Computational Science and Engineering

Technische Universität München
Fakultät für Informatik

Coupling TherMoS with preCICE

Master's Thesis by Dominik Volland

1st examiner:	Prof. Dr. Hans-Joachim Bungartz
2nd examiner:	Prof. Dr. Ulrich Walter
1st advisor:	Benjamin Rüth
2nd advisor:	Matthias Killian
Submission Date:	July 19th, 2019

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Garching, July 19th, 2019 Dominik Volland

Acknowledgements

I wish to express my deep gratitude to my supervisors, Benjamin Rüth and Matthias Killian, for their great support.

In addition, preCICE developers Benjamin Uekermann and Gerasimos Chourdakis also provided lots of helpful ideas and suggestions, for which I am very grateful. Special thanks go to both of them and Benjamin Rüth for giving me the opportunity to present my preliminary results at the COUPLED 2019 conference.

Finally, I want to thank my family and my friends for supporting me on a personal level while I was finishing this thesis.

Abstract

The Thermal Moon Simulator (TherMoS) is a tool to compute transient thermal models for planning space missions on the Moon surface. It couples a thermal solver with a ray tracer running on GPUs to compute radiative heat fluxes.

The coupling in TherMoS was originally done in a monolithic way. In this work, we present and discuss a new implementation which uses the preCICE library for coupling the components. preCICE is a tool developed for black-box coupling of arbitrary physical solvers in multiphysics simulations.

The work is beneficial for both TherMoS and preCICE. While numerical experiments revealed that preCICE can not improve the performance of TherMoS, the tool could still be improved from a software engineering perspective by providing a clean partitioned interface in contrast to the previous monolithic coupling. Furthermore, preCICE gives access to implicit coupling schemes, which may be interesting to use in TherMoS in the future. For preCICE, the use in TherMoS is an interesting use case, as it was hitherto mainly used in the fields of FSI and CHT.

Glossary of abbreviations

MATLAB	MATrix LABoratory
MEX	MATLAB executable
GPU	Graphics Processing Unit 1
TherMoS	Thermal Moon Simulator 1
preCICE	Precise Code Interaction Coupling Environment 2
FSI	Fluid Structure Interaction 2
CHT	Conjugate Heat Transfer 2
RHD	Radiation Hydrodynamics 5
FDM	Finite Difference Methods 10
FEM	Finite Element Methods 10
FVM	Finite Volume Methods 10
LPM	Lumped Parameter Methods 10
MPI	Message Passing Interface 11
TUM	Technische Universität München 11
CUDA	Compute Unified Device Architecture 25
PCT	Parallel Computing Toolbox 45

Contents

Acknowledgements v

Abstract vii

Glossary of abbreviations ix

1. Introduction 1
- 1.1. Motivation and Goals . . . 2
- 1.2. Thesis structure . . . 3

2. Background 5
- 2.1. Multiphysics problems . . . 5
  - 2.1.1. Overview . . . 5
  - 2.1.2. Numerical approaches . . . 5
  - 2.1.3. Coupling techniques . . . 6
- 2.2. Heat transfer and thermal modeling . . . 7
  - 2.2.1. Foundations of heat transfer . . . 8
  - 2.2.2. Heat transfer in space . . . 9
  - 2.2.3. Numerical methods . . . 10
- 2.3. preCICE . . . 11
  - 2.3.1. Overview . . . 11
  - 2.3.2. Configuring preCICE . . . 12
  - 2.3.3. A preCICE adapter . . . 14
  - 2.3.4. Mapping . . . 14
  - 2.3.5. Communication and process mapping . . . 17
- 2.4. Ray tracing . . . 18
  - 2.4.1. Reflection of rays . . . 19
- 2.5. Interfacing MATLAB with C++ . . . 19
  - 2.5.1. MEX and Engine APIs . . . 19
  - 2.5.2. C MEX API and C Matrix API . . . 20
  - 2.5.3. C++ MEX API and C DATA API . . . 21
  - 2.5.4. Data handling in MATLAB and in the MEX APIs . . . 21
  - 2.5.5. Out-of-process execution . . . 23
- 2.6. Existing implementation of TherMoS . . . 23
  - 2.6.1. Overview . . . 23
  - 2.6.2. Solver implementation . . . 24
  - 2.6.3. Ray tracer implementation . . . 25

2.6.4. The memory map based interface . . . 25
2.6.5. Multi-GPU parallelization . . . 26

3. Implemenation **29**
3.1. Matlab bindings . . . 29
3.1.1. Requirements discussion and API choice . . . 29
3.1.2. Description of the implementation . . . 30
3.1.3. Out-of-process variant . . . 33
3.2. Coupling Configuration . . . 34
3.3. MATLAB solver adapter . . . 35
3.4. Ray tracer adapter . . . 36
3.4.1. Floats in the ray tracer . . . 36
3.4.2. Broadcast-reduction approach in the ray tracer . . . 37

4. Results and conclusions **41**
4.1. Results . . . 41
4.1.1. Test system . . . 41
4.1.2. Validation of Results . . . 41
4.1.3. Performance Tests . . . 42
4.2. Conclusion . . . 44
4.3. Future work . . . 45
4.3.1. On the MATLAB bindings . . . 45
4.3.2. On TherMoS . . . 46

A. Full code for TherMoS coupling **49**
A.1. Configuration file for TherMoS . . . 49
A.2. TherMoS solver adapter . . . 52
A.3. TherMoS ray tracer adapter . . . 55
A.3.1. Common function . . . 55
A.3.2. Original variant . . . 57
A.3.3. Conservatie mapping variant . . . 62

1. Introduction

It happens that on the very day that this thesis is being submitted, exactly fifty years have passed since the Saturn V rocket of the Apollo 11 mission entered a lunar orbit[1]. One day later, it landed on lunar surface, allowing the first human beings to set foot on the Moon and thus marking a momentous milestone in aerospace endeavors.

In these fifty years, the world has changed incredibly. Technological advancements have pushed the limitations to the capabilities of humanity further and further, often beyond our own imagination. Among countless others, these advancements have, of course, also impacted the domain of spaceflight. Nowadays, humanity is gearing itself up for landing on Mars.

Planning a space mission poses numerous challenges: The absence of breathable air in space and the huge distances to overcome may be the most evident examples. An aspect no less important than these is the necessity to cope with extreme temperatures: The Sun's radiation heats the surface of celestial bodies with distressing rapidity, while in the absence of sunlight, temperatures drop to barely imaginable lows. This environment is hostile not only to astronauts, but also to the spacecrafts themselves and other equipment. Consequently, thermal simulation is a vital part of preparing aerospace missions. With the increase of computational power, the computation of *transient* thermal models has become possible and gained significance.

There are several peculiarities to thermal models on the Moon as opposed to the Earth. First of all, the aforementioned lack of an atmosphere in space means there is no or negligible heat transfer via convection, which is often the dominant heat transfer mechanism on Earth. Thus, especially radiative heat transfer plays a core role in lunar thermal simulations. Secondly, the countless smaller celestial bodies that have impacted the Moon's surface over the eons have rendered it a very rugged and rough surface. While some areas are exposed to full sunlight, other spots only a few meters away may be shadowed completely, resulting in huge temperature differences. This means that high spatial accuracy is required for lunar thermal models.

The task of simulating and computing radiative heat transfer is very complex, as it involves determining view factors between the surfaces exchanging heat via radiation. Computing these analytically is nigh impossible. For this reason, it has become common to use ray tracing for this task. This is implemented, for example, in one of the leading softwares in aerospace industry, the ESATAN-TMS thermal modeling suite[33].

Ray tracing is a task which can be performed particularly efficiently on Graphics Processing Units (GPUs). In 2013, Philipp Hager proposed a new thermal simulation tool specifically geared towards transient models for moving objects on the Moon surface[21]. His tool, named Thermal Moon Simulator (TherMoS), was mainly developed in the widespread MATLAB software[25], but it uses an interface to a ray tracer, which

is based on NVIDIA's OptiX framework[35]. This framework is based on CUDA[36], which is NVIDIA's language for general purpose computing on GPUs, thus allowing TherMoS to exploit the advantages of GPUs.

The use of multiple components in a software always poses an extra challenge: How can the components be coupled? This question is even more important in scientific software, where good performance is naturally required. The existence of an interface always induces overhead due to synchronization and communication, and it is desirable to minimize this overhead. The original implementation simply used comma-separated text files for communication, which was easy to implement, but extremely slow. In early 2018, Mohammad Alhasni improved TherMoS by extending the existing implementation to work with multiple GPUs instead of just one. In the course of his work, he also developed a new interface using memory mapped files, which yielded a significant speed up.

A collaboration of scientific computing chairs at German universities develops a library named Precise Code Interaction Coupling Environment (preCICE) allowing users to couple scientific codes[6]. The library in mainly used in the fields of Fluid Structure Interaction (FSI) and Conjugate Heat Transfer (CHT), but it can be used in other fields as well. The main aim of this thesis is to delegate the coupling necessary in TherMoS to preCICE and to discuss the results.

1.1. Motivation and Goals

A simple question arises: Why? Which benefit comes from replacing the existing interface with the preCICE library?

From the perspective of TherMoS. First of all, the change may improve TherMoS from a software engineering perspective. The current interface provides only a straightforward data exchange mechanism and is restricted to the existing components. In contrast, preCICE is designed to allow black-box coupling between arbitrary solvers. Thus, if either of the components is replaced by different implementations in the future, or if further functionalities are extracted from the MATLAB component into separate components, this can be performed easily, as opposed to completely restructuring the code. Furthermore, preCICE does not only offer a simple interface for communication. It comes with a number of built-in coupling features that can be used without writing extra code. Among these features is a palette of parallel and implicit coupling schemes including post-processing methods to increase the stability of the coupled algorithm and the accuracy of solutions. This is something which may be interesting for TherMoS: While TherMoS is not reliant on the increased stability coming with implicit coupling schemes, it would still be interesting to try whether more sophisticated coupling techniques increase the performance or the accuracy of solutions. Implementing these techniques without preCICE would require a complete restructuring of the implementation - with preCICE, this comes down to a simple change in the configuration.

From the perspective of preCICE. However, beyond that, the work done in this thesis is also interesting for the developers of preCICE. As mentioned earlier, preCICE understands itself as general-purpose multi-physics coupling library, but up to now, it has hardly been used outside FSI and conjugate heat transfer. The coupling for TherMoS is thus a welcome case study to survey the usability of preCICE in a hitherto untested physical setup.

Apart from that, the usage of preCICE for TherMoS also requires the implementation of MATLAB bindings for preCICE. This is a feature that had been requested for a while, since MATLAB is one of the most widespread languages[1] in engineering today.

Finally, TherMoS also runs on hybrid architecture due to one component running on the CPU and the other component running on GPUs. To the best of out knowledge, preCICE has not been used on hybrid architectures before.

1.2. Thesis structure

The thesis is organized as follows. Chapter 2 will introduce the necessary prerequisites. We start with the theoretical background of coupled problems and heat transfer and move on to the tools for tackling these problems: preCICE as a library to solve coupled problems and ray tracing to compute radiative heat fluxes. Next, we introduce the C++ APIs provided by MATLAB, which are used in both the previous implementation of TherMoS and the MATLAB bindings for preCICE. We close the chapter by describing the existing implementation of TherMoS.

In Chapter 3, we move to the author's implementation. We describe the MATLAB bindings and the MATLAB adapter for TherMoS based on them. We also describe the configuration of the coupling in TherMoS and the adapter for the ray tracer. We will highlight some problems arising from our situation and mention possible solutions.

Finally, in Chapter 4, we present and discuss the results of applying preCICE to TherMoS, with a focus on performance aspects. We also shed light on some challenges for the future.

[1] While MATLAB is mainly a software, it also provides an extensive high-level programming language. We will therefore refer to it as both language and software throughout this thesis.

2. Background

2.1. Multiphysics problems

This section gives a brief explanation of the terms "multiphysics" and "coupled". It provides a stub into solution approaches to these kinds of problem, allowing the reader to classify the approach pursued by preCICE in a broader context.

2.1.1. Overview

The term "multiphysics" is defined differently in the literature [29]. In an attempt to sum up, a multiphysics problem consists of two[1] different components, physical models or physical phenomena, which have to be taken into account for solving one problem. Examples include

- Fluid Structure Interaction problems, which model the interaction between a fluid and a flexible solid[7]. For instance, simulating the flow of blood in an artery is a FSI problem.
- Radiation Hydrodynamics (RHD), which deals with electromagnetic radiation affecting the behavior of a fluid[8].

To solve a multiphysics problem, it is necessary to treat not only the two individual components, but also the interaction between them, which is known as coupling. Multiphysics problems can be subdivided into two classes, depending on where the coupling occurs[27]: The coupling may either span the whole physical domain, as it is the case in RHD, which is termed bulk coupling or volume coupling. Or, it can occur only at an interface of lower dimension, like for example in FSI.

2.1.2. Numerical approaches

In general, two different approaches are possible to tackle coupled problems numerically. On the one hand, we can follow a *monolithic* approach. I.e., the problem is formulated in one global system of equations and solved with a solver exclusively devoted to the problem as a whole. Alternatively, it is possible to follow a *partitioned* approach: For each component, we can use a solver which is devoted to dealing with one of the components involved, and couple these solvers in a black-box way. To implement this, we use

[1] Naturally, a multiphysics problem may also consist of more than two components. However, for the sake of simplicity, we confine ourselves with two components.

a library or framework which enables communication between two solvers and provides techniques to couple the solvers. Several such tools are available, like C-Coupler[32] or the preCICE library we will use.

Both approaches have advantages and disadvantages. Partitioned problems often suffer from robustness issues or require a longer run time[34]. The monolithic approach can deal with the structure of the problem in a better way. Solving the arising equiation system can be challenging, but a number of well-studied preconditioners can be found in literature, see, for example, [17].

The great downside of monolithic solvers, however, is the necessity to develop a solver from the beginning. A good reason to follow the partitioned approach is that in many cases, well-developed solvers for the individual components of the problem already exist. The approach enables us to reuse these solvers. This greatly reduces the amount of code that needs to be written and maintained. In the case of interface coupling, performing a black-box coupling furthermore does not require knowledge of or access to all internal details of the solvers – it suffices to know about the discretization at the coupling interface. Beyond that, using a flexible black-box interface also allows to modify or even replace the solvers on either side without the need to adapt the other solver. In other words, the partitioned approach implies low coupling of the software components, which is desirable from a software engineering perspective[5].

There are also approaches that lie between the two extremes of fully partitioned and fully monolithic coupling, e.g., Mortar methods[3].

2.1.3. Coupling techniques

We will now briefly discuss different coupling techniques for black-box coupling as described above. That is, we assume that we have two solvers for our subproblems at hand. Roughly speaking, the question we want to answer is how often and in what order we will execute our solvers. The discussion here is based on the third chapter of [42], which is specifically about FSI, but it applies to other multiphysics problems as well.

We formally write the two solvers as two operators F and G acting on the coupling scheme:

$$F : \mathbb{R}^n \to \mathbb{R}^m, F(x) = y, \text{ and } G : \mathbb{R}^m \to \mathbb{R}^n, G(y) = x,$$

where $m, n \in \mathbb{N}$ depends on the size of the coupling interface and the amount and dimension of data communicated. The application of F resp. G corresponds to the computation of a single time step in one of the solvers. One solver takes a certain physical quantity x as input and returns a different quantity y, while the other takes y as input and returns x. In FSI, for example, a structure solver computes displacements on the structure x from the forces y acting on the structure. The fluid solver, on the other hand, computes velocities and forces y arising from the displacements x.

We now have to choose a *coupling scheme* which describes how the coupling of the components is carried out. Coupling schemes can be categorized in two ways: They can be either explicit or implicit, and both explicit or implicit schemes can be either serial or parallel. In the following, let $x^{(j)}$ and $y^{(j)}$ be the solutions computed in the j-th time

step. We start with explicit schemes.

Serial means that the solvers are executed in a staggered way, i.e. either solver blocks while the other one is running, and results are exchanged between the runs. This means that the results of the next steps are obtained as

$$y^{(j+1)} = F\left(x^{(j)}\right),\ x^{(j+1)} = G\left(y^{(j+1)}\right).$$

Parallel coupling, on the other hand, means that the solvers are applied simultaneously to the results of the previous time step, allowing them to run in parallel. This can, for example, be done as

$$y^{(j+1)} = F\left(x^{(j)}\right),\ x^{(j+1)} = G\left(y^{(j)}\right)$$

or

$$x^{(j+1)} = (G \circ F)\left(x^{(j)}\right),\ y^{(j+1)} = (F \circ G)\left(y^{(j)}\right)$$

depending on the structure of the problem. Note that it is important to distinguish between a parallel solver and parallel coupling: One or both of our solvers might already be implemented to run in parallel on multiple cores. However, they can still be coupled to the other solver either in a serial way or in a parallel way, if additional cores are available.

As stated previously, the equations above resemble **explicit** coupling: All solvers are only executed once or a fixed number of times per simulation time step. In contrast to this, **implicit** coupling means that the solvers are executed multiple times during a single simulation time step to reach an equilibrium solution for this time step. In other words, we solve the fixed-point equation

$$y^{(j+1)} = F\left(x^{(j+1)}\right),\ x^{(j+1)} = G\left(y^{(j+1)}\right).$$

Note that the $x^{(j+1)}$ and $y^{(j+1)}$ computed by the explicit schemes do in general *not* satisfy this equation. To solve this equation, we apply a fixed-point iteration - i.e., the solvers are run multiple times to compute iterative solutions $x_k^{(j)}$, $y_k^{(j)}$ for $k = 0, 1, 2, \ldots$ which converge to the equilibrium solution. Since simple fixed-point iterations may converge very slowly or not at all[28], it is common to apply a post-processing method after each iteration to improve the current iterate in some sense. There are multiple post-processing methods, ranging from simple underrelaxation schemes to sophisticated Quasi-Newton methods[42].

Clearly, explicit coupling is a lot cheaper, since the number of calls to the solver is limited. However, implicit coupling is usually more stable and gives more accurate results. Depending on the problem, explicit coupling is often not sufficient to give a stable solution. A well-known example for this phenomenon is the added-mass effect in FSI[9].

2.2. Heat transfer and thermal modeling

This section will give a brief introduction to heat transfer. Since our work focuses on the implementation of the coupling, we will omit most details and refer the interested

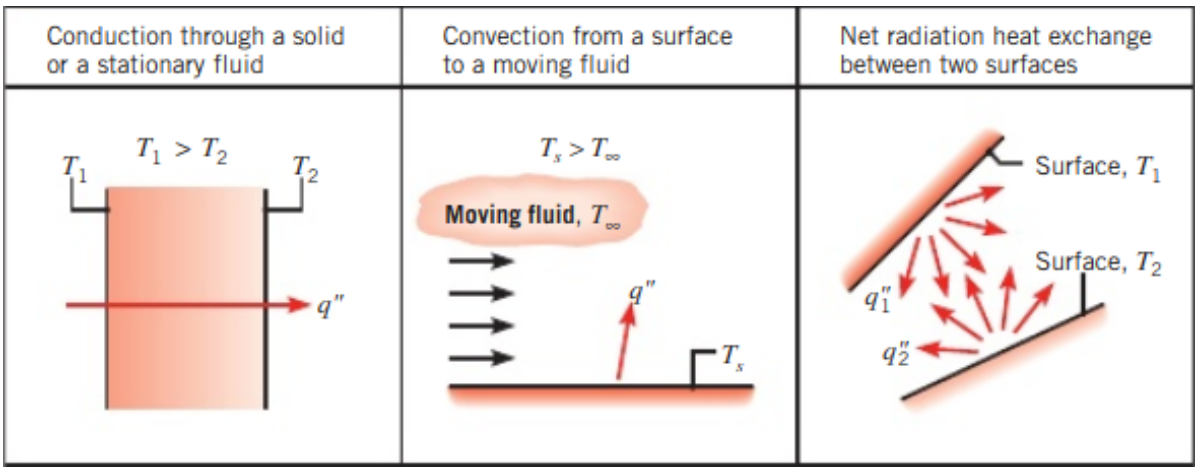


Figure 2.1.: Illustration of the different mechanisms of heat transfer. Figure taken from [26]

reader to our literature references. The introduction given here is derived from the books of Çengel [10] and Incropera [26]. The interested reader may consult these books for further information; to the German reader, we further recommend [37].

2.2.1. Foundations of heat transfer

Heat is a form of energy that can be transferred between physical systems of different temperature. The temperature gradient causes energy to be transferred from one system to the other, which in turn causes the temperatures to change.

Heat is usually denoted by Q. The speed at which heat is transferred, i.e., the amount of heat transferred per unit of time, is called *heat transfer rate* and denoted by $\dot{Q}$. Its unit is W. The quantity we will be concerned with most of the time is the *heat flux*, which is the heat transfer rate per unit of area orthogonal to the direction of heat transfer. We denote it by q and its unit is $\mathrm{W} \cdot \mathrm{m}^{-2}$. [10, Section 1-3]

In general, there are three different types of heat transfer, which we will briefly describe in the following paragraphs. Figure 2.1 illustrates the different mechanisms.

Conductive heat transfer occurs through solids or stationary fluids. The molecules of each body with a temperature above 0 Kelvin are in constant movement, which causes collisions with other molecules. Due to these collisions, energy is transferred from molecules with higher energy to molecules with lower energy. As a consequence, heat transfer occurs on a macroscopic scale in the direction of decreasing temperature.

Conductive heat transfer is quantified in Fourier's law of heat conduction. This law states that the conductive heat transfer rate $\dot{Q}_{cond}$ in direction x across an area A is given by

$$\dot{Q}_{cond} = -kA\frac{\mathrm{d}T}{\mathrm{d}x},$$

where $\frac{\mathrm{d}T}{\mathrm{d}x}$ is the temperature gradient in direction x and k is the *thermal conductivity*, a constant depending on the material.

Convective heat transfer describes heat exchanged between a surface and a moving fluid or gas which is in contact with the surface. This actually consists of two different mechanisms: On the one hand, energy is exchanged on a microscopic scale through molecular motion. On the other hand, the macroscopic motion of the fluid resp. gas transports heat as well. Convective heat transfer is governed by Newton's law of cooling, which we omit since we will not need it in the following.

Radiative heat transfer is the exchange of energy by electromagnetic waves. Changes in the electronic configuration of atoms or molecules induce the emission of radiation at different wave lengths. In contrast to the other mechanisms, the radiation requires no medium to travel and thus also occurs in vacuum. The Stefan-Boltzmann equation states that the radiative heat-transfer rate $\dot{Q}_{rad}$ emitted by a surface of area A in vacuum is given by

$$\dot{Q}_{rad} = \varepsilon A \sigma T_s^4,$$

where T_s is the *absolute temperature* of the surface, σ is a constant called *Stefan-Boltzmann constant* and $0 \leq \varepsilon \leq 1$ is, again, a constant depending a material, the so-called emissivity[2].

When radiation from a different body arrives at an opaque surface, part of it is absorbed, while the remaining part is being reflected[3]. The fraction of incoming radiation absorbed by a body is termed absorptivity α. It holds $0 \leq \alpha \leq 1$.

For two given surfaces, the radiative flux between them can be computed as

$$q_{ij} = -\alpha_j \varepsilon_i \sigma F_{i,j} (T_j^4 - T_i^4)$$

where the subscript indices are used to denote the two different surfaces. Here, $F_{i,j}$ is the *view factor* (or configuration factor) between the surfaces, which is obtained as

$$\frac{\cos\theta_i \cos\theta_j}{\pi d_{ij}^2} A_j.$$

Here, d_{ij} is the length of the straight line between the centers of the surfaces, and θ_i is the angle between this line and the surface normal of the respective surface.

2.2.2. Heat transfer in space

The goal of our application is to perform thermal modeling on the Moon surface. This has a very important consequence: Since the atmosphere of the Moon is negligible and there are no fluids on the Moon, convective heat transfer is negligible. In contrast, radiative

[2]In fact, emissivity and also the absorptivity discussed later depend on the material's temperature, but this is ignored in most applications.

[3]For transparent surfaces, a part of the radiation will also pass through the surface.

heat transfer becomes a very important mechanism, since radiation can travel through the vacuum unhindered, while it is attenuated in other physical media. Conductive heat transfer also occurs in the interior of a celestial body (the Moon, in our case) or a spacecraft.

Spacecrafts and equipment involved in a space mission are all subject to certain temperature constraints. For example, electronic devices will only operate as long as their temperature remains in certain ranges. Of course, the same applies to astronauts in manned space missions. This gives rise to the necessity of *thermal control*: When planning the mission, steps must be taken to ensure that the temperatures stay within the constraints. This can be done passively by a careful choice of materials and surface coatings to impact the absorptivity and emissivity of the equipment, or actively by adding heating or cooling devices.

To determine the requirements of thermal control, programs like TherMoS are used to perform thermal modeling. I.e., the temperatures that the equipment will be exposed to during its mission are computed numerically.

2.2.3. Numerical methods

For numerical computations, the physical domain in question is discretized. There are various different numerical approaches to perform the computations. The most popular of these are Finite Difference Methods (FDM), Finite Element Methods (FEM) and Finite Volume Methods (FVM). While FEM have been on the rise in the last decades due to their accuracy and low requirements on regularity, the discipline of spacecraft thermal modeling is dominated by a special variant of FVM known as Lumped Parameter Methods (LPM).

In LPM, the physical environment is discretized into a number of elements, mostly triangles or quads for surfaces and cubes for three-dimensional bodies. As a simplification, all quantities important for the simulation – temperature, heat fluxes, material properties, and the like – are averaged across the element and assumed to be lumped at a thermal node, hence the name of the method. The thermal node is usually placed in the center of the element.

To compute a transient thermal model, temperatures in each time step are used to compute conductive and radiative fluxes between the nodes. These are, in turn, used to compute the temperatures for the next time step.

Based on Fourier's law, the conductive heat transfer rate between to elements can be computed as

$$\dot{Q}_{ij} = -\frac{kA}{d}\left(T_i - T_j\right),$$

where A is the cross-sectional area of the interface between the elements and d is the distance between the thermal nodes. Computing these fluxes is straightforward: Since conductive heat transfer requires physical contact, it is sufficient to consider pairs of elements which are adjacent to each other. Two elements without a common boundary can not exchange energy conductively.

For radiative heat fluxes, this is significantly more involved: Any two elements can exchange heat via radiation as long as they can "see" each other. In other words, it is necessary to compute the view factor F_{ij} between the elements, and also to determine whether other elements located "between" the two elements in question obstruct the view. This is far from trivial, in fact, it is usually impossible to do analytically. As a consequence, several numerical solvers for thermal modeling perform this task via ray tracing.

2.3. preCICE

This section will give an introduction to preCICE, focusing on the perspective of the users and giving small insights into the implementation where it is important for our work.

2.3.1. Overview

preCICE is being developed jointly by the Chair of Scientific Computing at the Technische Universität München (TUM) and the Institute for Parallel and Distributed Systems at the University of Stuttgart. The current release and the whole source code are available on GitHub[38], along with an extensive wiki[40]. preCICE is a library mainly implemented in C++ which allows partitioned black-box coupling for arbitrary physical solvers. The library is geared towards surface coupling as opposed to volume coupling described in Section 2.1. In principle, however, it could also be used for bulk coupling. preCICE also supports the coupling of more than two solvers, but like in Section 2.1, we will assume that we only have to deal with two solvers for the sake of simplicity.

As opposed to a framework approach, preCICE follows a more flexible library approach: It is not an application itself, rather, it provides an API which is called by the solvers. The user needs a so-called adapter calling the library functions for each solver he wishes to couple. For many popular both open source and commercial solvers, like OpenFOAM, ANSYS or FEniCS, there exist ready-to-use adapters which are actively maintained by the preCICE developers. For other solvers or in-house codes, users have to write adapters themselves. The preCICE wiki offers a step-by-step tutorial detailing how to do this.

preCICE is written in C++, hence, the original API is in written in C++, too. To simplify writing adapters for solvers in different languages, preCICE offers binding for other languages, like Python and Fortran, allowing solvers written in these languages to access the library without writing C++ code.

One of the main ambitions of preCICE is to support massively parallel simulations. To achieve this, preCICE allows easy integration into existing Message Passing Interface (MPI) environments. A unique selling point is the use of a parallel peer-to-peer concept: The processes on either side of the coupling communicate directly with each other without requiring a central instance.

Most of the library revolves around a class named `SolverInterface`. An adapter instantiates an object of this class and configures it using an XML configuration file. Afterwards, the coupling is initialized, and the solver blocks until a connection to the other solver has been established. Once the solvers are successfully connected, they can optionally set initial values for the coupling data before entering a computational loop, which performs the actual time stepping.

For a better understanding of the capabilities of preCICE and the steps to develop codes coupled with preCICE, we will detail the structure of configuration files and adapters in the following subsections.

2.3.2. Configuring preCICE

An example configuration file is shown in Listing 1. We give a rough overview here, for details we refer to the corresponding pages in the preCICE wiki[40].

We first provide the physical dimension of the solver, which defines the dimension of vector data and the coordinates of mesh vertices. Currently, preCICE only supports two or three dimensions. Next, we set up a nomenclature for the data to be communicated between the solvers.

Afterwards, meshes are introduced. We specifiy the names of meshes involved in the simulation and declare the data associated with each mesh. Usually, there will be one mesh for each solver, but additional meshes can be specified as desired. At this point, we do not yet declare the geometry of the solvers. This is done in the adapters instead.

Once meshes and data are defined, it is possible to declare the solvers participating in the simulation. For each solver, we should specify which meshes they use and which data they read and write. In this context, "writing" means that the data is made available to other solvers, while "reading" means that data written by a different solver is obtained. Each mesh must be *provided* by one solver, which means that this solver is responsible for declaring the geometry of the mesh to preCICE. Solvers may use meshes that were provided by other participants. In this case, the `from` attribute is used to specify the participant that provided the mesh. In addition, it is necessary to define a mapping between the meshes. Details about mapping configuration are given below.

The next thing to do is specifying the communication between the participants. This is done in the `m2n` tag. preCICE supports communication either via TCP/IP sockets or MPI Ports, a feature included in version 2.0 of the MPI standard[20]. While MPI ports are usually faster, the preCICE developers recommend using sockets due to stability issues of the ports[40]. If either of the solvers is parallel, a master tag has to be added to inform preCICE about the communication inside the solver. If the solver is parallelized using MPI, the easiest way to do this is to set `<master:mpi-single>`.

Finally, we can define a coupling scheme. preCICE supports the whole array of methods discussed in Section 2.1: Both explicit and implicit as well as both serial and parallel coupling are available, along with a number of post-processing schemes for the implicit methods.

```
<precice-configuration>

  <solver-interface dimensions="2" >

    <data:vector name="x"  />
    <data:scalar name="y"  />

    <mesh name="MeshOne">
      <use-data name="x" />
      <use-data name="y" />
    </mesh>

    <mesh name="MeshTwo">
      <use-data name="x" />
      <use-data name="y" />
    </mesh>

    <participant name="SolverOne-serial">
      <use-mesh name="MeshOne" provide="yes"/>
      <write-data name="x" mesh="MeshOne" />
      <read-data  name="y" mesh="MeshOne" />
    </participant>

    <participant name="SolverTwo-parallel">
      <master:mpi-single/>
      <use-mesh name="MeshOne" from="SolverOne"/>
      <use-mesh name="MeshTwo" provide="yes"/>
      <mapping:nearest-neighbor   direction="write" from="MeshTwo" to="MeshOne"
      ↪  constraint="conservative"/>
      <mapping:nearest-projection direction="read"  from="MeshOne" to="MeshTwo"
      ↪  constraint="consistent" />
      <write-data name="y" mesh="MeshTwo" />
      <read-data  name="x" mesh="MeshTwo" />
    </participant>

    <m2n:sockets from="SolverOne-serial" to="SolverTwo-parallel"
    ↪  distribution-type="gather-scatter"/>

    <coupling-scheme:serial-explicit>
      <participants first="SolverOne-serial" second="SolverTwo-parallel" />
      <max-timesteps value="10" />
      <timestep-length value="1.0" />
      <exchange data="x" mesh="MeshOne" from="SolverOne-serial"
      ↪  to="SolverTwo-parallel" />
      <exchange data="y" mesh="MeshOne" from="SolverTwo-parallel"
      ↪  to="SolverOne-serial"/>
    </coupling-scheme:serial-explicit>

  </solver-interface>

</precice-configuration>
```

Listing 1: Example for a preCICE configuration file. Adapted from [38].

2.3.3. A preCICE adapter

Listing 2 gives pseudo code demonstrating the structure of preCICE adapters.

As mentioned before, the API revolves around an instance of the `SolverInterface` class. First of all, we call `configure()` to read and process the configuration file written previously.

Afterwards, we declare the coupling mesh to preCICE by setting coordinates for all vertices that carry data to be communicated in the coupling. The `setMeshVertices` function returns IDs that preCICE internally attributes to the specified vertices. These IDs are used to refer to the vertices in the following.

Calling `initialize()` launches the coupling and returns as soon as a connection to the other solver has been established. Once this is successful, preCICE will compute the mappings between the meshes as specified in the configuration file.

Before entering the computational loop itself, the solver may initialize data as specified in the configuration file. After completing this, it calls `initializeData()`. Much like `initialize()`, this function blocks until all solvers have called it.

The loop itself is controlled by the `isCouplingOngoing()` function. In every loop iteration, the adapter should first read data from the other solver, then run the solver itself, and write the data back. Afterwards, they call the `advance` function. This function steers the execution based on the coupling scheme specified in the configuration file.

After both solvers exited the computational loop, `finalize()` is called to clean up.

The input arguments `rank` and `size` to the `SolverInterface` constructor allow for initializing preCICE in an existing MPI environment. The user should call `MPI_Init()` before instantiating the interface. Thus, each process will call the constructor by itself. The size of `MPI_COMM_WORLD` and the rank of the process within this communicator should then be passed to the constructor. It is assumed that the coupling mesh has been partitioned and distributed among the processes previously. Each process should only declare its part of the coupling mesh to preCICE. The entire coupling mesh consists of the parts specified individually by each process.

preCICE itself uses MPI to establish communication via MPI ports, and to determine the master of a parallel participant if the `<master:mpi-single>` tag was used. For this reason, preCICE will initialize MPI if this was not done by the solver previously. Notably, preCICE will call `MPI_Init()` even if sockets are used for communication. This can cause unexpected problems. For example, the author experienced an issue when writing the MATLAB bindings discussed in section 3.1. Certain builds of OpenMPI cause MATLAB to crash if `MPI_Init()` is called from the MATLAB process due to compatibility issues. As a consequence, the bindings could not be used if MATLAB was built with OpenMPI. If users face issues of this kind, the only remedy is to compile preCICE itself without MPI.

2.3.4. Mapping

As stated before, preCICE is designed for interface coupling. The two solver will usually operate on entirely different domains. As a consequence, the interface meshes of the

```
turnOnSolver(); // e.g. some setup steps

precice::SolverInterface interface("SolverTwo",rank,size);
interface.configure("precice-config.xml"); // configuration file

// mesh declaration
int dim = interface.getDimensions();
int meshID = interface.getMeshID("MeshTwo");
int numVertices = getInterfaceSize();
double* coordinates = new double[vertexSize*dim];
getCoordinates(coordinates);
int* vertexIDs = new int[vertexSize];
interface.setMeshVertices(meshID, vertexSize, coords, vertexIDs);

// preparing buffers to hold coupling data
int xID = interface.getDataID("x", meshID);
int yID = interface.getDataID("y", meshID);
double* x = new double[vertexSize*dim];
double* y = new double[vertexSize];

double dt; // timestep size

dt = interface.initialize();

// Initializing coupling data
getInitialData(x); // initial Data is returned in x
if(interface.isActionRequired(actionWriteInitialData())){
    interface.writeBlockVectorData(xID, vertexSize, vertexIDs, x); //
    ↪ writes data in x to the other solver
    interface.fulfilledAction(actionWriteInitialData());
}
interface.initializeData();

while (interface.isCouplingOngoing()){ // time loop
    endTimeStep();
    interface.readBlockScalarData(yID, vertexSize, vertexIDs, y); //
    ↪ data from the other solver is stored in y
    executeSolver(y,dt,x);
    interface.writeBlockVectorData(xID, vertexSize, vertexIDs, x);
    dt = interface.advance(dt);
    endTimeStep();
}
interface.finalize(); // free memory, close communication
turnOffSolver();
```

Listing 2: Example for a preCICE adapter. Adapted from [40].

two solvers will usually not match. The interface meshes might coincide, but their vertices could still be at different positions if either mesh is coarser than the other one. Alternatively, they may even exhibit a completely different geometry, and thus overlap. Figure 2.2 shows examples. Hence, it is necessary to find a way of mapping data from one mesh to the other. For this purpose, preCICE offers different mapping methods that vary in their complexity and accuracy.

The easiest method is **nearest neighbor** mapping. In this case, vertices in one mesh will simply be mapped to the closest vertex in the other mesh. This mapping is cheap to compute and simple, but also a first-order method and thus usually very inaccurate. In most applications, it will not give sufficiently accurate results.

For better results, preCICE provides the second order **nearest projection** method. In this method, a vertex of one mesh is projected onto a face of the other mesh. Afterwards, interpolation is used to compute the actual mapping. Nearest neighbor and nearest projection mapping are illustrated in Figure 2.3. Precice further offers the more sophisticated **radial basis function** mapping, which we will not discuss here. We refer the interested reader to [31].

Either of the mapping techniques may be used either for **consistent** or **conservative** mapping. The difference between these constraints is visualized in Figure 2.4. For exact definitions, we refer to Chapter 4 of [42]. Conservative mapping ensures that the *sum* over the values at all nodes is conserved in the mapping. In contrast, the crucial property of consistent mapping is the preservation of constant functions. This is desirable for properties that are normalized in some sense.

If either of the solvers is parallel, then preCICE only allows consistent read mappings and conservative write mappings, since these mappings are always unique. Consistent write mappings resp. conservative read mappings may be ambiguous, as illustrated in [42]. It is, therefore, good practice to always confine ourselves to these kinds of mapping.

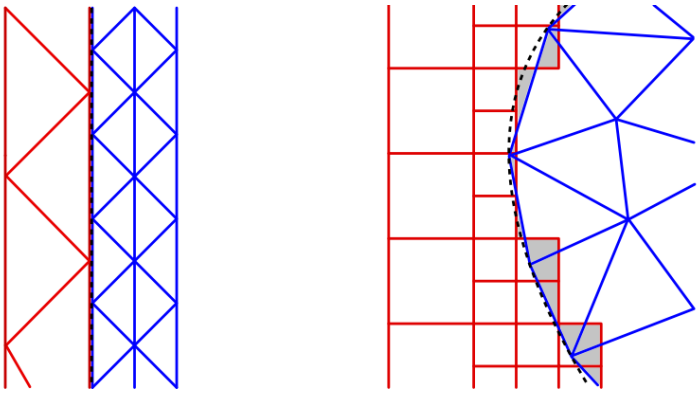


Figure 2.2.: Examples of non-matching coupling meshes. Figure taken from [16]

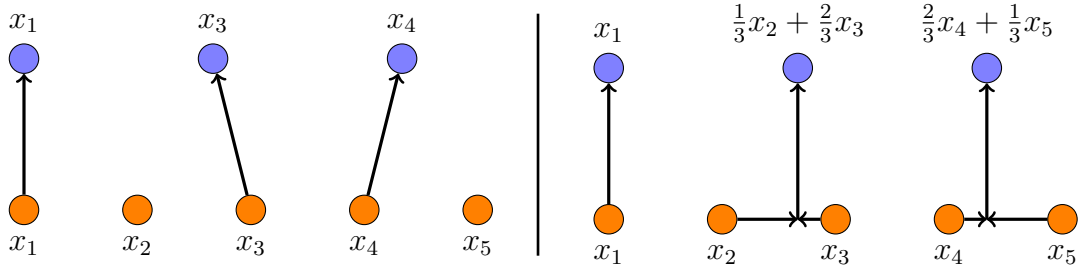


Figure 2.3.: Nearest neighbor mapping (left) versus nearest projection mapping.

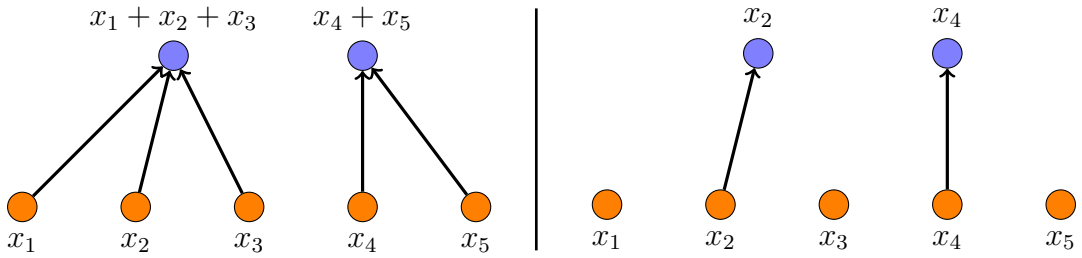


Figure 2.4.: Conservative mapping (left) versus consistent mapping (right)

2.3.5. Communication and process mapping

As mentioned earlier, preCICE follows a peer-to-peer concept. The processes of solvers coupled via preCICE communicate directly with processes in the other solver instead of communicating their data via a central instance[39]. Thus, the user should not think of preCICE as a broker between the peers. Rather, the library simply abstracts away the explicit implementing of ports, sockets or other interprocess communication tools. This approach prevents the creation of a bottleneck at a central instance and enables massively parallel simulations.

This feature is reflected in the way preCICE handles data mapping for parallel solvers. If two parallel solvers are coupled, both have distributed their mesh among their processes. In most cases, the submesh of a certain process in one solver overlaps only with a small subset of the submeshes of the other solver's processes. In other words, each process needs to communicate only with a subset of the other solver's processes. This is taken into account by preCICE: When computing a mapping of the meshes, it also computes a mapping of the processes to reduce communication overhead as far as possible.

In the current preCICE implementation (version 1.5.0), the mapping is computed only once during the initialization phase, before entering the computational loop. Therefore, preCICE can not directly handle adaptive mesh refinement. However, this is planned to be supported in the future.

2.4. Ray tracing

This section gives a brief introduction to the principles of ray tracing. Extensive material on the topic is found for example in [19] and [41].

The method of ray tracing was mainly developed for rendering images, e.g. in computer games. It follows a very simple and straightforward idea: First, the rendering method sets up the geometry of the scene and an image plane representing the view of the observer. Afterwards, the method casts rays representing photons and computes their paths through the scene. Whenever the ray hits an object in the scene, part of its energy is absorbed and the rest is reflected in a specular and/or diffusive way, depending on the material properties of the object hit.

Ray tracing methods fall into two categories[19]: In *forward ray tracing* methods, the rays are cast from the light sources in the scene. Rays that hits the image plane are accounted for to compute the color value at the respective coordinate to finally render the whole image. In contrast to that, *backward ray tracing* methods cast rays from the image plane and trace them until they hit light sources. This variant is significantly cheaper, since the number of rays cast is bounded by the size of the image plane. Therefore, image rendering methods today almost exclusively use backward ray tracing[19] or hybrid techniques[22]. However, backward ray tracing is only applicable if we are only interested in the image plane. If we are interested in results on the whole scene, using forward ray tracing is imperative.

Since radiative heat transfer is based on electromagnetic waves, the method of ray tracing can be used to compute radiative heat transfer between the elements in thermal modeling: Rays are launched from each element in the discrete scene. They carry a certain amount of energy depending on the thermal energy of the element they were launched from. They are traced until they leave the scene or hit other elements. When hitting a surface, part of the energy is conferred to the surface hit, while the remaining energy is reflected in the form of other rays cast from the surface. This way, we can compute the radiative heat fluxes. For transient models, the fluxes have to be computed in each time step. There is an alternative to this: If the elements in the scene are stationary with respect to each other, the view factors F_{ij} in the radiative heat transfer equation are not time-dependent. Hence, it is possible to compute them only once, store them and insert them into the equation whenever the fluxes are being computed. This approach is used in ESATAN, for example[21].

Since TherMoS is geared towards simulating the movement of a spacecraft across the Moon surface, it follows the approach mentioned first. Apart from that, storing the view factors may be problematic anyway, as it consumes lots of memory. While some of the entries are equal to 0, the matrix $(F_{ij})_{ij}$ is in general not sparse. This is easy to imagine: Assume that our scene contains the surface of a concave element, e.g., a crater. From the definition of the view factors, we can see that in this case, all view factors corresponding to two elements of the crater are non-zero. Hence, the storage required for the view factor will usually be quadratic in the number of elements. This can lead to storage problems for large scenes.

2.4.1. Reflection of rays

If radiation hits an opaque surface, part of it will be absorbed, while the remaining part will be reflected[4]. There are two main mechanisms of reflection: Rough surfaces reflect radiation in a *diffuse* way, which means that the reflected radiation is scattered in every direction. Mirrors reflect light in a *specular* way, which means that an incoming ray is reflected into only one specific direction, the mirror direction. Most actual surfaces are neither perfect mirrors nor perfectly diffuse [11]. Therefore, both types of reflection have to be taken into account.

2.5. Interfacing MATLAB with C++

As stated earlier, the TherMoS ray tracer is implemented in C++, while the remaining components are written in MATLAB. This means that, in some way, it is necessary to interface between MATLAB and C++ for TherMoS. Fortunately, MATLAB comes with a number of possibilities to do this.[5] This chapter will introduce the two APIs offered by MATLAB for interfacing to C++ code. It serves as preparation to explain the existing implementation on the one hand and the MATLAB bindings on the other hand.

The contents of this chapter are to a large extent derived from the online MATLAB documentation[24].

2.5.1. MEX and Engine APIs

Roughly speaking, there are two main options: One can either interface from MATLAB to C++ or from C++ to MATLAB. On the one hand, users can access MATLAB from C++ code. For this purpose, MATLAB provides so-called Engine APIs. These APIs can start new or connect to existing MATLAB sessions, invoke MATLAB functions and exchange data with the work spaces of the session. On the other hand, users may wish to invoke C++ routines or libraries from MATLAB. Doing this is possible by using MEX APIs. MEX is short for MATLAB executable. The User has to write a C++ source file with a certain prescribed structure, called source MEX file. Afterwards, he invokes the `mex` command in MATLAB, which interfaces to a C++ compiler to create a special executable, the binary MEX file. This file can be run by the MATLAB interpreter and thus be called just like a normal MATLAB function. `mex` is compatible e.g. with the GNU compiler for Linux systems[15][6] and numerous compilers on Windows and Mac systems[23].

[4] For transparent surfaces, another part of the radiation will be refracted or pass through the surface.

[5] MATLAB also offers interfaces to other languages, like Java or Python, and many of the things stated here hold good for these interfaces as well. However, we will only discuss C++ interfaces in our work.

[6] It should be noted that MATLAB R2019a officially only supports the GNU compiler version 6.3.x. For later compilers, a warning is issued when calling the `mex` command. However, the author encountered no problems using versions 8.x and 9.x of the GNU compiler for this work.

The choice between these options mainly depends on the question which of the languages the application is based on - MATLAB based applications should use the MEX APIs while C++ based applications should use the Engine APIs.

TherMoS is strongly based on MATLAB. For this reason, MEX functions were used to interface to the ray tracer. In addition, the purpose of the MATLAB bindings for preCICE we will discuss later is to provide an interface for existing MATLAB-based applications. For these reasons, we will only discuss the MEX APIs in the following.

Recent MATLAB releases support two entirely different and non-compatible APIs for writing C++ MEX files. The original C MEX API has been part of MATLAB for multiple years. It is used together with the C matrix API to handle MATLAB data. In R2018a, the new C++ MEX API was introduced alongside another new data handling API called C Data API. In the following two subsections we will briefly introduce these APIs.

2.5.2. C MEX API and C Matrix API

Source files written in the C MEX API must include a function named `mexFunction` with the following signature:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
↪  *prhs[])
```

This function is called gateway routine and essentially replaces the usual main function. The integers `nlhs` and `nrhs` are the numbers of input resp. output arguments the function is called with from MATLAB, while `plhs` and `prhs` are arrays of pointers to the arguments themselves.

The `mxArray` data type is defined in the related C Matrix API. It serves as a wrapper around any kind of MATLAB data.

A problem when using the C MEX API is the lack of support for embedding C++ objects into MATLAB. While it is possible to use object oriented programming inside the MEX function, there is no support for making objects created in such a function persistent between multiple calls to the function. However, this is exactly what we will need for the bindings in preCICE: The concept of preCICE is to create a `solverInterface` object and repeatedly calling this object's member functions throughout the solver code to steer the communication with the other solver. To port this to MATLAB, we need a MATLAB solver interface object which wraps around an internally managed C++ `solverInterface` object and interfaces to its methods. Doing this, however, requires creating an object that persists between calls to the C++ code.

It is possible to work around this problem in several ways [14]. Arguably, the best solution known to this problem is to create the desired object using dynamically allocated memory, cast a pointer to the object to an integer using `reinterpret_cast` and return this integer to MATLAB. There, the integer can be stored as a private property of a handle object. When calling the same MEX function again, passing the integer allows to access the object back.

This approach works fine and is also safe if the user implements a destructor for the wrapper class that deletes the C++ object and frees the associated memory. If this is done properly, there will be no memory leaks as soon as the wrapper object is destroyed in MATLAB.

Nevertheless, this approach involves intentionally creating memory leaks in the MEX functions and using `reinterpret_cast` to cast a pointer to an integer, which must be managed by the programmer. A cleaner solution which does not require using these concepts is desirable.

Beyond this difficulty, the data handling of the C MEX API can be tricky to users and cause unexpected side effects. We will detail this in Subsection 2.5.4.

2.5.3. C++ MEX API and C DATA API

The second and much younger API is geared towards a better support for object orientation and modern C++ features. The calls, classes and function used in this API are completely disjoint to those used in the C MEX API.

C++ MEX API functions are in fact classes, not functions. They inherit from a superclass `matlab::mex::Function` defined in the `mex.hpp` header. When the user calls the MEX function for the first time, MATLAB internally creates and stores an instance of this class which persists after the function returns. This object remains until `clear mex` is called or MATLAB is closed. Whenever the function is called again, the previously created object is reused.

The execution flow of the MEX function is defined in the `()` operator of the MexFunction object. This operator must be overridden in the definition to take two arguments of type `matlab::mex::ArgumentList`, which will hold pointers to the inputs and outputs of the function call in MATLAB, respectively. The data itself are encapsulated in objects of type `const matlab::data::array`.

Beyond the restrictions on overriding the `operator()`, the user may add further members to the mexFunction class as he pleases. As stated above, these members will persist between calls to the function, unless the user intentionally removes them by issuing the `clear mex` call. This allows to circumvent the problem discussed above: Objects created in the mexFunction can be retained without any difficulty, as long as no `clear` call is issued.

This advantage exhibits the C++ MEX API as a better choice for writing the preCICE bindings. The following two subsections list two further advantages of using this API instead of the C MEX API.

2.5.4. Data handling in MATLAB and in the MEX APIs

This subsection contains a digression to explain a further advantage of the new C++ MEX API. We refer the interested reader to [30].

MATLAB is based on copy-on-write semantics. To explain this, let us have a look at the following example.

```
>> a = [1,2,3];
>> b = a; % No data is copied: b and a reference the same physical
↪  memory
>> b(1) = 0; % Copy-on-write: data is copied, a remains unchanged
```

A newly created variable `a` internally stores a reference to a certain memory location. If we create a second variable `b` and initialize it to be equal to `a`, then `b` will store a reference to the same memory location – the data itself is not copied. Only if either of the variables is changed, a copy of the data will be created to make sure that the other variable remains unchanged.

```
1 function x = foo(x)
2   x(1) = 0; % Copy-on-write triggered
3 end
```

The very same thing happens in functon calls: If we call `foo(a)` with the function defined above, the variable `x` inside the function remains shared with `a`. As soon as the function reaches line 2, the data is copied, such that the input argument `a` remains unchanged. As a consequence, *in-place editing* of variables is not possible in MATLAB functions: Even if the function above is called via `a = foo(a)`, a copy will always be created in line 2 of the function.

In the C MEX API, this is different. By using the `mxGetPr` function, it is possible to obtain a non-`const` pointer to the values of an `mxArray` passed as input argument. It is now possible to write on the address this pointer points to. MATLAB does not notice this, thus, copy-on-write is not triggered. As a consequence, it is possible to modify a variable passed to a C MEX function in such a way that the modification affects the original variable outside the function.

This can, on the one hand, be advantageous: Copying data always implies overhead. So, if users willingly and intentionally performs in-place operations, they can avoid this overhead using MEX functions. On the other hand, this behavior is extremely hazardous as it can easily cause unexpected and confusing results. Consider, for example, the following code snippet:

```
>> a = [1,2,3];
>> b = a; % Recall that b and a reference the same memory!
>> inPlaceEdit(a);
```

In this case, if `inPlaceEdit` is a MEX function modifying the contents of `a`, then *it will modify the contents of* `b` *as well*! So, even though `b` did not even appear in the call, its value was changed.

The C++ MEX API handles this problem in a much cleaner way: If an input to a MEX function is accessed without the `const` keyword, MATLAB will understand this as a write access and hence create a copy of the data. If we wish to use an input argument in a read-only way, we can assign it to a `const matlab::data::array`. In this case, pointers to the data in the array must be `const`, too. Otherwise, `mex` will

throw a compile-time error. MATLAB will not create a copy of data accessed with the the `const` keyword, preventing unnecessary overhead.

Apart from that, the C++ API even supports in-place modifications of arrays to a degree using the `std::move` operator. For details, we refer to the documentation [24, "Avoid Copies of Arrays in MEX Functions"].

2.5.5. Out-of-process execution

In general, a MATLAB instance runs as a single-process application. If a MEX function is called in MATLAB, then it does not create a new process for the MEX function to run. Rather, the MEX function is executed directly within the MATLAB process. As a consequence, subsequent calls to the same MEX function might affect each other even if the call `clear mex` is issued in the mean time.

This fact can have a number of unexpected implications which are counter-intuitive at the first glance. An example for this is the use of `MPI_Finalize()`. The MPI specification demands that once `MPI_Finalize()` was called, no other MPI functions may be called. Now, if a MEX function calls `MPI_Finalize()` either explicitly or implicitly via a library it uses, then MPI functions can't be called again later by the same process. Thus, if the user calls this MEX function twice, the process will abort, causing MATLAB to crash.

One of the features of the C++ MEX API is the fact that it allows *out-of-process execution*. Recent MATLAB version come with the `mexhost` function. This function spawns a separate process accessible from MATLAB via a `matlab.mex.MexHost` object. MEX functions can now be executed on in this process.

This has a number of advantages: Pitfalls like the example mentioned above can be prevented by running the MEX function on a new process, if necessary. Moreover, using `mexhost` processes allows the use of debuggers and libraries which are not compatible with MATLAB. Finally, crashes in the MEX function will not cause MATLAB itself to crash, but merely affect the MEX host process.

On the down side, there is one big disadvantage: Using out-of-process execution causes huge overhead which may degrade the performance severely if multiple function calls on the `mexhost` process are issued.

2.6. Existing implementation of TherMoS

This section describes the original implementation of TherMoS by Philipp Hager [21] and the changes implemented by Mohammad Alhasni [2].

2.6.1. Overview

The purpose of TherMoS is the planning of space missions that involve a moving object on the Moon surface. This object can, for example, be a rover or an astronaut, and is referred to as sample in the nomenclature of TherMoS. The user can define a scene

representing part of the lunar surface and a path for the sample to move along over a prescribed time. TherMoS then computes a transient thermal model for the specified time interval, returning the computed temperatures and fluxes for the time steps requested by the user.

TherMoS consists of a number of different modules, almost all of which are implemented in MATLAB. The important modules for our work are the solver module and the ray tracer module, which are jointly responsible for performing the simulation.

TherMoS is controlled by using a graphical user interface. The user specifies the location of interest on the Moon in terms of coordinates, along with the time span to be simulated. From this information, TherMoS computes the position of the Sun and generates a terrain geometry representing a patch of the lunar surface. Afterwards, the user can create a sample object or load a previously defined one. The user can further prescribe the path the sample will move along during the simulation.

The lunar terrain and the sample are discretized by subdividing their surface into triangles. Each triangle corresponds to a thermal node in the sense of the LPM discussed in Section 2.2. The patch size should usually be in the magnitude of multiple km – the smaller it is, the less accurate the computations for the sample will be, since radiation originating outside the patch is not computed and thus neglected. As a consequence, the size of the sample is usually very small in comparison to the surface patch. Near the sample, it is important to use a high resolution – in other words, small triangles – for the terrain to achieve sufficient accuracy at the sample. To save computation time, we use a lower resolution for the parts of the terrain that are sufficiently far away from the sample. Thus, the terrain mesh consists of a high resolution part in the middle of the patch and a low resolution part on the remaining patch.

Once the preparations have been made, the user calls geometric and thermal preprocessing. Geometric preprocessing writes the vertices and triangles of both terrain and sample geometry to `csv` files, which can be read by the ray tracer later on. Thermal preprocessing computes the initial temperatures and writes them to `csv` files, along with the material properties required for performing the simulation. Once these steps are done, the user may run the simulation.

2.6.2. Solver implementation

Performing the simulation consists in repeatedly calling the ray tracer and the solver in turns. The ray tracer computes the radiative heat fluxes, which is detailed below. The matlab solver, on the other hand, computes conductive heat transfer rates. To model the heat exchange between the surface and the interior of the Moon, TherMoS introduces 9 additional thermal nodes below the thermal node of each surface triangle. Thus, each triangle encompasses a total of 10 thermal nodes which exchange heat by conduction, while only one of these nodes is taken into account for radiative heat transfer.

The radiative heat fluxes computed by the ray tracer are converted into heat transfer rates. Afterwards, the solver finally computes the new temperatures by solving the heat balance equation arising from the lumped parameter method[18].

The original implementation communicated with the ray tracer simply by launching

it in each time step. Results of both components would be written to `csv` files and read in by the respective other component. Alhasni replaced this approach by using a memory mapped file for communication and a mutex for synchronization. Details are given below.

2.6.3. Ray tracer implementation

The ray tracer is built around NVIDIA's OptiX framework [35]. This framework abstracts away most of the Compute Unified Device Architecture (CUDA) implementation on the GPU itself, allowing programming on a higher level.

The computation happens in a class named `optixSimulator`. In the original implementation, this class read the temperatures computed by the solver from `csv` files and wrote the results back to `csv` files. Alhasni implemented two subclasses in his work, `optixSimulatorStandalone` and `optixSimulatorSynchronized`, the former of which contains the original first implementation and the latter of which implements a synchronized execution of the ray tracer alongside the MATLAB solver.

The radiative heat fluxes computed in the ray tracer fall into two categories, namely, solar and infrared fluxes.

Solar radiation represents the heat emitted by the Sun. Due to the enormous distance between the Sun and the Moon, the individual rays arriving on the lunar surface patch modeled in TherMoS are nearly parallel. Therefore, the computation is done by introducing an auxiliary parallelogram, the *Sun parallelogram*, which is located between the actual position of the Sun and the lunar surface and orthogonal to the direction of the solar radiation. Rays are then launched from this parallelogram towards the lunar surface. Figure 2.5 visualizes the setup. A small secondary parallelogram is introduced in the center of the main parallelogram to take into account the higher resolution in the center of the scene.

Infrared radiation represents the heat emitted by the lunar terrain itself. Rays are sampled in random directions from each triangle of the terrain mesh using a cosine-weighted sampling based on Lambert's law, which is also used for diffusive reflection. [2].

Once any ray hits one of the triangles, part of it is absorbed and contributes to the total energy of the triangle. The remaining part of the ray is reflected back into the scene. Due to the rocky surface of the Moon, we only take into account diffusive reflection for the terrain. For the sample, both diffuse and specular reflection are used.

To accelerate the computation, the ray tracer further uses bounding boxes. We will not detail this here. The implementation is described in [2]. For further information about acceleration structures, we refer the reader to [41].

2.6.4. The memory map based interface

The original interface suffered from two major performance drawbacks:

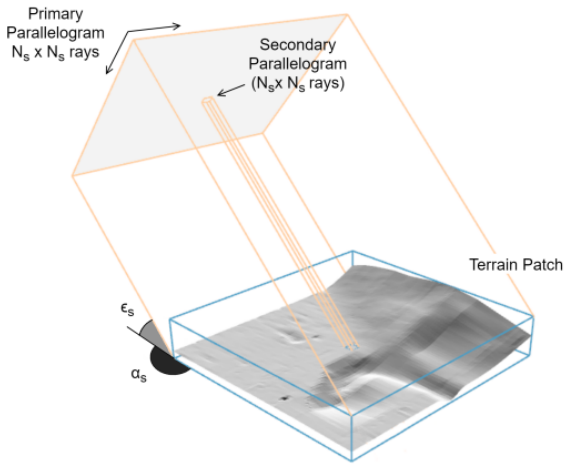


Figure 2.5.: Visualization of the solar parallelogram. Its position is determined based on the solar coordinates, the azimuth α_s and the elevation ϵ_s. Figure taken from [2].

- The exchange of data via `csv` files requires repeated accesses to the hard disk, which comes with high latency.
- The repeated launching of the ray tracer in each time step generates additional overhead.

Alhasni remedied both of these problems in his work by introducing the interface based on a memory mapped file[2]. To implement this, he used the C++ boost library[4].

On the solver side, he introduced the `heat_synchronizer`, a MEX function written in the C MEX API. This function first creates a memory mapped file using the boost library and initializes a mutex. Afterwards, it calls the synchronized version of the ray tracer.

After an initial handshake between the components was successful, the function enters a computational loop. In each step, it waits for the ray tracer to unlock the mutex. Once the mutex is unlocked, it reads in data from the memory mapped file, performs its computation and writes data back to the file.

On the ray tracer side, the `optixSimulatorSynchronized` class does not just compute one time step, but enters a computational loop itself. The geometry of the scene and the sample are still read from csv files, since this step happens only once during setup. The ray tracer keeps running until it recieves a signal to terminate from the solver after the simulation has finished.

2.6.5. Multi-GPU parallelization

Apart from enhancing the interface, Alhasni's main work was to parallelize the ray tracer on multiple GPUs. The parallelization is based on MPI. The meshes of the surface and

the sample are distributed among the process. During the computation, each process only launches and traces rays from its own sumbeshes.

While each processes casts rays only from its own submesh, the rays can still hit triangles belonging to submeshes of different processes. Consequently, each process computes flux contributions to the entire mesh, not only its own submesh. After each time step, the resulting contributions are added using `MPI_Reduce`.

3. Implemenation

This chapter describes the implementation done by the author.

Section 3.1 discusses the MATLAB bindings for preCICE written in the course of this work. The subsequent sections describe the coupling of the TherMoS components via preCICE.

3.1. Matlab bindings

It was mentioned in Section 2.3 that preCICE comes with bindings for various languages. These bindings allow users to couple simulations which are not written in C++. They internally interface to C++ to call the preCICE routines, thus relieving the user of writing C++ code himself and allowing him to use the language of the simulation.

MATLAB is a very popular software with a large community of users in Science, Engineering and Mathematics. Therefore, MATLAB bindings for preCICE had been requested in an issue [13] in early 2018. However, when the author's work on TherMoS started, no progress on the implementation itself had been made. There were only some theoretical considerations.

For this reason, a MATLAB API for preCICE was developed in the course of this work. The API was developed in a fork of the preCICE repository and will be subject to a pull request in late July.

3.1.1. Requirements discussion and API choice

The primary goal of the bindings is to provide users an interface in the native language of their application. The reader is asked to note that this requires a different approach than, for example, the `heat_synchronizer` in TherMoS. The `heat_synchronizer` in TherMoS steers the entire application from C++ code. The MATLAB routines of the solver are invoked using the `mexCallMatlab` function instead of just calling them from MATLAB code. This is perfectly fine, but it does not agree with the philosophy of the bindings. If a similar approach was used there, users of the bindings would have to embed the calls to their solvers into C++ code. In most cases, this would imply a major restructuring of their code instead of simply writing an adapter in the native language of the code.

Hence, we wish to implement a set of routines in MATLAB that allows users to invoke all routines in the C++ API. Due to the design of preCICE, this means that a `SolverInterface` object can be created from MATLAB and persist during subsequent calls to MATLAB functions on the one hand and calls to the preCICE routines on the

other hand. This gives rise to the necessity to make C++ objects persist between calls to the bindings.

In Section 2.5, we discussed the interfaces provided by MATLAB for interfacing to C++ code. As was already mentioned there, the modern the C++ MEX API allows realizing persistent objects in a much cleaner way than the C MEX API. Namely, an objects can be declared as member of the `MexFunction` object. As soon as the compiled function is called for the first time, an instance of the `MexFunction` object is created and stored internally by MATLAB.

So, our approach will be to encapsulate all functionalities of preCICE into a single MEX function which holds the solver interface persisting between the calls.

3.1.2. Description of the implementation

To wrap around the C++ interface, the bindings consist of two MATLAB classes named `SolverInterface` and `Constants`. Both classes are located in the MATLAB style package folder `+precice`. This package folder itself is contained in the path `src/precice/bindings/matlab` on the preCICE repository. As soon as the user adds this folder to the MATLAB path, they are accessible in the name space `precice`. In the following, we confine ourselves to describing the MATLAB `SolverInterface` class. This class has private access to a MEX function named `preciceGateway`.

To avoid confusion in the following description, we introduce the following nomenclature:

- *MATLAB interface* is the MATLAB `SolverInterface` class and its instances
- *`SolverInterface`* is the C++ `SolverInterface` class and its instances
- *gateway object* is the `mexFunction` object representing the `preciceGateway` MEX function created and stored internally by MATLAB
- *gateway routine* is the operator `()` of the gateway object, which is invoked whenever the MEX function is called from MATLAB
- *interface* is the `SolverInterface` instance stored inside the gateway object.

The reader is asked to become aware of the differences between these concepts before proceeding.

Listing 3 shows the structure of the gateway object. It holds a pointer to the C++ interface. Note that the constructor of the gateway object does not initialize the interface yet. This is not possible, since input arguments are needed for the construction, which are not available to the constructor – they can only be passed to the gateway routine. For this reason, the gateway object also holds a bool, which is set to `true` as soon as the interface constructor was called, and to `false` after it was destroyed.

For the the MATLAB bindings, every member function of the `SolverInterface` class is associated with a unique unsigned 8-bit integer ID. The gateway routine always expects the function ID as first argument. Depending on the ID, the gateway routine calls the

```
using namespace precice;
using namespace matlab::data;
using matlab::mex::ArgumentList;
using matlab::engine::MATLABEngine;

class MexFunction: public matlab::mex::Function {
private:
    SolverInterface* interface; // C++ solver interface
    ArrayFactory factory; // responsible for creating output arrays
    bool constructed; // true after interface was constructed
    // gives access to the MATLAB instance, required by myMexPrint
    std::shared_ptr<MATLABEngine> matlabPtr = getEngine();

    // routine for printing to the MATLAB command window
    void myMexPrint(const std::string text);

public:
    // constructor
    MexFunction(): constructed{false}, factory{}, interface{NULL} {}

    // gateway routine
    void operator()(ArgumentList outputs, ArgumentList inputs);
```

Listing 3: Basic structure of the `MexFunction` privately available to the MATLAB `SolverInterface` class

corresponding member function of the interface. The remaining arguments are passed to this function. The output argument of the function, if existent, is then returned by the gateway object. Listing 4 shows an example.

The MATLAB interface class itself has exactly the same member functions as the original class `SolverInterface`. Each function in the MATLAB interface simply calls the gateway routine with the respective ID and the input arguments and returns the output arguments to the caller. As an example, the MATLAB interface version of the `advance` function is shown in Listing 5. The MATLAB interface is a handle class, i.e., it inherits from the MATLAB `handle` superclass. This means that a MATLAB variable holding an instance of this class always only holds a reference to the actual object. A new instance of the class can only be created by explicitly calling the constructor. MATLAB manages a garbage collector, which deletes the objects as soon as no more references point to it. The constructor and destructor of the MATLAB interface call the constructor resp. denstructor of the `SolverInterface` class via the gateway routine. This way, we ensure that the interface encapsulated in the gateway object lives exactly as long as the corresponding MATLAB interface.

The syntax of the MATLAB interface class and the `SolverInterface` class are largely

```
void operator()(ArgumentList outputs, ArgumentList inputs) {
    // determine the functionID
    const TypedArray<uint8_t> functionIDArray = inputs[0];
    int functionID = functionIDArray[0];

    /* Abort if constructor was not called yet, or if constructor is
    ↪  called a second time */
                /* ... */

    switch (functionID) {
        /* ... */
        case 12: //advance
        {
            const TypedArray<double> dt_old = inputs[1];
            double dt = interface->advance(dt_old[0]);
            outputs[0] = factory.createArray<double>({1,1}, {dt});
            break;
        }
        /* ... */
    }
}
```

Listing 4: Exemplary structure of the gateway routine

```
function dt = advance(obj,dt)
    dt = preciceGateway(uint8(12),dt);
end
```

Listing 5: Functions in the MATLAB interface

identical with a few exceptions:

- Pointers are replaced by MATLAB vectors and matrices, since MATLAB-style vectors are the usual way to handle float data in MATLAB.
- Some `SolverInterface` member functions take pointers to output buffers as input arguments. These are replaced by output arguments in MATLAB.

For example, the signature of the function `setMeshVertices` already seen in Section 2.3 reads

```
void setMeshVertices (int meshID, int size, const double *positions, int
↪  *ids)
```

in the `SolverInterface` class and becomes

```
function ids = setMeshVertices(meshID, size, positions)
```

in the MATLAB interface.

The reader may have noticed a drawback of our implementation: In the way described above, only one MATLAB interface can exist at a time. The reason for this is that the gateway object is created only once – all instances of the MATLAB interface will access the same gateway object when invoking the gateway routine, thus all of them would also use the same interface.

To prevent problems like this, the gateway object holds the `constructed` variable. If the user tried to create a second MATLAB interface object before destroying the first one, gateway routine would notice that the constructor is called again, but the `constructed` variable holds `true`. In this case, it will abort on an error. Thus, the second MATLAB interface object will run into an error in the constructor, causing it to be destroyed again.

Technically speaking, this is a very ill behavior. In practice however, users will hardly run in a situation where they do need two MATLAB interface objects within the same MATLAB instance. For this reason, the problem described here was not addressed up to now. A simple solution of this difficulty would be to replace the `SolverInterface* interface` variable in the gateway object by a list of pointers to `SolverInterface` objects and adding another input argument to the gateway routine specifying the index of the `SolverInterface` object to be used. This may be done in the future.

3.1.3. Out-of-process variant

As discussed in Subsection 2.5.5, the C++ MEX API further gives access to out-of-process execution. For this reason, a subclass of the `SolverInterface` was created, named `SolverInterfaceOOP`. Objects of this class hold a `mexhost` object created upon construction and run all calls to the gateway routine on this object.

Apart from the advantages already stated in Subsection 2.5.5, this remedies the problem mentioned above: Multiple instances of `SolverInterfaceOOP` can exist at the same time without interfering. It is even possible to run two MATLAB solvers from the same MATLAB instance using `SolverInterfaceOOP`.

However, the overhead of out-of-process execution poses a massive drawback. Since the preCICE bindings will call the gateway routinefrequently, they are strongly affected by the overhead. The author noted that executing the MATLAB solverdummy out of process can take more than 10 times as long as executing it in the main process.

Therefore, using the `SolverInterfaceOOP` is only advisable if the user encounters major problems when running them in the main MATLAB process. Apart from that, developers may find `SolverInterfaceOOP` useful for debugging: If crashes occur in bindings when running out of process, they will only cause the `mexhost` object to crash – the MATLAB instance will not be affected.

3.2. Coupling Configuration

To couple TherMoS, three steps are necessary: We have to write a configuration file for preCICE and we have to write adapters for both components. In this section, we will start with the configuration. Appendix A.1 contains the full configuration file for the TherMoS coupling. In this section, we will briefly discuss and comment it.

We first define the data to be exchanged. The actual coupling data consists of temperatures computed by the solver and the fluxes computed by the ray tracer. Both of these are computed both on the terrain and on the sample, so we define separate data sets for terrain and sample.

In addition, however, we need to exchange global data which is related to neither of the geometric meshes:

- The solar angles, azimuth and elevation, are determined in the solver but required by the ray tracer to compute the solar parallelogram
- A 4×4 matrix, the transformation matrix, defining a transformation of the sample, if necessary

preCICE currently does not support exchanging global data, though. There is a GitHub issue about this, which is still open at the time this thesis is written[12]. As a workaround, we define an extra data set called auxiliary data which encompasses the global data.

Second, we declare the meshes. Geometrically, we have two meshes: The terrain mesh and the sample mesh. This sheds light on one of the peculiarities of our setup: *Both ray tracer and solver use the same meshes.*

In Subsection 2.3.4, we saw the challenges posed by interface coupling with non-matching meshes. We also discussed the mapping capabilities of preCICE to deal with these problems. For TherMoS, however, these capabilities simply have no use: There is no need for nor advantage in defining separate meshes for both components and computing a mapping between them.

Unfortunately, it is not possible for two preCICE solvers to share the same mesh. Applications with two solvers operating on the same mesh are not intended preCICE. Therefore, preCICE does not support this. For two serial solvers, the author verified that using one mesh does in fact work. However, if either of the solvers is parallel, preCICE will throw errors when trying to use a single mesh. The reason is that in this case, preCICE tries to compute a process mapping as described in Subsection 2.3.5, which does not work in the internal implementation if only one mesh is used.

As a consequence, it is not possible in our case to define only one terrain and sample mesh for both solver and ray tracer. Instead, both need to define their own instance of the mesh. This comes with an inconvenient disadvantage: preCICE will compute a mapping between the two meshes even though this is not necessary. In theory, it is sufficient to simply map data at each vertex to the corresponding vertex with the same index. However, in the current implementation, there is no way to simply tell this to preCICE.

Computing the mapping causes unnecessary overhead. This problem is not critical for the overall performance, as the mapping computation happens only once during the initialization, but it is still a drawback.

Beyond the geometric meshes, we need a further mesh to communicate the auxiliary data. As a result, we end up with a total of six meshes: Terrain mesh, solver mesh and auxiliary mesh; each of them for both participants.

After defining data and meshes, we can define the participants. Since we can not avoid the mapping, we have to define mappings in one of the participants. Here, we must make sure that every vertex on either side is mapped exactly to the corresponding neighbor mapping. This can be achieved by choosing nearest neighbor mapping: Since the vertices in the two meshes are located at the exact same positions, each vertex will be mapped to the corresponding vertex on the other mesh.

Since the mapping will identify the vertices in a bijective way, it does not matter whether we choose conservative or consistent mapping. The two strategies differ only if multiple vertices on either side would be mapped to the same vertex on the other side. For reasons for that will become clear later, we declare the solver→ray-tracer-mapping as consistent and the ray-tracer→solver-mapping as conservative. Due to the restrictions on parallel participants, we therefore do the mappings on the ray tracer side.

The communication can be done either with sockets or ports. Both have been tested successfully.

It remains to choose the coupling scheme. Recall that the original implementation calls the two components in turns and blocks while the other component is running. Each component is called once per time step. Recalling Section 2.1, we notice that this corresponds exactly to serial-explicit coupling. Thus, we use this coupling.

3.3. MATLAB solver adapter

With the MATLAB bindings at hand, writing the adapter for the MATLAB solver is mostly straightforward. The reulting adapter is found in the appendix A.2. The coupling is done in a MALTAB script named `precice_synchronizer.m` which replaces the `heat_synchronizer.cpp`.

In many scientific codes, discrete data is located on the vertices of a mesh. However, recall that in our LPM, the data is in fact located on nodes that correspond to the *faces* of our mesh. Hence, we need to declare the faces resp. nodes as vertices to preCICE.

It is necessary to provide vertex coordinates for defining the geometry to preCICE. Geometrically, the correct choice for vertex coordinates would be the centers of the faces, where the LPM nodes are located. Yet, these vertices are required by preCICE solely for the purpose of mapping. Since we are only interested in identifying corresponding vertices with each other for TherMoS, the actual geometric location of the vertices declared to preCICE does in fact not matter – We only have to make sure that corresponding vertices are located at the same position.

For this reason, we perform a simplification: Instead of using meaningful coordinates, we simply declare the mesh vertices to be located at $(0,0,0), (1,0,0), (2,0,0), \ldots$ and so forth. By doing this in both adapters and using the nearest neighbor mapping, we make sure that the data is mapped correctly.

After calling the `initialize` function, the solver must write the intial temperatures. These are passed to the adapter as function arguments.

A further thing to note is that the total duration and the time step size for the simulation in TherMoS is declared in the GUI, while preCICE reads it from the simulation file. Consequently, dummy values were used in the configuration file. Upon starting the coupling, the TherMoS adapter reads the values from the GUI and creates a temporary copy of the configuration file which is used afterwards.

Apart from these things, the adapter is a standard preCICE adapter in all respects and closely follows the adapter example that was already seen in Figure 2.

To make the adapter accessible from TherMoS, the function responsible for running the simulation was modified in such a way that it opens a dialog window asking the user to choose between preCICE coupling and using the `heat_synchronizer.cpp`.

3.4. Ray tracer adapter

Since the ray tracer is entirely written in C++, implementing the ray tracer adapter was a straightforward matter. The resulting adapter is also found in the appndix A.3.

Our implementation is encapsulated into the class `optixSimulatorPrecice`, which is a new, third subclass to the `optixSimulator` class. The `SolverInterface` object itself and the IDs of meshes, data and vertices are members of this class.

Apart from hat, the structure of the new class is mostly identical to that of the original `optixSimulatorSynchronized` class. The accesses of the memory mapped file are replaced by calls to the preCICE API to communicate the data. The simulation is carried out by the run function of the new class.

3.4.1. Floats in the ray tracer

When communicating the data from and to the ray tracer via preCICE, we encounter a simple yet severe problem: The OptiX library used by the ray tracer is written to work exclusively with single precision floats, i.e. the C++ `float` data type. Yet, the preCICE API can only take pointers double precision floats, i.e. the C++ `double` data type, in all its calls for communicating data.

The restriction to doubles is a drawback of preCICE. Even though the majority of scientific codes today uses double precision, our example shows that there are still cases in which support for single precision would be preferable. Beyond this, it may be desirable in some cases to communicate completely different data types via preCICE, e.g. integers for integer quantities. Consequently, it would be a valuable feature to add functions for supporting different data types to the preCICE API. A completely

clean solution could be achieved by using templates, allowing the user to communicate arbitrary data types, even custom ones.

This change is far from trivial, though, and would have a huge impact on the implementation of preCICE. Therefore, it is currently out of scope.

For TherMoS, this means that we have to confine ourselves to communicating doubles for time being. If we are determined to use single precision, an option would be to use `reinterpret_cast` and simply communicate the buffers as if they consisted of $\frac{n}{2}$ doubles instead of n floats. This, however, would be a very dirty workaround resulting in bad software design. Also, it would overthrow the concept of meshes in preCICE, since the user would have to define a new artificial mesh with no geometric meaning. It would further prohibit any kind of mapping, since using the `reinterpret_cast` renders any arithmetic operations on the data meaningless. In fact, the mapping features of preCICE are not necessary for TherMoS as discussed above, but if it is used in a future change, the code would become invalid.

Thus, we instead bear to convert all buffers used int the ray tracers to doubles before writing them to the solver and back to floats after reading. For this purpose, we use three `std::vector<double>`s – one for each of the meshes – to store data temporarily as doubles. To save memory, we allocate the buffers only once when constructing the simulator, and reuse them whenever needed. The data inside these buffers is cast to floats using `static_cast<float>` before using them in the ray tracer, and back to doubles afterwards.

It should be remarked that conversion from float to double data also happened in the existing versions of the ray tracer. While MATLAB can do most of its operations with floats, it uses doubles by default in many built-in functions. In contrast to the new implementation, however, floats were communicated between the components in Alhasni's implementation. Therefore, a smaller amount of data was communicated in total.

3.4.2. Broadcast-reduction approach in the ray tracer

A second peculiarity of coupling TherMoS with preCICE is the parallelization approach of the ray tracer.

Many scientific codes use what can be called a *gather-scatter* approach. The computation domain is distributed spatially among the processes involved. Each process computes partial results for its own subdomain and communicates with processes responsible for adjacent subdomains, if necessary. After finishing the computation, the results are usually gathered for visualization or post processing.

As was indicated before, preCICE is designed for this kind of work sharing. If preCICE is initialized in an existing MPI environment, the individual processes declare only their own part of the mesh. preCICE then determines all pairs of processes that need to communicate with each other.

What about TherMoS? Recall the description in Subsection 2.6.5. The TherMoS ray tracer also distributes the mesh among the different GPUs. However, due to the

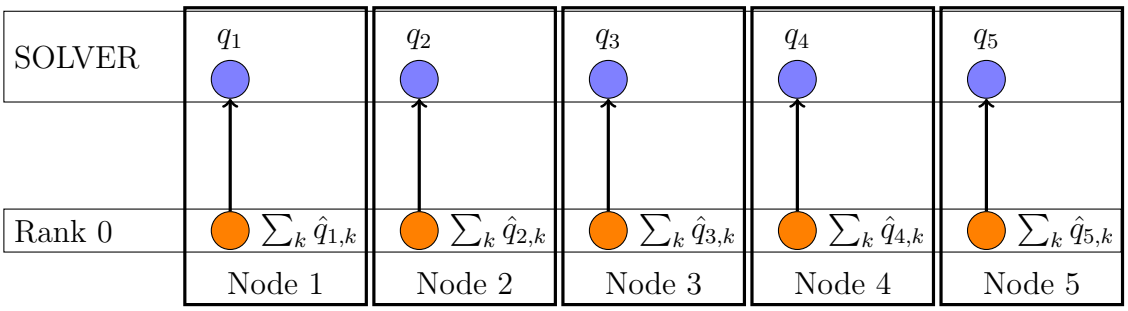


Figure 3.1.: Straightforward implemntation of broadcast-reduction approach. Rank 0 holds the mesh and communicates the sum of all flux contributions

nature of ray tracing, each process still computes *contributions to the entire mesh*. These contribution can not simply be gathered, instead they must be added up – i.e., reduced via `MPI_Reduce()` – at the master process to obtain the total flux computed in one time step. Before the computational step, it is necessary to broadcast the temperatures on the entire mesh to all processes.

In summary, the ray tracer follows a *broadcast-reduction* approach rather than a gather-scatter approach. This kind of work sharing is not supported by preCICE. How can we remedy this?

The simple and evident solution is to let only the master process communicate. I.e., we adopt the MPI communication performed in Alhasni's version. The temperatures are broadcast to the individual ray tracer processes before the computation using `MPI_Broadcast()` and reduced at the process with rank 0 afterwards using `MPI_Reduce()`. In this case, only rank 0 communicates with the solver. This approach is implemented in the code seen in Appendix A.3.

Benjamin Uekermann suggested a very elegant alternative we will discuss in the following subsection: The reduction can also be performed by abusing the mapping techniques implemented in preCICE.

Reducing via conservative mapping

The idea is as follows. Let n be the number of mesh vertices and p be the number of processes. Instead of letting all processes declare only disjoint parts of the mesh to preCICE, we let each process declare each mesh point. This means that, from the perspective of preCICE, the ray tracer mesh does not consist of n vertices, but in fact of $n\dot{p}$ vertices. For each of the n geometrical mesh vertices, there are p copies sharing the same coordinates. The serial MATLAB solver, on the other hand, defines each vertex only once in its mesh.

Now, if we define a nearest neighbor mapping from the ray tracer mesh to the solver mesh, then for each vertex v in the solver mesh, all p corresponding vertices in the ray tracer mesh $v_1, \ldots, v_n$ will be mapped to v. If we choose a conservative mapping, then the flux values computed at $v_1, \ldots, v_n$ will be added up at v. But this is exactly what we want! By performing the conservative nearest neighbor mapping, preCICE adds up

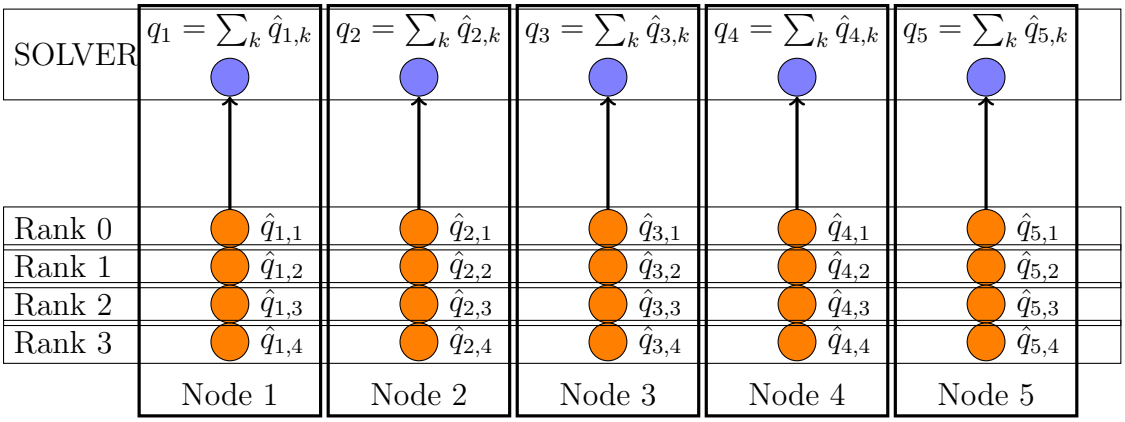


Figure 3.2.: Alternative implementation. Each rank holds the mesh, each rank sends its own flux contributions to the solver, where they are summed up via conservative mapping

the contributions computed by the individual processes to obtain the correct result at the solver. Figure 3.2 illustrates the approach.

Recall that for mapping temperatures from the ray tracer to the solver, we defined a consistent read mapping on the ray tracer side. preCICE determines this mapping by transposing the operator for the conservative mapping[42]. In our case, this means that the values at each node on the solver will be mapped to all the nodes on the ray tracer side. Thus, the consistent mapping automatically broadcasts the values to the ranks in the ray tracer.

In summary, the approach stated here delegates the entire MPI communication in the ray tracer to preCICE, which is very convenient from a design perspective. No more explicit broadcasting or reduction are required, all we have to do in the ray tracer is calling the preCICE API functions and performing the computation themselves. The resulting code for this variant is shown in Appendix A.3.3. It is, clearly, much shorter than the code in Appendix A.3.2. Nevertheless, it comes with an important disadvantage: At least in the current implementation of preCICE, all data will be sent separately to the solver and the individual contributions will be summed up on the solver side, not the ray tracer side. This increases the amount of data to be transferred and may affect the performance.

4. Results and conclusions

4.1. Results

In this section, we report on results with the new implementation. We show some visualizations of the computations done with the new implementation and compare the run time with the original implementation.

4.1.1. Test system

The tests were run on a simulation server provided by the Chair of Astronautics at the TUM. The server is equipped with the following hardware:

- Intel(R) Xeon(R) Silver 4110 CPU
- An NVIDIA GeForce GTX 1080 GPU
- An NVIDIA Quadro P6000 GPU
- 64 GB RAM

4.1.2. Validation of Results

Naturally, the first thing to check is whether the new implementation yields correct results. As our work consisted in replacing the existing interface by a new one, we confine ourselves to comparing the results to those computed with the original implementation. The results computed by the original implementation were already tested against validation data in the works of Hager and Alhasni[2][21].

The terrain used for validation consisted of a quadratic patch of lunar surface with a side length of 8 km. An astronaut space suit was used as a sample. The suit was positioned in the center of the patch and would not move during the simulation. The simulation was run over a time window of 10 minutes. A single time step spanned 20 seconds, resulting in 30 steps.

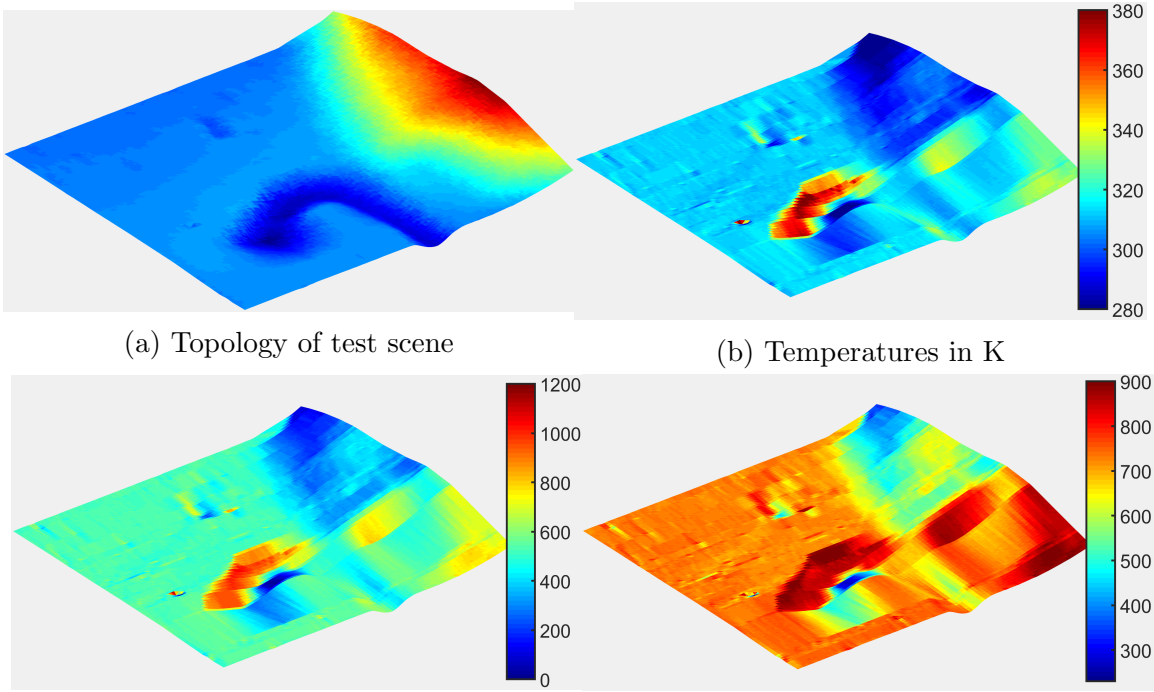
Figure 4.1 shows multiple different views of the test scene. The first image is colored according to height to demonstrate the topographic structure of the terrain. The second image shows the temperatures after 60 seconds. The third and fourth images show the computed solar and infrared fluxes at the same time.

The results were computed in six different ways. On the one hand, three different implementations were used:

- The original implementation based on memory mapped files
- The "simple" preCICE implementation, in which broadcasting and reduction is done explicitly
- The alternative preCICE implementation, in which these steps are done via conservative mapping, called "preCICE b" in Table 4.1.

Each of the implementations was tested both on a single GPU and on both GPUs.

Averaged across the nodes, the results deviated from those results computed with the memory mapped file by a magnitude of 10^{-5}. This is acceptable due to the stochastic nature of ray tracing and confirms the correctness of the new implementation.



(a) Topology of test scene

(b) Temperatures in K

(c) Solar fluxes in $\mathrm{W} \cdot \mathrm{m}^{-2}$

(d) Infrared fluxes in $\mathrm{W} \cdot \mathrm{m}^{-2}$

Figure 4.1.: Visualization of results for a test case.

4.1.3. Performance Tests

After checking the validity of the computations, we are mainly interested in the question how the performance of TherMoS with preCICE compares to the implementation without preCICE. Again, we compare all three implementations with both one and two GPUs. Our test scene consisted of a total of roughly 100000 faces. The implementation was run for a total of 30 time steps. Table 4.1 shows a variety of run times measured for the different implementations.

Type	GPUs	Total compute	Reading Temp.s	Writing Fluxes	Casting	Mapping	Filtering
Original	1	138.02353	0.00054	0.00010			
Original	2	101.22182	0.00065	0.00011			
preCICE	1	147.00487	0.00964	0.02657	0.00012	84.75008	
preCICE	2	111.42825	0.00973	0.02819	0.00013	89.32058	212.08111
preCICE B	1	146.93492	0.00934	0.02637	0.00012	84.57572	
preCICE B	2	113.97372	0.00977	0.02782	0.00013	89.26862	213.19065

Table 4.1.: Comparison of run times for test scene

First of all, to obtain an overview over the differences in time spent by the implementations, we compare the overall run time of the computational loop. The time does *not* include the time spent in setup steps, it only accounts for the computational loop itself.

However, our implementation made no changes to the ray tracer or the solver itself. We only changed the interface. Thus, the most important thing to do is comparing the run times of the interface. In preCICE, reading and writing data between the solvers happens via the corresponding function. In the previous implementation, this corresponds to reading and writing data to the memory mapped file. Therefore, we also measure the times taken by these steps. Table 4.1 lists the average time taken in one time step for reading temperatures and writing fluxes.

Furthermore, recall that for the preCICE interface, it is necessary to cast the data to floats and back. We therefore also measure the wall clock time taken by the casting to find out how much this drawback affects our performance. Like the reading and writing times, this time was averaged over all time steps.

Finally, recall for using preCICE, it is necessary to compute a mapping between the meshes involved. Even though this step occurs only once during initialization, it is interesting to check how much time is consumed by this step. We therefore also measure the time taken for the mapping and filtering by preCICE.

There is a number of interesting observations. First of all, we see that the interface using preCICE is in general slower than the one based on the memory mapped file. The overall simulation time is slightly longer for the preCICE example, and we see that the communication times are higher than the times for accessing the memory mapped file.

Looking at the times consumed for casting the values to from floats to doubles in the ray tracer, it is also evident that casting is not the problem. In fact, this takes very little time. The additional time required in the preCICE version originates from the preCICE API calls themselves. The communication performs overall slower.

On the other hand, the time required for communication is negligible in contrast to the time spent with solving and ray tracing in either case. Even though the preCICE version is no match for the memory map version, both versions are perfectly usable in terms of the time spent in the computational loop.

Furthermore, it is noteworthy that the mapping takes quite long. The time spent in mapping, especially the time required for filtering the meshes, is huge compared to the time spent in a single iteration. However, we should not forget that mapping occurs only once per simulation. For a simulation over a much higher number of time steps, the mapping overhead will be shadowed by the total computation time.

Finally, note that using the conservative mapping for broadcasting and reduction does not increase the computation time. In fact, the performance in both variants of the preCICE based implementation is roughly identical.

4.2. Conclusion

In the past, the preCICE library was almost exclusively used for FSI and CHT. Albeit its intention has always been to be used in general multiphysics applications, the library has rarely been applied in other fields to the best of our knowledge. In this thesis, we have seen preCICE make its way into a new physical domain, thus proving itself capable of a broader range of applications.

It is evident from our results that using preCICE in TherMoS is not a perfect fit. We had already encountered a number of difficulties in the implementation: The special work sharing approach in the TherMoS ray tracer, the necessity to perform explicit type casts and the computation of a mapping which is pointless in our application. Nevertheless, preCICE still performs considerably well in our application.

Beyond that, just like people can learn from their mistakes, preCICE can also draw inspiration from the peculiarities that were faced when interfacing TherMoS. We have realized that there are applications in which communicating double precision floats is not sufficient, and we have encountered a situation in which the extensive mapping capabilities of preCICE pose an obstruction rather than an advantage. Regardless of the question whether implementing corresponding modifications is in the scope of preCICE or not, these observations are valuable for the further development of preCICE.

Furthermore, preCICE has been successfully used on hybrid architecture for TherMoS. While our work did not include explicit CUDA programming due to using the OptiX framework, it is still noteworthy that the coupling between CPU and GPUs succeeded without any problem.

What about TherMoS? While we could not improve TherMoS with respect to performance, we can say that the use of preCICE poses a substantial improvement for the software design of TherMoS. On the solver side, the need to steer the communication in a monolithic way from a MEX file – in other words, a function written in an external language – has been eliminated. The solver is now fully written in MATLAB, making it easier to maintain even for developers who have little or no experience with MEX functions or C++ in general. Furthermore, the partitioned approach has simplified updating or even replacing either or the components.

Finally, preCICE has opened the ability to use different coupling schemes. In the original implementation, performing implicit coupling would have meant a complete restructuring of the heat synchronizer. Thanks to preCICE, this now comes down to

changing some tags in the configuration. While explicit coupling has proven to work fine for the purposes of thermal modeling, it will be interesting to see how the stability and quality of the results could be improved by using the implicit coupling strategies of preCICE. This question must be left open here and remains an exciting challenge for the future.

4.3. Future work

Besides experimenting with other coupling schemes, there is a number of other interesting questions that could be addressed in the future. This final section is devoted to shed light on some of the possibilities for future work and research.

4.3.1. On the MATLAB bindings

Parallel programming in MATLAB One of the important selling points of preCICE is its very easy integration into existing MPI implementation. The m2n communication allows to fully exploit the advantages of existing parallel implementations and minimizes communication overhead.

While MATLAB does not directly support MPI, there exists a number of parallel libraries for MATLAB based on message passing. Besides several third-party implementations, like pMatlab or MatlabMPI, MathWorks offers the official Parallel Computing Toolbox (PCT). Using this toolbox, the user can create parallel pools on a local machine or in a cluster, which internally start additional MATLAB sessions on different processes, called MATLAB workers. To distribute workload among the workers, the user can access a range of methods, from high level parallel programming constructs, like parallel for loops, to explicit message passing via abstractions of the functions of the underlying MPI implementation.

The preCICE bindings written for this work only support serial MATLAB solvers directly, since this was sufficient for TherMoS. However, it would certainly be desirable to extend them in such a way that they support the PCT. This way, the full potential of parallel computing could also be used by MATLAB applications coupled with preCICE.

The `clibgen` package In the very newest release, R2019a, MATLAB added the new `clibgen` package which allows to import C++ libraries directly into MATLAB. Since preCICE is in fact a C++ library, this might be an extremely comfortable way to dispose the MATLAB bindings entirely.

The usage of `clibgen` can be described roughly as follows. First, the user must identify the `.so` library file and the corresponding header files. Using these, he can user the function `clibgen.generateLibraryDefinition`, which automatically determines the functions and classes defined in the library and defines corresponding wrappers in MATLAB. These definitions are stored in a human readable `.mlx` file called the library definition file. Now, the user must manually review this file to complement information that could not be determined automatically by `clibgen`. This mostly concerns pointers:

Wherever a C++ function expects a pointer, MATLAB can not automatically determine whether the function will only read from the memory, only write to it or do both. It also does not know the size of the associated buffer out of the box. The user must therefore specify the pointer as input or output (or both) and the buffer size.

After completing the definition file, the user may call `clibgen.build` to import the file to MATLAB. This will create a MATLAB package in the `clibgen` name space from which all the library functionality can be accessed.

Since the package was not yet released when the MATLAB bindings for this work were implemented, it was not considered for use. However, it may be the best option to interface preCICE to MATLAB in the long run. Importing preCICE using `clibgen` is not completely trivial, there are a few things that have to be dealt with:

- Several C++ features are not supported by `clibgen`. The developers must carefully review whether this will affect preCICE.
- For several functions in the preCICE API, the size of the input buffer depends on the problem dimension, which is accessible via a member function of the solver interface object. However, in the definition file, MATLAB only allows hard-coded constants or other function input arguments to specify the size of the buffer accessed by an input pointer. This does not suffice for preCICE and must be worked around in some way.
- In the long run, the bindings should recieve a proper building process. Reviewing the definition should be done automatically or not be necessary at all.

However, these are minor problems. Once they are solved, the `clibgen` import can likely replace the existing MATLAB bindings.

4.3.2. On TherMoS

Radiosity method for fluxes Martin Schreiber from the Chair of Computer Architecture and Parallel Systems at TUM has suggested to use a radiosity method instead of a ray tracing method for computing fluxes. The information given on this method here is derived from [11, Chapter 6].

It was stated in Section 2.4 that ray tracing methods are mostly used to render images. For this application, it suffices to use backward ray tracing methods, which essentially ignore parts of the scene that are not visible from the image plane. In contrast to this, radiosity methods use different approaches to compute a complete representation of the diffuse illumination of the scene.

The classic radiosity method is a FEM based on the radiosity integral equation. It has, in fact, been originally used in heat transfer, and found its way into image rendering in the 1980s. It is likely to outperform the currently used ray tracing method with respect to performance.

The partitioned implementation with preCICE allows to replace either of the components of TherMoS without much difficulty. Therefore, a possible use for the new interface

could be to replace the ray tracer by a radiosity method. It would be interesting to see how this would affect the performance of TherMoS.

A. Full code for TherMoS coupling

A.1. Configuration file for TherMoS

```
<precice-configuration>

  <solver-interface dimensions="3" >

    <data:scalar name="TerrainTemperatures"  />
    <data:scalar name="TerrainFluxesSolar"  />
    <data:scalar name="TerrainFluxesIR"  />
    <data:scalar name="SampleTemperatures"  />
    <data:scalar name="SampleFluxesSolar"  />
    <data:scalar name="SampleFluxesIR"  />
    <data:scalar name="AuxiliaryData"  />

    <mesh name="TerrainMeshSolver">
      <use-data name="TerrainTemperatures" />
      <use-data name="TerrainFluxesSolar" />
      <use-data name="TerrainFluxesIR" />
    </mesh>

    <mesh name="TerrainMeshRaytracer">
      <use-data name="TerrainTemperatures" />
      <use-data name="TerrainFluxesSolar" />
      <use-data name="TerrainFluxesIR" />
    </mesh>

    <mesh name="SampleMeshSolver">
      <use-data name="SampleTemperatures" />
      <use-data name="SampleFluxesSolar" />
      <use-data name="SampleFluxesIR" />
    </mesh>

    <mesh name="SampleMeshRaytracer">
      <use-data name="SampleTemperatures" />
      <use-data name="SampleFluxesSolar" />
      <use-data name="SampleFluxesIR" />
```

```
35      </mesh>
36
37      <mesh name="AuxiliaryMeshSolver">
38        <use-data name="AuxiliaryData" />
39      </mesh>
40
41      <mesh name="AuxiliaryMeshRaytracer">
42        <use-data name="AuxiliaryData" />
43      </mesh>
44
45      <participant name="Raytracer">
46        <use-mesh name="TerrainMeshRaytracer" provide="yes"/>
47        <use-mesh name="SampleMeshRaytracer" provide="yes"/>
48        <use-mesh name="AuxiliaryMeshRaytracer" provide="yes"/>
49        <use-mesh name="TerrainMeshSolver" from="Solver"/>
50        <use-mesh name="SampleMeshSolver" from="Solver"/>
51        <use-mesh name="AuxiliaryMeshSolver" from="Solver"/>
52        <mapping:nearest-neighbor direction="write"
          ↪  from="TerrainMeshRaytracer" to="TerrainMeshSolver"
          ↪  constraint="conservative" />
53        <mapping:nearest-neighbor direction="read"
          ↪  from="TerrainMeshSolver" to="TerrainMeshRaytracer"
          ↪  constraint="consistent" />
54        <mapping:nearest-neighbor direction="write"
          ↪  from="SampleMeshRaytracer" to="SampleMeshSolver"
          ↪  constraint="conservative" />
55        <mapping:nearest-neighbor direction="read" from="SampleMeshSolver"
          ↪  to="SampleMeshRaytracer" constraint="consistent" />
56        <mapping:nearest-neighbor direction="write"
          ↪  from="AuxiliaryMeshRaytracer" to="AuxiliaryMeshSolver"
          ↪  constraint="conservative" />
57        <mapping:nearest-neighbor direction="read"
          ↪  from="AuxiliaryMeshSolver" to="AuxiliaryMeshRaytracer"
          ↪  constraint="consistent" />
58        <write-data  name="TerrainFluxesSolar" mesh="TerrainMeshRaytracer"
          ↪  />
59        <write-data  name="TerrainFluxesIR" mesh="TerrainMeshRaytracer" />
60        <write-data  name="SampleFluxesSolar" mesh="SampleMeshRaytracer"
          ↪  />
61        <write-data  name="SampleFluxesIR" mesh="SampleMeshRaytracer" />
62        <read-data name="TerrainTemperatures" mesh="TerrainMeshRaytracer"
          ↪  />
63        <read-data name="SampleTemperatures" mesh="SampleMeshRaytracer" />
64        <read-data name="AuxiliaryData" mesh="AuxiliaryMeshRaytracer" />
```

```
65      </participant>
66
67      <participant name="Solver">
68        <use-mesh name="TerrainMeshSolver" provide="yes"/>
69        <use-mesh name="SampleMeshSolver" provide="yes"/>
70        <use-mesh name="AuxiliaryMeshSolver" provide="yes"/>
71        <write-data name="TerrainTemperatures" mesh="TerrainMeshSolver" />
72        <write-data name="SampleTemperatures" mesh="SampleMeshSolver" />
73        <write-data name="AuxiliaryData" mesh="AuxiliaryMeshSolver" />
74        <read-data  name="TerrainFluxesSolar" mesh="TerrainMeshSolver" />
75        <read-data  name="TerrainFluxesIR" mesh="TerrainMeshSolver" />
76        <read-data  name="SampleFluxesSolar" mesh="SampleMeshSolver" />
77        <read-data  name="SampleFluxesIR" mesh="SampleMeshSolver" />
78      </participant>
79
80      <m2n:sockets from="Raytracer" to="Solver"
        ↪  distribution-type="gather-scatter"
        ↪  exchange-directory="./25_Precice_Module/"/>
81
82      <coupling-scheme:serial-explicit>
83        <participants first="Raytracer" second="Solver" />
84        <max-time value="DUMMY-DURATION" />
85        <timestep-length value="DUMMY-TIMESTEP" />
86        <exchange data="TerrainTemperatures" mesh="TerrainMeshSolver"
          ↪  from="Solver" to="Raytracer" initialize="yes" />
87        <exchange data="TerrainFluxesSolar"  mesh="TerrainMeshSolver"
          ↪  from="Raytracer" to="Solver" />
88        <exchange data="TerrainFluxesIR"     mesh="TerrainMeshSolver"
          ↪  from="Raytracer" to="Solver" />
89        <exchange data="SampleTemperatures"  mesh="SampleMeshSolver"
          ↪  from="Solver" to="Raytracer" initialize="yes" />
90        <exchange data="SampleFluxesSolar"   mesh="SampleMeshSolver"
          ↪  from="Raytracer" to="Solver" />
91        <exchange data="SampleFluxesIR"      mesh="SampleMeshSolver"
          ↪  from="Raytracer" to="Solver" />
92        <exchange data="AuxiliaryData"       mesh="AuxiliaryMeshSolver"
          ↪  from="Solver" to="Raytracer" initialize="yes" />
93      </coupling-scheme:serial-explicit>
94
95    </solver-interface>
96
97  </precice-configuration>
```

A.2. TherMoS solver adapter

```
function coupleWithPrecice( azimuth, elevation, transform,
    terrainTempsInit, sampleTempsInit, solverFunction,
    resultsDirectoryPath, configFilePath, iDuration, iStepLength )

    global oSimulation;

    %% STEPS
    %
    % 1. Instantiate interface object and configure it.
    % 2. Geometric preprocesing: Mesh is written to preCICE.
    % 3. Initialization: Raytracer is being contacted
    % 4. Data initializazion/Thermal preprocessing: Writing initial
    % temperatures
    % 5. Actual coupling: Beginning to compute.
    %

    %% STEP 1
    disp("Launching preCICE coupling for TherMoS");
    disp("Pre-inizialization setup");

    % Replace dummy value in config file by value from GUI
    modifyConfig(configFilePath, iDuration, iStepLength);

    % Setup the precice interface
    preciceInterface = precice.SolverInterface("Solver");
    preciceInterface.configure("__THERMOS_CONFIG_TEMP__.xml");

    %% STEP 2
    % Get terrain, sample and aux sizes.
    terrainSize = length(oSimulation.GEOM.mfFaces);
    sampleSize = oSimulation.GEOM_sample.n_element;
    auxSize = 18;
    maxSize = max([terrainSize sampleSize auxSize]);

    % Get the mesh IDs
    terrainMeshID = preciceInterface.getMeshID("TerrainMeshSolver");
    sampleMeshID = preciceInterface.getMeshID("SampleMeshSolver");
    auxMeshID = preciceInterface.getMeshID("AuxiliaryMeshSolver");

    % Set vertex positions to [0 0 0], [1 0 0], ...
    vertexPositions = [0:(maxSize-1);zeros(2,maxSize)];
```

```
40      terrainVertexIDs =
        ↪ preciceInterface.setMeshVertices(terrainMeshID,...
41          terrainSize,vertexPositions(:,1:sampleSize));
42      sampleVertexIDs = preciceInterface.setMeshVertices(sampleMeshID,...
43          sampleSize,vertexPositions(:,1:sampleSize));
44      auxVertexIDs =
        ↪ preciceInterface.setMeshVertices(auxMeshID,auxSize,...
45          vertexPositions(:,1:auxSize));
46
47      %% STEP 3
48      dt = preciceInterface.initialize();
49      tInit = tic;
50
51      %% STEP 4
52      % Data IDs
53      terrainTempID = preciceInterface.getDataID
        ↪ ("TerrainTemperatures",terrainMeshID);
54      sampleTempID = preciceInterface.getDataID
        ↪ ("SampleTemperatures",sampleMeshID);
55      auxDataID = preciceInterface.getDataID("AuxiliaryData",auxMeshID);
56      terrainSolarID = preciceInterface.getDataID
        ↪ ("TerrainFluxesSolar",terrainMeshID);
57      terrainIRID =
        ↪ preciceInterface.getDataID("TerrainFluxesIR",terrainMeshID);
58      sampleSolarID = preciceInterface.getDataID
        ↪ ("SampleFluxesSolar",sampleMeshID);
59      sampleIRID =
        ↪ preciceInterface.getDataID("SampleFluxesIR",sampleMeshID);
60
61          % Prepare aux data
62      auxDataInit = [azimuth;elevation;transform(:)];
63
64          % Write initial data
65      sActionInit = precice.Constants.actionWriteInitialData();
66      if preciceInterface.isActionRequired(sActionInit)
67          preciceInterface.writeBlockScalarData⌋
            ↪ (terrainTempID,terrainSize,terrainVertexIDs,terrainTempsInit);
68          preciceInterface.writeBlockScalarData⌋
            ↪ (sampleTempID,sampleSize,sampleVertexIDs,sampleTempsInit);
69          preciceInterface.writeBlockScalarData⌋
            ↪ (auxDataID,18,auxVertexIDs,auxDataInit);
70          preciceInterface.fulfilledAction(sActionInit);
71      end
72
```

```
73      preciceInterface.initializeData();
74
75      %% STEP 5
76      tComp = tic;
77      while(preciceInterface.isCouplingOngoing())
78              % Reading the fluxes
79          tRead = tic;
80          terrainSolar = preciceInterface.readBlockScalarData
            ↪ (terrainSolarID,terrainSize,terrainVertexIDs,true);
81          terrainIR = preciceInterface.readBlockScalarData
            ↪ (terrainIRID,terrainSize,terrainVertexIDs,true);
82          sampleSolar = preciceInterface.readBlockScalarData
            ↪ (sampleSolarID,sampleSize,sampleVertexIDs,true);
83          sampleIR = preciceInterface.readBlockScalarData
            ↪ (sampleIRID,sampleSize,sampleVertexIDs,true);
84          fprintf('READING FLUXES : %9.5f seconds.\n', toc(tRead))
85
86          % Call the solver
87          tSolve = tic;
88          [azimuth, elevation, transform, terrainTemps, sampleTemps] =
            ↪ feval(solverFunction,terrainSolar, sampleSolar, terrainIR,
            ↪ sampleIR);
89          fprintf('SOLVING        : %9.5f seconds.\n', toc(tSolve))
90
91          % Write Data
92          tWrite = tic;
93          auxData = [azimuth;elevation;transform(:)];
94          preciceInterface.writeBlockScalarData
            ↪ (terrainTempID,terrainSize,terrainVertexIDs,terrainTemps);
95          preciceInterface.writeBlockScalarData
            ↪ (sampleTempID,sampleSize,sampleVertexIDs,sampleTemps);
96          preciceInterface.writeBlockScalarData
            ↪ (auxDataID,18,auxVertexIDs,auxData);
97          fprintf('WRITING TEMPS  : %9.5f seconds.\n', toc(tWrite))
98
99          % Advance
100         dt = preciceInterface.advance(dt);
101     end
102
103     % Print times
104     fprintf('---------------------------------');
105     fprintf('COMPUTING TIME : %9.5f seconds.\n', toc(tComp))
106     fprintf('COMPLETE TIME  : %9.5f seconds.\n', toc(tInit))
107     fprintf('---------------------------------');
```

```
108
109     % Finalize preCICE
110     preciceInterface.finalize();
111
112     % Remove temporary config file
113     delete("__THERMOS_CONFIG_TEMP__.xml");
114 end
```

A.3. TherMoS ray tracer adapter

A.3.1. Common function

The following code is the `run` function of the `OptixSimulator` class, which is called by the `main()` function and used in both variants of the ray tracer.

```
1  void OptixSimulatorPrecice::run()
2  {
3          int size, rank;
4          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5          MPI_Comm_size(MPI_COMM_WORLD, &size);
6
7          // Configuration of preCICE
8          preciceInterface.configure(scene_->settings
           ↪  ().precice_config_file.c_str());
9
10         // Declaring geometry etc. to preCICE
11         declare_geometry_to_precice(rank);
12
13         // Initializing preCICE
14         double dt,t,t_total,t_comploop,t_cast,t_init;
15         t_init = MPI_Wtime();
16         dt = preciceInterface.initialize();
17         t_init = MPI_Wtime() - t_init;
18         printf("INITIALIZATION : %9.5f seconds.\n",t_init);
19         t = MPI_Wtime();
20         t_total = t;
21
22         // Get data IDs
23         terrainTempID = preciceInterface.getDataID
           ↪  ("TerrainTemperatures",terrainMeshID);
24         sampleTempID = preciceInterface.getDataID
           ↪  ("SampleTemperatures",sampleMeshID);
25         auxDataID =
           ↪  preciceInterface.getDataID("AuxiliaryData",auxMeshID);
```

```
26          terrainSolarID = preciceInterface.getDataID
            ↪  ("TerrainFluxesSolar",terrainMeshID);
27          terrainIRID =
            ↪  preciceInterface.getDataID("TerrainFluxesIR",terrainMeshID);
28          sampleSolarID = preciceInterface.getDataID
            ↪  ("SampleFluxesSolar",sampleMeshID);
29          sampleIRID =
            ↪  preciceInterface.getDataID("SampleFluxesIR",sampleMeshID);
30
31          // Initialize data
32          preciceInterface.initializeData();
33          t = (MPI_Wtime() - t);
34          printf("INITIAL SETUP  : %9.5f seconds.\n",t);
35
36          // Now the coupling can start
37          t_comploop = MPI_Wtime();
38          while (preciceInterface.isCouplingOngoing())
39          {
40                  // read solver results - read_input calls the precice
                    ↪  functions
41                  t = MPI_Wtime();
42                  read_input(rank);
43                  t = (MPI_Wtime() - t);
44                  printf("READING TEMPS  : %9.5f seconds.\n",t);
45
46                  // Computation step
47                  t = MPI_Wtime();
48                  simulate(rank, size);
49                  t = (MPI_Wtime() - t);
50                  printf("RAYTRACING     : %9.5f seconds.\n",t);
51
52                  // write raytracer results - rank 0 calls send_output,
                    ↪  which calls preCICEs write functions
53                  t = MPI_Wtime();
54                  t_cast = send_output(rank);
55                  t = (MPI_Wtime() - t);
56                  printf("WRITING FLUXES : %9.5f seconds.\n",t);
57                  printf("CASTING FLUXES : %9.5f seconds.\n",t_cast);
58
59                  printf("Advancing...\n");
60                  dt = preciceInterface.advance(dt);
61          }
62
63          // Finalize MPI and print final times
```

```
64          t_comploop = (MPI_Wtime() - t_comploop);
65          t_total = (MPI_Wtime() - t_total);
66          printf("---------------------------------\n");
67          printf("COMPUTING TIME : %9.5f seconds.\n",t_comploop);
68          printf("COMPLETE TIME  : %9.5f seconds.\n",t_total);
69          printf("---------------------------------\n");
70          preciceInterface.finalize();
71          printf("Raytracer sync ended.\n");
72  }
```

A.3.2. Original variant

The following code contains the helper functions called by the `run` function as they are implemented in the simple ray tracer variant with explicit broadcasting and reducing.

```
1   void OptixSimulatorPrecice::declare_geometry_to_precice(int mpi_rank)
2   {
3           if(mpi_rank==0) {
4                   // Get the mesh IDs
5                   terrainMeshID =
                    ↪  preciceInterface.getMeshID("TerrainMeshRaytracer");
6                   sampleMeshID =
                    ↪  preciceInterface.getMeshID("SampleMeshRaytracer");
7                   auxMeshID = preciceInterface.getMeshID
                    ↪  ("AuxiliaryMeshRaytracer");
8
9                   // Declare terrain geometry
10                  std::vector<double> positions(3*terrainSize);
11                  for (int j = 0; j < terrainSize; j++) {
12                          positions[3*j] = j;
13                          positions[3*j+1] = 0;
14                          positions[3*j+2] = 0;
15                  }
16                  preciceInterface.setMeshVertices(terrainMeshID,
                    ↪  terrainSize, positions.data(),
                    ↪  terrainVertexIDs.data());
17
18                  // Declare terrain geometry
19                  for (int j = 0; j < sampleSize; j++) {
20                          positions[3*j] = j;
21                          positions[3*j+1] = 0;
22                          positions[3*j+2] = 0;
23                  }
```

```
                preciceInterface.setMeshVertices(sampleMeshID,
                ↪ sampleSize, positions.data(),
                ↪ sampleVertexIDs.data());

                // Declare auxiliary mesh geometry
                for (int j = 0; j < auxSize; j++) {
                        positions[3*j] = j;
                        positions[3*j+1] = 0;
                        positions[3*j+2] = 0;
                }
                preciceInterface.setMeshVertices(auxMeshID, auxSize,
                ↪ positions.data(), auxVertexIDs.data());
        }
}
```

```
void OptixSimulatorPrecice::read_input(int mpi_rank)
{
        double t = MPI_Wtime();

        float azimuth;
        float elevation;
        float sample_transform[16];
        float *terrain_temps =
        ↪ static_cast<float*>(terrain_temp_buf_->map());
        float *sample_temps =
        ↪ static_cast<float*>(sample_temp_buf_->map());

        // Rank 0 reads and broadcasts to the other ranks
        MPI_Request requests[5];
        if (mpi_rank==0) {
                preciceInterface.readBlockScalarData(terrainTempID,
                ↪ terrainSize, terrainVertexIDs.data(),
                ↪ terrainDouble.data());
                for (int j = 0; j<terrainSize; j++) {
                        terrain_temps[j] =
                        ↪ static_cast<float>(terrainDouble[j]);
                }
                MPI_Ibcast(terrain_temps, terrainSize, MPI_FLOAT, 0,
                ↪ MPI_COMM_WORLD, requests+3);
                preciceInterface.readBlockScalarData(sampleTempID,
                ↪ sampleSize, sampleVertexIDs.data(),
                ↪ sampleDouble.data());
                for (int j = 0; j<sampleSize; j++) {
```

```
21                          sample_temps[j] =
                             ↪  static_cast<float>(sampleDouble[j]);
22                  }
23                  MPI_Ibcast(sample_temps, sampleSize, MPI_FLOAT, 0,
                     ↪  MPI_COMM_WORLD, requests+4);
24                  preciceInterface.readBlockScalarData(auxDataID, auxSize,
                     ↪  auxVertexIDs.data(), auxDouble.data());
25                  azimuth = static_cast<float>(auxDouble[0]);
26                  elevation = static_cast<float>(auxDouble[1]);
27                  for (int j = 0; j<16; j++) {
28                          sample_transform[j] =
                             ↪  static_cast<float>(auxDouble[2+j]);
29                  }
30                  MPI_Ibcast(&azimuth, 1, MPI_FLOAT, 0, MPI_COMM_WORLD,
                     ↪  requests);
31                  MPI_Ibcast(&elevation, 1, MPI_FLOAT, 0, MPI_COMM_WORLD,
                     ↪  requests+1);
32                  MPI_Ibcast(sample_transform, 16, MPI_FLOAT, 0,
                     ↪  MPI_COMM_WORLD, requests+2);
33          } else {
34                  MPI_Ibcast(terrain_temps, terrainSize, MPI_FLOAT, 0,
                     ↪  MPI_COMM_WORLD, requests+3);
35                  MPI_Ibcast(sample_temps, sampleSize, MPI_FLOAT, 0,
                     ↪  MPI_COMM_WORLD, requests+4);
36                  MPI_Ibcast(&azimuth, 1, MPI_FLOAT, 0, MPI_COMM_WORLD,
                     ↪  requests);
37                  MPI_Ibcast(&elevation, 1, MPI_FLOAT, 0, MPI_COMM_WORLD,
                     ↪  requests+1);
38                  MPI_Ibcast(sample_transform, 16, MPI_FLOAT,
                     ↪  0,MPI_COMM_WORLD, requests+2);
39          }
40          MPI_Waitall(5,requests,MPI_STATUSES_IGNORE);
41
42          // unmap buffers
43          terrain_temp_buf_->unmap();
44          sample_temp_buf_->unmap();
45
46          // set sample transform
47          scene_->set_sample_transform(sample_transform);
48
49          // recalculate sun parallelograms
50          scene_->set_sun_angles(azimuth, elevation);
51  }
52
```

```
53 void OptixSimulatorPrecice::simulate(int mpi_rank, int mpi_size)
54 {
55         ((OptixScene*)scene_)->setup_params();
56 
57         std::fill(ir_sample_.begin(), ir_sample_.end(), 0.0f);
58         std::fill(ir_terrain_.begin(), ir_terrain_.end(), 0.0f);
59         std::fill(solar_sample_.begin(), solar_sample_.end(), 0.0f);
60         std::fill(solar_terrain_.begin(), solar_terrain_.end(), 0.0f);
61 
62         MPI_Request requests[4];
63 
64         // Infrared
65         launch_infrared_rays(mpi_rank, mpi_size);
66 
67         if (mpi_rank == 0)
68         {
69                 MPI_Ireduce(MPI_IN_PLACE, &ir_sample_[0],
                    ↪  ir_sample_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[0]);
70                 MPI_Ireduce(MPI_IN_PLACE, &ir_terrain_[0],
                    ↪  ir_terrain_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[1]);
71         }
72         else
73         {
74                 MPI_Ireduce(&ir_sample_[0], &ir_sample_[0],
                    ↪  ir_sample_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[0]);
75                 MPI_Ireduce(&ir_terrain_[0], &ir_terrain_[0],
                    ↪  ir_terrain_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[1]);
76         }
77 
78         // Solar
79         launch_solar_rays(mpi_rank, mpi_size);
80 
81         if (mpi_rank == 0)
82         {
83                 MPI_Ireduce(MPI_IN_PLACE, &solar_sample_[0],
                    ↪  solar_sample_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[2]);
84                 MPI_Ireduce(MPI_IN_PLACE, &solar_terrain_[0],
                    ↪  solar_terrain_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[3]);
```

```
85          }
86          else
87          {
88                  MPI_Ireduce(&solar_sample_[0], &solar_sample_[0],
                    ↪  solar_sample_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[2]);
89                  MPI_Ireduce(&solar_terrain_[0], &solar_terrain_[0],
                    ↪  solar_terrain_.size(), MPI_FLOAT, MPI_SUM, 0,
                    ↪  MPI_COMM_WORLD, &requests[3]);
90          }
91
92          MPI_Waitall(4, requests, MPI_STATUSES_IGNORE);
93  }
94
95  double OptixSimulatorPrecice::send_output(int mpi_rank)
96  {
97          if(mpi_rank==0) {
98                  double t_cast,t_temp;
99                  t_temp = MPI_Wtime();
100                 for (int j = 0; j<terrainSize; j++) {
101                         terrainDouble[j] =
                            ↪  static_cast<double>(solar_terrain_[j]);
102                 }
103                 for (int j = 0; j<sampleSize; j++) {
104                         sampleDouble[j] =
                            ↪  static_cast<double>(solar_sample_[j]);
105                 }
106                 t_cast = MPI_Wtime()-t_temp;
107                 preciceInterface.writeBlockScalarData
                    ↪  (terrainSolarID,terrainSize,terrainVertexIDs.data
                    ↪  (),terrainDouble.data());
108                 preciceInterface.writeBlockScalarData
                    ↪  (sampleSolarID,sampleSize,sampleVertexIDs.data
                    ↪  (),sampleDouble.data());
109
110                 t_temp = MPI_Wtime();
111                 for (int j = 0; j<terrainSize; j++) {
112                         terrainDouble[j] =
                            ↪  static_cast<double>(ir_terrain_[j]);
113                 }
114                 for (int j = 0; j<sampleSize; j++) {
115                         sampleDouble[j] =
                            ↪  static_cast<double>(ir_sample_[j]);
116                 }
```

```
117                 t_cast = t_cast+ MPI_Wtime()-t_temp;
118                 preciceInterface.writeBlockScalarData
                    ↪ (terrainIRID,terrainSize,terrainVertexIDs.data
                    ↪ (),terrainDouble.data());
119                 preciceInterface.writeBlockScalarData
                    ↪ (sampleIRID,sampleSize,sampleVertexIDs.data
                    ↪ (),sampleDouble.data());
120                 return t_cast;
121         }
122 }
```

A.3.3. Conservatie mapping variant

This code contains the helper functions as they are implemented in the ray tracer variant using conservative mapping.

```
1  void OptixSimulatorPrecice::declare_geometry_to_precice(int mpi_rank)
2  {
3          // Get the mesh IDs
4          terrainMeshID =
           ↪ preciceInterface.getMeshID("TerrainMeshRaytracer");
5          sampleMeshID =
           ↪ preciceInterface.getMeshID("SampleMeshRaytracer");
6          auxMeshID =
           ↪ preciceInterface.getMeshID("AuxiliaryMeshRaytracer");
7
8          // Declare terrain geometry
9          std::vector<double> positions(3*terrainSize);
10         for (int j = 0; j < terrainSize; j++) {
11                 positions[3*j] = j;
12                 positions[3*j+1] = 0;
13                 positions[3*j+2] = 0;
14         }
15         preciceInterface.setMeshVertices(terrainMeshID, terrainSize,
           ↪ positions.data(), terrainVertexIDs.data());
16
17         // Declare terrain geometry
18         for (int j = 0; j < sampleSize; j++) {
19                 positions[3*j] = j;
20                 positions[3*j+1] = 0;
21                 positions[3*j+2] = 0;
22         }
23         preciceInterface.setMeshVertices(sampleMeshID, sampleSize,
           ↪ positions.data(), sampleVertexIDs.data());
```

```
24
25          // Declare auxiliary mesh geometry
26          for (int j = 0; j < auxSize; j++) {
27                  positions[3*j] = j;
28                  positions[3*j+1] = 0;
29                  positions[3*j+2] = 0;
30          }
31          preciceInterface.setMeshVertices(auxMeshID, auxSize,
            ↪  positions.data(), auxVertexIDs.data());
32  }

1   void OptixSimulatorMapHack::read_input(int mpi_rank)
2   {
3           double t = MPI_Wtime();
4
5           float azimuth;
6           float elevation;
7           float sample_transform[16];
8           float *terrain_temps =
            ↪  static_cast<float*>(terrain_temp_buf_->map());
9           float *sample_temps =
            ↪  static_cast<float*>(sample_temp_buf_->map());
10
11          preciceInterface.readBlockScalarData(terrainTempID, terrainSize,
            ↪  terrainVertexIDs.data(), terrainDouble.data());
12          for (int j = 0; j<terrainSize; j++) {
13                  terrain_temps[j] = static_cast<float>(terrainDouble[j]);
14          }
15          preciceInterface.readBlockScalarData(sampleTempID, sampleSize,
            ↪  sampleVertexIDs.data(), sampleDouble.data());
16          for (int j = 0; j<sampleSize; j++) {
17                  sample_temps[j] = static_cast<float>(sampleDouble[j]);
18          }
19          preciceInterface.readBlockScalarData(auxDataID, auxSize,
            ↪  auxVertexIDs.data(), auxDouble.data());
20          azimuth = static_cast<float>(auxDouble[0]);
21          elevation = static_cast<float>(auxDouble[1]);
22          for (int j = 0; j<16; j++) {
23                  sample_transform[j] =
                    ↪  static_cast<float>(auxDouble[2+j]);
24          }
25
26          // unmap buffers
27          terrain_temp_buf_->unmap();
```

```
28          sample_temp_buf_->unmap();
29
30          // set sample transform
31          scene_->set_sample_transform(sample_transform);
32
33          // recalculate sun parallelograms
34          scene_->set_sun_angles(azimuth, elevation);
35  }
36
37  void OptixSimulatorMapHack::simulate(int mpi_rank, int mpi_size)
38  {
39          ((OptixScene*)scene_)->setup_params();
40
41          std::fill(ir_sample_.begin(), ir_sample_.end(), 0.0f);
42          std::fill(ir_terrain_.begin(), ir_terrain_.end(), 0.0f);
43          std::fill(solar_sample_.begin(), solar_sample_.end(), 0.0f);
44          std::fill(solar_terrain_.begin(), solar_terrain_.end(), 0.0f);
45
46          // Infrared
47          launch_infrared_rays(mpi_rank, mpi_size);
48
49          // Solar
50          launch_solar_rays(mpi_rank, mpi_size);
51  }
52
53  double OptixSimulatorMapHack::send_output(int mpi_rank)
54  {
55          double t_cast,t_temp;
56          t_temp = MPI_Wtime();
57          for (int j = 0; j<terrainSize; j++) {
58                  terrainDouble[j] =
                    ↪  static_cast<double>(solar_terrain_[j]);
59          }
60          for (int j = 0; j<sampleSize; j++) {
61                  sampleDouble[j] = static_cast<double>(solar_sample_[j]);
62          }
63          t_cast = MPI_Wtime()-t_temp;
64          preciceInterface.writeBlockScalarData⌋
            ↪  (terrainSolarID,terrainSize,terrainVertexIDs.data⌋
            ↪  (),terrainDouble.data());
65          preciceInterface.writeBlockScalarData⌋
            ↪  (sampleSolarID,sampleSize,sampleVertexIDs.data⌋
            ↪  (),sampleDouble.data());
66
```

```
67          t_temp = MPI_Wtime();
68          for (int j = 0; j<terrainSize; j++) {
69                  terrainDouble[j] = static_cast<double>(ir_terrain_[j]);
70          }
71          for (int j = 0; j<sampleSize; j++) {
72                  sampleDouble[j] = static_cast<double>(ir_sample_[j]);
73          }
74          t_cast = t_cast+MPI_Wtime()-t_temp;
75          preciceInterface.writeBlockScalarData
            ↪ (terrainIRID,terrainSize,terrainVertexIDs.data
            ↪ (),terrainDouble.data());
76          preciceInterface.writeBlockScalarData
            ↪ (sampleIRID,sampleSize,sampleVertexIDs.data
            ↪ (),sampleDouble.data());
77          return t_cast;
78  }
```

Bibliography

[1] National Aeronautics and Space Administration. Apollo 11 mission report, 1969.

[2] Mohammad Alhasni. Multi-gpu parallelization of dynamic heat transfer model on the moon. Master's thesis, Department of Informatics, Technical University of Munich, Munich, March 2018.

[3] C. Bernardi, Y. Maday, and A. T. Patera. *Domain Decomposition by the Mortar Element Method*, pages 269–286. Springer Netherlands, Dordrecht, 1993.

[4] Boost. Boost C++ Libraries. `http://www.boost.org/`, 2019. Accessed 15-July-2019.

[5] Bernd Brügge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[6] Hans-Joachim Bungartz, Florian Lindner, Bernhard Gatzhammer, Miriam Mehl, Klaudius Scheufele, Alexander Shukaev, and Benjamin Uekermann. preCICE – a fully parallel library for multi-physics surface coupling. *Computers and Fluids*, 141:250–258, 2016. Advances in Fluid-Structure Interaction.

[7] Hans-Joachim Bungartz and Michael Schäfer, editors. *Fluid-Structure Interaction - Modelling, Simulation, Optimisation*. Number 53 in Lecture Notes in Computational Science and Engineering. Springer-Verlag, Berlin, Aug 2006. editor: Bungartz, Hans-Joachim; Schäfer, Michael¡br¿address: Berlin¡br¿series: Lecture Notes in Computational Science and Engineering¡br¿.

[8] J.I. Castor. *Radiation Hydrodynamics*. Radiation Hydrodynamics. Cambridge University Press, 2004.

[9] Paola Causin, Jean-Frédéric Gerbeau, and Fabio Nobile. Added-mass effect in the design of partitioned algorithms for fluid-structure problems. (RR-5084), 2004.

[10] Yunus A. Çengel. *Heat Transfer: A Practical Approach*. McGraw-Hill series in mechanical engineering. McGraw-Hill, 2003.

[11] Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006.

[12] Benjamin Uekermann et al. Handling global coupling data, 2018. [Online; accessed 15-July-2019].

[13] Gerasimos Chourdakis et al. Implement a matlab api, 2018. [Online; accessed 15-July-2019].

[14] Oliver Woodford et al. Making c++ objects persistent between mex calls, and robust. [Online; accessed 14-July-2019].

[15] Inc. Free Software Foundation. Gcc, the gnu compiler collection. [Online; accessed 14-July-2019].

[16] Bernhard Gatzhammer. *Efficient and Flexible Partitioned Simulation of Fluid-Structure Interactions*. Dissertation, Technische Universität München, München, 2014.

[17] M. W. Gee, U. Küttler, and W. A. Wall. Truly monolithic algebraic multigrid for fluid–structure interaction. *International Journal for Numerical Methods in Engineering*, 85(8):987–1016, 2011.

[18] D.G. Gilmore. *Spacecraft Thermal Control Handbook: Fundamental technologies*. Spacecraft Thermal Control Handbook. Aerospace Press, 2002.

[19] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, UK, 1989.

[20] William Gropp, Anthony Skjellum, and Ewing L. Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 2nd edition, 1999.

[21] Philipp Hager. *Dynamic thermal modeling for moving objects on the Moon*. PhD thesis, Technische Universität München, 2013.

[22] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. 24(4):145–154, August 1990.

[23] The MathWorks Inc. Compilers - matlab & simulink. [Online; accessed 14-July-2019].

[24] The MathWorks Inc. Matlab documentation, 2019. [Online; accessed 18-July-2019].

[25] The MathWorks Inc. MATLAB R2019a, 2019. Software. URL `http://mathworks.com`.

[26] F.P. Incropera, T.L. Bergman, D.P. DeWitt, and A.S. Lavine. *Fundamentals of Heat and Mass Transfer*. Wiley, 2013.

[27] David Keyes, Lois C. McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, Emil Constantinescu, Donald Estep, Katherine Evans, Charbel Farhat, Ammar Hakim, Glenn

Hammond, Glen Hansen, Judith Hill, Tobin Isaac, and Barbara Wohlmuth. Multiphysics simulations: Challenges and opportunities. *International Journal of High Performance Computing Applications*, 27, 02 2013.

[28] Ulrich Küttler and Wolfgang A. Wall. Fixed-point fluid–structure interaction solvers with dynamic relaxation. *Computational Mechanics*, 43(1):61–72, Dec 2008.

[29] Zhen Leo Liu. *Multiphysics in Porous Materials*. 07 2018.

[30] Peter Li. Matlab mex in-place editing — undocumented matlab. [Online; accessed 14-July-2019].

[31] Florian Lindner, Miriam Mehl, and Benjamin Uekermann. *Radial Basis Function Interpolation for Black-Box Multi-Physics Simulations*. 05 2017.

[32] L. Liu, C. Zhang, R. Li, B. Wang, and G. Yang. C-coupler2: a flexible and user-friendly community coupler for model coupling and nesting. *Geoscientific Model Development*, 11(9):3557–3586, 2018.

[33] ITP Engines UK Ltd. Esatan-tms, 2019. [Online; accessed 18-July-2019].

[34] Matthias Mayr. *A Monolithic Solver for Fluid-Structure Interactionwith Adaptive Time Stepping and a HybridPreconditioner*. Dissertation, Fakultät für Maschinenwesen, Technische Universität München, 2016.

[35] NVIDIA. Optix™ ray tracing engine 6.0.0, 2018. `https://developer.nvidia.com/optix`.

[36] NVIDIA. CUDA toolkit documentation v10.1.168, 2019. `https://developer.nvidia.com/cuda-toolkit`.

[37] Wolfgang Polifke and Jan Kopitz. *Wärmeübertragung : Grundlagen, analytische und numerische Methoden*. Pearson, München [u.a.], 2009.

[38] preCICE. precice – a coupling library for partitioned multi-physics simulations on massively parallel systems, 2019. [Online; accessed 13-July-2019].

[39] preCICE. precice :: Home, 2019. [Online; accessed 13-July-2019].

[40] preCICE. precice/precice wiki • github, 2019. [Online; accessed 13-July-2019].

[41] Kevin Suffern. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., Natick, MA, USA, 2007.

[42] Benjamin Uekermann. *Partitioned Fluid-Structure Interaction on Massively Parallel Systems*. Dissertation, Institut für Informatik, Technische Universität München, October 2016.