SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Extending the FEniCSx Adapter for the Coupling Library preCICE

Philip Hildebrand

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Extending the FEniCSx Adapter for the Coupling Library preCICE

Erweiterung des FEniCSx-Adapters für die Kopplungsbibliothek preCICE

Author:	Philip Hildebrand
Supervisor:	Prof. Dr. Hans-Joachim Bungartz
Advisors:	M.Sc. Ishaan Desai (University of Stuttgart), M.Sc. (hons) Benjamin Rodenberg
Submission Date:	March 15, 2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, March 15, 2023 Philip Hildebrand

Acknowledgments

I want to thank my advisors Ishaan and Benjamin who guided me through my work during the last months. Before starting my thesis, I had no knowledge about the topic and they helped me understand it a lot. I also want to thank my friends and family for the continuous support they gave me.

Abstract

In partitioned multi-physics simulations individual single-physics solvers are coupled together to cooperatively solve a multi-physics problem. In order to help with code coupling and data exchange between individual solvers there is a broad range of coupling tools. Since different solvers may have different data representations, it is an additional task of the coupling tool to mediate between them. However, the coupling tool can not be expected to know the internal works of every solver. In the case of the coupling tool preCICE, this issue is solved by the use of adapters which connect preCICE with a solver and allow it to steer the coupled simulation while treating the solver as a black-box. One of these adapters is the FEniCSx-preCICE adapter which links preCICE and the FEM library FEniCSx together. Prior to this work it existed as a sketch that was ultimately unusable. In this thesis I finish the implementation of the FEniCSx-preCICE adapter. Additionally, I modify an existing preCICE test case consisting of a partitioned setup of the heat equation to be compatible with the adapter and use it to evaluate the adapter's performance.

Contents

Acknowledgments iii

Abstract iv

1 Introduction 1

2 Mathematical background 3
2.1 Differential equations 3
2.2 Boundary conditions 3
2.2.1 Dirichlet boundary conditions 3
2.2.2 Neumann boundary conditions 4
2.3 Finite Element Method 4
2.4 Heat Equation 4
2.4.1 Analytical solution 6
2.4.2 Heat flux 6
2.4.3 A partitioned approach 7

3 Software components 9
3.1 preCICE 9
3.1.1 Uni-directional or bi-directional coupling 9
3.1.2 Coupling schemes 10
3.1.3 Data mapping 10
3.1.4 Python bindings 11
3.2 FEniCSx 11
3.2.1 Creating a mesh 11
3.2.2 Creating a function space and functions 12
3.2.3 Defining boundary conditions 12
3.2.4 Creating boundary conditions 13
3.2.5 Defining a variational problem 13
3.2.6 Performing the timestepping 13
3.3 FEniCS-preCICE adapter 14

4 Implementation **15**
4.1 FEniCSx-preCICE-adapter . . . 15
4.2 Previous state of the adapter . . . 16
4.3 Changes to the adapter . . . 16
4.3.1 Changing FEniCSx-to-preCICE communication . . . 16
4.3.2 Adjusting the adapter's scope . . . 17
4.4 Solver for the partitioned heat equation . . . 19
4.5 Changes to the error computation . . . 20

5 Testing and Evaluation **22**
5.1 Performance . . . 22
5.1.1 Increasing ny . . . 22
5.1.2 Increasing nx . . . 23
5.1.3 Coupling non-matching meshes . . . 24
5.2 Comparison with the FEniCS-preCICE adapter . . . 25
5.3 Accuracy . . . 26
5.4 Discussion . . . 28

6 Conclusion **29**
6.1 Summary . . . 29
6.2 Future work . . . 30

List of Figures **31**

List of Tables **32**

Bibliography **33**

1 Introduction

When performing simulations of physical phenomena[1], there are two general approaches [2]. The first is the monolithic approach, where the behavior of the system to be simulated is expressed in one all-encompassing system of equations which is then solved by one software, called solver. One example for this is the solver turtleFSI which can be used to simulate Fluid-Structure Interactions (FSI) in a monolithic way[3]. The other is the partitioned approach, where the system is split up into smaller subsystems, each expressed by their own systems of equations. These subsystems are then solved by independent solvers and subsequently coupled together to perform the simulation of the entire system. While monolithic approaches are applicable for well defined systems that are not expected to be changed, they show clear deficiencies when parts of the system are modified or removed or new parts are added. In these cases, the entire system of equations would need to be redefined [2]. For those scenarios, using a partitioned approach proves to be efficient, since the solvers of the subsystems that are left untouched stay unchanged as well.

As an example, a solver in a partitioned setup could employ the open-source computing platform FEniCSx[1], which allows solving systems by using the Finite Element Method (FEM) [4].

To coordinate the individual solvers in partitioned simulations one can use the multi-physics coupling library preCICE [5], which defines a common communication standard with which solvers send their data to and receive data of other solvers from. In order to do this however, both preCICE and the solvers need to receive the data in a format they understand.

In this thesis, I focus on the FEniCSx-preCICE adapter. In the scope of preCICE, an adapter is a piece of software that integrates preCICE with a solver in a minimally-invasive way[6]. The FEniCSx-preCICE adapter allows solvers using FEniCSx to read data from and write data to preCICE by translating between their differing data representations. Prior to this work, the adapter existed as a draft but was not usable in practice. The main goal of this thesis is to turn the adapter into a working piece of software. An additional goal is to test its accuracy and efficiency. For this, I take one of many tutorial cases offered by preCICE, a partitioned setup of the heat equation[2], and

[1]https://fenicsproject.org/

[2]https://github.com/precice/tutorials/tree/develop/partitioned-heat-conduction

modify it to be usable with the FEniCSx-preCICE adapter.

Chapter 2 explains differential equations, in particular the heat equation, as well as boundary conditions and the Finite Element Method and introduces a partitioned setup of the heat equation. Chapter 3 deals with preCICE and FEniCSx and presents their core features. Chapter 4 addresses the implementation of the adapter and the solver for the partitioned heat equation and shows the major changes compared to the FEniCS-preCICE adapter. In chapter 5 the implementation is discussed in terms of accuracy and computing time. Finally, chapter 6 offers a summary of the findings of this thesis and provides an outlook on future work.

2 Mathematical background

In this chapter I go over the mathematical background that is needed to understand the following chapters. In section 2.1 I shortly explain partial differential equations. In section 2.2 I cover boundary conditions which further define the behaviour of differential equations. I use the concepts shown in these sections to present the Finite Element Method which solves partial differential equations by discretizing their domain in section 2.3. Lastly, I show the heat equation as an example that can be solved with the Finite Element Method as well as a partitioned approach for it in section 2.4.

2.1 Differential equations

Differential equations are equations that contain an unknown function as well as at least one derivative of that function. Partial differential equations (PDEs) are a subset of differential equations where the function depends on multiple variables and the equation contains derivatives with respect to only certain quantities which affect that function.

2.2 Boundary conditions

A differential equation describes how a system changes, and to study this change one needs to define an initial state and some constrains which can then be changed as per the equation. One type of such conditions are boundary conditions that prescribe the behaviour of the equation on the boundary of its domain. While there are many types of boundary conditions, we focus on two in particular, the Dirichlet and the Neumann boundary condition.[1]

2.2.1 Dirichlet boundary conditions

A Dirichlet boundary condition specifies the value of the unknown function u on the boundary domain. It has the form

$$u(x) = f(x), x \in \partial\Omega \tag{2.1}$$

[1]https://en.wikipedia.org/wiki/Boundary_value_problem

with function domain Ω, domain boundary $\partial\Omega$ and function $f(x)$ defining the values of x on the domain boundary.

2.2.2 Neumann boundary conditions

A Neumann boundary condition specifies the value of the derivative of the unknown function u on the boundary domain. It has the form

$$\frac{\delta u(x)}{\delta n} = f(x), x \in \partial\Omega \tag{2.2}$$

with n being the unit normal to $\partial\Omega$.

2.3 Finite Element Method

One approach to solve PDEs is the Finite Element Method (FEM). Its main principle is to divide the continuous domain of the PDE into discrete parts, which are the finite elements. Their form depends on the dimensions of the domain, in a two-dimensional domain for example elements are usually triangles or quadrilaterals. This way we obtain a discrete number of points, which is called a mesh, with a single point being called node or vertex. This discretization is necessary to be able to solve the PDE. The next step is to create smaller equations for every element that specify their behaviour using interpolation functions. Generally, choosing interpolation functions with higher degrees leads to a more precise approximation but also a higher computational cost. We then collect all element equations in one system of equations that specifies the behaviour of the entire domain. Lastly, we impose the boundary conditions which makes us able to solve the system of equations [2].

2.4 Heat Equation

This section follows this tutorial[3] by J. S. Dokken which itself is an adaptation of [4]. The heat equation is a PDE which describes the stationary distribution of heat in a body to a time-dependent problem. It is defined as:

$$\frac{\delta u}{\delta t} = \nabla^2 u + f \quad \text{in } \Omega \times (0, T] \tag{2.3}$$

$$u = u_D \qquad \text{in } \partial\Omega \times (0, T] \tag{2.4}$$

$$u = u_0 \qquad \text{at } t = 0 \tag{2.5}$$

[2]http://hplgit.github.io/fem-book/doc/pub/book/pdf/fem-book-4screen.pdf
[3]https://jsdokken.com/dolfinx-tutorial/index.html

Here, Ω is the spacial domain, $\partial\Omega$ the border of Ω and ∇ the gradient. $u = u(x, y, t)$ is an unknown function, also referred to as trial function, and $f = f(x, y, t)$ a prescribed function, both of which vary with space and time. Furthermore, u_D and u_0 are Dirichlet boundary condition and initial condition, respectively. We only consider two-dimensional spacial domains since our simulation is two-dimensional as well.

In preparation of using the Finite Element Method, we first discretize the time derivative by a finite difference approximation. We introduce superscript n which lets us specify a value at a certain time step. For example, u^n means u at time step t_n with n being an integer counting time steps beginning at 0. We now sample the PDE at some time step, for instance t_{n+1}:

$$\frac{\delta u}{\delta t}^{n+1} = \nabla^2 u^{n+1} + f^{n+1} \tag{2.6}$$

The left hand side can be approximated by a difference quotient, resulting in:

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla^2 u^{n+1} + f^{n+1} \tag{2.7}$$

with time discretization parameter Δt. This procedure is called *implicit Euler* or *backwards Euler* discretization and gives us the time-discrete version of (2.3)

We can now reorder the equation such that the left-hand side only consists of terms containing the unknown u^{n+1} and the right-hand side consists of only computed terms. This results in a sequence of stationary problems for u^{n+1} given that u^n is known from the previous time step:

$$u^0 = u_0 \tag{2.8}$$

$$u^{n+1} - \Delta t \nabla^2 u^{n+1} = u^n + \Delta t f^{n+1}, n = 0, 1, 2, ... \tag{2.9}$$

We are now able to transform the equation into its weak form. For this, we multiply the equation with a test function v, integrate the result over Ω and perform integration by parts of terms with second order derivatives.
This yields:

$$a(u^{n+1}, v) = L_{n+1}(v), \tag{2.10}$$

where

$$a(u^{n+1}, v) = \int_\Omega (u^{n+1}v + \Delta t \nabla u^{n+1} \cdot \nabla v) dx \tag{2.11}$$

$$L_{n+1}(v) = \int_\Omega (u^n + \Delta t f^{n+1}) \cdot v \, dx \tag{2.12}$$

For the initial condition we get

$$a_0(u, v) = L_0(v), \tag{2.13}$$

with

$$a_0(u, v) = \int_\Omega uv \, dx \tag{2.14}$$

$$L_0(v) = \int_\Omega u_0 v \, dx \tag{2.15}$$

Obtaining the weak form of the equation helps us solving it.

2.4.1 Analytical solution

Solving PDEs using the Finite Element Method does not give the exact solution but rather an approximation of it. We therefore have to compare the result with the analytical solution at every time step and ensure they are sufficiently close. For this, we follow [4] and choose

$$u = 1 + x^2 + \alpha y^2 + \beta t \tag{2.16}$$

with arbitrary parameters α and β. This function ensures its computed values at the nodes to be exact, regardless of the size of the elements and δt, as long as the mesh is uniformly partitioned [4]. Inserting 2.16 into 2.3 gives us a solution for f:

$$f = \beta - 2 - 2\alpha \tag{2.17}$$

For the Dirichlet boundary value u_D and the initial value u_0 we get:

$$u_D(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t \tag{2.18}$$

$$u_0(x, y) = 1 + x^2 + \alpha y^2 \tag{2.19}$$

2.4.2 Heat flux

As postprocessing we can use the temperature u in order to compute the heat flux $q = \nabla u$, which we obtain by solving the problem

$$a(w, v) = L(v), \tag{2.20}$$

where

$$a(w, v) = \int_\Omega w \cdot v \, dx \tag{2.21}$$

$$L(v) = \int_\Omega \nabla u \cdot v \, dx \tag{2.22}$$

with functions v and w living in a vector version of the function space of u [4].

2.4.3 A partitioned approach

Until now we have considered the heat equation on a singular spacial domain Ω. We now want to make this a coupled setup, so we split the domain into two parts, resulting in two subdomains Ω_D on the left and Ω_N on the right which share the vertical coupling interface $\Gamma = \Omega_D \cap \Omega_N$. For this example we choose $\Omega = [0,2] \times [0,1]$, which leads to $\Omega_D = [0,1] \times [0,1]$ and $\Omega_N = [1,2] \times [0,1]$. We now have a coupled setup where

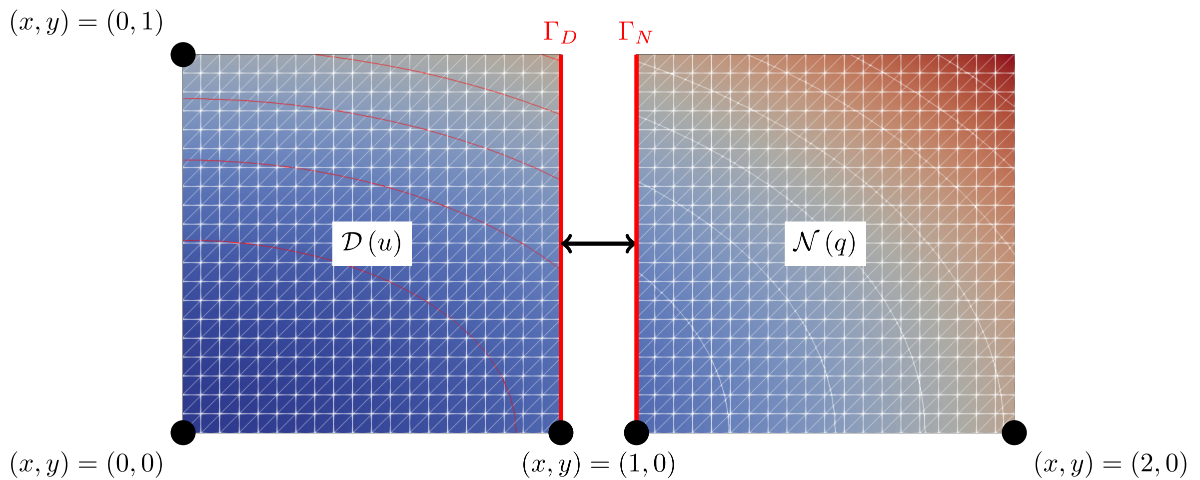

Figure 2.1: Illustration of the partitioned domain. Taken from [7]

each side solves the heat equation on its own and only communicates with the other by exchanging data at the coupling interface. In particular, the solver on domain Ω_D, the Dirichlet solver, takes a Dirichlet boundary condition from the solver on domain Ω_N, the Neumann solver, while the Neumann solver takes a Neumann boundary condition from the Dirichlet solver. This is known as the partitioned heat equation [7]. We keep the Dirichlet boundary condition from the unpartitioned heat equation and use the heat flux to form the Neumann boundary condition. Since the coupling interface is a

[4] http://hplgit.github.io/INF5620/doc/pub/fenics_tutorial1.1/tu2.html#tut-poisson-gradu

vertical line, we can discard the y-component of the heat flux vector by projecting the vector onto its x-component by multiplying it with its normal vector n, resulting in the Neumann boundary condition:

$$c_n = n \cdot \nabla u \tag{2.23}$$

Furthermore, we need u and q to be continuous on Γ to assure a correct transition between the two sides at every time step. For this, we start with initial guesses u_D^0 for the solution of the Dirichlet side and u_N^0 for the solution of the Neumann side. We then use the value of u_N^0 at the left boundary of Ω_N as a Dirichlet boundary condition for the computation of u_D^1 on Ω_D. Afterwards we use the slope of u_D^1 at the right boundary of Ω_D as a Neumann boundary condition to compute u_N^1 on Ω_N which finishes one iteration. We continue until iteration k where the solutions u_D^k and u_N^k agree at the interface. This procedure is called Dirichlet-Neumann coupling [8].

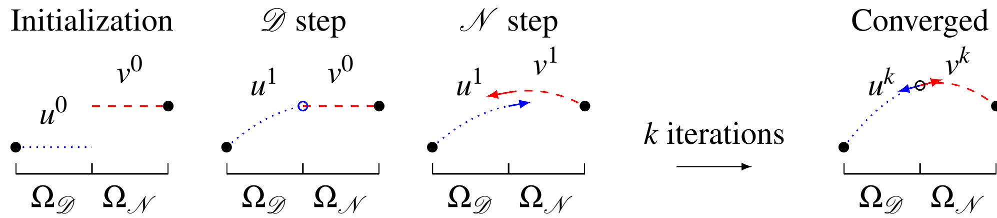

Figure 2.2: Illustration of the Dirichlet-Neumann coupling. u and v refer to u_D and u_N, respectively. Picture created by Benjamin Rodenberg.

3 Software components

In this chapter I present the software components used for this thesis. In section 3.1 I cover the coupling library preCICE and give an overview over its core features. In section 3.2 I go over the Finite Element Method library FEniCSx and give an example of how to use it. Lastly, in section 3.3, I show the FEniCS-preCICE adapter for FEniCS, the predecessor of FEniCSx, which serves as a template for the implementation covered in chapter 4.

3.1 preCICE

preCICE is a multi-physics coupling library that allows single physics solvers to be coupled to perform multi-physics simulations. The individual solvers, also called participants, are viewed as "black-boxes", which means that preCICE does not need to know their internal structure. preCICE receives their output, modifies it in a mathematically sensible way and sends it to another solver. This approach makes it easy to exchange solvers for testing purposes. It also simplifies adding new coupling components by only requiring few changes to them. The repository can be found at[1]. In the following we will look at some of the functionalities offered by preCICE; this and more information can be found in [5].

3.1.1 Uni-directional or bi-directional coupling

preCICE differentiates between two general coupling types, namely uni-directional coupling and bi-directional coupling. Uni-directional coupling happens when one participant depends on data received from the other but not the other way around. This happens for example when coupling an acoustic far field with a flow simulation where the former receives acoustic perturbations at the coupling interface but no acoustic waves travel to the latter. In bi-directional coupling both participants are dependant on data from the other, for example the two domains of the partitioned heat equation.

[1]https://github.com/precice
[2]http://precice.org/docs.html

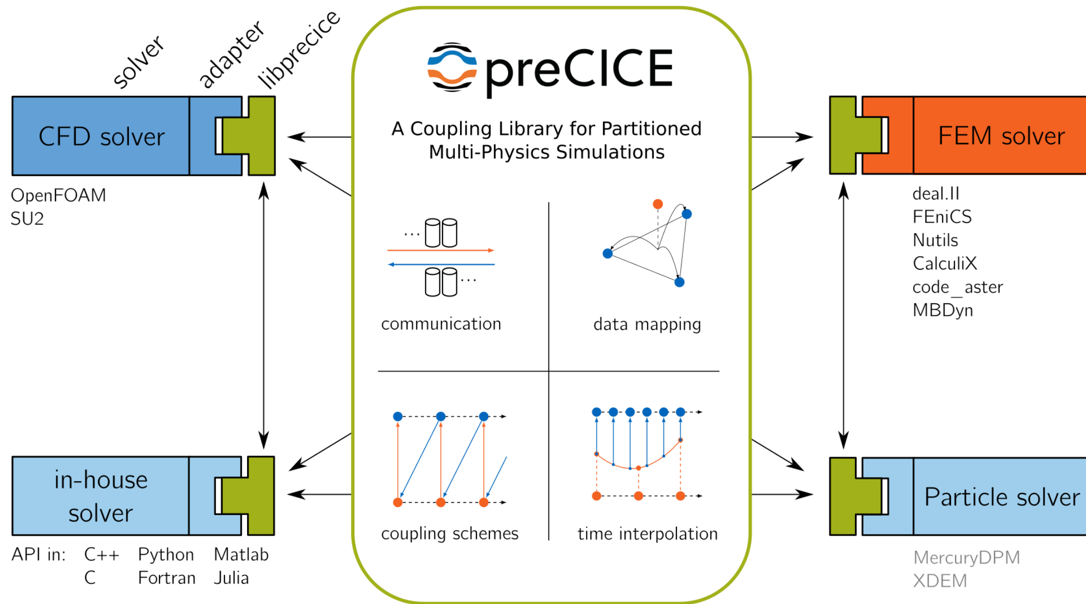

Figure 3.1: General structure of a coupled multi-physics simulation using preCICE. Taken from [2].

3.1.2 Coupling schemes

preCICE offers several options of how to perform the coupling. Which schemes are more suitable than others is dependant on the type of simulation to be performed.

Explicit or implicit coupling

When using explicit coupling each participant is executed once per time step, while implicit coupling lets every participant execute several iterations per time step until all computed values fulfill the coupling conditions.

Serial or parallel coupling

Serial coupling refers to the execution of each participant one after another while parallel coupling allows multiple participants to be executed at the same time.

3.1.3 Data mapping

preCICE manages the data exchange between the two participants by receiving data from the nodes that lie on the coupling interface of one participant and sending it to

the nodes on the coupling interface of the other. When both participants have matching meshes, their nodes on the coupling interface also match and the data can easily be transferred between matching nodes. However, this is often not the case, at which point preCICE uses one of the many available mapping methods in order to ensure every node on the coupling domain still has a partner node it can send data to or receive from.

3.1.4 Python bindings

Our implementation is done in Python. Because preCICE is written in C++, we additionally need to install the Python bindings provided by preCICE in order to translate between the two languages[3].

3.2 FEniCSx

FEniCSx is an open-source computing platform for solving partial differential equations (PDEs) by using the Finite Element Method (FEM)[9][10]. The main force behind FEniCSx is its computational environment DOLFINx, which offers high level interfaces in Python and C++. Another important component is the Unified Form Language (UFL) library which offers an easy to use interface for expressing finite elements and weak forms of PDEs[11]. Its documentation can be found here[4]. Additionally to the weak form, a user needs to specify a mesh, the function space for trial and test functions as well as the boundary conditions in order to run a simulation. In case of time dependent PDEs, FEniCSx also requires a timestepping scheme. In this section we will look at how to express examples of these components using FEniCSx. It roughly follows the FEniCSx tutorial[5] with some adjustments.

3.2.1 Creating a mesh

Initializing a simple mesh can be done in a single line of code. For example, a mesh over the unit square $[0,1] \times [0,1]$ consisting of 9×9 rectangles can be created with

```
mesh = create_unit_square(MPI.COMM_WORLD, 9, 9, CellType.quadrilateral)
```

`MPI.COMM_WORLD` refers to the communicator of the Message Passing Interface (MPI)[6] standard used for mediating between parallel processes. The communicator has to be

[3]https://github.com/precice/python-bindings
[4]https://fenics.readthedocs.io/projects/ufl/en/latest/index.html
[5]https://jsdokken.com/dolfinx-tutorial/index.html
[6]https://en.wikipedia.org/wiki/Message_Passing_Interface

provided even if the code is not being run in parallel. More information about the different cell types can be found in the UFL documentation.

3.2.2 Creating a function space and functions

Once the mesh is created, it can be used to create a function space, for example with

```
V = FunctionSpace(mesh, ("CG", 1))
u = TrialFunction(V)
v = TestFunction(V)
```

Here we create an object of the class `FunctionSpace` which is initialized with the mesh and a tuple consisting of the type of the finite element and its degree. In this particular example, `"CG"` refers to the standard Lagrange family of finite elements. Afterwards, we create u and v as objects of classes `TrialFunction` and `TestFunction`, respectively with both functions living in the function space V.

3.2.3 Defining boundary conditions

In order to define a boundary condition we need to define a Python class that represents the exact solution. We now take equation (2.16) as an example and define the class as

```
class exact_solution():
    def __init__(self, alpha, beta, t):
        self.alpha = alpha
        self.beta = beta
        self.t = t
    def __call__(self, x):
        return 1 + x[0]**2 + self.alpha * x[1]**2 + self.beta * self.t
u_exact = exact_solution(alpha, beta, t)
u_D = Function(V)
u_D.interpolate(u_exact)
```

Here, `x` is the vector of coordinates with `x[0]` being the x-coordinate and `x[1]` being the y-coordinate while `u_D` is a Function object that interpolates `u_exact`
We then define a function that takes a point of the domain and returns whether or not the point lies on the domain boundary. If, for example, the domain is the unit square we could define this function as

```
def boundary(x):
    is_left_wall = numpy.isclose(x[0], 0)
    is_right_wall = numpy.isclose(x[0], 1)
```

```
    is_bottom_wall = numpy.isclose(x[1], 0)
    is_top_wall = numpy.isclose(x[1], 1)
    is_any_wall = is_left_wall | is_right_wall | is_bottom_wall | is_top_wall
    return is_any_wall
```

3.2.4 Creating boundary conditions

After defining it, we can create the Dirichlet boundary condition with

```
dofs = locate_dofs_geometrical(V, boundary)
bc = dirichletbc(u_D, dofs)
```

The function `locate_dofs_geometrical` takes a function space and the function `boundary` as arguments and returns an array of degrees of freedoms that lie on the domain boundary. Finally, the function `dirichletbc` takes the function containing the exact solution and the array of degrees of freedom as arguments and returns the Dirichlet boundary condition.

3.2.5 Defining a variational problem

As an example we consider the terms a, L and f of the heat equation as seen in equations (2.11), (2.12) and (2.17), respectively. We can define these with

```
f = Constant(mesh, beta - 2 - 2 * alpha)
F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)
```

Here we again make use of some of the functionalities provided by the UFL library, `grad(argument)` computes the gradient of the argument and multiplying a term with `dx` represents the integration over the domain. We initialize f as an object of type `Constant` that takes the mesh and the value of the constant as arguments and then define `a` and `L` as the left hand side and right hand side of `F`, respectively.

3.2.6 Performing the timestepping

A simple timestepping scheme in FEniCSx could be implemented as

```
u = Function(V)
t = 0
for n in range(num_steps):
    # Update current time
    t += dt
```

```
u_D.t = t
# Solve variational problem
linear_problem = LinearProblem(a, L, bc)
u = linear_problem.solve()
# Update previous solution
u_n.interpolate(u)
```

The class `LinearProblem` receives the terms `a`, `L` as well as the boundary condition and instantiates a linear problem that gets solved in the next line and interpolated by Function object `u_n`.

3.3 FEniCS-preCICE adapter

Generally, preCICE does not have the same data representation as the solvers it couples, making it impossible to connect them directly. To circumvent this, preCICE offers so-called adapters which integrate preCICE with a specific solver[6][12]. The adapters are minimally-invasive which means that no preCICE code needs to be altered to make preCICE compatible with the adapters. One of these adapters is the FEniCS-preCICE adapter[13] for FEniCS[4], which is the predecessor of FEniCSx. This adapter allows preCICE to couple solvers that employ FEniCS. Since not all classes and methods from the FEniCS library are included in FEniCSx, we can not use the FEniCS-preCICE adapter to couple preCICE with solvers using FEniCSx. However, we can use it as a template for an adapter for FEniCSx solvers since it has a fitting API and many contents of FEniCS are still present in FEniCSx.

Additionally, we can make use of the tutorial cases offered by preCICE[7]. These are generally small setups which require no prior knowledge about the underlying solvers from the user and teach them how to perform partitioned simulations with preCICE. They also exemplify the black-box philosophy of preCICE since many tutorials allow the users to choose a combination of available solvers at will, for example FEniCS. Similar with the FEniCS-preCICE adapter, we can use the tutorials working with FEniCS as a template when making them compatible with FEniCSx. This would allow us to test the adapter for FEniCSx and preCICE as well.

[7]https://github.com/precice/tutorials

4 Implementation

In this chapter, I cover the FEniCSx-preCICE adapter and its implementation. In section 4.1 I explain the purpose of the adapter. Afterwards I describe the state of the adapter before this thesis in section 4.2. I then justify the changes made between the FEniCSx-preCICE and the FEniCS-preCICE adapter in section 4.3. In section 4.4, I implement a solver for the partitioned heat equation which uses the FEniCSx-precice adapter 4.4. I finish the chapter by explaining the solver's error computation in section 4.5.

4.1 FEniCSx-preCICE-adapter

On their own, preCICE and FEniCSx can not communicate with each other because of their different internal data representation. In order to solve this issue we need an adapter to link them together, called the FEniCSx-preCICE-adapter. The adapter obtains data from preCICE at the nodes of the coupling interface and translates it into FEniCSx code and vice versa.

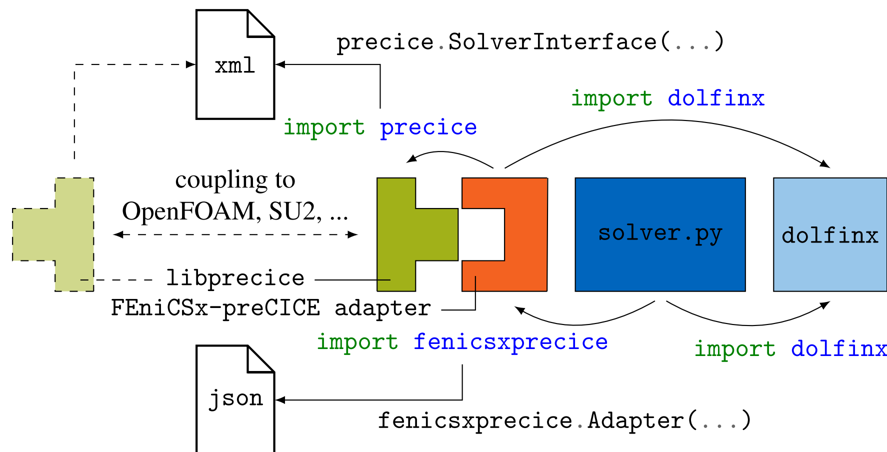

Figure 4.1: Overview of the software architecture. From left to right: preCICE, FEniCSx–preCICE adapter, application code solver.py, and DOLFINx. Adapted from [13].

4.2 Previous state of the adapter

Initially, the FEniCSx-preCICE adapter[1] was created as a fork of the FEniCS-preCICE adapter[2]. While some modifications have already been made by then, such as replacing some calls to the FEniCS API with calls to the FEniCSx API, the FEniCSx-preCICE adapter in its entirety was not in a usable state[3].

4.3 Changes to the adapter

Generally, the FEniCSx-preCICE adapter follows the same structure as the FEniCS-preCICE adapter. That is, methods that fulfill the same purpose also have the same name, with all occurrences of "fenics" being replaced by "fenicsx". Readers are encouraged to first have a look at the FEniCS-preCICE adapter paper [13] in order to get an overview of its core methods. In the following we will focus on those methods whose implementation had to be adjusted in order to work with FEniCSx, disregarding very minor changes like slightly different syntax, and on methods which have been removed altogether.

4.3.1 Changing FEniCSx-to-preCICE communication

For the FEniCSx-precice adapter, adjustments had to be done in the procedure how the solver using FEniCSx can send its data to preCICE. The method `write_data` takes the `Function` object `write_function` and calls the method `convert_fenics_to_precice` in turn, which takes `write_function` as well as an array of local indices of vertices on the coupling interface called `local_ids` as arguments. `convert_fenics_to_precice` then samples the values of `write_function` on all vertices of the mesh and returns only the values of the vertices on the coupling interface. Afterwards, `write_data` writes the result to preCICE. The sketched code of `convert_fenics_to_precice` in the FEniCS-preCICE adapter is given by:

```
precice_data = []
    sampled_data = write_function.compute_vertex_values()
    if len(local_ids):
        for lid in local_ids:
            precice_data.append(sampled_data[lid])
    return numpy.array(precice_data)
```

[1] https://github.com/precice/fenicsx-adapter
[2] https://github.com/precice/fenics-adapter
[3] https://github.com/precice/fenicsx-adapter/pull/15

The method compute_vertex_values from the FEniCS library computes and returns the values of the caller function at all mesh vertices. This method, however, has been removed in FEniCSx, making a workaround necessary:

```
x_mesh = write_function.function_space.mesh.geometry.x
x_dofs = write_function.function_space.tabulate_dof_coordinates()
if(self._mask == None):
    self._mask = [] # where dof coordinate == mesh coordinate
    for i in range(x_dofs.shape[0]):
        for j in range(x_mesh.shape[0]):
            if numpy.allclose(x_dofs[i, :], x_mesh[j, :], 1e-15):
                self._mask.append(i)
                break
```

Depending on the type of finite elements, there may be more degrees of freedom than vertices meaning some degrees of freedom do not lie on vertices. We therefore have to filter only those degrees of freedom which align with a vertex since we only evaluate data on the vertices. We accomplish this with a nested for loop over the array of degree-of-freedom coordinates we obtain from the method tabulate.dof_coordinates and the array of vertex coordinates we get from the member mesh.geometry.x. The mask consists of the indices of the degrees of freedom that lie on a vertex. It then gets passed to convert_fenicsx_to_precice as an additional argument. The sketched version of the method in FEniCSx looks like this:

```
precice_data = []
sampled_data = write_function.x.array[mask]
if len(local_ids):
    for lid in local_ids:
        precice_data.append(sampled_data[lid])
return numpy.array(precice_data)
```

The member x.array returns a vector containing the value of all degrees of freedom. We then apply the mask to filter only the sampled values from the vertices. Afterwards, the procedure is the same as in the FEniCS-preCICE adapter.

4.3.2 Adjusting the adapter's scope

Since the adapter was not usable before I started working on it, we decided that getting it to work at all is our main priority. Therefore, we opted for a minimal working version which involved removing features present in the FEniCS-preCICE adapter that are not necessary for the adapter to function in general. It would have been out of scope for

this thesis to get every additional functionality to work. Our goal is to re-implement them in the future, however.

Removal of support of vector-valued functions

As a result of the changes made to the way we sample the data to be written to preCICE in the `convert_fenicsx_to_precice` method, only scalar values can be sampled. Therefore, the possibility to use functions that return vector values has been taken out. This is acceptable because the solver for the partitioned heat equation does not use them. As mentioned in section 2.4.3, while the heat flux is a two-dimensional vector, we can discard its y-component because of the vertical coupling boundary, resulting in a scalar value. For different setups where the coupling boundary is not vertical, this would not be possible and vector-valued functions would have to be supported again, however.

Removal of support of point sources

The FEniCS library implements a class called `PointSource`[4] which represents a heat source with no spatial extension. The FEniCS-preCICE adapter uses this for example in the method `get_point_sources` which can be used to create a list of point sources using the coupling data as input. In FEniCSx however, point sources have not been implemented yet[5]. Therefore, all methods related to them had to be removed.

Removal of support of parallelism

The FEniCS-preCICE adapter uses imports from the mpi4py package[6] which allow the handling of inter-process communication by letting each of multiple processes of a solver solve a different part of the mesh. It uses these, for example, in its `communicate_shared_vertices` method which lets solver processes send vertex data to other solver processes with sharing vertices. For the FEniCSx-preCICE adapter, this and other methods regarding parallelism have been removed, meaning a solver only has one process.

[4] https://fenicsproject.org/olddocs/dolfin/1.5.0/python/programmers-reference/cpp/fem/PointSource.html
[5] https://fenicsproject.discourse.group/t/pointsource-in-dolfinx/8337
[6] https://pypi.org/project/mpi4py/

4.4 Solver for the partitioned heat equation

For testing, but also as a starting point for new users I developed a solver for the partitioned heat equation that communicates with preCICE using the FEniCSx-preCICE adapter[7]. This solver is based on the FEniCS solver of the partitioned heat equation tutorial case from preCICE [8].

In order to run a simulation with preCICE, an xml configuration file which specifies the solver participants and the coupling schemes, among other settings, has to be provided. An overview of how this file is arranged can be found on the preCICE website[9]. The simulation of the partitioned heat equation consists of two separate solver instances, solving the Dirichlet and Neumann side, respectively, that exchange data on the coupling interface via bidirectional serial-implicit coupling. Since both solvers run the same code, the parts that are specific to one participant are not executed by the other.

The meshes are setup the same way as described in section 2.4.3. Subsequently, the parameters $\alpha = 3$ and $\beta = 1.3$ as well as the function spaces are initialized.

Afterwards, the Dirichlet and Neumann boundary conditions are defined and the adapter is initialized with the coupling boundary, the function space it reads data from as well as the function object it writes data to. In the case of the Dirichlet solver, the read function space is the function space of the temperature and the write object is the function object of the Neumann boundary condition, while for the Neumann solver the read function space is the function space of the heat flux and the write function object is the function object of the Dirichlet boundary condition.

Now, the adapter handles the implicit coupling by setting a checkpoint by storing the current solution and time step for both solvers at the beginning of each time step. First, the Dirichlet solver solves the problem on its side and sends the heat flux at the coupling interface to the Neumann solver. In turn, the Neumann solver solves the problem on its side and sends the temperature at the coupling interface to the Dirichlet solver

```
if problem is ProblemType.DIRICHLET:
    # reads temperature and writes flux on boundary to Neumann problem
    flux = determine_gradient(V_g, u_np1)
    flux_x = Function(W)
    flux_x.interpolate(flux.sub(0))
    precice.write_data(flux_x)
```

[7]https://github.com/precice/tutorials/pull/317
[8]https://github.com/precice/tutorials/tree/master/partitioned-heat-conduction/fenics
[9]https://precice.org/configuration-overview.html

```
elif problem is ProblemType.NEUMANN:
    # reads flux and writes temperature on boundary to Dirichlet problem
    precice.write_data(u_np1)
```

`determine_gradient` is a method that computes the heat flux according to section 2.4.2, `V_g` a vector function space and `W` a one-dimensional subspace of `V_g`. The function spaces have to be setup this way because the flux itself is a vector that gets reduced to its x-component by using the method `flux.sub(0)`.

If the data at the coupling interface did not converge yet, the adapter rolls back to the checkpoint. Once convergence is reached, the solutions are updated and the solvers advance to the next time step. The implicit coupling process repeats until the last time step is finished.

4.5 Changes to the error computation

The error computation, that allows us to check if the numerical solution is sufficiently close to the analytical one, is implemented in the FEniCS solver as follows:

```
# compute pointwise L2 error
error_normalized = (u_ref - u_approx) / u_ref
# project onto function space
error_pointwise = project(abs(error_normalized), V)
# determine L2 norm to estimate total error
error_total = sqrt(assemble(inner(error_pointwise, error_pointwise) * dx))
assert (error_total < 10 ** -4)
return error_total
```

Here, `u_approx` is the numerical solution function, `u_ref` the analytical solution function and `V` the function space they live in. The relative error between the functions is computed and then projected onto `V` using the FEniCS method `project` resulting in the error function `error_pointwise`. Lastly, the L2 norm of the error function is computed which yields the total error. The method `project` is not part of FEniCSx, however, resulting in the following changes for the FEniCSx solver[10]:

```
mesh = u_ref.function_space.mesh
# compute total L2 error between reference and calculated solution
error_pointwise = form(((u_approx - u_ref) / u_ref) ** 2 * dx)
error_total = sqrt(mesh.comm.allreduce(assemble_scalar(error_pointwise), MPI.SUM))
assert (error_total < 10 ** -4)
return error_total
```

[10] https://github.com/precice/fenicsx-adapter/pull/14

Here, `error_pointwise` is a function that takes the relative difference of the reference and approximate solutions in the L2 norm. The `error_total` is computed by assembling the system which resolves `error_pointwise` using the method `assemble_scalar` and taking the square root over the sum of all values. This sum is computed with the method `allreduce` which takes the summation operator `MPI.SUM` as the second argument and executes it on its first argument.

5 Testing and Evaluation

In this chapter I test the preCICE-FEniCSx adapter and evaluate its results. In section 5.1 I execute the preCICE tutorial case using the adapter with several different configurations and address their computation time. Afterwards, I compare the adapter's performance with the FEniCS-preCICE adapter in section 5.2. Furthermore, I assess the accuracy of the results in section 5.3. Lastly, I discuss the findings of the chapter in section 5.4.

5.1 Performance

To test our implementation we run the tutorial case for the partitioned heat equation where we consider the same domain dimensions as described in subsection 2.4.3 with varying cell counts in the meshes. In all test cases we compute 10 time steps. In the following we refer to the number of cells of a mesh in x-direction and y-direction as `nx` and `ny`, respectively.

5.1.1 Increasing ny

For this test case, we set `nx` = 9 for both domains and simultaneously increase `ny` for both domains. In figure 5.1 we observe a big spike between the times for `ny` = 36 and `ny` = 72 where the total time spent by the Dirichlet and Neumann participant rises by over 500% for both. The simulation loop time after the initialization rises by over 1600% and over 3700% for Dirichlet and Neumann participant respectively. It makes up between 20% and 52% of the total time of the Dirichlet participant and between 7% and 42% for the Neumann participant.

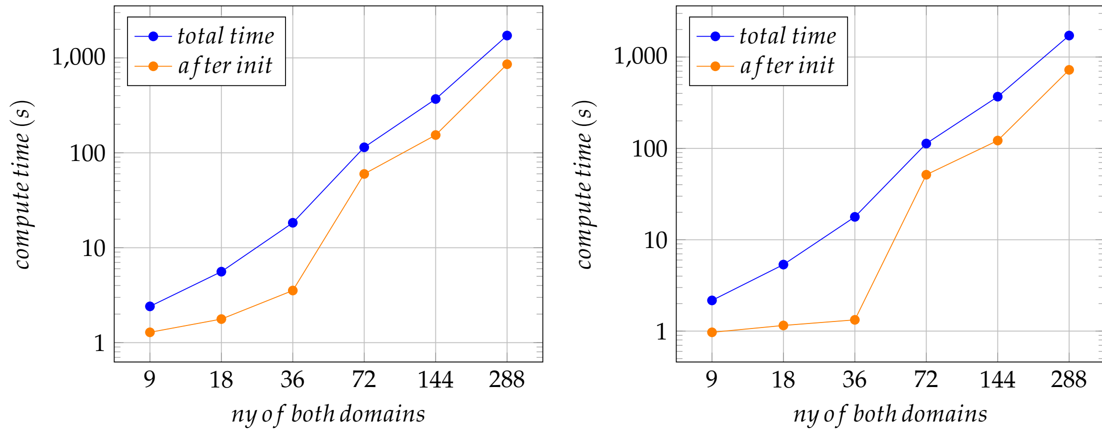

Figure 5.1: Runtime of the tutorial with `nx` = 9 and simultaneously increasing `ny` for both domains: The Dirichlet participant is on the left side, the Neumann participant on the right.

5.1.2 Increasing nx

Now, we reverse the setup of subsection 5.1.1 and set `ny` = 9 for both domains and simultaneously increase `nx` for both domains. In figure 5.2 we observe that both the total time taken by the simulation as well as the time spent by the simulation loop is lower than seen in subsection 5.1.1. Also, we do not observe any sudden spikes in the computational time. The simulation loop now makes up between 13% and 50% for the Dirichlet participant. For the Neumann participant it makes up a marginal amount for high `nx` with only 0.3% for `nx` = 288.

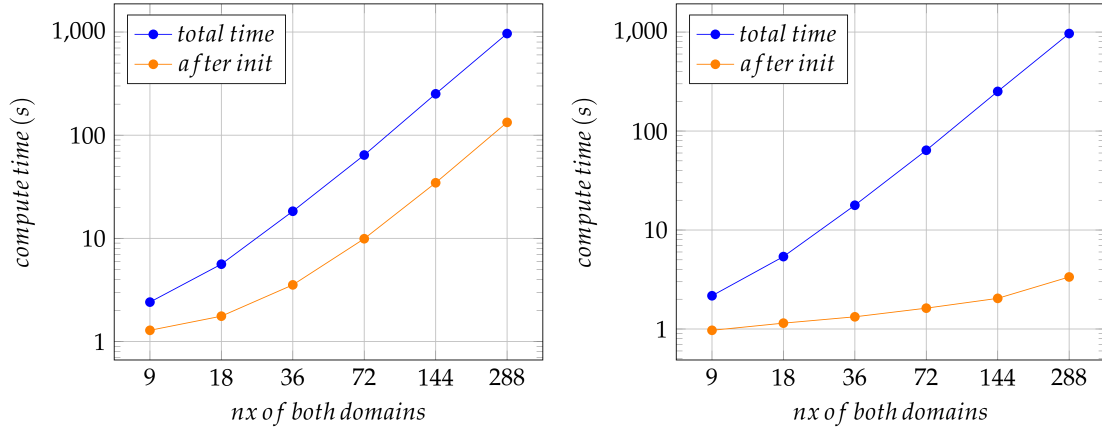

Figure 5.2: Runtime of the tutorial with `ny` = 9 and simultaneously increasing `nx` for both domains: The Dirichlet participant is on the left side, the Neumann participant on the right.

5.1.3 Coupling non-matching meshes

In this test case we choose different ny for Ω_D and Ω_N which leads to a differing amount of nodes on the coupling domain at which point preCICE employs data mapping. As we can see in figure 5.3 the total computation time stays below the times seen in subsections 5.1.1 and 5.1.2 due to the lower total amount of nodes. Again, there is a big spike between the times for `ny` = 36 and `ny` = 72. Also, for `ny` >= 72 the simulation loop now takes up most of the time of the simulation with over 97% for the Dirichlet participant and over 72% for the Neumann participant.

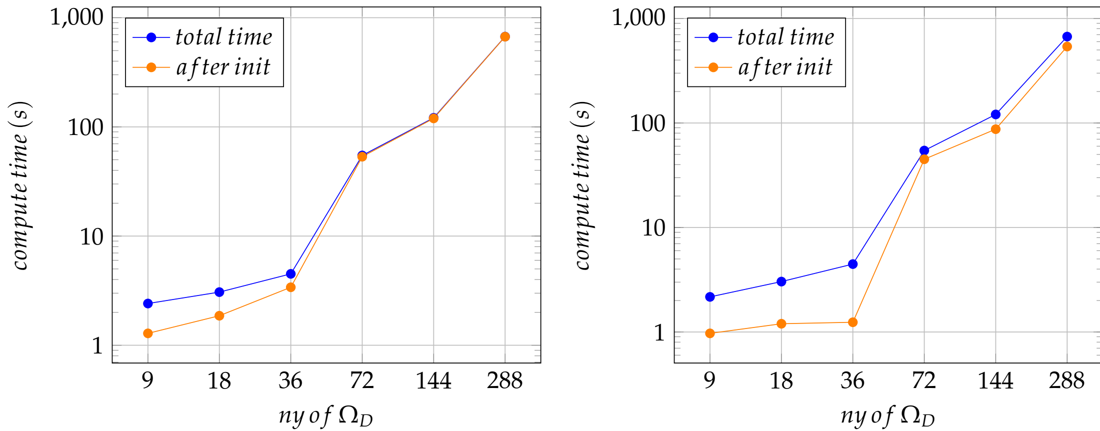

Figure 5.3: Runtime of the tutorial with `nx` = 9 for both domains, `ny` = 9 for Ω_N and increasing `ny` for Ω_D: The Dirichlet participant is on the left side, the Neumann participant on the right.

5.2 Comparison with the FEniCS-preCICE adapter

In order to see the effects of the changes made between the FEniCS-preCICE adapter and the FEniCSx-preCICE adapter, we run the FEniCS tutorial case of the partitioned heat equation with the same configurations as in subsection 5.1.1. The FEniCS tutorial case was performed using the preCICE virtual machine[1] which, among others, has the tutorial case for the partitioned heat equation installed. Figure 5.4 shows, that for `ny` up until 72, the FEniCS case takes less than 10 seconds total. For `ny` = 288, the FEniCS case finishes in slightly above 40 seconds, which is roughly 2.3% of the time needed for the FEniCSx case.

[1]http://precice.org/installation-vm.html

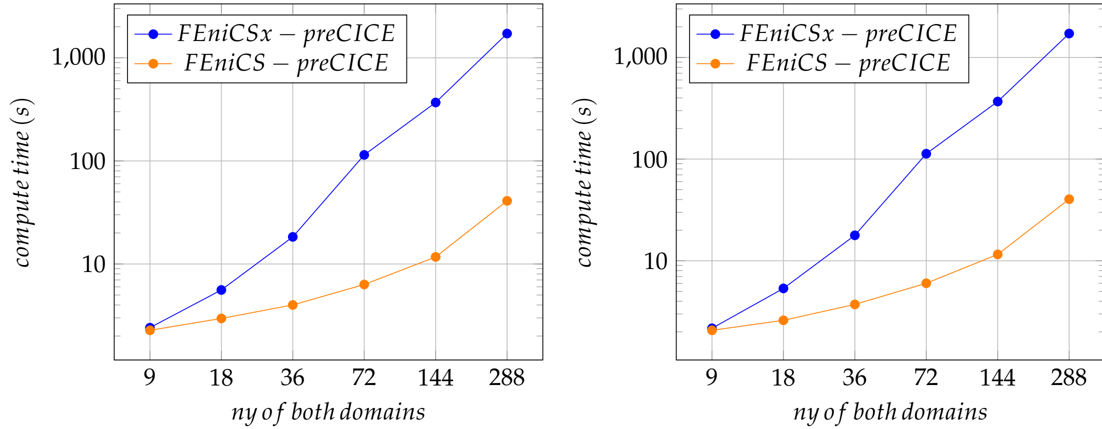

Figure 5.4: The setup is identical to the one presented in subsection (5.1.1). Again, the Dirichlet participant is on the left, the Neumann participant on the right.

5.3 Accuracy

All tested configurations shown in sections 5.1 and 5.2 produce the same output whose initial and final configuration can be seen in figures 5.5 and 5.6, respectively. The output also coincides with the output produced by the analytical solution, showing that the FEniCSx-preCICE adapter works correctly. This also means that we do not lose accuracy compared to the FEniCS-preCICE adapter. Furthermore, we can observe that the data is continuous even at the coupling interface, showing the successful usage of the Dirichlet-Neumann coupling presented in section 2.4.3. This is also the case for the configuration with non-matching meshes as seen in subsection 5.1.3 which demonstrates the correctness of the data mapping employed by preCICE.

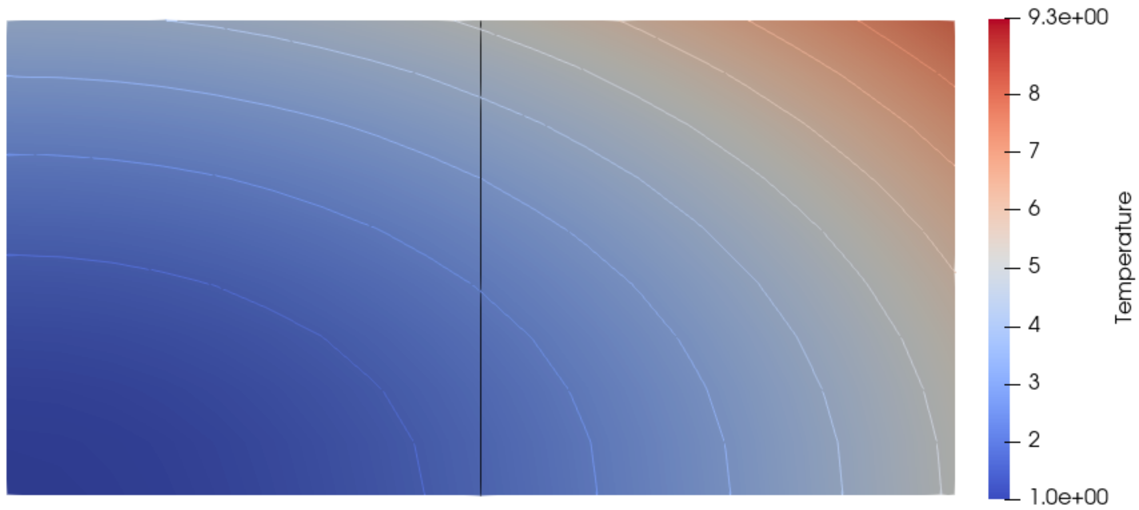

Figure 5.5: The initial heat distribution across the domain; the black line indicates the coupling interface

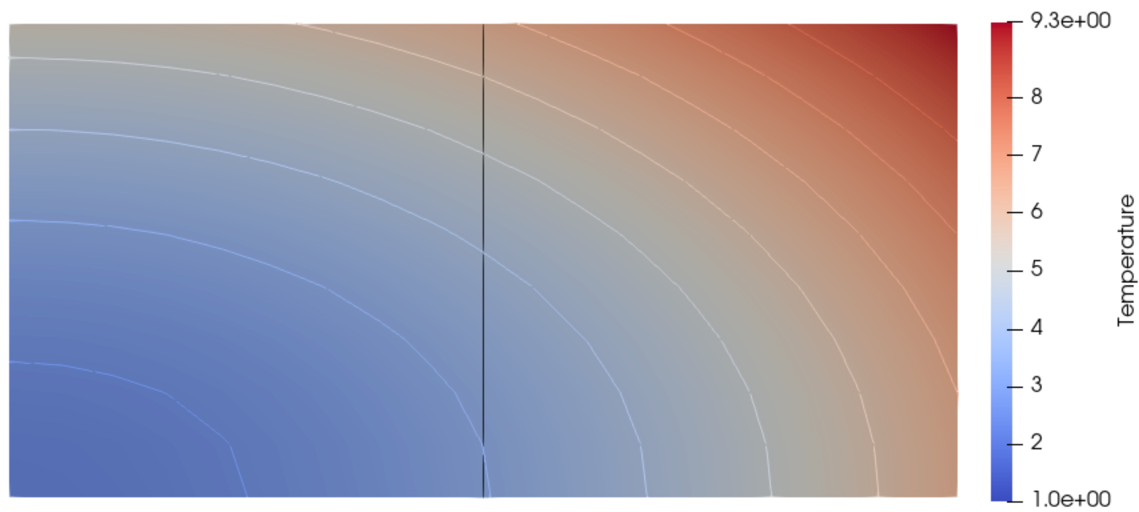

Figure 5.6: The heat distribution across the domain after the coupling has finished; the black line indicates the coupling interface

5.4 Discussion

The simulations using the FEniCSx-preCICE adapter produce accurate results, meaning we fulfilled our goal of not losing accuracy compared to the FEniCS-preCICE adapter. However, their computational time is significantly increased compared to simulations using the FEniCS-preCICE adapter for bigger meshes. I suspect there are two main reasons for this. One seems to be the mask computation that had to be introduced as described in subsection 4.3.1 and its nested for loop over the degrees of freedom and the mesh vertices during the initialization phase. This would result in a considerable overhead that is not present in the FEniCS-preCICE adapter. I assume the other reason to be the simulation time loop itself which generally also has a significantly higher computational cost compared to the time loop in the FEniCS-preCICE simulations. One reason for this could be the circumstance that not every functionality offered by FEniCS is implemented in FEniCSx yet, making workarounds necessary that negatively impact performance. In order to make more accurate assumptions further investigations of this topic are necessary, however. One exception to this is that the Neumann solver still scales well for meshes with few cells in y-direction and a high number of cells in x-direction as seen in subsection 5.1.2. This could be explained by the fact that, in contrast to the Dirichlet solver, the Neumann solver does not have to compute the heat flux. This is, in fact, the reason why the computation loop of the Neumann solver always takes less time than the one of the Dirchlet solver. Also, since in this scenario there are few vertices at the coupling interface and therefore few data values that are exchanged via preCICE each time step, this suggests that the data exchange with preCICE has a rather high computational cost.

6 Conclusion

To conclude this thesis, I summarize its content and give an outlook on future work. In section 6.1 I recapitulate the main points of the thesis, in particular the FEniCSx-preCICE adapter and its evaluation. In section 6.2 I discuss parts of the adapter that have room for improvement and look at other preCICE tutorial cases that could be made compatible with FEniCSx.

6.1 Summary

In multi-physics simulations, partitioned approaches, which simulate the considered system by dividing it into several subsystems that are subsequently solved individually by independent solvers, are useful because changes to the system only result in changes to the affected subsystems, leaving the others untouched. For this, the independent solvers need to be coupled together, which can be achieved by tools such as preCICE, a coupling library which treats individual solvers, such as the Finite Element Method (FEM) library FEniCSx, as black boxes.

In this thesis, I turn the FEniCSx-preCICE adapter, which is used to integrate preCICE with FEniCSx in a minimally-invasive way, from a draft into a functioning software. In chapter 2, I explain the mathematical background needed for understanding the Finite Element Method such as boundary conditions, address the heat equation in particular and present a partitioned setup for it. In chapter 3, I give an overview over the features of preCICE in section 3.1 and show how to solve partial differential equations by employing the Finite Element Method with FEniCSx in section 3.2. In section 3.3, I consider the FEniCS-preCICE adapter for FEniCS, the predecessor of FEniCSx, as the basis for the FEniCSx-preCICE adapter. In chapter 4, I focus on the implementation details of the FEniCSx-preCICE adapter. I describe major changes compared to the FEniCS-preCICE adapter in sections 4.3 and 4.4. Furthermore I take an existing preCICE tutorial case consisting of a partitioned setup for the heat equation and make it compatible with FEniCSx in section 4.5. The chapter shows the minimally-invasive nature of the adapter since no preCICE code had to be altered. Afterwards, I use it in chapter 5 to test the adapter's performance. Section 5.3 shows that the results produced by the adapter are as accurate as the analytical solution as well as the FEniCS-preCICE adapter. I was therefore successful at developing a working version of

the adapter as well as making it usable with a FEniCSx solver. The results of sections 5.1 and 5.2 show that there is still room for improvement regarding their performance for big meshes, however. I suspect that the workarounds explained in section 4.3, which had to be employed due to functionalities from FEniCS that have been cut from FEniCSx, are at least partly at fault for this.

6.2 Future work

The first step of improving the adapter is to further investigate the significantly higher computational cost compared to the FEniCS-preCICE adapter as seen in section 5.2. Also, since the main goal of developing the FEniCSx-preCICE adapter was to create a minimal functioning version that is usable for the partitioned-heat-equation tutorial case, there are many features that are yet to be implemented as explained in subsection 4.3.2. Right now, the adapter only works in serial and does not support parallelization. Similarly, it is currently not possible to use vector-valued functions. The tutorial does not use them and we could therefore afford to omit them. Additionally, point sources can not be used with the adapter because there is currently no implementation for them in FEniCSx. They are necessary for the simulation of fluid-structure interaction and are therefore of great importance. All listed functionalities are supported by the FEniCS-preCICE adapter, so they should be rather straightforward to bring to the FEniCSx-preCICE adapter once all necessary functionalities of FEniCS are brought to FEniCSx. Furthermore, there are other preCICE tutorial cases such as fluid-structure interaction[1] and conjugate heat transfer[2] that are compatible with the FEniCS and are therefore suitable candidates to be implemented in the future once all features supported by FEniCS return to FEniCSx. Since the FEniCSx library is still being expanded, future added functionalities could improve the adapter as well.

[1]http://precice.org/tutorials-perpendicular-flap.html
[2]http://precice.org/tutorials-flow-over-heated-plate.html

List of Figures

2.1 Illustration of the partitioned domain. Taken from [7] 7
2.2 Illustration of the Dirichlet-Neumann coupling. u and v refer to u_D and u_N, respectively. Picture created by Benjamin Rodenberg. 8

3.1 preCICE-picture 10

4.1 adapter-picture 15

5.1 Runtime of the tutorial with `nx` = 9 and simultaneously increasing `ny` for both domains: The Dirichlet participant is on the left side, the Neumann participant on the right. 23
5.2 Runtime of the tutorial with `ny` = 9 and simultaneously increasing `nx` for both domains: The Dirichlet participant is on the left side, the Neumann participant on the right. 24
5.3 Runtime of the tutorial with `nx` = 9 for both domains, `ny` = 9 for Ω_N and increasing `ny` for Ω_D: The Dirichlet participant is on the left side, the Neumann participant on the right. 25
5.4 The setup is identical to the one presented in subsection (5.1.1). Again, the Dirichlet participant is on the left, the Neumann participant on the right. 26
5.5 The initial heat distribution across the domain; the black line indicates the coupling interface 27
5.6 The heat distribution across the domain after the coupling has finished; the black line indicates the coupling interface 27

List of Tables

Bibliography

[1] D. E. Keyes, L. C. McInnes, and C. Woodward. “Multiphysics simulations: Challenges and opportunities.” In: *The International Journal of High Performance Computing Applications* 27 (2013), pp. 4–83. DOI: https://doi.org/10.1177/1094342012468181.

[2] B. Uekermann. “Partitioned Fluid-Structure Interaction on Massively Parallel Systems.” available at https://mediatum.ub.tum.de/doc/1320661/document.pdf. PhD thesis. Technische Universität München, 2016.

[3] A. W. Bergersen, A. Slyngstad, S. Gjertsen, et al. “turtleFSI: A Robust and Monolithic FEniCS-based Fluid-Structure Interaction Solver.” In: *Journal of Open Source Software* 5.50 (2020), p. 2089. DOI: 10.21105/joss.02089.

[4] H. P. Langtangen and A. Logg. *Solving PDEs in Python*. Springer, 2017. ISBN: 978-3-319-52461-0. DOI: 10.1007/978-3-319-52462-7.

[5] G. Chourdakis, K. Davis, B. Rodenberg, et al. *preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]*. Open Res Europe 2022, 2:51. DOI: https://doi.org/10.12688/openreseurope.14445.2.

[6] B. Uekermann, H.-J. Bungartz, L. Yau, et al. “Official preCICE Adapters for Standard Open-Source Solvers.” In: *7th GACM Colloquium on Computational Mechanics for Young Scientists from Academia and Industry* (Stuttgart, Germany). Ed. by M. v. Scheven, M.-A. Keip, and N. Karajan. Institute for Structural Mechanics, University of Stuttgart, Oct. 2017. DOI: http://dx.doi.org/10.18419/opus-9334.

[7] B. Rüth, B. Uekermann, M. Mehl, et al. “Quasi-Newton waveform iteration for partitioned surface-coupled multiphysics applications.” In: *International Journal for Numerical Methods in Engineering* 122 (2020), pp. 5236–5257. DOI: https://doi.org/10.1002/nme.6443.

[8] A. Monge and P. Birken. “A time-adaptive Dirichlet-Neumann waveform relaxation method for coupled heterogeneous heat equations.” In: *Proceedings in Applied Mathematics and Mechanics* 19 (2019). DOI: https://doi.org/10.1002/pamm.201900206.

[9] J. McDonagh, N. Palumbo, N. Cherukunnath, et al. "Modelling a permanent magnet synchronous motor in FEniCSx for parallel high-performance simulations." In: *Finite Elements in Analysis and Design* 204 (2022). DOI: `https://doi.org/10.1016/j.finel.2022.103755`.

[10] M. Alnæs, J. Blechta, J. Hake, et al. "The FEniCS Project Version 1.5." In: *Archive of Numerical Software* 3 (2015). DOI: `https://doi.org/10.11588/ans.2015.100.20553`.

[11] M. Alnæs, A. Logg, and K. B. Ølgaard. "Unified form language: A domain-specific language for weak formulations of partial differential equations." In: *ACM Transactions on Mathematical Software* 40 (2014). DOI: `https://doi.org/10.1145/2566630`.

[12] G. Chourdakis, D. Schneider, and B. Uekermann. "OpenFOAM–preCICE: Coupling OpenFOAM with External Solvers for Multi-Physics Simulations." In: *OpenFOAM®Journal* 3 (2023), pp. 1–25. DOI: `https://doi.org/10.51560/ofj.v3.88`.

[13] B. Rodenberg, I. Desai, R. Hertrich, et al. "FEniCS–preCICE: Coupling FEniCS to other simulation software." In: *SoftwareX* 16 (2021), p. 100807. ISSN: 2352-7110. DOI: `https://doi.org/10.1016/j.softx.2021.100807`.