master process of (intraparallel) component, can be configured, while each of the rest of the processes of (intraparallel) component is assigned the TCP port equal to the sum of the “base” TCP port and the corresponding process rank. Clearly such an approach is impractical for production runs due to possibly high risks of collision with TCP ports that are already in use by other applications (see Section 3.2 on how this problem was solved in preCICE).

Fourth, the way ASCoDT manages intercommunication parallelism in each process of (intraparallel) component is inflexible and likely inefficient. In fact, it allocates the fixed number of “worker” threads which continuously listen to their respective sockets for incoming RMI signals. There is nothing particularly harmful for toy applications here, but for real-world HPC applications this approach is merely far from being mature for the following reasons:

— Complexity — it implicitly requires application programmer to manually manage synchronization of those (internal) “worker” threads since the corresponding RMIs will occur exactly in those “worker” threads.

— Inflexibility — running a fixed number of “worker” threads is simply inappropriate for scenarios when a number of possible incoming connections is either not known beforehand or changes dynamically (at run time).

— Inefficiency — running a fixed number of “worker” threads can quite negatively impact performance and waste precious resources such as memory in general. For example, this means that at least a very careful distribution of MPI-based (intraparallel) components across CPUs and/or CPU cores in a massively parallel environment has to take place and could potentially lead to outrageously cumbersome distribution patterns.

Fifth, same socket objects cannot be reused to do both, send RMI signals and receive RMI signals. In other words, separate socket objects have to be created, what can again be considered a waste of resources.
Last but not least, there is no mature deployment support for standalone distributed components (executables). This job is again forwarded to application programmer and may cause a great deal of trouble to set up, configure, and maintain.

### 1.2 ADVanced ENgineering analysis Tool for Ultra large REal world (ADVENTURE)

The **ADVanced ENgineering analysis Tool for Ultra large REal world** (ADVENTURE) [18] is an open-source general-purpose computational mechanics system for large-scale analysis and design. Recently, a supplementary FSI coupling tool called **ADVENTURE_Coupler** has been developed [19]. The corresponding communication model of ADVENTURE_Coupler is summarized in Figure 1.1. The intracommunication model follows the classic approach based on MPI, while the intercommunication model appears to be a more advanced variation of the preCICE client/server intercommunication technique (Subsection 2.1.1) in a sense that, in this case, the server allocates \((m + n)\) MPI processes each of which dedicates itself to either one of \(m\) processes of fluid solver or one of \(n\) processes of solid (structure) solver.

![Communication Model of ADVENTURE_Coupler](image)

In fact, ADVENTURE_Coupler plays the role of a parallel coupling server itself. The data intercommunication and the interpolation of the interface variables between the two coupled solvers are executed within the server MPI processes. Handling of the case of non-matching grids between the two (interface) meshes is supported too [19]. Certainly, this strategy with a parallel coupling server employed by ADVENTURE_Coupler can solve the data throughput problem that preCICE is facing with its current implementation of the coupling server. Nevertheless, several possible disadvantages to this approach should be noted.

For example, running a separate parallel coupling server in addition to coupled solvers obviously doubles the overall number of processes involved. When deployed to a massively parallel environment this might lead to problems with load balancing of CPUs and/or CPU cores if the

---

Footnote: which don’t depend on Eclipse plugin to manage their proper interconnection and execution
The distribution of both the solver and the server processes across the computing nodes is not carefully arranged. In particular, assuming that the coupled solvers and the coupling server exhibit roughly even load balancing across their respective MPI processes, one has only two feasible options how to properly distribute all these processes across CPU cores of a massively parallel environment:

1. map (pin) each process to a (separate) single dedicated CPU core, i.e. \(2 \cdot (m + n)\) free CPU cores in total are necessary to run the simulation;

2. map (pin) both each process of a coupled solver and its corresponding process of the coupling server (Figure 1.1) to a (separate) single dedicated CPU core, i.e. only \((m + n)\) free CPU cores in total are necessary to run the simulation.

Thus, it is clear that the (possibly) severe disadvantage of **Option 1** is the doubling of the total number of CPU cores required. This might be troublesome when

— the budget of available computational resources is limited;

— the processes of either the coupled solvers or the coupling server are unable to fully load their respective dedicated CPU cores, what is very likely to be the case especially for the processes of the coupling server because when the initialization phase is finished, the coupling server does not do much computational work, but mostly does the data interpolation and forwards all the data intercommunication between the coupled solvers, and that is unlikely to be enough to fully load modern CPU cores in an overwhelming majority of cases; hence, at least half of the dedicated CPU cores may be wasted in terms of efficient utilization of their computational resources;

— \(n\) and \(m\) include not only interface processes\(^{14}\) but also the rest of the processes\(^{15}\), what would make the waste of computational resources even more dramatic.

The main disadvantage of **Option 2** is the nontrivial difficulty to properly define such a mapping. In other words, it is a pure usability problem.

To wrap it up, it is worth to mention that all of these issues are avoided by preCICE with two simple design decisions to

1. implement it as a library that should be directly linked into the coupled (intraparallel) solvers;

2. migrate the (partitioned) coupling server responsibilities into the respective processes of the coupled (intraparallel) solvers.

This can be thought of as **Option 2** but without the usability problem described above since each process already combines both the numerical computations\(^{16}\) and the data intercommunication\(^{17}\) (Section 2.2).

One potential advantage of ADVENTURE_Coupler over preCICE is that while the dedicated coupling server is busy, the coupled solvers could be free to do some other useful work.

---

\(^{12}\) which would not introduce load balancing problems

\(^{13}\) e.g. a possibility of simply running out of free CPU cores

\(^{14}\) which are responsible for computations over those mesh parts that are located at the coupling interface

\(^{15}\) unclear from \(^{19}\)

\(^{16}\) done by the solver routines

\(^{17}\) done by the preCICE routines
1.3 Coupling With Interpolation Parallel Interface (CWIPi)

The Coupling With Interpolation Parallel Interface (CWIPi) [20–24] is an open-source library that aims at providing a fully parallel intercommunication layer for mesh-based coupling between multiple parallel codes via MPI (i.e. TCP/IP is not supported unlike in preCICE). Like ADVENTURE_Coupler (Section 1.2), EMPIRE (Section 1.4), MpCCI (Section 1.5), OASIS (Section 1.6), preCICE [2], and most of the other existing libraries with multiexecutable coupling approach, CWIPi is a static coupler [21, 23], i.e. all of the simulation participants are launched together at the beginning of the simulation, intercommunicate data during the simulation phase, and terminate together at the end of the simulation. Since the API is available in C, C++, or Fortran programming languages, CWIPi can couple codes written in different programming languages.

Linear, surface, or volume couplings are available. CWIPi is capable of working with all types of geometrical elements; there is no hard requirement about the mesh structure. CWIPi has the following obligations:

— control of coupled processes;
— construction of the intercommunication graph between the distributed mesh interfaces through geometrical localization;
— interpolation of non-matching grids between the two (interface) meshes;
— efficient intercommunication of coupling fields;
— building of visualization files.

As preCICE, CWIPi also supports asynchronous communication [22, “16.7.2 Asynchronous communication”, p. 112]. According to [24, “Box 2 — CWIPi”, p. 7], the coupled participants must be launched in the same MPI environment because during the initialization phase, CWIPi creates a separate MPI intracommunicator for each participant. For every coupling defined by the user, the corresponding MPI intercommunicators are created between the coupled participants accordingly. Thus, from the user’s perspective, the intercommunications become completely transparent even if the participants are parallelized (with MPI) themselves. In fact, these functionality concepts are very similar to those of preCICE, but their implementation is slightly more generic in preCICE thanks to the well-designed class hierarchy of communication abstractions (Chapter 3). This allows preCICE to avoid the flexibility disadvantage of CWIPi that the coupled participants must be launched in the same MPI environment. In other words, when preCICE is used for coupling, the coupled participants are free to be launched in either the same or different MPI environments regardless of whether the chosen underlying intercommunication implementation is based on TCP/IP sockets (Section 3.2) or MPI ports (Section 3.3).

---

18 also known as solvers, components, or executables
line, polygon, polyhedral, etc.

20 something similar to the preCICE concept of LCMs (Subsection 3.6.4)

21 something similar to the preCICE data mapping [2, “4.2 Data Mapping for Non-Matching Grids”, pp. 141–151]

22 optimized for a massively parallel environment
It should be noted that the CWIPI library is the part of OpenPALM (Section 1.7), a more advanced and comprehensive code coupling solution that is arguably one of the leaders in the industry.

### 1.4 Enhanced Multi Physics Interface Research Engine (EMPIRE)

The Enhanced Multi Physics Interface Research Engine (EMPIRE) [25, 26] is an open-source code coupler that enables co-simulation with n-codes to solve general multiphysics problems. As shown in Figure 1.2, EMPIRE is divided into two components:

1. The Emperor code coupling server — couples all the participating simulation codes together to perform co-simulation.
2. The EMPIRE_API library — delivers functions for connection establishment with Emperor and intercommunication of various data (e.g. mesh, degree of freedom, signal, etc.) with Emperor.

![Co-Simulation with EMPIRE](image)

In fact, EMPIRE follows the client/server intercommunication technique with Emperor acting as the code coupling server and the simulation codes linked against the EMPIRE_API library acting as the clients. The intercommunication between the server and the clients is performed using MPI ports (i.e. TCP/IP sockets are not supported unlike in preCICE).

### 1.5 Mesh-based parallel Code Coupling Interface (MpCCI)

The Mesh-based parallel Code Coupling Interface (MpCCI) [27–29] is a commercial code coupling environment for partitioned (multiphysics) simulations. It implements the intercommunication of data between the meshes of two (or more) simulation codes participating in a coupled simulation. Since the meshes belonging to different simulation codes are not compatible in general, MpCCI implicitly performs an interpolation. In case of intraparallel codes, MpCCI keeps track of the distribution of the domains across the parallel processes. MpCCI supports the intercommunication of nearly any kind of data between the coupled codes, e.g. energy, momentum,
material properties, mesh definitions, or global quantities. The details of the data intercommunication are hidden behind the concise API of MpCCI. The architecture of MpCCI workflow is summarized in Figure 1.3.

![Figure 1.3](image)

With MpCCI the intercommunication between the coupled codes is done through the MpCCI coupling server on behalf of code adapters for each simulation code participating in a coupled simulation (Figure 1.3). These code adapters are nothing else but auxiliary libraries which have to be linked to the respective simulation codes either statically or dynamically. A code adapter itself consists of three major parts (Figure 1.3):

1. The **Coupling Manager** — this generic part is used in each code adapter and is responsible for a consistent run-time behavior. It reads the application-specific model configuration and (during coupling process) controls which model regions have to intercommunicate which boundary values with the other coupled code.

2. The **MpCCI (Intercommunication) Client** — the small module that implements the intercommunication through a distributed heterogeneous network. According to the client/server intercommunication technique of MpCCI (Figure 1.4), the clients are directly connected to the MpCCI coupling server using TCP/IP sockets (i.e. MPI ports are not supported unlike in preCICE).

3. The **Code Driver** — acts as an interface between the data structures of the coupled code and the MpCCI coupling manager. The access to the coupled code’s data can be implemented through e.g. Fortran common blocks, C subroutines, etc. The code driver is the only code-specific part in the adapter.
Another brief review of MpCCI can be found in [2, “2.6.1 Software Projects”, p. 93].

### 1.6 OASIS3-MCT

The OASIS coupler [23, 30] is an open-source code coupler for massively parallel climate modeling. **OASIS3-MCT**, the new version of the OASIS coupler based on the Model Coupling Toolkit (MCT) [31], offers a fully parallel implementation of coupling field interpolation and intercommunication. Low-intrusiveness, portability, and flexibility are claimed to be the key design concepts as for all previous OASIS versions. The important difference with respect to the previous OASIS3 version is that there is no separate coupler executable any longer, i.e. OASIS3-MCT is now a coupling library that has to be directly linked to the **component models** (solvers), with the main purpose of interpolating and intercommunicating the coupling fields between these (coupled) component models. As a result, it turns out that the OASIS coupler has also evolved in the direction similar to the one that preCICE is following now, i.e. getting rid of the coupling server to avoid the issues possibly haunting ADVENTURE_Coupler (Section 1.2) and other performance-related problems [23].

OASIS3-MCT supports coupling of general two-dimensional fields. Unstructured grids and three-dimensional grids are also supported through a one-dimensional representation of the two- or three-dimensional data structures. Thanks to MCT, all transformations, including regridding (remeshing), are executed in parallel on the set of **source** or **target** component processes. Furthermore, all coupling intercommunication is now executed in parallel directly between the (coupled) components via MPI (i.e. TCP/IP is not supported unlike in preCICE). In addition, OASIS3-MCT also supports file I/O using **NetCDF**. In the current version, however, the implementation of this functionality is not parallelized, i.e. the read/write operations of the fields are performed by the master processes only.
1.7 OpenPALM

The OpenPALM software \cite{21–23, 32–34} is an open-source dynamic code coupler for massively parallel simulations of multiphysics and general multiexecutable HPC applications that require extensive data intercommunication during their execution. The key feature of OpenPALM, compared to other coupling solutions presented in this survey and preCICE as well, is dynamic coupling, i.e. facilities for scheduling of participants execution either sequentially or concurrently. In other words, with OpenPALM, a participant can be launched and terminated at any moment in time during the simulation, while available computational resources (required memory and the number of dedicated CPUs and/or CPU cores) are handled by the coupler. This allows one to formulate the so-called coupling algorithms of arbitrary complexity in order to implement certain simulation scenarios where the participants execution order and the data intercommunication patterns cannot be completely defined before execution.

Since the API is available in C, C++, or Fortran programming languages, OpenPALM can couple codes written in different programming languages. Furthermore, OpenPALM itself consists of three complementary components:

1. The CWIPI library (Section 1.3) — provides a fully parallel (process-to-process) intercommunication technique \cite{21, 33} for mesh-based coupling between multiple parallel codes via MPI (i.e. TCP/IP is not supported unlike in preCICE).

2. The PALM library \cite{21–23} — provides the framework to split applications into elementary components and adapt them to dynamic coupling; implements and handles all the scheduling responsibilities described above; manages synchronization, connection, and intercommunication between the coupled units. With regards to intercommunication of PALM with external codes \cite{22, “17 Connection of an external code to a PALM application”, pp. 115–129], similarly to preCICE (Chapter 3), PALM supports both MPI ports \cite{22, “17.2 How it works”, p. 116} and TCP/IP sockets \cite{22, “17.5 To go further: IP connection of an external code”, pp. 122–129].

3. The PreP ALM application \cite{21–23} — provides GUI to ease the definition of the coupling schemes, the execution algorithms, the parallel sections, the data intercommunication patterns, and the algebraic treatments. After all configurations are finished, it exports the input configuration file for the coupler executable and the source code of the wrappers of the coupled components. This GUI application can also be used at run time to monitor the simulation status, perform the post-processing, and provide some profiling statistics.

1.8 Other Software

Aside from the software that has been discussed in Chapter 1 and \cite[“2.6 Existing Coupling Software for Partitioned Fluid–Structure Interaction”, pp. 92–96]{2}, there exist other projects when participants can be executed in loops or under logical conditions (even based on the results returned by one or many participants) which take care to set up the intercommunication context properly without a need to modify the source code of the coupled components.
which seem to share some common goals with preCICE (at least partially) and could be worth looking into at some point:

— Multiscale Coupling Library and Environment (MUSCLE) [35, 36];
— Open Numerical Engineering LABoratory (ONELAB) [37];
— SALOME [38].
2 Background

This chapter summarizes theoretical considerations, previous experiences (from the software discussed in Chapter 1 and preCICE itself), and possible issues of various intercommunication models along with their corresponding realization techniques in context of code coupling for partitioned (multiphysics) simulations.

2.1 The Centralized Intercommunication Model (CICM)

In a general sense, the centralized intercommunication model (CICM) is characterized by the mandatory existence of a central entity that provides intercommunication service for consumer entities which have to exchange data between each other. That is this central entity acts as an intermediary, properly forwarding all of the intercommunication requests between consumer entities. Most of the software reviewed in Chapter 1, ADVENTURE_Coupler (Section 1.2), EMPIRE (Section 1.4), MpCCI (Section 1.5), and OpenPALM (Section 1.7), employ CICM in a certain form. Prior efforts of this thesis (Chapter 3), the only available intercommunication model in preCICE was CICM too.

2.1.1 The Client/Server Intercommunication Technique

The client/server intercommunication technique is a classic approach to implement CICM. Its current implementation within preCICE is thoroughly described in [2, “3.4.2 Parallel Deployment”, pp. 106–108], and the corresponding sketch is illustrated in Figure 2.1.

![Figure 2.1 The Client/Server Intercommunication Technique](image-url)
In fact, not only preCICE but also ADVENTURE Coupler (Section 1.2), EMPIRE (Section 1.4), MpCCI (Section 1.5), and OpenPALM (Section 1.7) follow similar approach. The key factor of client/server intercommunication performance is whether the server handles intercommunication forwarding between the clients in parallel or not. For instance, ADVENTURE Coupler, MpCCI, and OpenPALM have their code coupling servers parallelized, while EMPIRE and preCICE not. Usage of synchronous (blocking) communication methods exclusively further aggravated the intercommunication performance problem in preCICE. The primary reason why data intercommunication was not parallelized on the preCICE server adapter (Figure 2.1) in the first place was that the other two essential operations, equation coupling and data mapping (Figure 2.1), were not yet parallelized as well [2, “3.4.2 Parallel Deployment”, pp. 106–108]. All of these shortcomings resulted in the performance bottleneck (Figure 2.2) for the same test case as the one exercised in Subsection 4.3.2.

![Figure 2.2](image1)

**Figure 2.2** The preCICE Performance Bottleneck Induced by the Non-Parallel Client/Server Intercommunication Technique (Strong Scaling) [3, 4]

### 2.1.2 The Master/Slave Intercommunication Technique

Following the general ideas for improvements previously formulated in [2, “6.2 Ideas for Extensions”, pp. 245–246], both equation coupling and data mapping were parallelized. Furthermore, for the reasons argued in Section 1.2 (using ADVENTURE_Coupler as an example) and [2, “3.4.1 General Considerations”, pp. 104–105], it was decided not to implement this parallelization in a server adapter (Figure 2.1), but rather get rid of it completely in favor of integrating all of the coupling functionality into the coupled (intraparallel) solvers themselves (Section 1.2). This is how preCICE evolved to the so-called master/slave intercommunication technique, another popular variant of CICM depicted in Figure 2.3.

![Figure 2.3](image2)
As a result of this effort, all three feature groups (equation coupling, data mapping, and data communication) are executed over the distributed (across the processes of the coupled solvers) data. This was a huge leap forward that drastically reduced preCICE impact on the performance of coupled simulations. Nevertheless, the issue with sequentialization of data intercommunication still remained. That is data intercommunication was done in a gather/scatter manner, using synchronous (blocking) communication methods, with master processes acting as forwarders (Figure 2.3), similarly to server adapters (Figure 2.1). Thus, the data throughput bottleneck, which would limit bandwidth between the coupled solvers, remained.

2.2 The Distributed Intercommunication Model (DICM)

In a general sense, the distributed intercommunication model (DICM) allows entities to directly (without a need for a central entity) exchange data between each other. The two key advantages of DICM over CICM (Section 2.1) are:

1. Elimination of intermediary intercommunication forwarder(s), i.e. the data does not have to be intercommunicated multiple times until it reaches the destination entity, but is rather directly transferred from the source entity to the destination entity. This was also mentioned in [2, “3.4.1 General Considerations”, pp. 104–105].

2. Division of the intercommunication load among all of the entities, so that they all contribute some processing power and memory. Besides, intraparallel nature of the coupled solvers could be reused to parallelize data intercommunication over the distributed data more efficiently (compare Section 1.2).

Among the software presented in Chapter 1, ASCoDT (Section 1.1), CWIPI (Section 1.3), and OASIS3-MCT (Section 1.6) follow DICM.

2.2.1 The Process-to-Process (P2P) Intercommunication Technique

The introduction of the process-to-process (P2P) intercommunication technique was the next natural evolutionary step from CICM (Section 2.1), based on the master/slave intercommuni-
cation technique (Subsection 2.1.2), to implement DICM in preCICE. The general illustration of the P2P intercommunication technique is shown in Figure 2.4.

Within this approach, data intercommunication between the processes of (intraparallel) solvers is supposed to be fully parallel. That is the parallelism should come not only from the fact that the processes of solvers can run in parallel themselves but also from parallelization of intercommunication requests inside each of the processes in the first place. This can only be achieved if asynchronous (non-blocking) communication routines are utilized. Otherwise, each process has to execute a loop of synchronous (blocking) intercommunication requests which would continuously block each other. Hence, at the very best, only as many intercommunication requests could be executed in parallel as there are intercommunicating processes running in total, while in reality, as it heavily depends on the intercommunication pattern (which is usually quite complex in practice), most of the intercommunicating processes would spend decent amounts of time simply being blocked and waiting for some other processes to deal with their intercommunication requests. Furthermore, in order to efficiently utilize the underlying networking hardware (such as network interface controllers, host adapters, interconnect, etc.), the bandwidth must be kept as close to the corresponding saturation level as possible. Therefore, overlapping as many asynchronous (non-blocking) communication requests as possible (in a single intercommunication session) ensures that this objective is fulfilled. Consequently, without using asynchronous (non-blocking) communication routines for implementation of the P2P intercommunication technique, its performance benefits are at least minimal and at most questionable at all (e.g. under some conditions even the master/slave intercommunication technique could do better). This issue was also very well examined and solved accordingly for OpenPALM (Section 1.7) in [21, “4.2. Communication phase”, pp. 9–13] and [33, “OpenPALM Direct communication scheme”, p. 7].
Intercommunication Deadlock

Importance of asynchronous (non-blocking) communication routines for implementation of the P2P intercommunication technique is also dictated by the risk of intercommunication deadlocks. For instance, consider the situation demonstrated in Figure 2.5 and assume that all data transfer operations are synchronous (blocking).

Figure 2.5 Example of The Process-to-Process (P2P) Intercommunication Deadlock

It turns out that the intercommunication deadlock is inevitable if the (synchronous) data transfer operations occur in the following order:

— The process $A_1$
  1. calls send operation to send some data via the intercommunication channel 1;
  2. calls send operation to send some data via the intercommunication channel 3.
— The process $A_2$
  1. calls send operation to send some data via the intercommunication channel 2;
  2. calls send operation to send some data via the intercommunication channel 4.
— The process $B_1$
  1. calls receive operation to receive some data via the intercommunication channel 4;
  2. calls receive operation to receive some data via the intercommunication channel 1.
— The process $B_2$
  1. calls receive operation to receive some data via the intercommunication channel 3;
  2. calls receive operation to receive some data via the intercommunication channel 2.

There are two possible solutions to this problem:
1. Continue to use synchronous (blocking) communication routines and enforce safe order of data transfer operations in all of the intercommunicating processes. Nevertheless, as described previously, synchronous (blocking) communication routines for implementation of the P2P intercommunication technique are more than suboptimal.
2. Utilize asynchronous (non-blocking) communication routines for at least one of the sides of intercommunication session, that is for either all sending processes or all receiving processes, while the other side may use either blocking or non-blocking communication routines.
2.3 Conclusion

Taking into consideration the motivation of Chapter 2, the issues described in Chapter 2, the state-of-the-art approaches to implement a fully parallel data intercommunication employed by similar existing coupling software reviewed in Chapter 1, and the personal experiences during the development process, it was decided to perform parallelization of the P2P intercommunication technique by overlapping as many asynchronous (non-blocking) communication requests per CIP to its intercommunication partners as possible (in a single intercommunication session) on both sender and receiver sides. The resulting implementation of P2P intercommunication technique is then expected to be efficiently parallelized, bandwidth-saturating, and proof against potential intercommunication deadlocks.
3 Implementation

This chapter contains implementation prerequisites and details of the fully parallel P2P intercommunication technique (Subsection 2.2.1) in preCICE according to analysis and conclusions made in Chapter 1 and Chapter 2. First, the updated architectural overview is given in Section 3.1. Second, the one-to-many TCP/IP socket-based communication implementation in Section 3.2. Third, the one-to-many MPI port-based communication implementation in Section 3.3. Fourth, the notion and implementation of asynchronous (non-blocking) communication requests are discussed in Section 3.4. Fifth, the concept and realization of one-to-many communication factories are explained in Section 3.5. Finally, the implementation details of the fully parallel (many-to-many) P2P intercommunication technique are disclosed in Section 3.6.

3.1 Architecture

A good architectural overview of communication methods implementation in preCICE has already been given in \citep[“4.3.1 Realization of Communication Methods”, pp. 152–157]{preCICE}. Nevertheless, thanks to efforts of this thesis, some new packages, classes, and interfaces have been added, while some implementation details of previously existing packages, classes, and interfaces have been modified. Thus, it is worth to provide the accordingly updated architectural overview, the complete picture of which is best visualized by means of class diagram in Figure 3.1.

As before, all of the (one-to-many) communication implementations are grouped in the precice::com package \citep[“4.3.1 Realization of Communication Methods”, pp. 152–157]{preCICE}. In order to decouple the rest of the preCICE code base from concrete communication mechanisms, the precice::com::Communication interface class is still used for the same purpose, but, following the rationale from Subsection 2.2.1, it has been supplemented with three different families of overloaded pure virtual methods:

1. aSend() — asynchronous (non-blocking) send operations;
2. aReceive() — asynchronous (non-blocking) receive operations;
3. broadcast() — synchronous (blocking) broadcast operations.

The two concrete classes implementing this interface class are precice::com::FileCommunication (unchanged) and precice::com::SocketCommunication (changed according to Section 3.2), while the only abstract class implementing this interface class is precice::com::MPICommunication (changed according to Section 3.3). Both precice::com::MPIDirectCommunication (unchanged) and precice::com::MPIPortsCommunication (changed according to Section 3.3) concrete classes use the send(), receive(), aSend(), aReceive(), and broadcast() implementations inherited from the precice::com::MPICommunication abstract class, i.e. they only differ in connection establishment methods (requestConnection() and acceptConnection()).
The `precice::com::Request` new interface class with two new implementing classes,
precice::com::SocketRequest and precice::com::MPIRequest, have been added. Their purposes and implementation details are clarified in Section 3.4.

The precice::com::CommunicationFactory new interface class with two new implementing classes, precice::com::SocketCommunicationFactory and precice::com::MPIPortsCommunicationFactory, have been added. Their purposes and implementation details are clarified in Section 3.5.

The precice::m2n new package has been added into preCICE, and its primary goal is to deliver concrete and interface classes realizing the many-to-many intercommunication techniques based on distributed data. Hence, the common interface class is precice::m2n::DistributedCommunication that is currently implemented by the two concrete classes, precice::m2n::GatherScatterCommunication corresponding to the master/slave intercommunication technique (Subsection 2.2.1) and precice::m2n::PointToPointCommunication corresponding to the fully parallel P2P intercommunication technique (Subsection 2.2.1). The design and implementation of the precice::m2n::PointToPointCommunication class and its prerequisites are two of the major efforts of this thesis, described in Section 3.6.

### 3.2 TCP/IP Sockets Communication

This section briefly presents the one-to-many communication implementation in preCICE on top of the TCP/IP layer. Before going into details, it should be noted that the BSD socket implementation is the root ancestor of all other existing socket implementations today. Basically, all other systems forked the BSD socket implementation at some point in time (or at least its interfaces) and then started to evolve it on their own. Of course the BSD socket implementation has been evolving as well at the same time, and thus, those systems which forked it later have some features that are lacking in those systems which forked it earlier. Nevertheless, understanding the BSD socket implementation is the key to understanding all other existing socket implementations, so it is important to consider its documentation even if there is no direct need to do network programming for a BSD system.

#### 3.2.1 Rationale

Unlike most of the coupling software discussed in Chapter 1, it has been deliberately chosen to support TCP/IP sockets in preCICE especially as an option for intercommunication between the coupled solvers. Compared to another option, MPI ports (Section 3.3), the two main reasons for this effort are:

1. **Portability** — the relevant subset of TCP/IP socket implementation is available in the vast majority of platforms, including massively parallel systems. Besides, there even exists the Internet Protocol over InfiniBand (IPoIB) protocol [39] that allows the Internet Protocol (IP) packets to be communicated via the high-performance interconnect, InfiniBand (IB), by encapsulating the IP packets in the IB packets, what makes transparent operation of TCP/IP socket-based applications over IB possible. For instance, the Linux kernel has the
ib_ipoib driver [40], which implements the IPoIB protocol. This is particularly relevant for this thesis since Linux is also the OS of the SuperMUC massively parallel system [7] that was used for verification (Chapter 4). Another merit of the TCP/IP stack is that, in contrast to MPI, there are no hard restrictions about which TCP/IP socket implementations in general and/or their versions in particular must be linked to the solvers in order to prevent run-time communication incompatibilities. As a result, it is still feasible to perform coupling and intercommunication through the TCP/IP layer instead of MPI in a situation when closed-source solvers, which have a mismatch of linked MPI implementations and/or their versions, are involved, and there is no way to relink them against matching MPI implementations.

2. **Reliability** — the TCP/IP stack is by far the most mature, bug-free, and stable implementation for network communication due to its widespread adoption and worldwide popularity. For example, the decision to support TCP/IP sockets for intercommunication has payed off immediately, even throughout the work on this thesis. In fact, all of the intercommunication during verification (Chapter 4) had to be done via TCP/IP instead of MPI because the two critical bugs pertaining to crashes in MPI client and server routines [41–43] had been revealed on SuperMUC in both the IBM and Intel MPI implementations simultaneously.

### 3.2.2 Basics

There are a couple of basics that are worth mentioning in the very beginning. A TCP connection is identified by a tuple of five parameters:

1. *protocol*,
2. *source address*,
3. *source port*,
4. *destination address*,
5. *destination port*.

In other words, any unique combination of these parameters unambiguously identifies a connection, and as a result, no two connections can have the same five parameters, otherwise the system would not be able to distinguish these connections any longer.

The protocol of a socket is set when a socket is created with the `socket()` function. The source address and port are set with the `bind()` function. The destination address and port are set with the `connect()` function. An unbound TCP socket is automatically bound before it will be connected.

### 3.2.3 Wildcards

If there is a need to explicitly bind a socket, then it is possible to bind it to the port 0, what essentially means "any port". Since a socket cannot really be bound to all existing ports, the system will have to automatically search and select a free port itself, usually from a predefined platform-specific range of available source ports. A similar wildcard exists for the source address, which can be "any address":

---

which could potentially result in an undefined behavior
1. $0.0.0.0$ for the IPv4 protocol,
2. :: for the IPv6 protocol.
Unlike in case of ports, a socket can really be bound to "any address", i.e. all available source IP addresses of all local NIs. If a socket is connected later on, the system has to automatically search and select a specific source IP address since a socket cannot be connected and be bound to any local IP address at the same time. Depending on a destination address and the content of a routing table, the system will choose an appropriate source IP address and replace the "any address" binding with a binding to the chosen source IP address.

### 3.2.4 Linger Time

The amount of time that OS will wait before it closes a socket, regardless of whether it still has pending data to be sent/received or not, is known as the linger time. The linger time is globally configurable on most OSs and by default is rather long, for instance, two minutes is a common value one may find on many OSs. Furthermore, it is also configurable per socket using the SO_LINGER socket option, which can be used to make the timeout shorter or longer and even to disable it completely. Disabling it completely is a very bad idea, though, since closing a TCP socket gracefully is a slightly complicated process and involves sending forth and back a couple of packets (as well as resending those packets in case they got lost), and this whole closing process is also limited by the linger time.

If lingering is disabled, a socket may not only lose pending data but it is also always closed forcefully instead of gracefully, what is usually not recommended. The details about how a TCP connection is closed gracefully are beyond the scope of this discussion. In any case, even if lingering is explicitly disabled with SO_LINGER and the process of a parent program terminates without explicitly closing the socket, on BSD (and most likely on other OSs too) the socket will linger nonetheless, ignoring what was configured. This will happen, for example, if a program merely calls exit() or std::abort() (pretty common for tiny toy server programs) or a process is killed by a signal (what includes the possibility that it simply crashes because of an illegal memory access). As a result, there is nothing one could do to make sure a socket will never linger under any circumstances.

### 3.2.5 Boost.Asio

*Boost.Asio* is an open-source cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach, e.g. helps to manage (heavy) I/O operations asynchronously without requiring programs to use “traditional” concurrency models based on threads and explicit locking. The overview of programming facilities delivered by Boost.Asio is summarized in Figure 3.2.
The Boost.Asio library is intended for systems programming with C++, where access to low-level OS functionality such as networking is of great need. In particular, Boost.Asio addresses the following goals:

— **Portability** — the library supports a range of commonly used OSs and ensures consistent behavior across these OSs.

— **Scalability** — the library facilitates the development of network applications that scale to thousands of concurrent connections, by exploiting native (for each supported OS) mechanisms that best achieve this scalability.

— **Efficiency** — the library supports advanced techniques such as scatter/gather I/O and allows programs to minimize data copying.

— **Familiarity** — the BSD socket API is widely implemented and understood, and is extensively covered in literature. Thus, as far as reasonable, Boost.Asio leverages the existing practice.

— **Ease of use** — the library provides a lower entry barrier for new users by taking a toolkit, rather than framework, approach. That is, it minimizes the up-front investment in time to just learning a few basic rules and guidelines.

Taking all of this into account, it was decided to implement the one-to-many TCP/IP socket-based communication on top of the Boost.Asio library.

### 3.2.6 The precice::com::SocketCommunication Class

The **precice::com::SocketCommunication** class implements communication in PRECICE via TCP sockets using the IPv4 protocol.

#### Constructors

There are two constructors available:
The **portNumber** parameter identifies the desired port to bind the underlying socket to. The default is port wildcard (0), and according to **Subsection 3.2.3**, it means that OS will automatically search and select a free port to bind the underlying socket to. In general, manually setting this parameter for typical use is strongly discouraged. The rationale behind this recommendation is that preCICE is primarily aimed at engineering audience, in which case explicitly dealing with such relatively low-level details as proper port manipulation could be both burdening and error-prone.

For example, naive manual configuration of **portNumber** might lead to the following issues:
1. Running an application, which uses `precice::com::SocketCommunication`, once and then running it again with the same **portNumber** being supplied and within the period of linger time (**Subsection 3.2.4**) will inevitably fail with error: “Accepting connection at port **portNumber** failed: bind: Address already in use”. This might turn out to be a very cumbersome limitation, especially during extensive testing.
2. Similar problem, but perhaps even more troubling, would occur when running an application not in a dedicated environment (i.e. alongside other third-party applications) because one would most likely be unsure whether the manually chosen **portNumber** is not bound by another application yet.

That is why the recommended default (0) for the **portNumber** parameter is an elegant and convenient solution that prevents both of these problems.

The **reuseAddress** flag indicates whether the address (a combination of port and IP address) of the underlying socket can be reused later to bind another socket, while the former one may still remain bound as well. This flag can solve **Issue 1**, however, it is considered a dangerous option and, in general, is not recommended to be set to **true**.

The **networkName** string specifies the name of a local NI to be used for communication. In fact, this name is used to query OS for the correct local IP address (that corresponds to this local NI), which then can be somehow *published* together with port (**portNumber**), so that other processes can *obtain* them in order to establish connection to the underlying socket exactly through the specified NI. Selecting the right NI can be crucial in terms of communication performance, especially when running applications in supercomputing environments. For instance, `ib0` is the name of a local NI dedicated for the IB interconnect on the SuperMUC massively parallel system, and failure to specify it might lead to a drastic letdown in bandwidth.

The **addressDirectory** string specifies the path to a directory where publishing/obtaining of addresses (combinations of ports and IP addresses) occurs. The default is the *current working directory* of a parent process (".").
Synchronous (Blocking) Connection Establishment Methods

There are two pairs of complementary synchronous (blocking) connection establishment methods:

```cpp
void acceptConnection(std::string const& nameAcceptor, 
                      std::string const& nameRequester, 
                      int acceptorProcessRank, 
                      int acceptorCommunicatorSize);
```

```cpp
void requestConnection(std::string const& nameAcceptor, 
                       std::string const& nameRequester, 
                       int requesterProcessRank, 
                       int requesterCommunicatorSize);
```

and

```cpp
void acceptConnectionAsServer(std::string const& nameAcceptor, 
                             std::string const& nameRequester, 
                             int requesterCommunicatorSize);
```

```cpp
int requestConnectionAsClient(std::string const& nameAcceptor, 
                             std::string const& nameRequester);
```

In order to establish connection between the two instances of the `precice::com::SocketCommunication` class, one of them must call the `acceptConnection()` (or `acceptConnectionAsServer()`) method, while another one must call the `requestConnection()` (or `requestConnectionAsClient()`) method. Both one-to-one (Figure 3.3) and one-to-many (Figure 3.4) connections are possible. Note that in case of one-to-many connection, the acceptor side always has one instance of the `precice::com::SocketCommunication` class, while the requester side can have many instances of the `precice::com::SocketCommunication` class.

Since in order for the requester side to request a connection to the acceptor side, it has to be aware of the IP address and the port of the acceptor side, as stated previously, address publishing and obtaining have to take place. For that purpose the `acceptConnection()` (or `acceptConnectionAsServer()`) method writes its address to an `address file` in the directory specified by the `addressDirectory` path. The address file has a special name pattern: `.<<nameRequester>-<nameAcceptor>.address`. Thanks to this name pattern, the `requestConnection()` (or `requestConnectionAsClient()`) method can locate the corresponding address file, read the address from it, and perform the intended connection request to the acceptor side.

---

26 further denoted as the acceptor side
27 further denoted as the requester side
28 given that `addressDirectory` on the requester side is identical to `addressDirectory` on the acceptor side
using precice::com::SocketCommunication;
SocketCommunication c;
c.acceptConnection(<A>, <R>, 0, 1);

Figure 3.3 One-to-One Connection Establishment Examples
for the precice::com::SocketCommunication Class

using precice::com::SocketCommunication;
SocketCommunication c;
c.requestConnection(<A>, <R>, 0, 1);

Acceptor Side   Requester Side

using precice::com::SocketCommunication;
SocketCommunication c;
c.acceptConnectionAsServer(<A>, <R>, 1);

c[0].requestConnectionAsClient(<A>, <R>);
c[1].requestConnectionAsClient(<A>, <R>);

Figure 3.4 One-to-Many Connection Establishment Examples
for the precice::com::SocketCommunication Class

Acceptor Side   Requester Side

Synchronous (Blocking) Communication Methods

There are two groups of complementary synchronous (blocking) communication methods:

1. Overloaded sending methods (send()), having the following properties:
   — accept a buffer (as an argument) to provide data to be sent;
   — implement blocking sending, that is they do not return until sending has finished;
   — after return the buffer can be immediately safely reused (read/write) further.
2. Overloaded receiving methods (receive()), having the following properties:

   either because OS has safely stored away (saved) the buffer’s data somewhere (for example, in the corresponding
   socket send buffer) or because the buffer’s data has already been successfully received by the destination side (does
   not necessarily imply that the corresponding receive() has finished, i.e. the data could simply be stored in the
   corresponding socket receive buffer on the destination side)
— accept a buffer (as an argument) to accommodate data to be received;
— implement blocking receiving, that is they do not return until receiving has finished;
— after return the buffer can be immediately safely reused (read/write) further.

Asynchronous (Non-Blocking) Communication Methods

There are two groups of complementary asynchronous (non-blocking) communication methods:

1. Overloaded sending methods ($\text{aSend}()$), having the following properties:
   — accept a buffer (as an argument) to provide data to be sent;
   — implement non-blocking sending, that is they do return immediately;
   — after return the buffer cannot be immediately safely reused (read/write) further;
   — return the corresponding instance of the $\text{precice::com::Request}$ interface class (Section 3.4), which is implemented by the $\text{precice::com::SocketRequest}$ class (Section 3.1).

2. Overloaded receiving methods ($\text{aReceive}()$), having the following properties:
   — accept a buffer (as an argument) to accommodate data to be received;
   — implement non-blocking receiving, that is they do return immediately;
   — after return the buffer cannot be immediately safely reused (read/write) further;
   — return the corresponding instance of the $\text{precice::com::Request}$ interface class (Section 3.4), which is implemented by the $\text{precice::com::SocketRequest}$ class (Section 3.1).

The main beneficial feature of asynchronous communication methods is their deterministic execution time, i.e. the guarantee that they can never block the execution for an unpredictable period of time, unlike their synchronous counterparts do. The two essential advantages brought by this feature are:

1. **Robustness** — allows to design deadlock-proof communication algorithms, especially relevant for the P2P intercommunication technique (Subsection 2.2.1).
2. **Efficiency** — allows to overlap some useful computational work (including communication) with communication, especially relevant for the P2P intercommunication technique (Subsection 2.2.1).

3.3 MPI Ports Communication

This section briefly presents the one-to-many communication implementation in preCICE on top of the MPI layer.

---

\[\text{the buffer has been successfully filled with the valid data}\]
\[\text{as soon as the actual task of sending has been scheduled, but there is no guarantee that the buffer’s data has been safely stored away (saved) somewhere}\]
\[\text{as soon as the actual task of receiving has been scheduled, but there is no guarantee that the buffer has been successfully filled with the valid data}\]
3.3.1 Rationale

Except for EMPIRE (Section 1.4) and OpenPALM (Section 1.7), none of the coupling software reviewed in Chapter 1 supports MPI ports. In contrast, preCICE deliberately supports MPI ports especially as another option for intercommunication between the coupled solvers along with TCP/IP sockets (Section 3.2), what brings more freedom and flexibility to users. The key advantage of using MPI ports instead of TCP/IP sockets is their high performance potential. For instance, the TCP/IP accelerated alternative for IB is the Sockets Direct Protocol (SDP) [39, 47] — a transport-agnostic network protocol to support stream sockets over Remote Direct Memory Access (RDMA) [48] network fabrics. As a result, most of the modern MPI implementations (e.g. MVAPICH, MVAPICH2, Open MPI, IBM MPI, and Intel MPI) either utilize SDP or even directly use the Mellanox Verbs Application Programming Interface (VAPI) [49, 50]. The essential (high-level) advantages of both SDP sockets and VAPI over TCP/IP sockets are:

1. Data communication can be initiated directly from the user space [51] to the underlying hardware, bypassing the kernel space [51] and avoiding the overhead of a system call [52].
2. The Host Channel Adapter (HCA) [53] can handle all of the network protocol on its own: breaking relatively large messages into packets, generating and handling of acknowledgments, retransmitting lost packets, and etc.; without burning a single CPU cycle on both the sender and the receiver sides (unlike for TCP/IP).

3.3.2 Basics

Assume that there are two sets of MPI processes that do not share a common MPI communicator, either two independent groups of processes of a single MPI application or even two independent MPI applications, and they need to intercommunicate with each other. The latter, by the way, is exactly what is typically achieved via network programming, e.g. with TCP/IP sockets (Section 3.2). In any case, implementation of this kind of intercommunication by means of MPI is possible starting from the version 2.1 of the MPI standard [43, 54]. For the purpose of this discussion, the two groups of processes willing to intercommunicate with each other will be distinguished as (intraparallel) client and (intraparallel) server, even though this has nothing to do with client/server communication model in the first place [55].

A server makes itself available with the MPI_OPEN_PORT and MPI_COMM_ACCEPT subroutines. It uses MPI_OPEN_PORT to establish a port (specified by port_name) at which it may be contacted, and it uses MPI_COMM_ACCEPT to accept connections from clients. MPI_COMM_ACCEPT is collective over the calling communicator. The MPI_CLOSE_PORT subroutine is used to release the port. A port can be reused after it has been released. [42]

A client uses MPI_COMM_CONNECT to establish communication with a server, at the port name specified with MPI_OPEN_PORT. MPI_COMM_CONNECT is collective over the calling communicator and returns an intercommunicator which refers to the remote group participated in calling the MPI_COMM_ACCEPT subroutine. [41]
3.3.3 The precice::com::MPIPortsCommunication Class

The precice::com::MPIPortsCommunication class implements communication in pre-
CICE via MPI ports.

Constructors

There is only one constructor available:

```cpp
MPIPortsCommunication(std::string const& addressDirectory = ".");
```

The `addressDirectory` string specifies the path to a directory where publishing/obtaining
of addresses (MPI port names) occurs. The default is the `current working directory`
of a parent process (".").

Synchronous (Blocking) Connection Establishment Methods

There are two pairs of complementary synchronous (blocking) connection establishment meth-
ods:

```cpp
void acceptConnection(std::string const& nameAcceptor,
                      std::string const& nameRequester,
                      int acceptorProcessRank,
                      int acceptorCommunicatorSize);

void requestConnection(std::string const& nameAcceptor,
                       std::string const& nameRequester,
                       int requesterProcessRank,
                       int requesterCommunicatorSize);
```

and

```cpp
void acceptConnectionAsServer(std::string const& nameAcceptor,
                              std::string const& nameRequester,
                              int requesterCommunicatorSize);

int requestConnectionAsClient(std::string const& nameAcceptor,
                              std::string const& nameRequester);
```

In order to establish connection between the two instances of the precice::com::MPI-
PortsCommunication class, one of them must call the acceptConnection() (or ac-
ceptConnectionAsServer()) method, while another one must call the requestCon-
nection() (or requestConnectionAsClient()) method. Both one-to-one (Figure 3.5)

---

further denoted as the acceptor side
further denoted as the requester side
and one-to-many (Figure 3.6) connections are possible. Note that in case of one-to-many connection, the acceptor side always has one instance of the precice::com::MPIPortsCommunication class, while the requester side can have many instances of the precice::com::MPIPortsCommunication class.

```cpp
using precice::com::MPIPortsCommunication;
MPIPortsCommunication c;
c.acceptConnection(<A>, <R>, 0, 1);
```

**Acceptor Side**

```cpp
using precice::com::MPIPortsCommunication;
MPIPortsCommunication c;
c.requestConnection(<A>, <R>, 0, 1);
```

**Requester Side**

**Figure 3.5** One-to-One Connection Establishment Examples for the precice::com::MPIPortsCommunication Class

```cpp
using precice::com::MPIPortsCommunication;
MPIPortsCommunication c;
c.acceptConnectionAsServer(<A>, <R>, 1);
```

**Acceptor Side**

```cpp
using precice::com::MPIPortsCommunication;
MPIPortsCommunication c[2];
c[0].requestConnection(<A>, <R>, 0, 2);
c[1].requestConnection(<A>, <R>, 1, 2);
```

**Requester Side**

```cpp
using precice::com::MPIPortsCommunication;
MPIPortsCommunication c[2];
c[0].requestConnectionAsClient(<A>, <R>);
c[1].requestConnectionAsClient(<A>, <R>);
```

**Acceptor Side**

**Figure 3.6** One-to-Many Connection Establishment Examples for the precice::com::MPIPortsCommunication Class

Since in order for the requester side to request a connection to the acceptor side, it has to be aware of the MPI port name of the acceptor side, as stated previously, address publishing and obtaining have to take place. For that purpose the acceptConnection() (or acceptConnectionAsServer()) method writes its address to an address file in the directory specified by the addressDirectory path. The address file has a special name pattern: .<nameRequester>-<nameAcceptor>.address. Thanks to this name pattern, the requestConnection() (or requestConnectionAsClient()) method can locate the corresponding address file, read the address from it, and perform the intended connection request to the acceptor side. **given that addressDirectory on the requester side is identical to addressDirectory on the acceptor side**
Synchronous (Blocking) Communication Methods

There are two groups of complementary synchronous (blocking) communication methods:
1. Overloaded sending methods (send()), having the following properties:
   — accept a buffer (as an argument) to provide data to be sent;
   — implement blocking sending, that is they do not return until sending has finished;
   — after return the buffer can be immediately safely reused (read/write) further.
2. Overloaded receiving methods (receive()), having the following properties:
   — accept a buffer (as an argument) to accommodate data to be received;
   — implement blocking receiving, that is they do not return until receiving has finished;
   — after return the buffer can be immediately safely reused (read/write) further.

Asynchronous (Non-Blocking) Communication Methods

There are two groups of complementary asynchronous (non-blocking) communication methods:
1. Overloaded sending methods (aSend()), having the following properties:
   — accept a buffer (as an argument) to provide data to be sent;
   — implement non-blocking sending, that is they do return immediately;
   — after return the buffer cannot be immediately safely reused (read/write) further;
   — return the corresponding instance of the precice::com::Request interface class (Section 3.4), which is implemented by the precice::com::MPIRequest class (Section 3.1).
2. Overloaded receiving methods (aReceive()), having the following properties:
   — accept a buffer (as an argument) to accommodate data to be received;
   — implement non-blocking receiving, that is they do return immediately;
   — after return the buffer cannot be immediately safely reused (read/write) further;
   — return the corresponding instance of the precice::com::Request interface class (Section 3.4), which is implemented by the precice::com::MPIRequest class (Section 3.1).

The main beneficial feature of asynchronous communication methods is their deterministic execution time, i.e., the guarantee that they can never block the execution for an unpredictable period of time, unlike their synchronous counterparts do. The two essential advantages brought by this feature are:
1. Robustness — allows to design deadlock-proof communication algorithms, especially relevant for the P2P intercommunication technique (Subsection 2.2.1).

---

37 either because OS has safely stored away (saved) the buffer’s data somewhere (for example, in the corresponding MPI send buffer) or because the buffer’s data has already been successfully received by the destination side (does not necessarily imply that the corresponding receive() has finished, i.e., the data could simply be stored in the corresponding MPI receive buffer on the destination side)
38 the buffer has been successfully filled with the valid data
39 as soon as the actual task of sending has been scheduled, but there is no guarantee that the buffer’s data has been safely stored away (saved) somewhere
40 as soon as the actual task of receiving has been scheduled, but there is no guarantee that the buffer has been successfully filled with the valid data
2. **Efficiency** — allows to overlap some useful computational work (including communication) with communication, especially relevant for the P2P intercommunication technique (Subsection 2.2.1).

### 3.4 Asynchronous (Non-Blocking) Communication Requests

As mentioned in Section 3.1, Section 3.2, and Section 3.3, instances of the `precice::com::Request` interface class represent handles to the asynchronous (non-blocking) communication requests returned by the corresponding asynchronous (non-blocking) communication methods invocations. A returned instance of the `precice::com::Request` interface class could be further used in order to properly wait (block execution) until the corresponding asynchronous (non-blocking) communication (send/receive) operation (triggered by the corresponding method invocation earlier) has finished by calling the `wait()` method on it. After the `wait()` method returns, the buffer, previously supplied to the corresponding asynchronous (non-blocking) communication method (`aSend()` or `aReceive()` as an argument, can be immediately safely reused (read/write) further. Thus, a common approach to implement parallel (overlapped) communication per process (e.g. Figure 3.7 in case of `precice::com`) is to perform

1. loop(s) over asynchronous (non-blocking) communication methods (`aSend()` or `aReceive()`) invocations;
2. loop(s) over the `wait()` method invocations on a set of the corresponding instances of the `precice::com::Request` interface class.

As seen from Subsection 2.2.1, this approach is the basis of efficient parallelization, bandwidth saturation, and deadlock-proof implementation of the P2P intercommunication technique (refer to Section 3.6 for concrete implementation details).

### 3.5 One-to-Many Communication Factories

During the design of the P2P intercommunication technique (Section 3.6), it became clear that numerous instances of the `precice::com::Communication` interface class have to be created while P2P connection establishment takes place and then aggregated in the corresponding instance of the `precice::m2n::PointToPointCommunication` class for further utilization in the P2P intercommunication methods (Section 3.6). Since the `precice::m2n::PointToPointCommunication` class must run on top of either the one-to-many TCP/IP socket-based communication implementation (Section 3.2) or the one-to-many MPI port-based communication implementation (Section 3.3), an extensible way to supply their instantiation mechanisms to the `precice::m2n::PointToPointCommunication` class was necessary, and preferably without code duplication because, in general, code reuse is very important for software robustness and consistency.

---

*conditions are the same as for synchronous (blocking) communication methods (`send()` or `receive()`) described in Section 3.2 or Section 3.3*
... precice::com::Communication::SharedPointer c = ...;
std::vector<com::Request::SharedPointer> requests;
...
for (...) {
  ...
  auto request = c->aSend(...);
  requests.push_back(request);
  ...
} ...
com::Request::wait(requests);
...

Parallel Asynchronous (Non-Blocking) Sending Session

... precice::com::Communication::SharedPointer c = ...;
std::vector<com::Request::SharedPointer> requests;
...
for (...) {
  ...
  auto request = c->aReceive(...);
  requests.push_back(request);
  ...
} ...
com::Request::wait(requests);
...

Parallel Asynchronous (Non-Blocking) Receiving Session

Figure 3.7 Examples of Parallel Asynchronous (Non-Blocking) Communication Sessions

It turned out to be an ideal scenario for application of a creational pattern [56] — the abstract factory pattern [57]. Essentially, the abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. The result of its application is reflected in Figure 3.1, where
— precice::com::CommunicationFactory is the abstract factory class that provides the newCommunication() (pure virtual) method to return instances of the precice::com::Communication interface class;
— precice::com::SocketCommunicationFactory is the concrete factory class that implements the newCommunication() method of precice::com::CommunicationFactory to create and return instances of the precice::com::SocketCommunication concrete class;
— precice::com::MPIPortsCommunicationFactory is the concrete factory class that implements the newCommunication() method of precice::com::CommunicationFactory to create and return instances of the precice::com::MPIPortsCommunication concrete class.

Thanks to abstract factory pattern, the precice::m2n package in general and the precice::m2n::PointToPointCommunication class in particular are fully abstracted from any concrete classes in the precice::com package, i.e. depend only on interface classes in the
3.6 Process-to-Process (P2P) Intercommunication

This section discloses the details of the fully parallel (many-to-many) P2P intercommunication technique (Subsection 2.2.1) implementation for preCICE.

3.6.1 Rationale

The bulk of rationale for realization of the fully parallel P2P intercommunication technique in preCICE has already been covered by Chapter 2.

3.6.2 The Toy Coupling Scenario

In order to better present the forthcoming implementation concepts, given the general illustration of the P2P intercommunication technique (Figure 2.4), assume the toy coupling scenario where the (intraparallel) solver $A$ has three processes and the (intraparallel) solver $B$ has five processes. Furthermore, for convenience and clarity, different colors are assigned to all these processes (Table 3.1).

<table>
<thead>
<tr>
<th>Processes</th>
<th>Solver $A$</th>
<th>Solver $B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

`Table 3.1  Color Notation for Processes of Coupled Intraparallel Solvers`

3.6.3 Vertex Distributions (VDs)

During the initialization phase, the solver $B$ sends its mesh to the solver $A$, where the data mapping for non-matching grids [2, “4.2 Data Mapping for Non-Matching Grids”, pp. 141–151] takes place between the two (interface) meshes (of solver $A$ and solver $B$ respectively) in order to generate the vertex distribution (VD) for the solver $A$, while the VD for the solver $B$ already exists. As a result, there are two VDs available, each stored within its corresponding solver. Essentially, a VD defines the scattering pattern of global vertex data (GVD) (i.e. the data that
is associated with each vertex, e.g. physical quantities, normals, etc.) across the (intraparallel) processes of the corresponding solver.

For instance, consider ten unique vertices which constitute the (interface) mesh between the coupled solvers in the case of the toy coupling scenario (Subsection 3.6.2). Each of these vertices is distinguished by the corresponding integer — the (zero-based) global vertex data index (GVDI), which also can be used to compute the corresponding (zero-based) global vertex data offset (GVDO) in order to properly fetch the corresponding GVD in some global vertex data array (GVDA). This example can be summarized in the form of Table 3.2.

Table 3.2  Example of VDs

<table>
<thead>
<tr>
<th>Vertex Distributions</th>
<th>Solver A</th>
<th>Solver B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GVDI</td>
<td>LVDI</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

By the time the initialization phase is finished, preCICE has already scattered all the GVDAs across the processes of the coupled solvers according to their corresponding VDs. Thus, from the perspective of each individual process, it further operates (numerical computations and intercommunication) only over its local vertex data (LVD), which is stored in some local vertex data array (LVDA) and can be properly accessed via the corresponding (zero-based) local vertex data offset (LVDO), which in turn can be computed from the corresponding (zero-based) local vertex data index (LVDI) (e.g. Table 3.2).

3.6.4 Local Communication Maps (LCMs)

Local communication map (LCM) is an auxiliary data structure that is internally used to efficiently implement the P2P intercommunication technique (Subsection 2.2.1). In fact, it defines the minimal amount of local (to each particular process) metadata that is needed to establish connections to remote (of another solver) processes and intercommunicate only the relevant parts of LVDAs with these remote processes.
In order to make the concept of LCMs clear, the mental recipe to deduce LCMs from VDs (e.g. Table 3.2) in the case of the toy coupling scenario (Subsection 3.6.2) is offered in Table 3.3. The important difference of Table 3.3 from Table 3.2 is that GVDIs are sorted in such a way that the equal GVDIs from both coupled solvers are stacked in the same rows. From this (sorted) representation of VDs, it can be immediately seen which processes have to establish connections between each other in general and which parts of LVDAs should be intercommunicated between them in particular. In other words, applying Table 3.3 to the general illustration of the P2P intercommunication technique (Figure 2.4), one would arrive to Figure 3.8.

Table 3.3 Deduction Example of LCMs

<table>
<thead>
<tr>
<th>GVDI</th>
<th>LVDI</th>
<th>GVDI</th>
<th>LVDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

Synthesizing Table 3.3 and the corresponding Figure 3.8, LCMs for processes of both coupled solvers would look as shown in Table 3.4. It should be noted that the actual algorithm for building LCMs, which can be found in Figure 3.9, is different from the mental deduction recipe presented in Table 3.3. The LCM building algorithm (Figure 3.9) is implemented as the precice::m2n::buildCommunicationMap() function, and its approximate complexity is $O(LVDIs_L \cdot GVDIs_R)$.

According to Table 3.4, the corresponding C++ datatype to represent a LCM is std::map<int, std::vector<int>>, which essentially defines a mapping from ranks of remote processes to LVDIs which indicate what potions of LVDAs have to be intercommunicated to and from these remote processes.

3.6.5 The precice::m2n::PointToPointCommunication Class

The precice::m2n::PointToPointCommunication class implements the P2P intercommunication technique (Subsection 2.2.1) in preCICE.
Constructors

There is only one constructor available:

```cpp
PointToPointCommunication(
    precice::com::CommunicationFactory::SharedPointer communicationFactory,
    precice::mesh::PtrMesh mesh);
```

The `communicationFactory` parameter specifies the one-to-many communication factory (Section 3.5), which is internally used to instantiate the one-to-many communication implementations, either the TCP/IP socket-based one (Section 3.2) or the MPI port-based one (Section 3.3).

The `mesh` parameter specifies the mesh according to which the appropriate LCMs (Subsection 3.6.4) are built. In fact, each mesh has a corresponding VD (in general, different for each coupled solver), which is accessible through the `getVertexDistribution()` method of the `precice::mesh::Mesh` class, and that is used further in the `precice::m2n::buildCommunicationMap()` function to build LCMs from VDs.

Synchronous (Blocking) Connection Establishment Methods

There are two complementary synchronous (blocking) connection establishment methods:
Table 3.4  Example of LCMs for Processes of Coupled Intraparallel Solvers

<table>
<thead>
<tr>
<th>Process</th>
<th>LVDIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{0}</td>
</tr>
<tr>
<td>2</td>
<td>{1}</td>
</tr>
</tbody>
</table>

\[ B_1 \]

<table>
<thead>
<tr>
<th>Process</th>
<th>LVDis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>{0, 2}</td>
</tr>
<tr>
<td>5</td>
<td>{1, 3}</td>
</tr>
</tbody>
</table>

\[ A_1 \]

<table>
<thead>
<tr>
<th>Process</th>
<th>LVDis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{0, 2}</td>
</tr>
<tr>
<td>2</td>
<td>{1}</td>
</tr>
</tbody>
</table>

\[ B_2 \]

<table>
<thead>
<tr>
<th>Process</th>
<th>LVDis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>{0}</td>
</tr>
<tr>
<td>3</td>
<td>{2, 3}</td>
</tr>
<tr>
<td>5</td>
<td>{4}</td>
</tr>
</tbody>
</table>

\[ A_2 \]

<table>
<thead>
<tr>
<th>Process</th>
<th>LVDis</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{0}</td>
</tr>
<tr>
<td>3</td>
<td>{1}</td>
</tr>
</tbody>
</table>

\[ B_3 \]

<table>
<thead>
<tr>
<th>Process</th>
<th>LVDis</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>{0}</td>
</tr>
<tr>
<td>5</td>
<td>{1}</td>
</tr>
</tbody>
</table>

\[ A_3 \]

<table>
<thead>
<tr>
<th>Process</th>
<th>LVDis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{0, 1}</td>
</tr>
<tr>
<td>2</td>
<td>{2}</td>
</tr>
<tr>
<td>3</td>
<td>{3}</td>
</tr>
</tbody>
</table>

\[ B_4 \]

```cpp
void acceptConnection(std::string const& nameAcceptor,
                      std::string const& nameRequester);

void requestConnection(std::string const& nameAcceptor,
                       std::string const& nameRequester);
```
\[ PR_L \leftarrow \text{rank of the current process} \]
\[ LCM_L \leftarrow 0 \]
\[ \text{for } GVDI_L \text{ in } (\text{GVDIs from } VD_L, \text{ where process rank is } PR_L) \text{ do} \]
\[ \text{for all } PR_R \text{ and } GVDI_R \text{ in } (\text{process ranks and GVDIs from } VD_R) \text{ do} \]
\[ \text{if } GVDI_L = GVDI_R \text{ then} \]
\[ LCM_L \leftarrow LCM_L \cup (PR_R \rightarrow LVDI_L) \]
\[ \text{end if} \]
\[ \text{end for} \]
\[ LVDI_L \leftarrow LVDI_L + 1 \]
\[ \text{end for} \]
\[ \text{return } LCM_L \]

where
\[ L \quad \text{[subscript]} \quad \text{— signifies that the data structure corresponds to the local (this) solver;} \]
\[ R \quad \text{[subscript]} \quad \text{— signifies that the data structure corresponds to the remote (another) solver.} \]

**Figure 3.9** The LCM Building Algorithm

In order to establish connection between the multiple instances of the precice::m2n:: PointToPointCommunication class, some of them must call the acceptConnection() method, while another ones must call the requestConnection() method. The many-to-many connection establishment example (between two acceptor sides and two requester sides) internally using the TCP/IP socket-based implementation (Section 3.2) is demonstrated in Figure 3.10. For the MPI port-based implementation (Section 3.3), the SocketCommunicationFactory identifier should be substituted by the MPIPortsCommunicationFactory identifier.

Address publishing and obtaining takes place according to the underlying one-to-many communication implementation, either the TCP/IP socket-based one (Section 3.2) or the MPI port-based one (Section 3.3). In order to disambiguate multiple acceptor sides from each other (i.e. prevent name collisions on their corresponding address files), more details have to be included in the name pattern: .<nameRequester>-<nameAcceptor>-<utils::MasterSlave::_rank>.address. This is achieved by invoking both the acceptConnectionAsServer() method and the requestConnectionAsClient() method of the corresponding one-to-many communication implementation with a proper value for the nameAcceptor parameter, internally during connection establishment, as shown in Figure 3.11.

**Synchronous (Blocking) Intercommunication Methods**

There are two groups of complementary synchronous (blocking) intercommunication methods:

1. Overloaded sending methods (send()), having the following properties:

[42] further denoted as the acceptor sides
[43] further denoted as the requester sides
Figure 3.10  The TCP/IP Socket-Based Many-to-Many Connection Establishment Example for the precice::m2n::PointToPointCommunication Class

— accept a buffer (as an argument) to provide data to be sent;
— implement blocking sending, that is they do not return until sending has finished.
void PointToPointCommunication::acceptConnection(std::string const& nameAcceptor, std::string const& nameRequester) {
    ...
    auto lcm = m2n::buildCommunicationMap(...);
    ...
    auto c = _communicationFactory->newCommunication();
    ...
    c->acceptConnectionAsServer(
        nameAcceptor + "-" + std::to_string(utils::MasterSlave::_rank),
        nameRequester,
        lcm.size());
    ...
}

Acceptor Side

void PointToPointCommunication::requestConnection(std::string const& nameAcceptor, std::string const& nameRequester) {
    ...
    auto lcm = m2n::buildCommunicationMap(...);
    ...
    for (auto& pair : lcm) {
        auto globalAcceptorRank = pair.first;
        ...
        auto c = _communicationFactory->newCommunication();
        ...
        c->requestConnectionAsClient(
            nameAcceptor + "-" + std::to_string(globalAcceptorRank), nameRequester);
        ...
    }
    ...
}

Requester Side

Figure 3.11 Implementation Details of the Connection Establishment
Methods of the precice::m2n::PointToPointCommunication Class

— after return the buffer can be immediately safely reused (read/write) further.
2. Overloaded receiving methods (receive()), having the following properties:
   — accept a buffer (as an argument) to accommodate data to be received;
   — implement blocking receiving, that is they do not return until receiving has finished;
   — after return the buffer can be immediately safely reused (read/write) further.

In fact, these synchronous (blocking) intercommunication methods are implemented according to the recipe described in Section 3.4, i.e. based on a loop over asynchronous (non-blocking)

either because OS has safely stored away (saved) the buffer’s data somewhere (for example, in the corresponding one-to-many communication implementation send buffer) or because the buffer’s data has already been successfully received by the destination side (does not necessarily imply that the corresponding receive() has finished, i.e. the data could simply be stored in the corresponding one-to-many communication implementation receive buffer on the destination side)

the buffer has been successfully filled with the valid data
communication methods (\texttt{aSend()} or \texttt{aReceive()}) invocations on a set of the instances of the \texttt{precice::com::Communication} interface class (Section 3.1), followed by a loop over the \texttt{wait()} method invocations on a set of the corresponding instances of the \texttt{precice::com::Request} interface class; see Figure 3.12.

**Figure 3.12** Implementation Details of the Synchronous (Blocking) Intercommunication Methods of the \texttt{precice::m2n::PointToPointCommunication} Class

```cpp
void
PointToPointCommunication::send(..., int size, int valueDimension) {
...  for (auto& mapping : _mappings) {
    mapping.offset = _buffer.size();
...    // Write the corresponding contiguous block of `_buffer' to be sent.
    mapping.request =
        mapping.communication->aSend(_buffer.data() + mapping.offset,
                                    mapping.indices.size() * valueDimension,
                                    mapping.localRemoteRank);
  }
  for (auto& mapping : _mappings) {
    mapping.request->wait();
  }
  _buffer.clear();
}
```

**Sender Side**

```cpp
void
PointToPointCommunication::receive(..., int size, int valueDimension) {
...  for (auto& mapping : _mappings) {
    mapping.offset = _buffer.size();
    _buffer.resize(_buffer.size() + mapping.indices.size() * valueDimension);
    mapping.request =
        mapping.communication->aReceive(_buffer.data() + mapping.offset,
                                        mapping.indices.size() * valueDimension,
                                        mapping.localRemoteRank);
  }
  for (auto& mapping : _mappings) {
    mapping.request->wait();
    ... // Read the corresponding contiguous block of `_buffer' that was received.
  }
  _buffer.clear();
}
```

**Receiver Side**
4 Verification

This chapter contains everything that is related to verification of the implemented P2P intercommunication technique (Section 3.6) in particular and its prerequisites (Chapter 3) in general. This chapter provides detailed descriptions of the test cases used for verification, the actual third-party simulation software used for coupling via preCICE, and the corresponding performance results supplemented with thorough analysis and explanations.

4.1 Unit Tests

The preCICE automated framework for unit tests [2, “3.6 Software Architecture”, pp. 116–120] has been used to implement the relevant unit tests for all of the new functionality that was added during this thesis into the precice::com and precice::m2n packages. For instance, the src/m2n/tests/PointToPointCommunication.cpp source file contains the implementation of the two unit tests for the precice::m2n::PointToPointCommunication class (Section 3.6), one for the TCP/IP socket-based communication implementation (Section 3.2) and one for the MPI port-based communication implementation (Section 3.3). Both unit tests cover a deadlock-prone intercommunication pattern (Subsection 2.2.1) and strictly validate the results of data intercommunication.

4.2 Integration Tests

The preCICE automated framework for integration tests [2, “3.6 Software Architecture”, pp. 116–120] has been used to extend the relevant integration tests with the utilization of the precice::m2n::PointToPointCommunication class. Specifically, the precice::tests::SolverInterfaceTest::testDistributedCommunications() integration test from the src/precice/tests/couplingmode/SolverInterfaceTest.cpp source file performs validation of the precice::m2n::PointToPointCommunication class (Section 3.6) in the coupling mode and ensures its behavior consistency with the precice::m2n::GatherScatterCommunication class in a toy coupled simulation scenario. Both the TCP/IP socket-based communication implementation (Section 3.2) and the MPI port-based communication implementation (Section 3.3) are validated.

4.3 Performance Tests

This section contains the details and results of scalability benchmarks and application scenarios profiled to examine the offered implementation of the P2P intercommunication technique (Section 3.6) for expected efficiency and possible drawbacks.
4.3.1 Hardware Configuration

In order to exploit the full potential of MPP and deliver high-quality results that would be meaningful to real-world applications, all the performance tests were done on the SuperMUC massively parallel system [7]. In particular, the thin nodes were used, and their respective hardware configuration can be found in [58].

4.3.2 The Ateles “Symmetric” Strong Scalability Test

Ateles is an advanced intraparallel CFD solver that is part of the Adaptable Poly-Engineering Simulator (APES) framework [8–10]. Ateles is characterized by an explicit time-stepping scheme and a discontinuous Galerkin (DG) scheme for hyperbolic conservation laws, which has proven to exhibit nearly linear strong scaling [3, 4] at least up to total of 512 mesh elements (corresponding to the mesh level \( l = 4 \)) with an eighth-order scheme on the MAC Cluster [59]. This is due to the merit of the underlying implementation of the DG method which requires only relatively small amounts of data to be intracommunicated between high-order elements in the mesh allowing the solver to scale down to even one single mesh element per process.

Rationale

Assume that there was some experiment for strong scalability performed with a single instance of Ateles which has eventually yielded roughly linear strong scaling; for instance, [3, 4, 8]. Then, theoretically, artificial splitting of the original experiment domain into two halves, to be simulated by the two corresponding Ateles instances coupled via the preCICE coupling interface (e.g. Figure 4.1), must not deteriorate the overall linear strong scaling trend as long as the underlying implementation of the intercommunication technique being used scales linearly as well. Thus, such testing approach could be applied to reliably verify whether the offered implementation of the P2P intercommunication technique (Section 3.6) is successful in terms of efficiency.

![Scenario Configuration: A cubic fluid domain is (artificially) divided into two halves in order to be simulated by the two corresponding Ateles instances coupled via the preCICE coupling interface](image1)

![Smooth Transition of a Spherical Density Pulse Through The (Artificial) Coupling Interface](image2)

**Figure 4.1** Simulation of a Traveling Spherical Density Pulse
Description

A simple spherical density pulse is transported with a constant fluid velocity through a cubic domain governed by the (nonlinear) compressible Euler equations,

\[
\frac{\partial}{\partial t} \begin{pmatrix} \rho \\ \rho \cdot \vec{v} \\ e \end{pmatrix} + \left( \rho \cdot \vec{v} \cdot \vec{v}^T + p \cdot I \right) \cdot \nabla = \vec{0},
\]  

(4.1)

where

\[
\begin{align*}
\vec{x} & \quad [m] \quad \text{— a point of space;} \\
t & \quad [s] \quad \text{— a moment of time;} \\
\vec{v}(\vec{x}, t) & \quad [m/s] \quad \text{— the fluid velocity at the point } \vec{x} \text{ of space and the moment } t \text{ of time;} \\
\rho(\vec{x}, t) & \quad [kg/m^3] \quad \text{— the fluid density at the point } \vec{x} \text{ of space and the moment } t \text{ of time;} \\
p(\vec{x}, t) & \quad [Pa] \quad \text{— the fluid pressure at the point } \vec{x} \text{ of space and the moment } t \text{ of time;} \\
e(\vec{x}, t) & \quad [J/m^3] \quad \text{— the fluid total energy density at the point } \vec{x} \text{ of space and the moment } t \text{ of time.}
\end{align*}
\]

This results in the underdetermined system — three partial differential equations and four unknowns. Thus, in order to fulfill the system, the application of the ideal gas law yields the additional relation between \( \vec{v}, \rho, p, \) and \( e \):

\[
p = \rho \cdot \frac{k_B}{m_p} \cdot T = (\gamma - 1) \cdot \left( e - \frac{\rho \cdot \vec{v}^2}{2} \right),
\]  

(4.2)

where

\[
\begin{align*}
k_B & \quad [J/K] \quad \text{— the Boltzmann constant;} \\
m_p & \quad [kg] \quad \text{— the mass of a (single) gas particle;} \\
T(\vec{x}, t) & \quad [K] \quad \text{— the fluid absolute temperature at the point } \vec{x} \text{ of space and the moment } t \text{ of time;} \\
\gamma & \quad \text{— the adiabatic index.}
\end{align*}
\]

As reasoned above, the domain is (artificially) divided into two halves in order to be simulated by the two corresponding Ateles instances coupled via the preCICE coupling interface (e.g. Figure 4.1). Since equations \( (4.1) \) and \( (4.2) \) with constant fluid velocity \( \vec{v} \) represent a pure transport phenomenon, the shape of the spherical density pulse should not vary over time (e.g. Figure 4.1). Furthermore, since both halves are absolutely symmetrical, the corresponding P2P intercommunication pattern is the symmetric one-to-one intercommunication pattern as depicted in Figure 4.2, and it is especially important to have this kind of pattern for the strong scalability test case because it does not introduce any intercommunication imbalance.
such scenario is not only suitable for validation of numerical results but also for performance tests. Besides, as this scenario has been studied and exercised before [3, 4] to confirm performance bottlenecks (Figure 2.2) induced by the non-parallel client/server intercommunication technique (Subsection 2.1.1), it is also reasonable to repeat the same experiment for the P2P intercommunication technique (Subsection 2.2.1). This time, however, the higher mesh level \( l = 6 \) is used to allow for a higher maximum number of processes to be tested, i.e. up to

\[
2^{3(l-1)} = 32768
\]

processes overall and 16384 per one Ateles instance accordingly.

\[\text{Figure 4.2} \quad \text{The Symmetric One-to-One Process-to-Process (P2P) Intercommunication Pattern}\]

**Results**

The first point to study is the impact of the offered implementation of the P2P intercommunication technique (Section 3.6) on the preCICE initialization phase. In particular, it is interesting to know how does the connection establishment (which takes place during the preCICE initialization phase) scale compared to the rest of the preCICE initialization phase. The corresponding bar chart for each major operation can be found on Figure 4.3. It should be noted that the operations which directly correspond to the P2P intercommunication connection establishment are

— "Request Connection" which represents the `PointToPointCommunication::requestConnection()` method and consists of

\[\text{in that execution order}\]

49
— “Synchronize” — the master process of the requester (this) side requests connection to the master process of the acceptor (another) side;
— “Exchange VDs” — the master process of the requester (this) side receives VD from the master process of the acceptor (another) side and sends VD of the requester (this) side to the master process of the acceptor (another) side;
— “Broadcast VDs” — the master process of the requester (this) side broadcasts both VDs to the rest of the (slave) processes of the requester (this) side;
— “Build LCM” — each process of the requester (this) side builds its individual LCM from the two VDs [Subsection 3.6.4];
— “Accept Connection” which represents the PointToPointCommunication::acceptConnection() method and consists of:
— “Synchronize” — the master process of the acceptor (this) side accepts connection from the master process of the requester (another) side;
— “Exchange VDs” — the master process of the acceptor (this) side sends VD of the acceptor (this) side to the master process of the requester (another) side and receives VD from the master process of the requester (another) side;
— “Broadcast VDs” — the master process of the acceptor (this) side broadcasts both VDs to the rest of the (slave) processes of the acceptor (this) side;
— “Build LCM” — each process of the acceptor (this) side builds its individual LCM from the two VDs [Subsection 3.6.4].

Furthermore, the “Initialize” operation (which represents the whole preCICE initialization phase) consists of:
— “Gather Mesh”,
— “Send Mesh”,
— “Initialize Data”, and
— “Request Connection” on the requester side and consists of:
— “Receive Mesh”,
— “Scatter Mesh”,
— “Filter Mesh”,
— “Initialize Data”, and
— “Accept Connection” on the acceptor side accordingly. No descriptions are provided for those operations because their implementation details are beyond the scope of this thesis and are irrelevant for further discussion anyway.

From the first glance on Figure 4.3, it might seem that the “Request Connection” operation does not scale, what in turn causes the “Initialization” operation on the requester side (Ateles “Left” instance) not to scale. However, this is not the case. In particular, it is clearly the “Synchronize” operation that does not scale and, as a result, makes the “Request Connection” operation not to scale. At the same time, the only reason why the “Synchronize” operation on the requester side
Figure 4.3  Initialization Decomposition (Strong Scaling)

...does not scale in the first place lies on the acceptor side (Ateles “Right” instance). In fact, it is the “Scatter Mesh” operation that does not scale\footnote{the major performance issue with the preCICE initialization phase due to simple and not optimized implementation}, and since it executes before the “Synchronize” operation on the acceptor side, it actually makes the requester side to become idle during the corresponding “Synchronize” operation for a relatively long period of time (approximately
the execution time of the “Scatter Mesh” operation). A good estimate of the “Synchronize” operation true scalability on the requester side is the scalability of the “Synchronize” operation on the acceptor side where it executes almost instantly compared to the one on the requester side due to the fact that the requester side is definitely already waiting for the connection to be accepted, so no execution time is wasted in the the “Synchronize” operation on the acceptor side. In fact, ideally, the execution time of the “Synchronize” operation does not depend on the number of processes involved and should stay approximately constant. Nevertheless, since, during the pair of “Synchronize” operations, address publishing and obtaining takes place according to the underlying one-to-many communication implementation, either the TCP/IP socket-based one (Section 3.2) or the MPI port-based one (Section 3.3), certain execution time fluctuations due to address file I/O operations are inevitable (Figure 4.3).

In general, when the number of CIPes increases, the sizes of the corresponding VDs increase too. Thus, the execution times of the “Exchange VDs” operations are expected to increase with the increasing number of processes involved. Although this is exactly what one could observe on Figure 4.3, the impact stays relatively (compared to the other operations) mild, and the execution times of the “Exchange VDs” operations exhibit quite slow growth.

The rise of the execution times of the “Broadcast VDs” operations with the increasing number of processes involved (Figure 4.3) is two-fold. On the one hand it is due to the same reason as for the “Exchange VDs” operations. On the other hand, it is due to the nature of the underlying (general) “broadcast” routine itself. Additionally to implementing P2P intercommunication technique in this thesis, special care was taken to get rid of the (previous) naive implementation of the “broadcast” routine and reimplement it in a more efficient manner. In particular, it is now based on the `MPI_Bcast()` routine, which offers a nearly constant communication time for large-scale IB clusters with most of the modern MPI implementations (e.g. Open MPI, IBM MPI, Intel MPI, etc.) [60]. During the subsequent performance tests it was revealed that such change had improved the performance of the “Broadcast VDs” operations by orders of magnitude, especially for relatively large numbers of processes used.

Contrary to the “Exchange VDs” and “Broadcast VDs” operations as the the sizes of the corresponding VDs increase, the execution times of the “Build LCMs” operations decline (Figure 4.3). The explanation of this phenomenon lies in the actual algorithm for building LCMs (Figure 3.9). Indeed, the size of the outer loop depends on how many GVDIs correspond to the (current) process, and since the number of GVDIs corresponding to each process decreases as the number of processes involved increases, it becomes clear that less (expensive) GVDI searching and matching operations have to be performed (in the inner loop), what results in faster execution.

All in all, it can be concluded that the connection establishment aspect of the offered implementation of the P2P intercommunication technique (Section 3.6) demonstrates relatively good
strong scaling characteristics. Thus, the major performance bottleneck of the preCICE initialization phase remains the “Scatter Mesh” operation.

Next comes the most interesting and important part of this performance analysis. Figure 4.4 shows the almost ideal (linear) strong scaling for a single time step. The trend follows all the way down from the $2^7$ processes per Ateles instance up to the $2^{14}$ processes per Ateles instance, what already confirms the (previously mentioned) fact that Ateles is indeed capable of maintaining roughly linear strong scaling down to even one single mesh element per process.

It is crucial to note that the execution times of a time step on Figure 4.4 consist not only of the Ateles computations but also of the preCICE overhead. Therefore, these results alone already indicate the success of the offered implementation of the P2P intercommunication technique (Section 3.6). Nevertheless, it is reasonable to decompose the time step execution times into the pure Ateles computations and the pure preCICE overhead contributions in order to have a more detailed view (Figure 4.5).

It might look confusing at the first glance, but (as expected) the overhead introduced by the preCICE “Advance” operation scales down to one single mesh element per process as well. The seemingly unusual fluctuations of the execution times of the preCICE “Advance” operations are the consequence of slightly different execution times of the Ateles “Compute” operations and a subsequent synchronization issue during the actual P2P intercommunication. For example, consider the execution times at the $2^{11}$ processes per Ateles instance in Figure 4.5. For the Ateles “Left” instance, the corresponding execution time of the Ateles “Compute” operation is 4070 ms, while for the Ateles “Right” instance, the corresponding execution time of the Ateles “Compute” operation is 4042 ms. Thus, “Right” is definitely going to wait for “Left” for at least $4070 - 4042 = 28$ ms to actually start the intercommunication. Then the intercommunication takes place for at most $15$ ms according to the execution time of the preCICE “Advance” operation on the Ateles “Left” instance, and so even the rough estimation of $28 + 15 = 43$ ms gives exactly the same result as the execution time of the preCICE “Advance” operation on the Ateles “Right” instance.

In fact, the exact origins and nature of those execution time fluctuations of the preCICE “Advance” operations are a combination of a broad range of other factors than mere differences in the execution times of the Ateles “Compute” operations and a subsequent synchronization issue. For instance, they also depend on whether the PointToPointCommunication::send() method blocks until the data has been actually received on the other side or it blocks only until the data has been saved in the corresponding one-to-many communication implementation.

---

those results are of course averaged across multiple time steps (ten in this case)

$l = 6 \iff 32768$ mesh elements overall $\iff 16384$ mesh elements per Ateles instance $\iff$ maximum of $16384 = 2^{14}$ processes per Ateles instance

caused by the “Advance” operation

the overwhelming part of which is the P2P intercommunication

since the first intercommunication step is always that “Right” receives from “Left”

since the last intercommunication step is always that “Left” receives from “Right”
Send buffer for future sending (Section 3.6), and these blocking times could differ substantially. Besides, whether it is one case or the other depends on both the size of the actual data being sent and the size of the corresponding one-to-many communication implementation send buffer. In fact, when the corresponding one-to-many communication implementation is the MPI port-based one (Section 3.3), things are complicated even further due to MPI message passing protocols [61]. Last but not least, execution environment volatility in general and cluster network

which is obviously changing with the changing number of processes used
volatility in particular also contribute to this phenomenon. In any case, the overall positive strong scaling trend of the offered implementation of the P2P intercommunication technique (Section 3.6) is clear from both Figure 4.4 and Figure 4.5.
Issues

One annoying limitation of the current implementation of the P2P intercommunication technique (Section 3.6) has been discovered with the help of this test case. In fact, it stems from a combination of two factors:

1. How address publishing and obtaining are implemented in the both underlying one-to-many communication implementations, the TCP/IP socket-based one (Section 3.2) and the MPI port-based one (Section 3.3). In particular, the fact that as many CIPes exist on the acceptor side, as many address files have to be created (and written) in the specified directory (addressDirectory).

2. The IBM General Parallel File System (GPFS) [62] has a limitation that “it is not optimal for handling large quantities of small files located in a single directory with parallel accesses” [63].

In other words, running large-scale simulations coupled via preCICE using the current implementation of the P2P intercommunication technique (Section 3.6) that performs address publishing and obtaining under either $SCRATCH or $WORK directory on SuperMUC [64] “will probably cause your application(s) to experience I/O errors (due to timeouts) and crashes” [63].

One possible solution is described in [65]. However, in practice, it did not improve the situation.

Another trivial solution, that works out of the box, is to perform address publishing and obtaining under $HOME directory where the NAS-based shared file system is mounted [64]. The only drawback is that the file created there will not be immediately visible to those processes which run on other computing nodes than the process that created the file. There is a fixed time interval (30 s on SuperMUC) between file system synchronizations periodically done by NAS. As a result, this might make the P2P connection establishment (Section 3.6) take longer than it would in case of using a “traditional” file system.

4.3.3 The Alya “Asymmetric” Strong Scalability Test

The Alya system [5, 11] is a multiphysics simulation code with emphasis on HPC, designed from the ground up to efficiently solve coupled problems in massively parallel environments. Alya has modular architecture: kernel, services, and modules; which can be separately compiled and linked. Each module corresponds to a different set of PDE, i.e. each module defines physics. In order to solve a coupled multiphysics problem, all the required modules have to be active and interact in a certain well-defined way. Alya’s kernel controls the execution, the I/O, and the mesh (geometry) manipulation. The services offer supplementary functionality, notably the parallelization service — master/slave parallelization strategy on top of the MPI layer. Alya has demonstrated scaling tendency with the increasing number of processes [11].

Rationale

Although the Ateles “symmetric” strong scalability test (Subsection 4.3.2) is inarguably good for verification of raw intercommunication scalability, it is also extremely artificial and quite
far from the real-world applications. In fact, the corresponding P2P intercommunication pattern (Figure 4.2) is too simple and limits the discovery and investigation of other possible issues besides raw scalability. Hence, it is reasonable to perform additional verification with the “asymmetric” strong scalability test, and coupling the two instances of the highly parallel Alya system with preCICE constitutes the relevant test case due to Alya’s support for unstructured grids. Furthermore, Alya has already been adapted for coupling with preCICE for the similar test case [5].

Description

The scenario consists of a flexible thin structure with a varying thickness clamped behind a fixed rigid non-rotating cylinder installed in a water channel (Figure 4.6). The cylinder has a diameter $D$. Its central axis is positioned exactly in the middle of a height $H$ of the domain, i.e. $H_c = H/2$, and at $L_c$ downstream of the inflow boundary. The domain has a length $L$ and a width $W$. The gravitational acceleration $g$ points in the $x$-direction in order to prevent asymmetries. The deformable structure used in the experiment behind the cylinder has a length $l$ and a width $w$. Since $w$ is chosen such that $w < W$, there is a small gap between the sides of the deformable structure and both lateral channel walls. The thickness of the deformable structure is $h$. The complete description of the test case, including the exact values of the parameters of the geometrical configuration, can be found in [66, “2. Description of the validation test case”, pp. 20–22].

The water channel is governed by the incompressible Navier–Stokes equations (in presence of the gravitational field) that are comprised of

1. the Cauchy momentum equation (the consequence of the balance of momentum):
\[
\frac{\partial \vec{\dot{v}}}{\partial t} + (\vec{\dot{v}} \cdot \nabla^\top) \cdot \nabla = \frac{1}{\rho} \cdot \nabla p + \nu \cdot (\vec{\dot{v}} \cdot \nabla^\top) \cdot \nabla + \vec{g},
\]
\hspace{1cm} (4.3)

where

- \( \vec{\dot{v}}(\vec{x}, t) \) — the fluid velocity at the point \( \vec{x} \) of space and the moment \( t \) of time;
- \( \rho \) — the constant fluid density;
- \( p(\vec{x}, t) \) — the fluid pressure at the point \( \vec{x} \) of space and the moment \( t \) of time;
- \( \nu \) — the constant fluid kinematic viscosity;
- \( \vec{g} \) — the gravitational acceleration acting on the fluid;

2. the continuity equation (the consequence of the conservation of mass):

\[
\nabla^\top \cdot \vec{\dot{v}} = 0.
\]
\hspace{1cm} (4.4)

The deformable structure is governed by the Cauchy momentum equation (in presence of the gravitational field) in the Lagrangian point of view (i.e. without a convective term):

\[
\frac{\partial^2 \vec{u}}{\partial t^2} = \frac{1}{\rho} \cdot S \cdot \nabla + \vec{g},
\]
\hspace{1cm} (4.5)

where

- \( \vec{u}(\vec{x}, t) \) — the material displacement at the point \( \vec{x} \) of space and the moment \( t \) of time;
- \( \rho \) — the constant material density;
- \( S \) — the second Piola–Kirchhoff stress tensor [2, “2.1.3 Structural Mechanics”, pp. 19–21];
- \( \vec{g} \) — the gravitational acceleration acting on the material.
Once again, following the concept of partitioned FSI, the idea is to separate the fluid solver and the structure solver to be executed by the two corresponding Alya instances coupled via the preCICE coupling interface. The following Alya’s modules are used [5]:

1. “NASTIN” (CFD) — solves the incompressible Navier–Stokes equations (4.3) and (4.4) using the large eddy simulation (LES) [67] mathematical model for turbulence;
2. “ALEFOR” (CMD) — moves the fluid mesh accordingly with the deformation of the solid mesh;
3. “SOLIDZ” (CSD) — solves the Cauchy momentum equation (balance of momentum) (4.5).

Hence, “NASTIN” is the fluid solver and the corresponding Alya instance executing it will be further called the Alya “NASTIN” instance, whereas “SOLIDZ” is the solid solver and the corresponding Alya instance executing it will be further called the Alya “SOLIDZ” instance accordingly.

As stated above, the goal is to have an asymmetric intercommunication pattern (e.g. Figure 4.7) and preferably throughput-imbalanced between the coupled solvers (but not inside of them) as well to study how this affects the performance. Thanks to the METIS library [68], which is utilized by Alya to implement the domain decomposition by partitioning the element graph [11], the resulting mesh distributions across the processes of the coupled solvers cannot be predicted beforehand, may vary significantly when the number of processes used is changed even slightly, and almost definitely lead to an arbitrary intercommunication pattern.

Last but not least, as the numerical complexity for the fluid solver is much higher than for the solid solver, it was decided that the number of processes dedicated for the Alya “NASTIN” instance remains fixed at 1024, while the number of processes dedicated for the Alya “SOLIDZ” instance varies from 3 up to 128. This not only allows to find the load balance ratio between the number of processes dedicated for the both coupled solvers in this particular test case but also intensifies the desired asymmetry of the intercommunication pattern, making the test case even more interesting.

Results

As discussed previously, the number of processes dedicated for the Alya “NASTIN” instance remains fixed at 1024, while the number of processes dedicated for the Alya “SOLIDZ” instance varies from 3 up to 128. The number of CIPes, the number of intercommunication partners per CIP (maximum, minimum, and average), and the number of LVDIs per CIP (maximum, minimum, and average) for the both coupled solvers are depicted on the corresponding bar charts in Figure 4.8 and Figure 4.9.

---

when the number of CIPes (the processes which participate in the P2P intercommunication, i.e. contain non-empty LCMs according to Subsection 3.6.4) is different between the both coupled solvers (in this case, the fluid solver and the solid solver respectively) and/or each CIP has a different number of intercommunication partners and/or each CIP has a different number of LVDIs to be intercommunicated with its intercommunication partners (Subsection 3.6.4)
To make it clear how these statistics are calculated, consider the toy coupling scenario from Subsection 3.6.2 and the corresponding LCMs for the processes of the solver \( A \) and the solver \( B \) from Table 3.4. For instance, for the CIP \( A_2 \), the overall number of LVDIs that have to be intercommunicated (with its intercommunication partners: \( B_1, B_2, B_3, \) and \( B_5 \)) is clearly 5, whereas for the CIP \( A_1 \), it is also 5 even though there is one duplicate LVDI (that is 1) because it is the amount of corresponding data to be intercommunicated that is counted in the first place. Following this logic, it can be concluded that the maximum and minimum numbers of LVDIs per CIP for the solver \( A \) are 5 and 2 respectively, while the maximum and minimum numbers of LVDIs per CIP for the solver \( B \) are 4 and 2 respectively. Note that in the case of the solver \( B \), the process \( B_4 \) is not even incorporated into the calculation because it contains an empty LCM, that is \( B_4 \) does not have any intercommunication partners, and hence is not a CIP. As a result, the number of CIPs for the solver \( A \) and the solver \( B \) is 3 and 4 respectively. Finally, the average number of LVDIs per CIP for the solver \( A \) and the solver \( B \) is \((5 + 5 + 2)/3 = 4\) and \((2 + 3 + 3 + 4)/4 = 3\) respectively. Analogous algorithm can be used to obtain the statistics on the intercommunication partners.

As expected, when the number of processes dedicated to a solver grows, so does the number of CIPs (Figure 4.8 or Figure 4.9), while the (maximum, minimum, and average) number of LVDIs per CIP declines accordingly (Figure 4.9). Besides, it should be noted that the METIS library does a great job of partitioning the solid mesh as the dispersion of the number of LVDIs per CIP remains relatively low (Figure 4.9). Thus, the Alya “SOLIDZ” instance is expected to

\[\text{yet depending on the domain decomposition algorithm}\]

\[\text{the range between the maximum and minimum numbers of LVDIs per CIP}\]
have fine load and throughput balance (inside of itself). The desired asymmetry and throughput imbalance of the intercommunication pattern is clearly present because
1. there are no equal values for the number of CIPes between the coupled Alya instances (Figure 4.8 or Figure 4.9),
2. there are different values for the number of intercommunication partners per CIP (Figure 4.8),
3. there are different values for the number of LVDIs per CIP (Figure 4.9).

Before disclosing the performance results, another important point has to be mentioned. Since the P2P intercommunication (Section 3.6) is fully parallel\footnote{see Subsection 2.2.1 for rationale},
measuring its performance accurately is a challenging task. Thus, the performance results, revealed further in this section, required auxiliary code instrumentation, especially with additional barriers across all of the processes of a single solver (intra-solver barriers), to precisely capture relevant execution events. Under these circumstances (with intra-solver barriers around the intercommunication routines), the maximum number of LVDIs per CIP (Figure 4.9) becomes the most important characteristic as the intercommunication time corresponding to the CIP with maximum number of LVDIs will dominate.

First, consider the scalability of a single time step shown in Figure 4.10. Similarly to the Ateles test case (Figure 4.4), the execution times of a time step include both the Alya computations and the preCICE overhead. Apparently, the positive scaling trend stops roughly at 6 processes per Alya “SOLIDZ” instance. Thus, the load balance ratio between the coupled Alya instances for this particular test case is roughly \( \frac{1024}{6} \approx 170 \) (or \( \frac{6}{1024} \approx 0.006 \)).

Second, consider the decomposition of the time step execution times from Figure 4.10 into the pure Alya computations and the pure preCICE overhead in Figure 4.11. As described above, to obtain these measurements, there is additional code instrumentation involved. In particular, the preCICE “Advance” operation contains a special inter-solver barrier, which synchronizes the coupled solvers exactly at the point when they are ready to intercommunicate. This allows to accurately capture the execution times of the following sending (PointToPointCommunication::send()) and/or receiving (PointToPointCommunication::receive()) operations and then accumulate them into the corresponding single characteristic — the time of the preCICE “Communicate” operation (Figure 4.11), which is part of the preCICE “Advance” operation.

Due to the inter-solver barrier, it is also possible to capture how much time is wasted in the preCICE “Advance” operation when the coupled solvers are load-imbalanced, i.e. execution times of their respective “Compute” operations are not equal. For example, as the number of processes dedicated for the Alya “SOLIDZ” instance increases from 3 up to 6, the execution time of the preCICE “Advance” operation in the Alya “NASTIN” instance drops (Figure 4.11) because the execution time of the Alya “Compute” operation in the Alya “SOLIDZ” instance scales down (Figure 4.11). Although further increase of the number of processes dedicated for the Alya “SOLIDZ” instance continues to moderately scale down the the execution time of its “Compute” operation at least up to 32 processes (Figure 4.11), this has no more impact on the execution time of the preCICE “Advance” operation in the Alya “NASTIN” instance (Figure 4.11) because the execution time of its “Compute” operation remains constant and is already larger than the execution time of its counterpart in the Alya “SOLIDZ” instance (Figure 4.11). Likewise, the execution time of the preCICE “Advance” operation in the Alya “SOLIDZ” instance grows for similar reasons (Figure 4.11). It should be emphasized once again, that this is that is both PointToPointCommunication::send() and PointToPointCommunication::receive() calls are fully parallel, both inside of each CIP (intercommunication with all of the intercommunication partners of this CIP is done asynchronously) and across the CIPes of a single solver (executions of these calls can freely overlap in different CIPes of a single solver)
not because the preCICE “Advance” operation has scaling issues, but because it contains the intercommunication phase, which inevitably introduces an implicit inter-solver synchronization point that causes one of the solvers to waste time (completely block the execution) in the preCICE “Advance” operation while waiting for another solver to reach the same synchronization point (i.e. being ready to intercommunicate). As a result, in order to efficiently utilize available

---

**Figure 4.10** Time Step (Strong Scaling)

---

even without the additional code instrumentation with an explicit inter-solver barrier
computational resources, it is important to know the inter-solver load balance ratio, especially for large-scale production runs.

Finally, consider the execution times of the preCICE “Communicate” operation corresponding to the intercommunication phase exclusively (Figure 4.11). The peculiar scaling pattern can be observed for the Alya “SOLIDZ” instance, and deserves a thorough explanation on its own. On the one hand, as the number of processes dedicated for the Alya “SOLIDZ” instance increases
from 3 up to 16, the execution time of its preCICE “Communicate” operation gradually rises from 3 ms up to 6 ms (Figure 4.11). On the other hand, as the number of processes dedicated for the Alya “SOLIDZ” instance increases further from 16 up to 128, the execution time of its preCICE “Communicate” operation gradually scales from 6 ms down to 2 ms (Figure 4.11). In order to understand this phenomenon, the hardware configuration (Subsection 4.3.1), the process distribution across the computing nodes, and the changes of the number of intercommunication partners (Figure 4.8) have to be accounted. In particular, according to Subsection 4.3.1, each thin node has 16 physical CPU cores. For every single run out of total of ten runs for this test case, there have been reserved the corresponding minimum number of thin nodes to accommodate exactly one process on each physical CPU core. However, note that for the first six runs, one single thin node, which has to accommodate processes dedicated for the Alya “SOLIDZ” instance, always remains underfilled, i.e. some of its physical CPU cores remain idle because there are simply not enough processes to occupy them. Thus, for the first seven runs, all of the processes dedicated for the Alya “SOLIDZ” instance end up running on one single thin node. At the same time, a single thin node is nothing else but a NUMA domain, i.e. each of its physical CPU cores shares access to a single HCA. Considering how the (total) number of intercommunication partners grows from 132 up to 327 with the first seven runs (Figure 4.8) and knowing that the number of intercommunication partners is equivalent to the number of concurrent (asynchronous) intercommunication requests to the underlying HCA (Section 3.6), it can be inferred that a single thin node (and its HCA) is simply being overwhelmed by the growing number of concurrent intercommunication requests what gradually rises the execution time of the preCICE “Communicate” operation from 3 ms up to 6 ms (Figure 4.11). As the number of thin nodes occupied by the Alya “SOLIDZ” instance consistently increases in the last three runs, while the (maximum, minimum, and average) number of intercommunication partners per CIP consistently decreases (Figure 4.8), it is no surprise that the execution time of the preCICE “Communicate” operation scales back, from 6 ms down to 2 ms (Figure 4.11).

This brings an interesting conclusion that in order to maximize the efficiency of the fully parallel P2P intercommunication (Section 3.6), there should be as less concurrent intercommunication requests per computing node as possible. Although it seems to contradict the purpose of the fully parallel P2P intercommunication, it is the natural trade-off to make and be aware of. For instance, one feasible improvement is to distribute the processes of the coupled solvers across the computing nodes in such a way that each computing node would run at most one CIP. However, this might be quite tricky to achieve in practice and does not seem to be worth trying since the actual performance impact of this issue looks negligible for a real-world application (Figure 4.11). All in all, the measurements presented in this section constitute a clear evidence that the offered implementation of the P2P intercommunication technique (Section 3.6) is indeed successful and ready for industrial consumption.

---

\[^{64}\] when the number of processes dedicated for the Alya “SOLIDZ” instance increases from 3 up to 8 (Figure 4.11)
\[^{65}\] when the number of processes dedicated for the Alya “SOLIDZ” instance increases from 3 up to 16 (Figure 4.11)
\[^{66}\] even though the (total) number of LVDIs being intercommunicated remains constant (Figure 4.9)
The main goal of this thesis was to get rid of the intercommunication bottleneck (Chapter 2) entirely and ensure nearly ideal strong and weak scaling of the preCICE intercommunication routines. This goal has been successfully attained thanks to a fully parallel P2P intercommunication technique and its prerequisites (Chapter 3). Following the rationale from Chapter 2 and the experiences with similar existing coupling software reviewed in Chapter 1, the parallelism is coming not only from the fact that the processes of coupled (intraparallel) solvers can run in parallel themselves but also from parallelization of intercommunication requests inside each of the processes in the first place. As stated in Section 2.3, this is achieved by overlapping as many asynchronous (non-blocking) communication requests per CIP to its intercommunication partners as possible (in a single intercommunication session). Thus, the resulting implementation of P2P intercommunication technique is efficiently parallelized, eagerly keeps bandwidth as close to the corresponding saturation level as possible, and avoids potential intercommunication deadlocks (refer to Section 3.6 for concrete implementation details).

Another important objective has been fulfilled, that is the implementation of the P2P intercommunication technique (Section 3.6) transparently supports both the TCP/IP and the MPI network communication standards (Chapter 3). This strengthens overall flexibility, reliability, and portability of the preCICE library.

The reliability of the implementation of the P2P intercommunication technique (Section 3.6) has been verified and confirmed with a combination of unit tests (Section 4.1) and integration tests (Section 4.2). The performance improvements have been verified and confirmed by profiling one scalability benchmark (Subsection 4.3.2) and one (real-world) application scenario (Subsection 4.3.3), which involve coupling of the actual third-party simulation software, on the SuperMUC massively parallel system (Subsection 4.3.1). The scalability benchmark has proven the ideal strong scaling capabilities of the implemented P2P intercommunication technique at least up to 32768 processes overall (Subsection 4.3.2), while the application scenario has proven that the overhead of the implemented P2P intercommunication technique scales and is extremely small (Subsection 4.3.3), compared to the rest of the simulation work, even in case of the imposed asymmetry and throughput imbalance of the intercommunication pattern.

Although, all of the initial goals have been successfully accomplished, there is still room for improvement and extensions of preCICE. There are two important ideas that are worth to mention. They both refer to large-scale simulations, and especially the upcoming exascale opportunities.

First of all, as observed in Subsection 4.3.3, for the implemented P2P intercommunication technique (Section 3.6), the current approach to address publishing and obtaining in the both underlying one-to-many communication implementations, the TCP/IP socket-based one (Section 3.2) and the MPI port-based one (Section 3.3), can pose serious problems with certain system configurations and will aggravate even further for these system configurations in case of exascale simulations. To solve this problem, a different mechanism of address publishing and obtaining has to be implemented, for example, based on (1) shared memory inter-process communication.
(IPC) [69], or (2) a dedicated address publishing server process with a known beforehand combination of port and IP address which can be queried by the requesting CIPes in order to obtain the P2P addresses accordingly.

Secondly, there is still a potential memory bottleneck present in preCICE. It is related to the fact that master processes of the coupled solvers have to load and store the whole meshes in their corresponding address spaces. Hence, on the computing nodes which run these master processes, the corresponding memory footprint could be substantial. Especially for exascale simulations, this might quickly lead to memory deficiency. Apparently, the solution is to distribute the meshes across the processes of the coupled intraparallel solvers and accordingly reimplement all algorithms for mesh processing to operate on a distributed form.

[70] better directly when reading the meshes by using MPI I/O
BIBLIOGRAPHY


http://dx.doi.org/10.1007/978-3-642-22244-3_7


http://eclipse.org/

http://sites.onera.fr/cwipi/


http://congress.cimne.com/marine2013/frontal/ProgSesion.asp?id=44

[27] http://www.mpcci.de/


http://dx.doi.org/10.1002/cpe.913


[38] http://www.salome-platform.org/


[53] http://wikipedia.org/wiki/Host_adapter#InfiniBand


http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf


[58] http://www.lrz.de/services/compute/supermuc/systemdescription/

http://www.lrz.de/services/compute/supermuc/filesystems/gpfs-usage/
http://www.lrz.de/services/compute/supermuc/filesystems/#TOC33

http://dx.doi.org/10.1016/j.compfluid.2014.04.020
http://wikipedia.org/wiki/Large_eddy_simulation
http://glaros.dtc.umn.edu/gkhome/views/metis
http://wikipedia.org/wiki/Inter-process_communication
http://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/node171.htm