Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Coupling Julia-based Simulations via preCICE

Pavel Kharitenko

Course of Study:	Softwaretechnik
Examiner:	Jun.-Prof. Dr. rer. nat. Benjamin Uekermann
Supervisor:	Ishaan Desai, M.Sc.
Commenced:	22. April 2021
Completed:	22. October 2021

Abstract

The coupling library preCICE allows to couple single-physics solvers to partitioned multi-physics simulations in a black-box fashion. preCICE is a C++ library, but it offers language bindings to access the preCICE API from solvers written in other languages, such as C, Python, Fortran and MATLAB. The Julia Programming Language designed for numerical computing is a strong candidate to be supported by preCICE. While Julia provides a wide set of tools for interfacing with other languages, including C++, porting a library such as preCICE that is made for High Performance Computing and runs on a huge number of processes, requires little to no compromises. Multiple ways of wrapping a C/C++ library are presented and implemented. In addition come Julia's own features, for example the Distributed base library, that deviate from classic standards of known scientific languages. To test the bindings, two dummy solvers are coupled and presented in an example setup, with an outlook on further development.

Kurzfassung

Die Kopplungsbibliothek preCICE ermöglicht es, Single-Physik Löser zu partitionierten Multi-Physik Simulationen in einer Black-Box-Manier zu koppeln. Zwar ist preCICE eine C++-Bibliothek, bietet sie Bindings für den Zugriff auf ihre API von Solvern, die in einer anderen Sprachen wie C, Python, Fortran und MATLAB geschrieben wurden, an. Die Programmiersprache Julia, die für die wissenschaftliche Programmierung entwickelt wurde, ist ein starker Kandidat für die Unterstützung von preCICE. Während Julia eine breite Auswahl für schnittstellen mit anderen Sprachen, einschließlich C++, bereitstellt, erfordert die Portierung einer Bibliothek wie preCICE, die für High Performance Computing entwickelt wurde und auf einer Vielzahl von Prozessesoren ausgeführt werdem kann, wenig bis gar keine Kompromisse. Es werden mehrere Möglichkeiten zum Portieren einer C/C ++-Bibliothek vorgestellt und implementiert. Hinzu kommen Julias eigene Features, zum Beispiel die Bibliothek Distributed, die von klassischen Standards bekannter Sprachen abweichen. Um die Bindings zu testen, werden zwei Dummy-Solver gekoppelt und in einem Beispielaufbau mit einem Ausblick auf die weitere Entwicklung vorgestellt.

Contents

List of Figures

1. Introduction

Multi-physics simulations are used to compute physical processes, such as fluid-structure interactions (e.g., an aircraft in a wind channel), where different physical phenomena have an influence on each other. Here, the solution of one domain is depending on the one they share a boundary with, by exchanging values like forces, heat, temperature, etc.

One way to formulate and solve these complex simulations is the monolithic approach: to create one physical domain, in which all interacting physics are computed and solved globally. There is also a partitioned approach: each physics is simulated by a single-physics solver specialized in their respective physical problem, and coupled by exposing their computed data to each other through a communication method during runtime. Both have their pros and cons, but the latter approach could provide reusability by using already established solvers that are devoted to simulating their part, and reducing the complexity of the physics simulation, since the domain is after all partitioned.

The open-source library preCICE[3] is exactly made for this kind of coupling. Two solvers can be coupled in a minimally invasive approach by making function calls to the library from their codes. While preCICE is a C++ library, it offers language bindings to access its API from solvers written in other languages, such as C, Python, FORTRAN and Matlab. The support for different languages widens the possibilities of composing a coupled simulation and enriches the selection of existing solvers for a particular domain.

The Julia Programming Language[1] is an open-source, dynamically typed programming language designed to be fast, targeting numeric and scientific computing. Its performance[2], which is reaching levels comparable to traditional statically-typed languages like C++ and FORTRAN, makes it a strong candidate to be supported by preCICE.

Developing a Julia binding would bring partitioned multi-physics simulations to the open-source community of Julia. That means, the various base and third-party Julia physics libraries could now be coupled to simulate multi-physics scenarios. But not only will Julia programmers be able to use preCICE among themselves. At the same

[1]https://julialang.org/
[2]https://julialang.org/benchmarks/

time, this is beneficial for the rest of the field of multi-physics problems, since preCICE enables to mix the different solvers for languages it supports. That would make solvers written with Julia's high performance[3] and scientific[4] libraries available to setups with solvers written in C++, for example OpenFOAM[5], and other languages. Since Julia was designed to bring the productivity and abstraction of dynamic languages to the field of numerical computing, which has benefited the least in the last decades as stated in [1], there is no better time to embed the still growing language like Julia into the emerging field of multi-physics simulations, which will only get closer in the future.

The aim of this thesis is to provide a solid investigation in coupling Julia code with the library preCICE. This includes implementing and testing a Julia language binding in the form of a Julia module for the preCICE library, and researching possible compatibility limitations between Julia's native features like process-level parallelization and the communication means of preCICE when coupling distributed solvers.

Structure of the thesis

The thesis consists of the following sections:

Chapter 2 – The Julia Programming Language: We dive in with an introduction to Julia and look at a FEM solver code.

Chapter 3 – The coupling library preCICE: Presents the features and an example of preCICE's API.

Chapter 4 – Coupling Julia solvers using preCICE: We investigate in how to access C++ code in Julia and implement Julia bindings for preCICE.

Chapter 5 – Testing the Julia bindings: The bindings are examined on a number of different coupling scenarios and demonstrate their functionality.

Chapter 6 – Conclusions and Outlook: The work is summed up and its contribution with an outlook on further challenges are addressed.

[3]https://docs.julialang.org/en/v1/manual/parallel-computing/
[4]http://www.juliafem.org/
[5]https://www.openfoam.com/

2. The Julia Programming Language

This chapter presents some major features of Julia and covers the language specific details needed for the understanding of implementations in later chapters.

2.1. Julia's background and core language components

Julia's development started at MIT in 2009 and it was released in 2012. As stated in[2], the overall aim is to achieve high performance of low-level languages such as C or Fortran while also providing the productivity or convenient abstraction that popular high-level languages offer to their users. This improvement should eliminate the habit found in numerical or scientific programming: using high-level environments for prototyping first and then rewriting an algorithm in a low-level language for an improved performance in speed and memory usage. Julia should finally allow technical computing to benefit from abstraction. Let see get an overview of Julia.

2.1.1. Julia's code selection paradigm

Julia is a dynamically typed language and its syntax reminds strongly of MATLAB, Python and other languages known in Scientific Computing. Julia is Just-In-Time compiled and build on top of the LLVM compiler framework[2]. A dynamic language derives its type information during runtime, and it still can be fast: Julia joined in 2017 the petaFLOPS club of our static C++ and Fortran languages[1], as it was used to write Celeste[7], a model categorizing astronomical objects from large data sets.

Julia derives its speed thanks to being designed around something any language needs to be good at: managing uncertainty[1, 2]. It has code selecting mechanisms on multiple levels of its design one of which is multiple dispatch. When we create a function in Julia, we also define a method. A function can have multiple methods, it is the implementation of the function for a certain combination of argument types, that Julia resolves during

[1]https://juliacomputing.com/media/2017/09/julia-joins-petaflop-club/

runtime on a function call. When calling `add(a,b)`, Julia looks at both types of `a` and `b` to decide which implementation to use. This is not method overloading, as Java or C++ generate during compile time a definite name for `add(a,b)`, something (e.g., _typea_typeb_add) called name mangling. They decide only on one, the runtime type of `obj` which implementation of `add` to call during execution.

In addition to that, Julia stores optimized code of a function for the argument type it was called. If during runtime, `func(1.0)` is called for the first time, Julia will cache native code for a program flow of the function where `Float64` was the type of the input argument, in a table for `func` (if all types can be inferred inside `func`). Calling `func` will be fast for the rest of the session, until it gets called with another concrete type (e.g., an Integer), then again, Julia will generate optimized code for that type. Having this in mind, consider a language that consists of nothing but function calls.

Julia is an imperative or procedural language, it has no classes, only structs and methods who operate on these structs. So over time in a session, for repetitive and predictable code (and in technical computing it usually is) the program turns to just deciding what native code to run next. But it prompts some compromises, as we have to make sure we write type safe routines and for very short prototyping or interactive use, the runtime compilation yields more inconvenience as a language like Python, that is using an interpreter.

2.1.2. Modules, Packages and Environments in Julia

There are a few concepts of Julia that need to be mentioned for understanding later shown code.

In a Julia module common defined functions, types and constants can be grouped in the `Module ... end`-block. Modules have their own scope, and when we can load them with `using` or import keyword, the former brings the exported definition inside our scope, as shown in Figure 2.1, where the latter only imports the module name in our namespace. After this chapter, this differentiation is ignored and we will write lines like `ModName.funcName(...` even if we loaded the packages with `using` to make clear to which module the definition belongs.

```
Module A
  export myFunc
  ...
end

using A
myFunc()
```

```
Module A
  export myFunc
  ...
end

import A
A.myFunc()
```

Figure 2.1.: Comparison of adding a module to one's scope

One or multiple Modules can be bundled in a Julia package. A Julia package is a module, that can contain further modules inside of it and has metadata for redistribution such as name, uuid, and so on. In this thesis, we will refer to packages as "`PackageName.jl`", this convention comes from the fact that the outermost module `ModuleName` in a package is written in a file `ModuleName.jl`.

Many programming languages come with the concept of local environments (e.g., virtual environments or `pipenvs` in Python) that list exact versions of all depending libraries. The Julia equivalent concepts are Julia projects, that are managed through the package manager "pkg" (similar to "pip" in Python).

Packages and projects are both portable environments, they contain a `Project.toml` file in their directory that contains metadata about dependencies, and a package in addition has `name`, `uuid` and `version` entry there.

The code examples from this thesis can be launched in a Julia REPL or as a script `julia <scriptname>.jl ...`, given that the packages in the code are in your global environment (`pkg> add PackageName`) or in a local project environment (`pkg> activate .`, `pkg> add PackageName`).

2.1.3. Calling C code from Julia

Since we are particularly interested in porting a C++ library, we should take a look at the closest feature we get from the standard library for doing that. Julia offers native ways to call functions of C or FORTRAN shared libraries. With the **`ccall`** function, which is part of the Base library, code in those languages can be called from Julia as long as they are compiled to shared libraries.

An example call of a C library `libmean` that computes the mean of two numbers would eventually look as shown in Figure 2.2.

It can be easily understood that the function **`ccall`** takes the function name and the path of the target library, possible arguments, and returns the result after execution.

```
julia> mean = ccall(
                    ("calcMean", "path/to/libmean.so"), # (function-library pair)
                    Cdouble,    # return type of C function
                    (Cdouble, Cdouble),     # argument types of C function
                    10.0, 20.0)     # actual Julia input arguments
15.0
```

Figure 2.2.: Example call to a C library

Before our actual arguments, as shown in the example, we pass `Cdouble` once and then again as a tuple. Julia offers alias types like the `Cdouble` specifically created to suit a type mapping from C/FORTRAN to Julia and the other way around.

Our `calcMean` function expects two arguments with the C++ type `double`, so we pass our two Julia arguments of the type `Float64` and tell `ccall` to cast them to `Cdouble` with that type tuple. That means, if a C function needs a float, we pass the Julia base type `Float32`, if a C function returns a `char[]` (a string), we state to expect the Julia alias type `Cstring` (that we later convert to a Julia string).

As long as `ccall` gets the corresponding type information, it will make a cast (where possible) before invoking the foreign function.

One more remark on the return types: We get in the example the mean as a `Cdouble` back, which in this case can be indeed just interpreted as a Julia `Float64`, but this is not trivial for other types. One has to be aware if the return type may need further conversion. For example, the returned types `Cshort`, `Cint` or `Cstring` can not be treated as Julia's `Integer` (`Int64` or `Int`), `AbstractFloat` or `AbstractString` types. They are actually Julia's `Int16`, `Int32` and `Ptr{Uint8}` types, so one is dealing with bitstypes or concrete types. A type correspondence table is provided in the documentation for ccall[2].

2.2. Example of Julia code: solving the heat equation

This section presents Julia code of an finite element method (FEM) solver to get a first look at Julia's syntax and a basic understanding of physics solvers we are interested in. The provided example solves the heat equation in 2D space, in other words, simulating heat transfer on a rectangular plate.

[2]https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/

2.2.1. Problem Definition

The distribution of temperature in space over a period of time is described through the heat equation, a partial differential equation. On a two-dimensional plate with an external heat source, the time-dependant heat equation has the following form:

$$\frac{\partial u}{\partial t} \;=\; k\,(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) + f \qquad (x,y) \in \Omega \tag{2.1}$$

where k is the thermal conductivity, f the heat source and $u(x,y,t)$ is the unknown function that describes the temperature at a certain time and point on the rectangular domain Ω. For this domain we consider the following fixed boundary condition:

$$u(x,y) = 0 \quad \text{for} \quad x,y \in \partial\Omega \tag{2.2}$$

where $\partial\Omega$ is the boundary of Ω, and is also called a Dirichlet boundary condition, that needs to be fulfilled by the solution u. Together, (2.1) and (2.2) describe a boundary value problem (BVP). We can solve this boundary value problem using FEM, a numerical method to compute an approximate solution.

2.2.2. Solver Code

The example code uses Ferrite.jl[3], a Julia library for finite element analysis. The solver describes the weak form of the heat equation as the variational form, an integrated version of the heat equation. This results in an variational problem, that reduces to solving a linear system. Figure 2.3 shows the important parts of the solver.

[3]https://ferrite-fem.github.io/Ferrite.jl/dev/

```julia
using Ferrite, SparseArrays # importing library methods into scope with 'using'

grid = generate_grid(Quadrilateral, (100, 100)) # create domain
...
# simulation parameters
max_temp = 100
Δt  = 1
T = 200  # total time steps

# define dirichlet boundary conditions on top and bottom edge of the domain
∂Ω₁ = setBoundaries(["top", "bottom"], (x,t) -> 0)
...
# temperature on left boundary is described as function a
a(x,t) = t*max_temp / T
∂Ω₂ = setBoundaries(["left"], a)
...
# assemble linear system
K, f, M, A = ... # stiffness matrix K, mass matrix M, system matrix A

uₙ = zeros(length(f))   # initialize first timestep

@time for t in 0:Δt:T
    update!(ch, t)     # update boundary condition values
    b = Δt .* f .+ M * uₙ    # compute right-hand side
    apply_rhs!(rhsdata, b, ch)     # apply boundary conditions of the current time step

    u = A \ b    # solve timestep
    uₙ .= u    # update solution
    saveData(u, t)     # write output file
end
```

Figure 2.3.: Solver snippet of the Ferrite.jl time dependant Heat Transfer example[1]. Note that Julia supports all characters of the UTF-8 encoding.

First meshes, simulation parameters, boundary conditions, and the equation are defined. Then, the initial state is set. The simulation's main part is the **for**-loop, where we compute u for the current timestep t, save it for visualization and then move on in time by the timestep-length Δt.

Regarding the Julia syntax: Methods such as `generate_grid`, `update!` and `apply_rhs!` come from the package `Ferrite.jl`. Function names with an exclamation marks indicate that they change the objects passed. Julia is not Object-Oriented so it consists throughout

[1]https://ferrite-fem.github.io/Ferrite.jl/dev/examples/transient_heat_equation/

of functions working on types. Note that the point of loading functions into one's namespace with `using` is to allow multiple packages to extend the same operation for different arguments. Other packages could provide the function `update!` but not for the types `Ferrite.jl` is using.

Another concept is the broadcast operator `.=`, `.*`, and so on, that applies the operator to every element of the first operand and the inverse divide operator \ that in this case is overloaded and solves the algebraic equation. Before the for loop we can see a `@time` macro expression. We will see it through the thesis a couple of times, it wraps the expression after it into a new one, giving it a new functionality with little change to the syntax. The `@time` macro is prompting the execution time and memory allocation of the for loop.

After running the simulation, `Ferrite.jl` provides `vtk_save` to write u to VTU files, that can be visualized with ParaView:

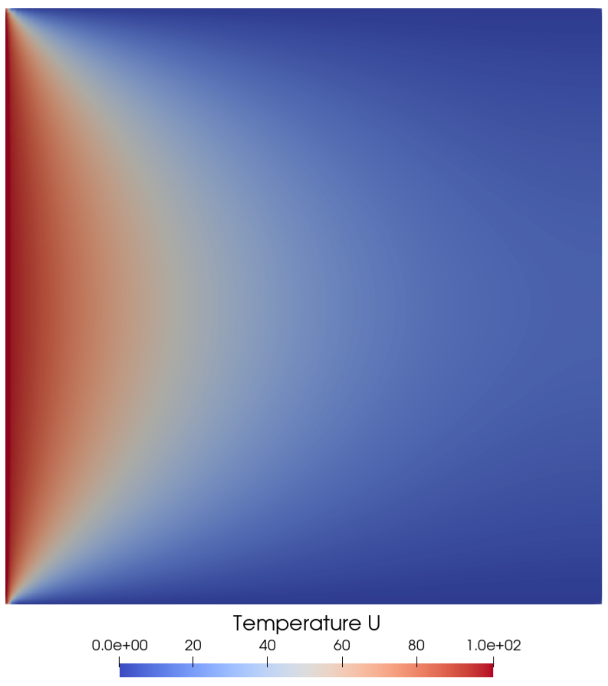

Figure 2.4.: Heat distribution at T = 200.

3. The coupling library preCICE

This chapter introduces the coupling library preCICE in a condensed form. It starts with a general overview of the features and then covers the library API.

3.1. Overview of preCICE

preCICE is an open-source C++ coupling library for partitioned multi-physics simulations, and is developed at the University of Stuttgart[1,2] and at the Technical University of Munich[3].

The idea of the library is to create a multi-physics simulation by taking already existing single-physics solvers and couple them in a partitioned manner. Since the individual solvers are capable of simulating a sub-part of the complete physics, joining them with preCICE is commonly termed as a partitioned simulation. This approach enables the reuse of popular solvers that are already well adapted in their field and provides advantages usually known from modular software architectures, such as flexibility and lower overall complexity of the code setup.

preCICE is not a framework or a software executable, it provides its library API which is accessed from the solver's code. The calls to the preCICE API, the library functions, which we will inspect in greater detail in the next sub-chapter, are minimally-invasive changes to the source code: preCICE couples the solvers in a black-box fashion, not requiring details of their physics or discretization.

preCICE is written in C++, but it offers language bindings to couple solvers written in other languages. At the time of this thesis there are bindings for Python, MATLAB, Fortran and C, the latter two being native bindings that are part of its core library. For popular solvers (e.g., OpenFoam, FEniCS, and more), there are also adapters, which are software packages that simplify coupling to the solvers by wrapping the preCICE API

[1]https://www.ipvs.uni-stuttgart.de/departments/us3/
[2]https://www.ipvs.uni-stuttgart.de/departments/sgs/
[3]https://www.in.tum.de/en/i05/home/

into the native style of the solver. Figure 3.1 shows overview of preCICE‘s ecosystem, and work in progress features in grey.

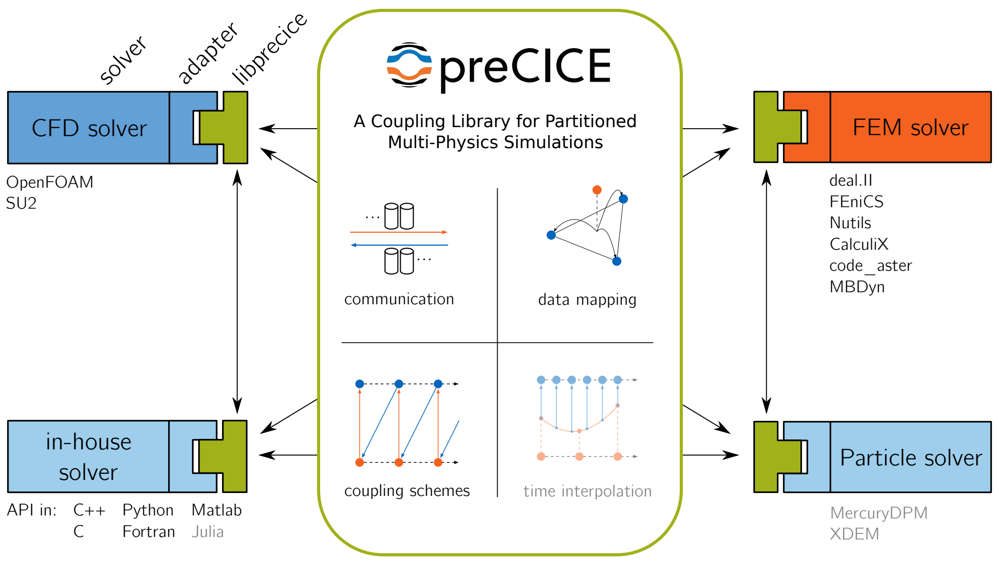

Figure 3.1.: Basic features and concepts. From [4], which is summarizing preCICE’s development from 2016 to 2021.

When solvers are coupled with preCICE, they do not "connect" a central instance. Conducting the partitioned simulation consists of configuring an XML file first, in which methods and parameters of the coupling are set. After that, the solvers are started in their respective environments on their own, each making calls to the shared library `libprecice` during runtime and start to couple. Think of preCICE as a distributed or peer-to-peer software enabling solvers to communicate, and a monolithic depiction throughout this thesis serves only as an abstraction.

preCICE offers, as seen in the Figure 3.1, data mapping, communication methods, coupling schemes; various features/components necessary for multi-physics coupling. To give an overview for these features it is best to consider what coupling solvers in a partitioned simulation would require numerically and technically:

Solvers are computing data values on points of their discretized domains (meshes) during runtime, that requires the data to be brought to the other solver’s domain, while their meshes may have non-matching grids. For that, preCICE offers **data mapping** options for interpolation (mapping types, mapping methods and more).

Then, the two solvers are computing the solutions of their equations that depend on each other, resutling in a coupled equation. Depending on the interactions of the involved physical phenomena, it is a weakly or a strongly coupled problem. The numerical solutions of the former case can be computed with a fixed number of solver executions per timestep, the latter requires multiple iterations due to stability issues. For these cases preCICE provides **coupling schemes**. A detailed explanation can be found in chapter 2 and 4 of [5]. While the details do not concern us for the rest of this thesis, note that the coupling schemes are a core part of preCICE since solving those strongly coupled problems is still conducted in a black-box coupling, contrary to a monolithic approach.

Finally our solvers might want to run on a large number of nodes, multiple processes for a parallel computation, what would require a scalable communication in terms of the number of parallel processes. And for that, preCICE offers means to establish a point-to-point **communication**, based on the geometry of the meshes from the two domains. We discuss this feature more in 5.2.

preCICE offers many more features for coupling, most of which are beyond the scope of this thesis. Further information can be found on the website[4]. These main features and further concepts such as multi-coupling are described accurately in [3]. But since preCICE is not only a library but also an open-source project, an extended description can be found in [4].

3.2. Using preCICE's API to couple a solver

The previous section explained that solvers are coupled by making calls to the preCICE API. Let us take a closer look at what that means. Figure 3.2 shows a simplified loop of the Julia solver from 2.2.

```
uₙ = zeros(length(f))

while t < T
    uₙ = solveTimeStep(Δt, uₙ)
    t += Δt
    saveData(uₙ, t)
end
```

Figure 3.2.: Structure of the solver's loop computing Temperature distribution u.

This solver computes some values temperature values u, and we want to exchange at the right boundary to another solver during their runtime. Technically, preCICE is a shared

[4]https://precice.org/

library, libprecice.so, a binary that we need to access from our solver's code. Either our programming language offers a way to access shared libraries or your are using languages supported by preCICE (C++, Python, and so on), in which case preCICE is available as a package or library in that language.

Let us assume that we have a preCICE package for Julia called PreCICE.jl, with which we can access the library (but note that the concept of the API is independent of the programming language). We would then extend our solver with the following calls:

```
Using PreCICE

PreCICE.createSolverInterface("temperatureSolver", "./precice-config.xml", 0, 1)

meshID = PreCICE.getMeshID("MeshA")
writeDataID  = PreCICE.getDataID("temperature", meshID)

vertices = [100,1, 100,2, 100,3, ..., 100,100] # right side of 100x100 grid
vertexIDs = PreCICE.setMeshVertices(meshID, 100, vertices)

Δt = PreCICE.initialize()

u_n = zeros(length(f))

while PreCICE.isCouplingOngoing()

    u_n = solveTimeStep(Δt, u_n)

    temperature .= u_n[100,:] # right boundary values
    PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, temperature)

    Δt = PreCICE.advance(Δt)
    saveData(u_n, t)
end

PreCICE.finalize()
```

Figure 3.3.: Solver extended to provide temperature values to coupling partner.

First, we create a `SolverInterface` instance. Besides our solver, we refer to a `precice-config.xml` where the coupling configurations (participants, mapping method, data and meshes, coupling schemes and communication) are set. It has the structure as in Figure 3.4.

```xml
<precice-configuration>
  <solver-interface dimensions="2">
   <data name="temperature" .../>
   <mesh name="meshA" .../>
   <mesh name="meshA".../>
   <participant name="SolverA" .../>
   <participant name="SolverB" .../>
   <m2n exchange-directory="." .../>
   <coupling-scheme .../>
 </solver-interface>
</precice-configuration>
```

Figure 3.4.: The basic contents of a configuration xml defining the coupling setup. Note that it is not limited to only two participants.

The call `createSolverInterface` requires the location of the configuration file and the name of the participant as which the solver takes the role in the coupling. We create an instance of the SolverInterface class in our process, that is a handle for steering the coupling. As seen in Figure 3.4, the computation loop is now depending on `isCouplingOngoing` and the next timestep is provided by `advance`. As well as `advance`, `initialize` and `finalize` are the steering methods, that reserve/write/free data structures and set up/close communication during the coupling.

Our solver writes temperature values to his mesh. For this we defined a coupling data and mesh in the configuration, that we access with `getMeshID` and `getDataID`. To write to our coupling mesh, we define, depending on the dimension set in the configuration, a set of 2D or 3D coordinates of vertices. We set them to the coupling mesh with `setMeshVertices`, and access them with `vertexIDs`, provided by the solver instance. Now we can begin with the computation.

With `initialize` we start the coupling process, where preCICE starts building up the communication with other participants. The solver stops at this line until the other solver "solverB" calls `initialize`. To establish communication the solvers place files in the directory set in the `<m2n />` tag. Our solver could run in parallel, where first a master-slave communication is established between the parallel solvers of one participant, before the participants start to communicate.

Inside the while, we use `writeBlockVectorData`, it is one of the functions to write data values to our mesh. The participant on the other side would use another calls for reading data from meshes, for example `readBlockVectorData` to retrieve them. In 5.1 we see a setup were a participant does read as well as write data.

After we read/write and are finished computing the timestep, we call `advance`, in which preCICE does the actual coupling: sending meshes, the interpolation, update data, and iterating (depending on the coupling scheme).

As we see, the library approach of preCICE is indeed minimal invasive, the only real changes we did was to the computational loop, where we use the timesteps that `advance` and `initialize` yield. In the configuration of a coupling, we define a total maximum time for the coupling, and timestep sizes. Each solver calls advance with the timestep size Δt it used, so when the timestep size from the configuration, the "time-window" is used, advance actually then does the coupling.

4. Coupling Julia solvers using preCICE

In 3.2 it was shown that in order to couple a Julia solver we have to make calls to the preCICE API, or more specifically, to run the functions of the C++ class `SolverInterface` from Julia. This implies the need of a Julia binding for preCICE, analogous to the other non-C++ languages that preCICE supports.

The desired outcome is a module that can be redistributed to all users, a Julia package. It has the structure of a directory `PreCICE` with a `Project.toml` file, and a subdirectory `src` with a `PreCICE.jl` file containing the Julia module `PreCICE`. The following chapters describe what we will write in that script.

This chapter documents the development of accessing the preCICE library in Julia. In 4.1 the goal in the context of Julia is stated. 4.2 presents ways to call C++ code in Julia, and 4.3 provides the final solution, demonstrated in an example case of coupling two Julia codes.

4.1. Calling C++ code in Julia

Julia offers a wide set of tools for interfacing with other languages such as C++, but there will be never complete a guide on how to port any library we want to use; each binding for a C library can vary according its functionality. We look take a look the possibilities Julia offers and investigate them to write the package, what as a side-effect eventually creates a guide or overview for bringing a C++ library to Julia.

4.1.1. Cxx.jl

When first searching for a way to run C++ code in Julia, one will very likely first encounter the Julia package `Cxx.jl`[1]. It provides C++ code interfacing, meaning we can integrate C++ code directly in Julia code. An example from its documentation:

[1]https://github.com/JuliaInterop/Cxx.jl

```
function playing()
    for i = 1:5
        icxx"""
            int tellme;
            std::cout<< "Please enter a number: " << std::endl;
            std::cin >> tellme;
            std::cout<< "\nYour number is "<< tellme << "\n" <<std::endl;
        """
    end
end
```

Figure 4.1.: Running C++ inside Julia code.

```
using Cxx
using Libdl
Libdl.dlopen("./libArrayMaker.so", Libdl.RTLD_GLOBAL)

cxxinclude("ArrayMaker.h")

# create array with 5 entries, that have the values 2^1 to 2^5
maker = @cxxnew ArrayMaker(5, 2.0)

arr = @cxx maker->fillArr()
```

Figure 4.2.: Using a C++ Class from a shared library.

`Cxx.jl` can also load shared libraries and it should be possible to write a C++ wrapper on “Julia’s side”. It seems very convincing. Bu it could become more complicated, since the final member call "`@cxx maker->...`" does not return a Julia data structure, but a pointer. We need to write a wrapper code to further process `arr` or more complicated data structures.

Furthermore, a concerning issue is that this package only supports Julia versions 1.1 to 1.3 at the time of this thesis work and prompted because of this building errors on the more recent stable releases like Julia 1.6.1:

```
ERROR: Error building `Cxx`:
ERROR: LoadError: could not load library "libLLVM-11.0.1"
libLLVM-11.0.1.so: cannot open shared object file: No such file or directory
Stacktrace:
...
```

Cxx.jl might change in the future but this and the support for the overall newer versions of Julia are still open issues, offering the only fix to it by using Julia versions 1.1 to 1.3. It is not an official Julia package and the future of this project seems unclear. Because of this, this solution was not further investigated. But for future references, we should add that if the support will continue, this Cxx.jl package should be considered in future bindings of C++ libraries.

4.1.2. CxxWrap.jl

`CxxWrap.jl`[2] is a Julia package for wrapping C++ libraries. Contrary to `Cxx.jl`, this package is intended to wrap complete libraries and create a Julia-equivalent package.

As we know from chapter 2, **`ccall`** is a native way to run C functions, but does not work for C++ symbols since C++ compilers use a different name mangling, that Julia does not expect when trying to access a function of the library.

The main idea is to store pointers of the desired C++ functions in an array that is then exposed to Julia by being returned in a C function, a function declared in the ’`extern "C"`’ block, that we know from 2.1.3 Julia can access with **`ccall`**. On the Julia side this array is extracted, in a module with equivalent functions, where they are accessing the according C++ function pointers. And for further C++ features like classes, standard and custom data structures, this array would have to be more complicated, but the concept remains.

The convenience of `CxxWrap.jl` is that the structure of the C++ exposing class and the creation of the Julia module are handled by it. There is only an additional C++ code part to write and compile. It is producing an auxiliary C++ library, exposing our desired library, and requires a short Julia part that is generating the Julia module which is linking to the auxiliary library.

The following example gives a short demonstration of the required programming:

In a `.cpp` file that includes the `jlcxx` header, we use the Module class, to which we add the desired C++ construct. In this example, we add a method by putting the address `&greet` of it together with the desired name `"greet"` we want to have on Julia’s side.

[2]https://github.com/JuliaInterop/CxxWrap.jl

```cpp
#include <string>
#include "jlcxx/jlcxx.hpp"

std::string greet()
{
   return "hello, world";
}

JLCXX_MODULE define_julia_module(jlcxx::Module& mod)
{
  mod.method("greet", &greet);
}
```

This C++ side is built and compiled in a CMake project, linking to `jlcxx`. On the Julia side, we create our own Module using the `CxxWrap.jl` package. Here we just have to load the compiled library with the `@wrapmodule` macro, and `CxxWrap.jl` will generate the respective Julia function `greet` together with the proper return and input argument types.

```julia
module CppHello
  using CxxWrap
  @wrapmodule("path/to/built/libgreet.so")
  function __init__()
    @initcxx
  end
end
# Now use the added function
CppHello.greet()
```

So this solution enables to write a wrapper mostly in C++ and demands less knowledge from the user about the mapping of C++ types to Julia types and vice versa. The `greet` function from the example will be annotated with the respective Julia String type as its return type, and if it would return a custom C++ class, `CxxWrap.jl` will also generate the appropriate Julia struct type.

The Julia Module will depend on this compiled wrapper C++ library (besides the C++ library we wanted to port to Julia) that needs to be distributed or compiled on other machines in addition to the Julia module.

In 4.3 a possible Julia-binding of the preCICE library with `CxxWrap.jl` will be presented and discussed.

4.1.3. Using Julia's build in ccall

As discussed in 2.1.3, Julia can call functions of C libraries with ccall, which is not compatible with C++ libraries without further work because of the different name mangling. But if a C++ library offers, similar to what the `CxxWrap.jl` solution is doing, a C interface, then ccall can be used to write a wrapper library entirely in Julia like for any ordinary C library.

The binding would essentially be a Julia module that contains low-level methods accessing the API of the C library with ccall. This approach, wrapping an entire C library with a corresponding Julia module may be the go-to solution for bringing many famous C libraries such as `MPI.jl` or `LibSerialPort.jl` to Julia, but for C++ libraries it should be considered whether writing a C interface and then the corresponding Julia module to it requires less effort than just generating the binding using `CxxWrap.jl`. The almost effortless creation of a Julia module that contains all the C++ functions with correctly typed argument annotations from `CxxWrap.jl` should not be overlooked at this point.

This solution of writing low-level **`ccall`** wrapper functions eventually works for preCICE. It has the exactly needed C interface since it already provides a C binding included in its core library. The possible solution of writing a binding for preCICE with **`ccall`** is described in 4.3.

4.2. The final Julia binding for preCICE

In the previous subsections, ways of writing a Julia binding for C++ libraries were presented. This chapter describes the final software solutions which were developed.

4.2.1. The CxxWrap.jl binding

`CxxWrap.jl` is, as stated in the previous subsection, a fast solution to create a Julia binding for a C++ library. However, the development and building process might appear confusing because it is scattered around two repositories, one[3] for the Julia and one[4] for the C++ part, called "libcxxwrap-julia". Another guide for it is a recording[5] of a workshop "Wrapping a C++ Library with CxxWrap.jl" from JuliaCon 2020. The

[3]https://github.com/JuliaInterop/CxxWrap.jl
[4]https://github.com/JuliaInterop/libcxxwrap-julia
[5]https://www.youtube.com/watch?v=VoXmXtqLhdo

steps required for wrapping the preCICE library were derived from the above-mentioned resources.

To create the C++ side of the wrapping, `CxxWrap.jl` offers the CMake project repository libcxxwrap-julia [6]. It needs to be built and compiled to `libcxxwrap_julia.so`. We create our own CMake project, that links to that `libcxxwrap_julia.so` build location and to the preCICE shared library in its `CMakeLists.txt`, as shown in Figure 4.3.

```
project(Juliaprecice)
# ...
set(JlCxx_DIR /path/to/libprecice.soDir/)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/lib")

find_package(precice REQUIRED CONFIG)
find_package(JlCxx)

get_target_property(JlCxx_location JlCxx::cxxwrap_julia LOCATION)
get_filename_component(JlCxx_location ${JlCxx_location} DIRECTORY)
set(CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib;${JlCxx_location}")

add_library(jlprecice SHARED juliaprecice.cpp)

target_link_libraries(jlprecice precice::precice)
target_link_libraries(jlprecice JlCxx::cxxwrap_julia)
# ...
```

Figure 4.3.: An abbreviated CMakeLists.txt for compiling the auxiliary library.

We write the C++ wrapper code in a juliaprecice.cpp file, in which we expose the SolverInterface class we want to the Julia side:

[6] https://github.com/JuliaInterop/libcxxwrap-julia

```cpp
#include "jlcxx/jlcxx.hpp"
#include <precice/SolverInterface.hpp>

using namespace precice;

JLCXX_MODULE define_julia_module(jlcxx::Module& mod)
{
    mod.add_type<SolverInterface>("SolverInterface")
        .constructor<const std::string, const std::string, int, int>()
        .method("initialize", &SolverInterface::initialize)
    ...
        .method("advance", &SolverInterface::advance)
        .method("finalize", &SolverInterface::finalize)
   mod.method("getVersionInformation", &getVersionInformation);
}
```

Figure 4.4.: Adding SolverInterface to the C++ class "mod" that representing a Julia module

The full code of Figures 4.3 and 4.4 can be found at A.2 and A.3, and a minimal template of the CMakeLists.txt is provided in an example repository[7]. The linking method from the documentation[8] of preCICE was followed to include its shared library to a CMake project.

After that, this CMake project is built and compiled to a shared library `libjlprecice.so`. This library is then referenced on the Julia side. Here we create a new Julia package “PreCICE”, and write in src/PreCICE.jl the module that imports CxxWrap.jl and uses its wrapmodule macro:

```julia
module PreCICE

    using CxxWrap
    @wrapmodule "path/to/libjlprecice.so"
    function __init__()
        @initcxx
    end

end
```

Figure 4.5.: Importing the auxiliary library in a Julia module.

[7]https://github.com/barche/libfoo
[8]https://precice.org/installation-linking.html

Now the Julia package contains all the exposed symbols added to the module entry on the C++ side that can be called like a Julia function:

```
import PreCICE
PreCICE.getVersionInformation()
```

Since Julia is not an Object-Oriented Language, we run functions of the `SolverInterface` class still from the module, but insert the instance of `SolverInteface` as the first argument:

```
using PreCICE
interface = SolverInterface(solverName, configFileName, commRank, commSize)
...
dt = initialize(interface)
...
dt = advance(interface, dt)
...
```

The convenience comes obviously from the auto generation of these Julia methods with the correct argument type annotations. The drawback here is that `PreCICE.jl` is depending on another, the `CxxWrap.jl` and the auxiliary library jlprecice.so. This means that along PreCICE.jl, one would have to distribute this binary too.

The Julia package manager and Julia packages are actually more than their equivalent in other languages. Julia offers a way for distributing binaries through so-called JLL packages, that are redistributed and downloaded like a Julia package.

For example, the first step where we compile `libcxxwrap.so`, can be omitted because it is available as a JLL package: We would download the binary `libcxxwrap_julia_jll` from Julia's general wrapper repository[9] instead. This JLL package detects the target operating system and builds the appropriate binary file, in this case the `libcxxwrap_julia.so`.

This made possible through a `binarybuilder.jl` script, such one we would in turn write for our own `libjlprecice.so`, where preCICE and its dependencies are installed, and again distribute it as `libjlprecice_jll`. Overall, the module would be written now as:

```
module PreCICE
    using CxxWrap
    @wrapmodule jlprecice_jll # package manager downloads the jll package automatically
...
```

[9]https://github.com/JuliaBinaryWrappers

In short, the building and redistribution of a binary can be reduced to a simple dependency on a JLL package. This is made possible through `BinaryBuilder.jl`[10]. A Julia script can be either written manually or through a wizard from `BinaryBuilder.jl`, that saves for which operating system platforms, where to download the archive, and how to build and compile the depending library.

This extra effort implies that while a `CxxWrap.jl` solution depends on additional binary artifacts, Julia offers native ways to redistribute them.

4.2.2. The native ccall binding

Writing the preCICE library with Julia's native `ccall` works just like wrapping any other C library, as described in 2.1.3, by calling functions of preCICE's C bindings.

Let us take a look at an example by wrapping the function `setMeshVertex`. This method's signature looks on the C interface side like this:

```
int precicec_setMeshVertex(int  meshID, const double *position);
```

`setMeshVertex` is used to create a single mesh vertex on a coupling mesh. The arguments are the ID of the target mesh (C integer) and a pointer to an array (C pointer to a container) holding the coordinates (C double) of the vertex. It returns an ID of the vertex created (C integer).

The Julia side method is simply wrapping `ccall` with the following arguments:

```
function setMeshVertex(meshID::Integer, position::AbstractArray{Float64})
    id = ccall(
                (:precicec_setMeshVertex, libprecicepath),
                Cint,
                (Cint, Ref{Float64}),
                meshID, position)
    return id
end
```

- The function name and library path tuple,
- `Cint` as the expected return type from `precicec_setMeshVertex`,
- a tuple containing the input types `Cint` and `Ref{Float64}` expected on the C end,
- and the actual values we want to call precicec_setMeshVertex with.

[10]https://binarybuilder.org/

The documentation of ccall[11] provides a table on what Julia alias types for which C/C++ types should be used. Here we can use `Cint` for int and `T*` (for an alias type `T`) is passed as `Ref{T}`, in this case `Ref{Cdouble}` or `Ref{Float64}` (since `Cdouble` is also an alias for `Float64`). Note that as these alias types are passed to `ccall`, the actual types of the arguments passed at the end of `ccall` such as `meshID` and `position` in the above example do not have to match those alias types exactly. This `ccall` function call is wrapped in a function that takes the necessary arguments annotated with the closest Julia equivalent type, such as Integer and `AbstractArray{Float64}` and not `Cint` and `Ref{Float64}` since `ccall` will implicitly convert them to the alias types, before finally passing them to `precicec_setMeshVertex`.

Writing the wrapper method or the signature of one can in some cases be less trivial, as in the case of `setMeshVertices`, a similar function as the one before, but now multiple vertices are defined on a coupling mesh at once. The original method signature looks like this:

```
void precicec_setMeshVertices(int meshID, int size, const double *positions, int * ids);
```

The difference to `setMeshVertex` is the new argument `size` of the amount of the new vertices one wants to set, and the `void` return type. Since C/C++ functions do not support returning multiple values at once, multiple references or just one reference of one array are passed as arguments that are set by the C function. This is the case here with `ids`, as an array is expected to be passed by reference that is set with all the ids of the created vertices after the function call. This should be done internally instead of demanding the caller of such a wrapper function to create an array himself, because that would require knowledge with which type the Julia array one would need to parametrize. For example, `AbstractArray{Integer}` would yield wrong conversions than `AbstractArray{Int32}`. So the Julia side wrapper method should look as shown in Figure 4.6.

```
function setMeshVertices(meshID::Integer,size::Integer, positions::AbstractArray{Float64})
    ids = Array{Int32, 1}(undef, size)
    ccall((:precicec_setMeshVertices, libprecicePath),
          Cvoid,
          (Cint, Cint, Ref{Cdouble}, Ref{Cint}),
          meshID, size, positions, ids)
    return vertexIDs
end
```

Figure 4.6.: Deviating from the library function's signature.

[11] https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/#man-bits-types

Internally, the method that wraps `setMeshVertices` first creates a Julia array with `Int32` (`Cint`) as its element type. This array is then passed on to `ccall` with `Ref{Cint}` as the expected type of the C function. This example demonstrates an exception where the Julia signature will have to change from the library function, since passing an array with anything other than `Int32`/`Cint` typed elements would return an array not with the ids but with corrupted values, and that would demand knowledge about the C function and the inner workings of ccall to get it right.

In similar ways exception handling could be built into the wrapper functions, but the error checking was left to `ccall` and the preCICE library itself for the initial solution.

The wrapped functions are included next in a Julia module PreCICE. This module embodies the core part of the Julia binding.

Some Julia bindings of other C libraries separate between the wrapper functions and their handling of type conversions or errors. An example structure of such a library is `LibSerialPort.jl`[12], that consists of multiple modules. `PreCICE.jl` contains only one module with the wrapper functions, to keep the maintenance of the code to that one of changing a single file and to correspond to the file "SolverInterface.cpp" of the core C++ library of preCICE. Should the complexity of the binding evolve in the future, a similar structure of `LibSerialPort` should be considered.

In addition to the wrapper functions, a way to provide the location of preCICE library location is required, since `ccall` depends on it in the first `(functionname, librarypath)` tuple argument. As discussed at the end of the `CxxWrap.jl` implementation, the Julia way of providing additional binary files is by creating a JLL package, that will be included at the top of our module with "`using preCICE_jll`". An alternative or more straight forward way is to create a global variable `libprecicePath`, that is referenced in every ccall.

The total code of PreCICE.jl for the ccall implementation can be found on the `julia-bindings`[13] repository.

We have seen above that ccall needs Julia alias types to know to what C types the provided Julia typed arguments have to be converted as it is invoking them on the shared library.

[12]https://github.com/JuliaIO/LibSerialPort.jl
[13]https://github.com/precice/julia-bindings

C method	Julia wrapper	ccall
int	`Integer`	`Cint / Int32`
double	`Float64`	`Cdouble / Float64`
char*	`String`	`Cstring / Ptr{UInt8}`
int*	`AbstractArray{Cint}`	`Ref{Cint} / Ptr{Cint}`
double*	`AbstractArray{Float64}`	`Ref{Cdouble} / Ptr{Cdouble}`
void*	`MPI.Comm`	`MPI.MPI_Comm / Ptr{Cvoid}`

Table 4.1.: Type translation of the ccall binding.

For the sake of completeness, Table 4.1 shows all the shows all the according Julia types chosen for the arguments of the wrapper methods.

It depicts the relation of a wrapper template

```julia
function wrapperMethodName(juliaArg1::TypeX)
    ccall((:cfunc, libpath), Cvoid, (TypeY,), juliaArg1)
    return vertexIDs
end
```

and the C function signature `void methodName(TypeZ cArg2);` where `TypeX`, `TypeY`, and `TypeZ` are the **Julia wrapper**, the **ccall**, and the **C method** types from the table. The last row in the table is for an alternative constructor of the SolverInterface class takes an MPI communicator, what can be replicated in Julia with the `Comm` and `MPI_Comm` types from the `MPI.jl` package. However, the type `Ptr{Cvoid}` is a valid type as well and does not make `MPI.jl` a dependency for `PreCICE.jl`.

4.3. Final remarks about the bindings

The last section described two ways of writing a Julia binding for preCICE. No comparison was intended, but chronologically the `CxxWrap.jl` solution was developed first. The `CxxWrap.jl` solution is included in the appendix of this thesis, but it is not preferred because of preCICE's C bindings being part of the core library, the native wrapper solution is not depending on any other Julia packages or binaries, besides `libprecice`.

Another remark is that the C bindings do not return a `SolverInterface` instance (but it is created as a field referenced by further calls inside the process), what makes the API calls closer to Julia's class-less paradigm. We remember, Julia is not Object-Oriented, `PreCICE.apicall()` notation is calling a method of namespace (the one of module PreCICE).

Let it also be said that the Julia bindings we are discussing are implemented to work on Linux. Note that preCICE is also available on other platforms, at the time of this thesis work for macOS; support for Windows is given at the moment through WSL (Windows Subsystem for Linux).

5. Testing the Julia bindings

This chapter demonstrates the Julia bindings on various coupling scenarios. First, two dummy Julia solvers are coupled in a minimal example setup. 5.1 is describing this setup in detail, whereas later in 5.2, the configuration and solver files are extended to show coupling scenarios using parallelism with MPI. In **??**, the coupling example from 5.2 is modified use Julia's native parallelism.

5.1. Coupling two dummy solvers

This example is based on the solverdummies[1] from preCICE's library, where two pseudo Julia solvers are coupled, that exchange arbitrary data. It is meant to serve as a minimal working example, without simulating any physics.

5.1.1. Configuration of preCICE

Consider two participants `SolverOne` and `SolverTwo` that exchange some data every timestep. For this setup, we create a `precice-config.xml` with the following contents.

1. We define the data values we want to exchange:

```
    <data:vector name="dataOne" />
    <data:vector name="dataTwo" />
```

In a proper scenario those values would be forces or temperature, that can also be scalars instead of vectors.

2. After we defined the type of data we want to exchange, we need to define the geometry or domain on which the solvers make their computation:

[1]https://github.com/precice/precice/tree/develop/examples/solverdummies

```
    <mesh name="MeshOne">
      <use-data name="dataOne" />
      <use-data name="dataTwo" />
    </mesh>

    <mesh name="MeshTwo">
      <use-data name="dataOne" />
      <use-data name="dataTwo" />
    </mesh>
```

We created two coupling meshes `MeshOne` and `MeshTwo`. Between these two meshes, the data mapping is performed by preCICE.

3. Now it is time to add the two participants themselves:

```
    <participant name="SolverOne">
      <use-mesh name="MeshOne" provide="yes" />
      <write-data name="dataOne" mesh="MeshOne" />
      <read-data name="dataTwo" mesh="MeshOne" />
    </participant>

    <participant name="SolverTwo">
      <use-mesh name="MeshOne" from="SolverOne" />
      <use-mesh name="MeshTwo" provide="yes" />
      <write-data name="dataTwo" mesh="MeshTwo" />
      <read-data name="dataOne" mesh="MeshTwo" />
    </participant>
```

The participants provide the respective meshes `MeshOne` and `MeshTwo` (make them visible), and in addition to that, `SolverTwo` uses `SolverOne`'s `MeshOne` because we choose to perform the data mapping on `SolverTwo` (the mapping has to be done on either one of the participants).

4. We add the nearest-neighbor mapping to `SolverTwo`:

```
    <participant name="SolverTwo">
      <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-neighbor
        direction="read" from="MeshOne" to="MeshTwo" constraint="consistent" />
      <write-data name="dataTwo" mesh="MeshTwo" />
      <read-data name="dataOne" mesh="MeshTwo" />
    </participant>
```

5. Finally, we define the coupling scheme, how our two solvers exchange their data after calling advance:

```
    <coupling-scheme:serial-implicit>
      <participants first="SolverOne" second="SolverTwo" />
      <max-time-windows value="2" />
      <time-window-size value="1.0" />
      <max-iterations value="2" />
      <min-iteration-convergence-measure
        min-iterations="5"
        data="dataOne"
        mesh="MeshOne" />
      <exchange data="dataOne" mesh="MeshOne" from="SolverOne" to="SolverTwo" />
      <exchange data="dataTwo" mesh="MeshOne" from="SolverTwo" to="SolverOne" />
    </coupling-scheme:serial-implicit>
```

While a `serial-implicit` coupling scheme is not the most minimal one, we set it instead of `serial-explicit` so we can observe multiple iterations of one timestep and convergence measures in the coupling too.

6. For the communication between the participants, we just add

```
    <m2n:sockets from="SolverOne" to="SolverTwo"/>
```

for a communication via sockets. How we set the `from-to` direction does not matter in this setup.

All the above tags are included inside `<solver-interface dimensions="3">` tag. The full `precice-config.xml` file can be found at A.4.

5.1.2. Julia code of solver dummies

For writing the two Julia solvers, we can create one `solverdummy.jl` file. In it we first import our PreCICE package and create an interface object for the participant we want to couple:

```
using PreCICE
PreCICE.createSolverInterface("SolverOne", "precice-config.xml", 0, 1)
```

Since `SolverOne` provides the mesh `MeshOne`, we need to define vertices on it. For that, we first get the meshID, create an array with the coordinates of the vertices and create them with `setMeshVertices`:

```
meshID = PreCICE.getMeshID("MeshOne")
vertexIDs = PreCICE.setMeshVertices(meshID, numberOfVertices, vertices)
```

`vertices` is an array of size `n` = `numberOfVertices * dimensions`, of the structure $[x_1, y_1, z_1, x_2, y_2, z_2, ...]$ since our domain is three-dimensional.

Now the initial "timestepping" part of our solver:

```julia
dt = PreCICE.initialize()

while PreCICE.isCouplingOngoing()

    # read data changed by other participant
    PreCICE.readBlockVectorData(readDataID, numberOfVertices, vertexIDs, readData)

    # compute data for this timestep
    for i in 1:(numberOfVertices * dimensions)
        writeData[i] = readData[i] + 1.0
    end

    # write new data to vertices
    PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, writeData)

    # finish timestep by telling it preCICE
    println("DUMMY: Advancing in time")
    dt = PreCICE.advance(dt)
```

This loop is where solvers usually solve some algebraic system for the current timestep, write the values, and then go to the next timestep. Here we just read data, update our write data, write them to the vertices, and call `PreCICE.advance`.

There are two more modifications to add. Since we are testing implicit coupling, a timestep is run multiple times, or sub-iterated, to reach a convergence threshold. preCICE needs us to revert to a previous state, an older timestep. That means we write (set variables to that of the previous timestep) or read a checkpoint.

We query when to save a state with the help of `isActionRequired`. We extend the loop to just log when reading or writing checkpoints would be required:

```
dt = PreCICE.initialize()

while PreCICE.isCouplingOngoing()

    if PreCICE.isActionRequired(PreCICE.actionWriteIterationCheckpoint())
        println("DUMMY: Writing iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionWriteIterationCheckpoint())
    end

    # read data changed by other participant
    PreCICE.readBlockVectorData(readDataID, numberOfVertices, vertexIDs, readData)

    # compute data for this timestep
    for i in 1:(numberOfVertices * dimensions)
        writeData[i] = readData[i] + 1.0
    end

    # write new data to vertices
    PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, writeData)

    # finish timestep by telling it preCICE
    dt = PreCICE.advance(dt)

    if PreCICE.isActionRequired(PreCICE.actionReadIterationCheckpoint())
        println("DUMMY: Reading iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionReadIterationCheckpoint())
    else
        println("DUMMY: Advancing in time")
    end
```

In addition to that, the time window size is defined in the configuration file as fixed. This means that preCICE prescribes the maximal timestep size until which the solvers can compute their next values for, before preCICE communicates the coupling data.

The solvers can use smaller timestep sizes, so preCICE provides `isWriteDataRequired` and `isReadDataAvailable` for reducing unnecessary calls while they subcycle:

```
dt = PreCICE.initialize()

while PreCICE.isCouplingOngoing()

    if PreCICE.isActionRequired(PreCICE.actionWriteIterationCheckpoint())
        println("DUMMY: Writing iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionWriteIterationCheckpoint())
    end

    if PreCICE.isReadDataAvailable()
        PreCICE.readBlockVectorData(readDataID, numberOfVertices, vertexIDs, readData)
    end

    for i in 1:(numberOfVertices * dimensions)
        writeData[i] = readData[i] + 1.0
    end

    if PreCICE.isWriteDataRequired(dt)
        PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, writeData)
    end

    dt = PreCICE.advance(dt)

    if PreCICE.isActionRequired(PreCICE.actionReadIterationCheckpoint())
        println("DUMMY: Reading iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionReadIterationCheckpoint())
    else
        println("DUMMY: Advancing in time")
    end
```

Finally, `PreCICE.finalize()` is called after the while-block when the simulation is complete, to close communication channels.

A similar script for `SolverTwo` can be written. Instead of this, we modify the solver-dummy.jl file so it can take the role of `"SolverOne"` as well as `"SolverTwo"`, depending on script arguments passed along.

This modified full `solverdummy.jl` file can be found in A.5, which is just one Julia solver for `SolverOne` as well as `SolverTwo` that are started with

```
julia ./solverdummy.jl ./precice-config.xml <SolverName>
```

in each terminal.

After launching one solver, it will execute until `initialize`, and then preCICE blocks and waits for the other solver:

```
~$ julia solverdummy.jl precice-config.xml SolverOne MeshOne
preCICE: This is preCICE version 2.2.0
preCICE: Revision info: v2.2.0
preCICE: Configuration: Release (Debug and Trace log unavailable)
preCICE: Configuring preCICE with configuration "precice-config.xml"
preCICE: I am participant "SolverOne"
preCICE: Setting up master communication to coupling partner/s
```

After starting `SolverTwo`, they start to read, write, advance until the maximum amount of time windows is reached. Regarding the implicit coupling: We also set in our `precice-config.xml` the `min-iterations-convergence-measure` to 5 and `max-iterations` to 2, so they will also iterate 2 times during each time-window, not reaching the convergence minimum.

5.2. Coupling two parallel dummy solvers in Julia that use MPI

The previous section demonstrates the bindings for coupling a solver that is running on one process. For large scale simulations, solvers make use of parallel computing.

5.2.1. Considerations for parallelism

Technically, this means that the algorithm of the solver allows a parallel computation of the domain. The mesh is spatially divided and parts of it are computed on different processes. This extends a coupled simulation where the solvers run on a different number of processes, called ranks. 5.1 sketches our situation:

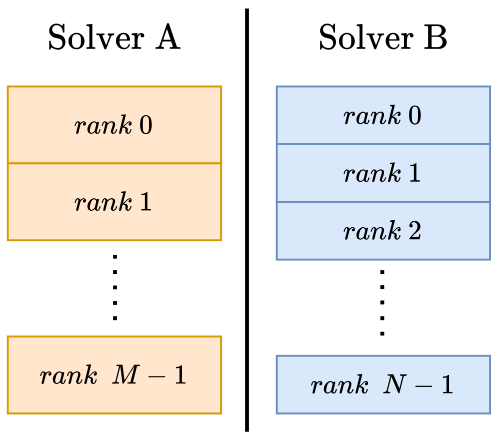

Figure 5.1.: Participants work on distributed data where a rank of A may not influence a rank of B

preCICE provides communication means so the coupling does not limit simulations that work on distributed data. It makes it possible through establishing a point-to-point connection between the ranks of all the participants. In an `M` to `N` parts decomposition of the domains, some sub-parts of Solver A's mesh may only touch with a certain subset of Solver B's meshes. This means that their respective ranks only need a communication between them, and preCICE can establish for that an M:N communication, a process mapping that establishes who communicates with whom.

The architecture of preCICE is distributed, where every rank `i` of a participant `"Solver"` creates a SolverInterface instance with

```
PreCICE.createSolverInterface("Solver", "./precice-config.xml", i, totalRankSize)
```

in its process.

To only establish connections between ranks from different domains that do touch, preCICE performs at the beginning a partitioning algorithm of the coupling meshes, based on how the geometries influence each other the M:N communication is established.

How the M:N communication and the mesh re-partitioning of preCICE work are described in [6, 8]. The important part for this sections is, that in order to be able to use solvers using the Julia bindings on massively parallel systems, we need to be able to integrate them in the M:N communication between ranks. Note that before the decentralized M:N communication is established, to send mesh information to the other participant for the mesh partitioning, another communication between ranks and between participants is created.

5.2.2. Coupling a Julia solver that uses MPI

When writing code logic for parallel computing, in our case a solver that works on a distributed mesh, a common standard is the Message Passing Interface[2] (MPI). Multiple C libraries exist that implement the API of MPI. They have the following application:

The solver code is started in n number of instances/n different processes. MPI's API provides for every process an identifying rank p_i, and further methods for the communication of those processes. The algorithms of the solver are then depending on their rank, for example, the solver computes only for points on its mesh at positions (x, p_i), but are otherwise the same. Such a model is also referred to as SPMD (Single Program Multiple Data).

preCICE's own M:N communication can be configured to work with MPI Ports or TCP/IP sockets, and supports an easy integration of existing MPI communications used by a solver. We take a look again at the solverdummy example, and modify it to use parallel computation with MPI.

MPI in Julia

`MPI.jl`[3] is a Julia wrapper library (just like PreCICE.jl) that provides an interface for common MPI implementations.

We first install MPI.jl into our Julia environment with either `add MPI` in the Pkg REPL or `import Pkg; Pkg.add("MPI")` in a Julia script. This will in addition to the package install and link against an MPI library. You can check the downloaded MPI implementation with `MPI.MPI_LIBRARY` and `MPI.MPI_LIBRARY_VERSION`. It downloaded the MPICH library version 3.4.2 at the time of writing.

Configuration of MPI.jl

The version of MPI used by MPI.jl may not be compatible with the one that comes with preCICE. An error is thrown when we couple a solver using `MPICH` 3.4.2 and `preCICE` 2.2.0:

```
preCICE:ERROR:  Accepting a socket connection at 127.0.0.1:63010 failed with the system
error: boost::filesystem::rename: No such file or directory
```

[2]https://www.mpi-forum.org/docs/
[3]https://github.com/JuliaParallel/MPI.jl

It was fixed when `MPI.jl` was set to use the MPI version on the system where this test was run (Debian OpenMPI 4.0.3). It can be done by setting an environmental variable and rebuilding MPI.jl, in short with the following command line:

```
julia -e 'ENV["JULIA_MPI_BINARY"]="system"; using Pkg; Pkg.build("MPI"; verbose=true)'
```

For further and more default information, consider the configuration section[4] of `MPI.jl`'s documentation.

5.2.3. Writing the solver

We can write a similar solverdummy as in 5.1, extend it to use MPI Communication:

```
using PreCICE
using MPI

MPI.Init()
comm = MPI.COMM_WORLD

commRank = MPI.Comm_rank(comm)
commSize = MPI.Comm_size(comm)

PreCICE.createSolverInterface("SolverOne", "./precice-config.xml", commRank, commSize)
```

In the previous solverdummy example, the rank was 0 and the communication size was 1, since we were not using parallelization and ran the solver for on one process. Here we first initiate an MPI computation and create `comm`, the communicator handle, that indentifies processes. There is nothing special, as we just use the basic MPI functions.

As for coupling to preCICE, we create in the current process a SolverInterface instance and specify our rank `commRank` and the total size `commSize`. All processes of `SolverOne` need to couple as `"SolverOne"`.

For a demonstration of computing distributed data, we create the vertices array containing their coordinates, and offset them by the rank so each process has its own "segment":

```
for i in 1:numberOfVertices, j in 1:dimensions
    offset = commRank * numberOfVertices
    vertices[ j + dimensions * (i-1)] = i-1 + offset
end
```

[4]https://juliaparallel.github.io/MPI.jl/latest/configuration/

That creates for rank 0 the array $[0, 0, 0, 1, 1, 1, 2, 2, 2]$, for rank 1 the array $[3, 3, 3, 4, 4, 4, 5, 5, 5]$ and so on.

At the end of the solver, we can add `MPI.Finalize()` to clean up MPI related state, but `MPI.jl` will call it either way when a Julia session ends.

The resulting solverdummy.jl that can be started as either as `SolverOne` or `SolverTwo` can be found at A.6.

Configuration

Regarding the configuration of preCICE, we can write and use the same `precice-config.xml` as in 5.1.

Running the coupling

We start the solver in N parallel processes with an MPI launcher, common are `mpirun, mpiexec`:

```
$ mpirun -n 3 julia solverdummy-parallel.jl config.xml SolverOne
preCICE: This is preCICE version 2.2.0
preCICE: Revision info: v2.2.0
preCICE: I am participant "SolverOne"
preCICE: Connecting Master to 2 Slaves
preCICE: Connecting Slave #0 to Master
preCICE: Connecting Slave #1 to Master
preCICE: Setting up master communication to coupling partner/s
preCICE: Setting up master communication to coupling partner/s
preCICE: Setting up master communication to coupling partner/s
```

As can be seen from the log, when we start a solver in 3 parallel processes, process 0, called here the master will wait until the other two, called Slaves, will connect, before they start communicating with the other solver. The Master-Slave division is made for the mesh re-partitioning algorithm to build the M:N communication later.

`MPI.jl` comes with is own `mpiexecjl` executable, that is just similar to `mpirun` or `mpiexec` but can take the `--project` argument, to start the Julia processes in a local Julia project environment.

To use the MPI implementation (if changed in the configuration of `MPI.jl`) of the current Julia project environment (similar if a "mpirunpy" would be venv or pipenv aware).

5.3. Using Julia's native parallelization

While the last section provided solutions to write Julia solvers that work on distributed data with MPI, it is not a default paradigm or feature of Julia. Julia provides primitives for distributed computing at every level:

The shared-memory parallelization of Julia is provided with the standard `Base.Threads` package for multi-threading. However, to initialize and use preCICE's parallelism we need separate memory spaces, so for example, starting a Julia script with `julia --threads n solvers.jl` and running

```
using Base.Threads
solvers = ["SolverOne", "SolverTwo"]
@threads for i = 1:2
    createSolverInterface(solvers[i], "precice-config.xml", i-1, 2)
end
```

would not work. We are particularly interested in Julia's process-level parallelism, which is provided with the standard `Distributed` module.

5.3.1. The Distributed module

`Distributed` is its own solution of managing processes and different than MPI. Let us shortly discuss its main concepts from the documentation[5].

There is one master process with id 1, managing all other processes that can be created on local or remote machines. The other processes but the master are called workers with ids 2 or higher. The communication appears to be "one-sided", as workers do not synchronize their state on their own (module or method definitions, global variables, and so on), the master has to explicitly load a module on each of them for example. For the rest of this chapter, we refer to the group containing the master and the workers together as "processes".

This "one-sided" approach becomes convincing when examining the`remote references` and `remote call` primitives. We do not need to work or inspect them directly, but Distributed is built on them. The former allow referencing data that is actually stored on another process; the latter are requests to execute a function on any other or on a particular process. We can already guess what convenience they in combination can provide, and let us see the higher-level concepts.

[5]https://docs.julialang.org/en/v1/manual/distributed-computing/

By starting the Julia REPL or script with "`julia -p k` (not `-t`, that starts threads) or by running `addprocs(k)` we have k workers ready to use. It is advised to set `k` to the number of logical cores on the machine. The `@spawnat p <expression>` macro evaluates an expression on process p, but when the symbol :any is passed as p, Julia will run it on an unspecified process:

```
using Distributed

remote_ref = @spawnat :any createABigSquareMatrix() # ref is now stored at some worker

det = @spawnat :any computeDeterminant(fetch(remote_ref)) # fetch is for copying locally
```

The second `@spawnat` does not copy `remote_ref` it to where it is called, it moves it directly to where `:any` will be, but in this case `@spawn` choose the process where `ref` is already stored.

So Julia's parallel computing is less about referencing specific processes, but we can still technically employ a parallel Julia solver in a preCICE coupling.

5.3.2. Writing the solver

While the master/worker division appears more "one-sided", we can create a solver-dummy, that works on his distributed data very similar to the MPI parallelism solution from the previous section.

Our practical considerations: We modify the resulting `solverdummy.jl` file from 5.1 to be started with `julia solverdummy.jl N ./precice-config.xml SolverName`, what launches N-1 additional workers and run the solverdummy code on N different processes, including the master process.

Our theoretical considerations: We tell every process of the solver to create the `SolverInterface` object and pass the total size N and the id of the process. This works analogous to the MPI example, instead of the rank id we pass the id of the master/worker process.

So first we import `PreCICE` and `Distributed` and define the parameters of the solver:

```
using Distributed
numberOfProcs = 10

ConfigFileName = "./precice-config-parallel.xml"
SolverName = "SolverOne"
MeshName = "MeshOne"
addprocs(numberOfProcs - 1)
```

We set the total number of processes to an arbitrary 10, and create for that 9 additional workers with `addprocs` that comes with `Distributed`. Note that all this is executed on the master process and in its scope.

Now comes the solver part, that is executed on every process:

```
@everywhere begin

using PreCICE
```

A **`begin`** `...` **`end`** block is just a way to bind multiple expressions into one. The `@everywhere` macro from `Distributed` is evaluating on all processes, including master, the expression after it, so our code inside **`begin`**.

On every process, we set the participant and rank specific parameters:

```
using PreCICE

commRank = myid() - 1
commSize = nprocs()

configFileName = $ConfigFileName
solverName = $SolverName
meshName = $MeshName
```

`commRank` is set to `myid()`, the process id, and decremented by one, because Julia uses 1-based indexing (master has id 1, and workers have 2 to n). preCICE is expecting to get ranks (numbered 0 to $n-1$) as in the MPI standard and will throw an error on the last process with id 10, stating that the size (`nprocs()` yields 10) cannot be less than a rank.

When we want to define the variables `configFileName`, `solverName` and `meshName`, we cannot use variables defined outside of `@everywhere` since the macro does not copy them to other processes, and they are not defined there. So we use the interpolation operator `$`, that does make the values of the variables as a part of the **`begin`** expression. The reason is that otherwise, `myid()` would return on every process 0, if not being strict to remote context.

After that, in every Julia process we create the `SolverInterface` object:

```
PreCICE.createSolverInterface(solverName, configFileName, commRank, commSize)

meshID = PreCICE.getMeshID(meshName)
dimensions = PreCICE.getDimensions()
...
```

Further from here on there is nothing different than in the solverdummy using MPI. The complete solver can be found in A.7. This example reproduced the SPMD paradigm of the MPI standard by running the computations with `@everywhere`. Figure 5.2 visualizes the Julia processes in a M:N coupling.

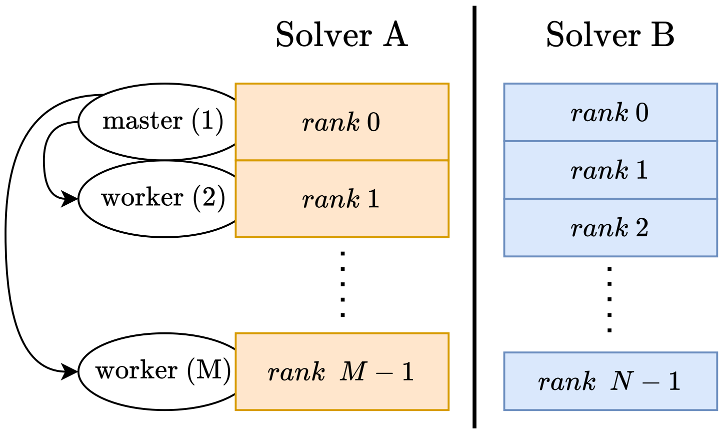

Figure 5.2.: Julia processes coupled as ranks; (n) denotes worker/master id

In the context of preCICE, there exists also a conception of a "master", it is the rank that is coupled as "0", whereas the other ranks of the same participant are called "slaves".

The differentiation is made in preCICE for its mesh re-partitioning algorithm. We want to add that the two meanings of "masters" are not related in any way, and the Julia master process here coupling as the first rank 0 is arbitrary. The Julia master process is not even required to participate at all, important is that every process of Julia participating in the coupling creates a SolverInterface to be included in the process mapping and calls further API methods of preCICE on them.

For a workers coupling only, we can use `@everywhere [...] <expr>`, where `[...]` is a subset of processes:

```
Using Distributed

addprocs(numberWorkers; exeflags="--project")

@everywhere workers() begin

using PreCICE

commRank = myid() - 2
commSize = nworkers()
...
```

where `workers()` yields all ids of the worker processes, and we decrement them by 2 since now our first rank 0, the worker, has an id of 2. We also pass the total communication size as `nworkers()` instead of `nprocs()`. This frees the master process and leads to the relation described in Figure 5.3.

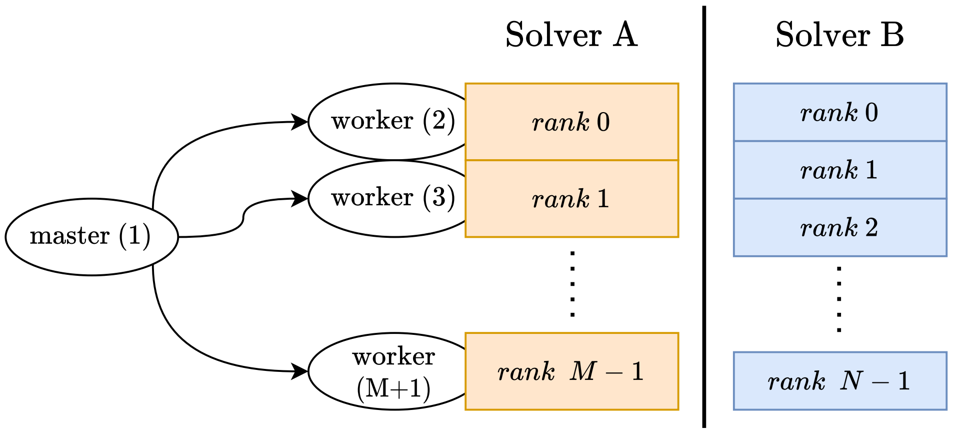

Figure 5.3.: Native parallelism with Julia worker processes only

And again, this is not limited to freeing the master process only. This means that a subset of Julia processes can connect as a participant to preCICE, the master and another set of processes left out. The full script for the workers only version can be found in appendix A.8.

This arrangement becomes very interesting; if the master process adds N additional processes, a Julia solver could couple as both participants, a coupling managed from one Julia script. Consider Figure 5.4, the "start" script, has all the control over what to execute where, as Julia's `Distributed` paradigm intended. We are reproducing the MIMD (Multiple Instruction Multiple Data) structure.

```
using Distributed

solverOne = ["SolverOne", "conf.xml", "DataOne", "MeshOne", 2:M]
solverTwo = ["SolverTwo", "conf.xml", "DataTwo", "MeshTwo", (M+1):(N+M)]

addprocs(M+N)
@everywhere using PreCICE

# SolverOne
@everywhere 2:M begin
    conf = $solverOne
    createSolverInterface(conf[1], conf[2], myid()-conf[end][1], conf[end][end])
    data_id = getMeshID(conf[3])
    mesh_id = getMeshID(conf[4])
    ...
end

# SolverTwo
@everywhere (M+1):(N+M) begin
    conf = $solverTwo
    createSolverInterface(conf[1], conf[2], myid()-conf[end][1], conf[end][end])
    data_id = getMeshID(conf[3])
    mesh_id = getMeshID(conf[4])
    ...
end
...
```

Figure 5.4.: Writing distributed coupling setup from one Julia file

Furthermore, we are not limited to process-level parallelism, as masters and workers can start threads, for example with `julia -p n -t k` or with `addprocs(n, execflags="--threads k")` starts n workers with k threads each, so a single rank of a Julia process could still use all the multi-threading functionality.

One thing that derivates from the MPI setup is the configuration.

5.3.3. Configuration for native parallelism

While the solver from looked very similar from the setup of MPI, we have to make changes to the configuration of the coupling. The way the intra-participant communication is built in preCICE, it is desirable to reuse a MPI communicator of a participant, if available. So by default, preCICE uses the global communicator `MPI_COMM_WORLD`, but when we use Julia's native parallelism it is not present, and an error will be thrown.

For this case preCICE also supports building its intra-participant communication with TCP/IP sockets. We add to the participant that uses Julia's native parallelism the `<master:sockets/>` tag in the `precice-configuration.xml`:

```
<participant name="SolverOne">
      <master:sockets/>
      <use-mesh name="MeshOne" provide="yes"/>
      <write-data name="dataOne" mesh="MeshOne" />
      <read-data  name="dataTwo" mesh="MeshOne" />
</participant>
```

6. Conclusions and Outlook

This thesis looked into two complementary open-source projects: Julia, being a high-level programming language bringing a fresh new approach to Scientific Computing, and preCICE, a state-of-the-art multi-physics coupling library for scientific discoveries in various important and advanced fields.

preCICE's API for coupling single-physics solvers in a multi-physics simulation is available for a number of different languages through language bindings. We researched ways to access the library through calls from Julia and presented two implementations. While CxxWrap.jl provided a template solution to wrap C++ libraries as Julia packages, an independent native solution could be developed due to preCICE's C bindings being a part of the core library and Julia's native support for calling to C libraries and its managing of C-Julia type conversions. The resulting Julia package PreCICE.jl was tested on dummy solvers, affirming a seamless integration of preCICE's basic API and parallelization using the package MPI.jl into the world of Julia. Finally, the Julia in-house features for distributed/parallel computing were explored upon compatibility with preCICE, where the workers of the Distributed package were employed to connect in a preCICE coupling setup as distributed solver processes, again providing at least a functional success, owing to preCICE upholding alternative options for solvers using a closed-source or modified communication in its configuration.

From a software developing perspective, this thesis provided an outline of how to port a full C or C++ library to Julia, summarizing available directions and multiple options into one of a single documented guide. For further development of Julia bindings for any C/C++ library, one is now able to build upon the outline from this thesis instead of starting from researching the options again.

The resulting Julia bindings brought the coupling library of preCICE into Julia's scope, making the creation of partitioned multi-physics simulations in Julia a viable option. Julia was intended to bring abstraction into the field of numerical computing, which translates into the field of multi-physics as users of preCICE now have a new paradigm ready for exploitation. For the open-source project of preCICE itself, the Julia bindings could serve as a new individual software package having its place besides the other languages rather than being an alternative, as the bindings seem to have integrated themselves into preCICE frictionless and without bitter compromises. For example, Julia's C interfacing has its limitations in certain areas, but these use cases never applied for preCICE; the parallelization provided its own Message Passing Interface, but coincidentally they offered everything preCICE needed; Julia supports distributing binaries through JLL packages, and all dependencies of preCICE are available as JLLs too, making a future preCICE_jll possible to compile, leading to install PreCICE.jl in the Julia REPL the simplest way to install preCICE and to get their hand on it (quicker than the quickstart[1] on preCICE's website!).

However, while this thesis provides the first code and working examples with arbitrary data, performance tests and real coupling scenarios using the API in action is further work that needs to be pursued to fully confirm the working state of the Julia bindings.

Same applies for the native parallelization of Julia on multiple points. The coupling as a distributed participant worked, but if there is any advantage to it instead of using `MPI.jl` remains until the end of this thesis an open question. There needs to be a further investigation of the "idea" of the Distributed package and how it fits within preCICE. The technical aspect too, as the ids of Julia processes use 1-based indexing, what makes the `createSolverInterface(solverName, config, myid()-offset, nprocs())` seem "hacky". If preCICE wants to allow a solver using its own version of MPI, then perhaps for languages such as Julia a more generic way to register as ranks of a participant could be considered, but for now it is only a matter of changing up values.

[1]https://precice.org/quickstart.html

A. Complete Code

A.1. Ferrite.jl Heat Transfer Solver

```
using Ferrite, SparseArrays # importing library methods into scope with 'using'
# create domain
grid = generate_grid(Quadrilateral, (100, 100))
dim = 2
ip = Lagrange{dim, RefCube, 1}()
qr = QuadratureRule{dim, RefCube}(2)
cellvalues = CellScalarValues(qr, ip)
dh = DofHandler(grid)
push!(dh, :u, 1)
close!(dh)
K = create_sparsity_pattern(dh)
M = create_sparsity_pattern(dh)
f = zeros(ndofs(dh))
function doassemble_K!(K::SparseMatrixCSC, f::Vector,
                       cellvalues::CellScalarValues{dim},
                       dh::DofHandler) where {dim}
    n_basefuncs = getnbasefunctions(cellvalues)
    Ke = zeros(n_basefuncs, n_basefuncs)
    fe = zeros(n_basefuncs)
    assembler = start_assemble(K, f)
    @inbounds for cell in CellIterator(dh)
        fill!(Ke, 0)
        fill!(fe, 0)
        reinit!(cellvalues, cell)
        for q_point in 1:getnquadpoints(cellvalues)
            dΩ = getdetJdV(cellvalues, q_point)
            for i in 1:n_basefuncs
                v  = shape_value(cellvalues, q_point, i)
                v = shape_gradient(cellvalues, q_point, i)
                fe[i] += v * dΩ
                for j in 1:n_basefuncs
                    u = shape_gradient(cellvalues, q_point, j)
                    Ke[i, j] += (v  u) * dΩ
                end
            end
        end
        assemble!(assembler, celldofs(cell), fe, Ke)
    end
    return K, f
end

function doassemble_M!(M::SparseMatrixCSC,
                       cellvalues::CellScalarValues{dim},
                       dh::DofHandler) where {dim}

    n_basefuncs = getnbasefunctions(cellvalues)
    Me = zeros(n_basefuncs, n_basefuncs)
    assembler = start_assemble(M)
    @inbounds for cell in CellIterator(dh)
        fill!(Me, 0)
        reinit!(cellvalues, cell)
        for q_point in 1:getnquadpoints(cellvalues)
            dΩ = getdetJdV(cellvalues, q_point)

            for i in 1:n_basefuncs
                v  = shape_value(cellvalues, q_point, i)
                for j in 1:n_basefuncs
                    u = shape_value(cellvalues, q_point, j)
                    Me[i, j] += (v  u) * dΩ
                end
            end
        end
        assemble!(assembler, celldofs(cell), Me)
    end
    return M
end

function saveData(u, t)
    vtk_grid("transient-heat-$t", dh) do vtk
        vtk_point_data(vtk, dh, u)
        vtk_save(vtk)
    end
end

# simulation parameters
max_temp = 100
Δt  = 1
T = 200
# define dirichlet boundary conditions on top and bottom edge of the domain
ch = ConstraintHandler(dh)
u_d(x,t) = 0
∂Ω₁ = union(getfaceset.((grid, ), ["top", "bottom"])...)
dbc = Dirichlet(:u, , u_d)
add!(ch, dbc)
# temperature on left boundary is described as function a
```

```
a(x,t) = t*max_temp / T
∂Ω₂ = union(getfaceset.((grid, ), ["left"])...)
dbc = Dirichlet(:u, ∂Ω₂, a)
add!(ch, dbc)
close!(ch)
update!(ch, 0.0)
# assemble linear system
K, f = doassemble_K!(K, f, cellvalues, dh)
M = doassemble_M!(M, cellvalues, dh)
A = (Δt .* K) + M  # system matrix A
rhsdata = get_rhs_data(ch, A)
uₙ = zeros(length(f))   # initialize first timestep
apply!(A, ch)   # apply boundary conditions
@time for t in 0:Δt:T
    update!(ch, t)  # update boundary condition values
    b = Δt .* f .+ M * uₙ    # compute right-hand side
    apply_rhs!(rhsdata, b, ch)  # apply boundary conditions of the current time step

    u = A \ b # solve timestep
    uₙ .= u # update solution
    saveData(u, t)  # write output file
end
```

A.2. CxxWrap.jl CMakeLists.txt for building the C++ wrapper library

```
project(Juliaprecice)
cmake_minimum_required(VERSION 2.8.12)
set(CMAKE_MACOSX_RPATH 1)
set(JlCxx_DIR /path/to/libprecice.soDir/)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/lib")
find_package(precice REQUIRED CONFIG)
find_package(JlCxx)
get_target_property(JlCxx_location JlCxx::cxxwrap_julia LOCATION)
get_filename_component(JlCxx_location ${JlCxx_location} DIRECTORY)
set(CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib;${JlCxx_location}")
add_library(jlprecice SHARED juliaprecice.cpp)
target_link_libraries(jlprecice precice::precice)
target_link_libraries(jlprecice JlCxx::cxxwrap_julia)
install(TARGETS
  jlprecice
LIBRARY DESTINATION lib
ARCHIVE DESTINATION lib
RUNTIME DESTINATION lib)
```

A.3. C++ side part of CxxWrap.jl

```
#include "jlcxx/jlcxx.hpp"
#include <precice/SolverInterface.hpp>


JLCXX_MODULE define_julia_module(jlcxx::Module& mod)
{
    using namespace precice;
    mod.method("getVersionInformation", &getVersionInformation);
    mod.add_type<SolverInterface>("SolverInterface")
        .constructor<const std::string, const std::string, int, int>()
        .constructor<const std::string, const std::string, int, int, void*>()
        // Steering Methods
        .method("initialize", &SolverInterface::initialize)
        .method("initializeData", &SolverInterface::initializeData)
        .method("advance", &SolverInterface::advance)
        .method("finalize", &SolverInterface::finalize)
        // Status Queries
        .method("getDimensions", &SolverInterface::getDimensions)
        .method("isCouplingOngoing", &SolverInterface::isCouplingOngoing)
        .method("isReadDataAvailable", &SolverInterface::isReadDataAvailable)
        .method("isWriteDataRequired", &SolverInterface::isWriteDataRequired)
        .method("isTimeWindowComplete", &SolverInterface::isTimeWindowComplete)
        // Action Methods
        .method("isActionRequired", &SolverInterface::isActionRequired)
        .method("markActionFulfilled", &SolverInterface::markActionFulfilled)
        // Mesh Access
        .method("hasMesh", &SolverInterface::hasMesh)
        .method("getMeshID", &SolverInterface::getMeshID)
        .method("setMeshVertex", &SolverInterface::setMeshVertex)
        .method("getMeshVertexSize", &SolverInterface::getMeshVertexSize)
        .method("setMeshVertices", &SolverInterface::setMeshVertices)
        .method("getMeshVertices", &SolverInterface::getMeshVertices)
        .method("getMeshVertexIDsFromPositions",
                &SolverInterface::getMeshVertexIDsFromPositions)
        .method("setMeshEdge", &SolverInterface::setMeshEdge)
        .method("setMeshTriangle", &SolverInterface::setMeshTriangle)
        .method("setMeshTriangleWithEdges", &SolverInterface::setMeshTriangleWithEdges)
        .method("setMeshQuad", &SolverInterface::setMeshQuad)
        .method("setMeshQuadWithEdges", &SolverInterface::setMeshQuadWithEdges)
        // Data Access
        .method("hasData", &SolverInterface::hasData)
        .method("getDataID", &SolverInterface::getDataID)
        .method("mapReadDataTo", &SolverInterface::mapReadDataTo)
        .method("mapWriteDataFrom", &SolverInterface::mapWriteDataFrom)
        .method("writeBlockVectorData", &SolverInterface::writeBlockVectorData)
        .method("writeVectorData", &SolverInterface::writeVectorData)
        .method("writeBlockScalarData", &SolverInterface::writeBlockScalarData)
        .method("writeScalarData", &SolverInterface::writeScalarData)
        .method("readBlockVectorData", &SolverInterface::readBlockVectorData)
        .method("readVectorData", &SolverInterface::readVectorData)
        .method("readBlockScalarData", &SolverInterface::readBlockScalarData)
        .method("readScalarData", &SolverInterface::readScalarData)
    ;
    mod.method("actionWriteInitialData", &constants::actionWriteInitialData);
    mod.method("actionWriteIterationCheckPoint",
                &constants::actionWriteIterationCheckpoint);
    mod.method("actionReadIterationCheckPoint",
                &constants::actionReadIterationCheckpoint);
}
```

A.4. Solverdummy configuration file

```
<?xml version="1.0" encoding="UTF-8" ?>
<precice-configuration>
  <log>
    <sink
      type="stream"
      output="stdout"
      filter="%Severity% > debug"
      format="preCICE:%ColorizedSeverity% %Message%"
      enabled="true" />
  </log>
  <solver-interface dimensions="3">
    <data:vector name="dataOne" />
    <data:vector name="dataTwo" />
    <mesh name="MeshOne">
      <use-data name="dataOne" />
      <use-data name="dataTwo" />
    </mesh>
    <mesh name="MeshTwo">
      <use-data name="dataOne" />
      <use-data name="dataTwo" />
    </mesh>
    <participant name="SolverOne">
      <use-mesh name="MeshOne" provide="yes" />
      <write-data name="dataOne" mesh="MeshOne" />
      <read-data name="dataTwo" mesh="MeshOne" />
    </participant>
    <participant name="SolverTwo">
      <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-neighbor
        direction="read"
        from="MeshOne"
        to="MeshTwo"
        constraint="consistent" />
      <write-data name="dataTwo" mesh="MeshTwo" />
      <read-data name="dataOne" mesh="MeshTwo" />
    </participant>
    <m2n:sockets from="SolverOne" to="SolverTwo" />
    <coupling-scheme:serial-implicit>
      <participants first="SolverOne" second="SolverTwo" />
      <max-time-windows value="2" />
      <time-window-size value="1.0" />
      <max-iterations value="2" />
      <min-iteration-convergence-measure
        min-iterations="5"
        data="dataOne"
        mesh="MeshOne" />
      <exchange data="dataOne" mesh="MeshOne" from="SolverOne" to="SolverTwo" />
      <exchange data="dataTwo" mesh="MeshOne" from="SolverTwo" to="SolverOne" />
    </coupling-scheme:serial-implicit>
  </solver-interface>
</precice-configuration>
```

A.5. Code for solverdummy.jl

```
using PreCICE
commRank = 0
commSize = 1
if size(ARGS, 1) < 2
    println("ERROR: pass config path, solver name and mesh name, example:
    julia solverdummy.jl ./precice-config.xml SolverOne MeshOne")
    exit(1)
end
configFileName = ARGS[1]
solverName = ARGS[2]
if solverName == "SolverOne"
    meshName = "MeshOne"
    dataWriteName = "dataOne"
    dataReadName  = "dataTwo"
```

```julia
else
    meshName = "MeshTwo"
    dataReadName  = "dataOne"
    dataWriteName = "dataTwo"
end
println("""DUMMY: Running solver dummy with preCICE config file
"$configFileName", participant name "$solverName", and mesh name "$meshName" """)
PreCICE.createSolverInterface(solverName, configFileName, commRank, commSize)
meshID = PreCICE.getMeshID(meshName)
dimensions = PreCICE.getDimensions()
numberOfVertices = 3
readDataID  = PreCICE.getDataID(dataReadName, meshID)
writeDataID = PreCICE.getDataID(dataWriteName, meshID)

readData = zeros(numberOfVertices * dimensions)
writeData = zeros(numberOfVertices * dimensions)
vertices = Array{Float64, 1}(undef, numberOfVertices * dimensions)

# create array of vertices v_i = (i,i,i)
for i in 1:numberOfVertices, j in 1:dimensions
        vertices[ j + dimensions * (i-1)] = i
end

vertexIDs = PreCICE.setMeshVertices(meshID, numberOfVertices, vertices)

let # setting local scope for dt outside of the while loop
dt = PreCICE.initialize()
while PreCICE.isCouplingOngoing()
    if PreCICE.isActionRequired(PreCICE.actionWriteIterationCheckpoint())
        println("DUMMY: Writing iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionWriteIterationCheckpoint())
    end
    if PreCICE.isReadDataAvailable()
        PreCICE.readBlockVectorData(readDataID, numberOfVertices, vertexIDs, readData)
    end
    for i in 1:(numberOfVertices * dimensions)
        writeData[i] = readData[i] + 1.0
    end
    if PreCICE.isWriteDataRequired(dt)
        PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, writeData)
    end
    dt = PreCICE.advance(dt)

    if PreCICE.isActionRequired(PreCICE.actionReadIterationCheckpoint())
        println("DUMMY: Reading iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionReadIterationCheckpoint())
    else
        println("DUMMY: Advancing in time")
    end
```

```
end # while
end # let
PreCICE.finalize()
println("DUMMY: Closing Julia solver dummy...")
```

A.6. Parallel solverdummy.jl with MPI Communication

```
using PreCICE
using MPI

if size(ARGS, 1) < 2
    println("ERROR: pass config path, solver name,
        example: julia solverdummy.jl ./precice-config.xml SolverOne")
    exit(1)
end

MPI.Init()
comm = MPI.COMM_WORLD
commRank = MPI.Comm_rank(comm)
commSize = MPI.Comm_size(comm)

configFileName = ARGS[1]
solverName = ARGS[2]
# set meshName depending on solverName
if solverName == "SolverOne"
    meshName = "MeshOne"
else
    meshName = "MeshTwo"
end

println("""DUMMY ($(MPI.Comm_rank(comm))): Running solver dummy with preCICE config file
        "$configFileName", participant name "$solverName", and mesh name "$meshName" """)
PreCICE.createSolverInterface(solverName, configFileName, commRank, commSize)

meshID = PreCICE.getMeshID(meshName)
dimensions = PreCICE.getDimensions()

numberOfVertices = 1

if solverName == "SolverOne"
    dataWriteName = "dataOne"
    dataReadName  = "dataTwo"
else
    dataReadName  = "dataOne"
    dataWriteName = "dataTwo"
```

```
end

readDataID  = PreCICE.getDataID(dataReadName, meshID)
writeDataID = PreCICE.getDataID(dataWriteName, meshID)

readData = zeros(numberOfVertices * dimensions)
writeData = zeros(numberOfVertices * dimensions)

vertices = Array{Float64, 1}(undef, numberOfVertices * dimensions)
# create different vertices coordinates for different procs
for i in 1:numberOfVertices, j in 1:dimensions
    offset = commRank * numberOfVertices
    vertices[j + dimensions * (i-1)] = i-1 + offset
end

vertexIDs = PreCICE.setMeshVertices(meshID, numberOfVertices, vertices)

let # setting local scope for dt outside of the while loop
dt = PreCICE.initialize()

while PreCICE.isCouplingOngoing()
    if PreCICE.isActionRequired(PreCICE.actionWriteIterationCheckpoint())
        #println("""DUMMY ($(MPI.Comm_rank(comm))): Writing iteration checkpoint""")
        PreCICE.markActionFulfilled(PreCICE.actionWriteIterationCheckpoint())
    end
    if PreCICE.isReadDataAvailable()
        PreCICE.readBlockVectorData(readDataID, numberOfVertices, vertexIDs, readData)
    end
    for i in 1:(numberOfVertices * dimensions)
        writeData[i] = readData[i] + 1.0
    end
    if PreCICE.isWriteDataRequired(dt)
        PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, writeData)
    end
    dt = PreCICE.advance(dt)
    if PreCICE.isActionRequired(PreCICE.actionReadIterationCheckpoint())
        println("""DUMMY ($(MPI.Comm_rank(comm))): Reading iteration checkpoint""")
        PreCICE.markActionFulfilled(PreCICE.actionReadIterationCheckpoint())
    else
        println("""DUMMY ($(MPI.Comm_rank(comm))): Advancing in time""")
    end
end # while
end # let
PreCICE.finalize()

println("""DUMMY ($(MPI.Comm_rank(comm))): Closing Julia solver dummy...""")
```

A.7. Solverdummy using native parallelism

```julia
using Distributed

if size(ARGS, 1) < 3
    println("ERROR: pass total processes number N, config path, solver name and
    mesh name, example: julia solverdummy.jl 5 ./precice-config.xml SolverOne")
    exit(1)
end

numberWorkers = parse(Int, ARGS[1]) - 1
ConfigFileName = ARGS[2]
SolverName = ARGS[3]

# add --project flag if PreCICE is installed in a local Julia environment
addprocs(numberWorkers)

@everywhere begin
using PreCICE

commRank = myid() - 1
commSize = nprocs()

configFileName = $ConfigFileName
solverName = $SolverName
# set meshName depending on solverName
if solverName == "SolverOne"
    meshName = "MeshOne"
else
    meshName = "MeshTwo"
end

println("""DUMMY ($commRank): Running solver dummy with preCICE config file
"$configFileName", participant name "$solverName", and mesh name "$meshName" """)

PreCICE.createSolverInterface(solverName, configFileName, commRank, commSize)

meshID = PreCICE.getMeshID(meshName)
dimensions = PreCICE.getDimensions()
numberOfVertices = 1

if solverName == "SolverOne"
    dataWriteName = "dataOne"
    dataReadName  = "dataTwo"
else
    dataReadName  = "dataOne"
    dataWriteName = "dataTwo"
end
```

```
readDataID  = PreCICE.getDataID(dataReadName, meshID)
writeDataID = PreCICE.getDataID(dataWriteName, meshID)
writeData = zeros(numberOfVertices * dimensions)
readData = zeros(numberOfVertices * dimensions)

vertices = Array{Float64, 1}(undef, numberOfVertices * dimensions)

# create different vertices coordinates for different procs
for i in 1:numberOfVertices, j in 1:dimensions
    offset = commRank * numberOfVertices
    vertices[j + dimensions * (i-1)] = i-1 + offset
end
let # setting local scope for dt outside of the while loop
vertexIDs = PreCICE.setMeshVertices(meshID, numberOfVertices, vertices)

dt = PreCICE.initialize()
while PreCICE.isCouplingOngoing()
    if PreCICE.isActionRequired(PreCICE.actionWriteIterationCheckpoint())
        println("DUMMY ($commRank): Writing iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionWriteIterationCheckpoint())
    end
    if PreCICE.isReadDataAvailable()
        PreCICE.readBlockVectorData(readDataID, numberOfVertices, vertexIDs, readData)
    end
    writeData = readData .+ 1.0
    if PreCICE.isWriteDataRequired(dt)
        PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, writeData)
    end
    dt = PreCICE.advance(dt)
    if PreCICE.isActionRequired(PreCICE.actionReadIterationCheckpoint())
        println("DUMMY ($commRank): Reading iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionReadIterationCheckpoint())
    else
       println("DUMMY ($commRank): Advancing in time")
    end
end # while
end # let
PreCICE.finalize()
println("DUMMY ($commRank): Closing Julia solver dummy...")

end
```

A.8. Native parallelism solverdummy, workers only

```julia
using Distributed
if size(ARGS, 1) < 3
    println("ERROR: pass total processes number N, config path, solver name and mesh
    name, example: julia solverdummy.jl 5 ./precice-config.xml SolverOne")
    exit(1)
end
numberWorkers = parse(Int, ARGS[1])
ConfigFileName = ARGS[2]
SolverName = ARGS[3]
# add --project flag if PreCICE is installed in a local Julia environment
addprocs(numberWorkers; exeflags="--project")

@everywhere workers() begin
using PreCICE

commRank = myid() - 2
commSize = nworkers()
configFileName = $ConfigFileName
solverName = $SolverName

# set meshName depending on solverName
if solverName == "SolverOne"
    meshName = "MeshOne"
else
    meshName = "MeshTwo"
end

println("""DUMMY ($commRank): Running solver dummy with preCICE config file
"$configFileName", participant name "$solverName", and mesh name "$meshName" """)
PreCICE.createSolverInterface(solverName, configFileName, commRank, commSize)
meshID = PreCICE.getMeshID(meshName)
dimensions = PreCICE.getDimensions()
dataWriteName = nothing
dataReadName = nothing

numberOfVertices = 1
if solverName == "SolverOne"
    dataWriteName = "dataOne"
    dataReadName  = "dataTwo"
end

if solverName == "SolverTwo"
    dataReadName  = "dataOne"
    dataWriteName = "dataTwo"
end
```

```julia
readDataID  = PreCICE.getDataID(dataReadName, meshID)
writeDataID = PreCICE.getDataID(dataWriteName, meshID)
writeData = zeros(numberOfVertices * dimensions)
readData = zeros(numberOfVertices * dimensions)
vertices = Array{Float64, 1}(undef, numberOfVertices * dimensions)
# create different vertices coordinates for different procs
for i in 1:numberOfVertices, j in 1:dimensions
    offset = commRank * numberOfVertices
    vertices[j + dimensions * (i-1)] = i-1 + offset
end
let # setting local scope for dt outside of the while loop
vertexIDs = PreCICE.setMeshVertices(meshID, numberOfVertices, vertices)
dt = PreCICE.initialize()
while PreCICE.isCouplingOngoing()
    if PreCICE.isActionRequired(PreCICE.actionWriteIterationCheckpoint())
        println("DUMMY ($commRank): Writing iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionWriteIterationCheckpoint())
    end
    if PreCICE.isReadDataAvailable()
        PreCICE.readBlockVectorData(readDataID, numberOfVertices, vertexIDs, readData)
    end
    writeData = readData .+ 1.0
    if PreCICE.isWriteDataRequired(dt)
        PreCICE.writeBlockVectorData(writeDataID, numberOfVertices, vertexIDs, writeData)
    end
    dt = PreCICE.advance(dt)

    if PreCICE.isActionRequired(PreCICE.actionReadIterationCheckpoint())
        println("DUMMY ($commRank): Reading iteration checkpoint")
        PreCICE.markActionFulfilled(PreCICE.actionReadIterationCheckpoint())
    else
       println("DUMMY ($commRank): Advancing in time")
    end
end # while
end # let scope
PreCICE.finalize()
println("DUMMY ($commRank): Closing Julia solver dummy...")
end
```

Bibliography

[1] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah. *Julia: A Fresh Approach to Numerical Computing*. 2015. arXiv: 1411.1607 [cs.MS] (cit. on pp. 12, 13).

[2] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman. *Julia: A Fast Dynamic Language for Technical Computing*. 2012. arXiv: 1209.5145 [cs.PL] (cit. on p. 13).

[3] H.-J. Bungartz, F. Lindner, B. Gatzhammer, M. Mehl, K. Scheufele, A. Shukaev, B. Uekermann. "preCICE – A fully parallel library for multi-physics surface coupling." In: *Computers and Fluids* 141 (2016). Advances in Fluid-Structure Interaction, pp. 250–258. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2016.04.003. URL: http://www.sciencedirect.com/science/article/pii/S0045793016300974 (cit. on pp. 11, 23).

[4] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H.-J. Bungartz, L. C. Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, O. Z. Koseomur. *preCICE v2: A Sustainable and User-Friendly Coupling Library*. 2021. arXiv: 2109.14470 [cs.MS] (cit. on pp. 22, 23).

[5] B. Gatzhammer. "Efficient and Flexible Partitioned Simulation of Fluid-Structure Interactions." Dissertation. Munich: Technical University of Munich, 2014 (cit. on p. 23).

[6] F. Lindner. *Data Transfer in Partitioned Multi-Physics Simulations: Interpolation Communication*. https://elib.uni-stuttgart.de/bitstream/11682/10598/3/Lindner%20-%20Data%20Transfer%20in%20Partitioned%20Multi-Physics%20Simulations.pdf. Dissertation. 2019 (cit. on p. 48).

[7] J. Regier, A. Miller, J. McAuliffe, R. Adams, M. Hoffman, D. Lang, D. Schlegel, Prabhat. *Celeste: Variational inference for a generative model of astronomical images*. 2015. arXiv: 1506.01351 [astro-ph.IM] (cit. on p. 13).

[8] B. W. Uekermann. "Partitioned Fluid-Structure Interaction on Massively Parallel Systems." Dissertation. Department of Informatics, Technical University of Munich, 2016. DOI: https://doi.org/10.14459/2016md1320661 (cit. on p. 48).

All links were last followed on October 19, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature