The coupling library for partitioned multi-physics simulations

Documentation version 2.2.1

Generated: September 24, 2021

preCICE is free/open-source software, using the GNU LGPL3 license. The code is publicly available and actively developed on GitHub.

Table of Contents

SU2

Couple your code

Step by step

Advanced topics

The preCICE documentation

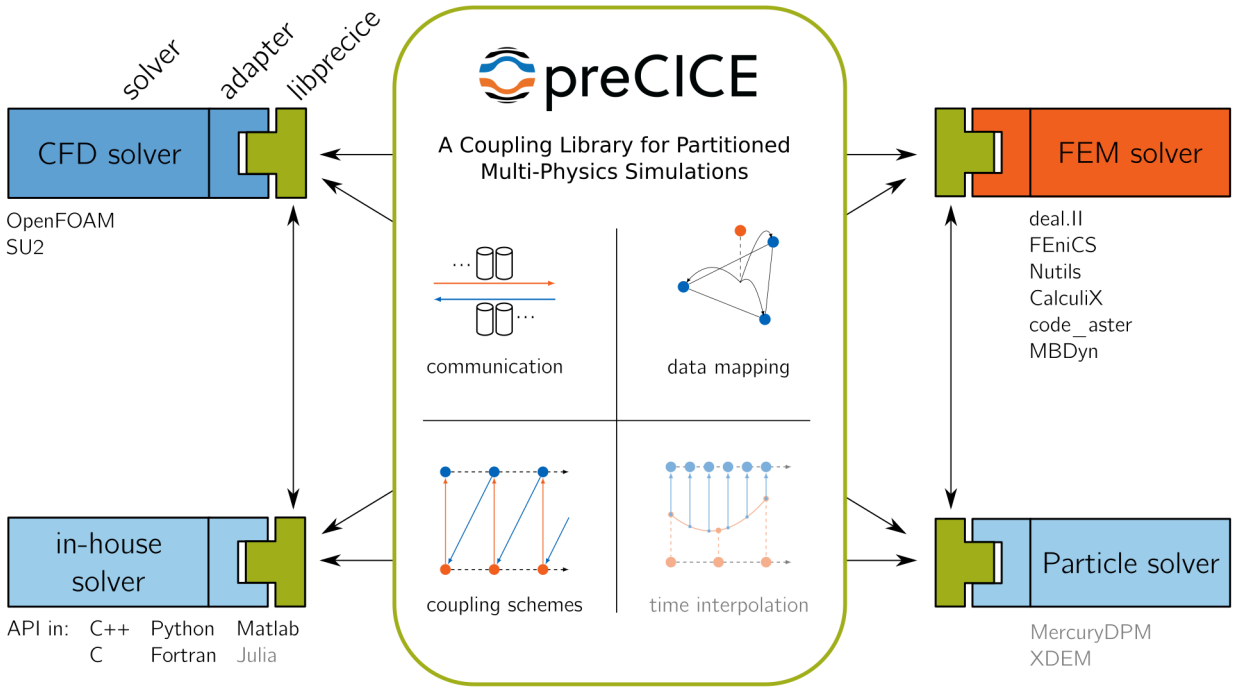
Summary: This page gives an overview of the complete preCICE documentation, including building, configuration, literature, the API, and many more.

The big picture

preCICE stands for Precise Code Interaction Coupling Environment. Its main component is a library that can be used for partitioned multi-physics simulations, including, but not restricted to fluid-structure interaction and conjugate heat transfer simulations. Partitioned (as opposite to monolithic) means that preCICE couples existing programs (solvers) which simulate a subpart of the complete physics involved in a simulation. This allows for the high flexibility that is needed to keep a decent time-to-solution for complex multi-physics scenarios, reusing existing components. preCICE runs efficiently on a wide spectrum of systems, from low-end laptops up to complete compute clusters and has proven scalability (page 8) on 10000s of MPI Ranks.

The preCICE library offers parallel communication means, data mapping schemes, and methods for transient equation coupling. Additionally, we are actively developing methods for time interpolation and more features (see our roadmap (page 10)). preCICE is written in C++ and offers additional bindings for C, Fortran, Python, and Matlab. Coupling your own solver is very easy, due to the minimally-invasive approach of preCICE. Once you add the (very few) calls to the preCICE library in your code, you can couple it with any other code at runtime. For well-known solvers such as OpenFOAM, deal.II, FEniCS, Nutils, CalculiX, or SU2, you can simply use one of our official adapters.

preCICE is free/open-source software, using the GNU LGPL3 license. This license ensures the open future of the project, while allowing you to use the library also in closed-source solvers. The code is publicly available and actively developed on GitHub.



Writing about preCICE? Get this image and more material.

Where to find what

This documentation explains how to use preCICE. We do not detail the numerical methods and HPC algorithms in the preCICE docs, but we refer to existing publications on preCICE for these topics. The literature guide (page 8) gives an overview of the most important preCICE literature.

The preCICE docs are organized in several sections:

- Installation (page 14): How to get and install preCICE on various systems.
- Configuration (page 62): At runtime, preCICE needs to be configured with an xml file. Here you learn how to do that.
- Tooling (page 161): Several helpful (but completely optional) tools around preCICE: tools for setting up your simulation, post-processing the results, and much more.
- Provided adapters (page 168): The preCICE community maintains ready-to-use adapters for many popular solvers. Here, you find the documentation of these adapters.
- Couple your code (page 222): Getting familiar with the preCICE API.
- Dev docs (page 0): References that developers use. Are you maybe also thinking of contributing (page 0)?

Before you start reading: there are just some preCICE-specific technical terms (page 6) that every user should read first.

Terminology

Summary: We often refer to the following terms, but they may not already be clear.

Partitioned approach

As already mentioned in the overview:

> Partitioned (as opposed to monolithic) means that preCICE couples existing programs (solvers) which simulate a subpart of the complete physics involved in a simulation.

The direct opposite is the (numerically) monolithic approach, in which the same software has to construct and solve a global system of equations for the complete domain.

There are several advantages and disadvantages in both approaches. The partitioned approach allows to reuse existing components, reducing the time from deciding to simulate a multi-physics scenario to getting accurate results (the real time to solution). It also allows to study different combinations of components that are already "experts" in each subdomain. The monolithic approach can have robustness and performance advantages in some cases, but with the current advanced partitioned coupling algorithms, the significance of any such difference should not always be taken for granted (see our literature guide (page 8)).

Solver and participant

By *solver*, we refer to a complete simulation code, which we want to couple. We do not mean a linear algebra solver. With *participant*, we refer to a solver in the context of a coupled simulation (e.g. "Fluid participant"). This term is also used in the preCICE configuration (page 62).

Library approach

preCICE follows a *library* approach. This basically means that preCICE is a library: each solver needs to call preCICE. This also means that preCICE runs in the same threads that the solvers run in. The opposite coupling approach is the *framework* approach. In that approach, the coupling tool calls all solvers, which need to implement a certain programming interface. The advantage of a library approach is that it is minimally invasive to the coupled codes. They do not need to be rewritten. Instead, you just need to insert the preCICE calls at the right places.

Peer-to-peer approach

preCICE also follows a peer-to-peer approach. If you already tried preCICE, you may have noticed that you only need to start all coupled solvers individually, in the same way you would start them to run each single-physics simulation. There is no other starting mechanism involved: no server-like coupling executable or anything similar.

Adapter

To call preCICE from your code, you need to call functions of the application programming interface of preCICE. You can directly do this in your code. In this case, you develop an adapted solver. The little software engineering purist in you prefers, however, to collect all calls to preCICE into one place. This could be a separate class or module in your code. This could also be a separate library, which you call from pre-defined callback hooks. We call this one place an *adapter*. Depending on the perspective, you would call it *preCICE adapter*, *MyCode adapter*, or *MyCode-preCICE adapter*; assuming that you want to couple a code named MyCode. preCICE comes with a few ready-to-use adapters (page 168). If you want to couple your own code, you basically want to develop an adapter for this code. Read more on adapter software engineering approaches (page 0).

Black-box coupling

preCICE also follows a *black-box* coupling approach. This is a numerical term. It means that preCICE treats the coupled solvers as black boxes. Only minimal information about these black boxes is available: what kind of data you can input, what you get as output, and how to repeat a timestep. At first, this is a drawback. With little information available, it is difficult to realize a robust coupling. That is why preCICE provides quite some numerical methods to overcome this hurdle (more information in the literature guide (page 8)). At second glance, however, black-box coupling turns out to be a very neat feature. First, it is very easy to couple a new code, as only little information needs to be provided. And second, you can easily exchange participants: for example, if you want to try a finite-element fluid solver instead of a finite-volume fluid solver.

We still need to discuss what black-box coupling means mathematically:

- The coupling algorithm only uses the input and output values of a coupled solver, not their derivatives. For example, the deformation (displacements) of a structural mechanics FEM code, but not its Jacobian matrix.
- The coupling algorithm only uses nodal values, meaning values at mesh vertices. For some algorithms, these mesh vertices can be extended with mesh connectivity (mesh edges). What preCICE does not use, however, is shape functions or anything similar.

Explicit and implicit coupling

preCICE offers a variety of coupling schemes (page 71). With “explicit coupling”, we mean that the participants exchange information only once per coupling time window. With “implicit coupling”, we mean that the participants are coupled iteratively, repeating each coupling time window until both sides of the interface have converged to the same values. The iterations of implicit coupling schemes can be greatly reduced with acceleration techniques, such as Aitken under-relaxation or interface quasi-Newton acceleration (which learns over time).

Literature guide

Summary: A guide to the main reference literature for each component and feature of preCICE

Wherever the information in this documentation is not enough (in this case, let us know), you may find a wide spectrum of additional information in publications. This page will guide you through it.

Starting points

- The main reference article for preCICE is preCICE - A Fully Parallel Library for Multi-Physics Surface Coupling. This may not be the best introduction for new users because of its condensed form. A very good first reading is the dissertations of the core preCICE developers.
- Bernhard Gatzhammer introduced preCICE in his dissertation Efficient and Flexible Partitioned Simulation of Fluid-Structure Interactions (2014). Chapters 1-4 give a detailed introduction of most of the preCICE features and are still valid to a large extend. Start here for an explanation of the different coupling schemes, of the different communication methods, or of the data mapping techniques. Note that the "geometry interface" and "server mode" features have been removed.
- Benjamin Uekermann introduced inter- and intra-solver parallelization in his dissertation Partitioned Fluid-Structure Interaction on Massively Parallel Systems (2016). Chapter 2 gives a compact introduction to preCICE. Furthermore, read here especially for the parallel coupling schemes, which allow a simultaneous execution of multiple solvers (Chapter 3) and the realization of all main features on distributed data (Chapter 4).

The list of completed dissertations also includes:

- Klaudius Scheufele: Coupling schemes and inexact Newton for multi-physics and coupled optimization problems. (page 0) (2018)
- Florian Lindner: Data Transfer in Partitioned Multi-Physics Simulations: Interpolation & Communication (2019)

and the story continues by the current team (page 0).

preCICE features

- **Coupling schemes** For an introduction to explicit and implicit coupling, as well as the various acceleration / post-processing techniques, have a look at the dissertations of Bernhard Gatzhammer (Sections 2.3 and 4.1) and Benjamin Uekermann (Chapter 3).
- **Data mapping** For an introduction to the various techniques, have a look at the dissertations of Bernhard Gatzhammer (Sections 2.4 and 4.2) and Benjamin Uekermann (Section 4.3). For a more condensed overview of RBF mapping, see Radial Basis Function Interpolation for Black-Box Multi-Physics Simulations (page 0)
- **Communication** For an introduction to the various techniques, have a look at the dissertation of Bernhard Gatzhammer (Section 4.3). Have a look also at the master's thesis of Alexander Shukaev: "A Fully Parallel Process-to-Process Intercommunication Technique for preCICE".
- **Time interpolation** This feature is currently under active development. Have a look at the publications, talks, and posters of Benjamin Rodenberg.

High-performance computing

- The initial effort for parallelization of preCICE is documented in Partitioned Fluid-Structure-Acoustics

Interaction on Distributed Data: Coupling via preCICE (2016).

- Further steps to speed up initialization are documented in ExaFSA: Parallel Fluid-Structure-Acoustic Simulation (2020).
- The parallelization of communication initialization is published in Efficient and Scalable Initialization of Partitioned Coupled Simulations with preCICE (2021).
- More details can be found in Benjamin's thesis (2016), Florian's thesis (2019), and Klaudius' thesis (page 0) (2019).

Adapters

- For the official adapters for open-source solvers, an overview is given in "Official preCICE Adapters for Standard Open-Source Solvers" (2017).
- For the first implementation of the OpenFOAM, CalculiX, and Code_Aster adapters, have a look at Lucia Cheung Yau's Master's Thesis Conjugate Heat Transfer with the Multiphysics Coupling Library preCICE (2016). Start here also for the physics of Conjugate Heat Transfer. The OpenFOAM adapter was then extended by Gerasimos Chourdakis in his Master's Thesis A general OpenFOAM adapter for the coupling library preCICE (2017). Start here for the structure of the current OpenFOAM adapter. For the additional functionality to support mechanical FSI simulations, have a look at Derek Risseeuw's thesis.
- For the SU2 adapter, read Alexander Rusch's Bachelor's Thesis Extending SU2 to Fluid-Structure Interaction via preCICE (2016). In this you can also find a quick introduction to Fluid-Structure Interaction.

Roadmap

> **Summary:** We are actively developing preCICE. These are some of the features you can expect in the future.

preCICE applies Semantic Versioning, introducing new functionality in minor and major releases. A minor release does not mean fewer changes than a major release, it only means that we add new functionality while keeping backwards compatibility. We release breaking changes only every few years, giving you time to focus on your project, keeping updates easy.

In this page, you can find information about features that we plan to introduce in next releases. This is not meant to be a strict schedule, but rather a hint on the directions that preCICE is heading towards. We also have a few issue milestones, which are updated more often. Issues and work packages of bigger features are generally grouped in projects.

If you are looking for features introduced already in the past, have a look at our Changelog.

Main feedback from the 1st preCICE Workshop

- Restructure the **precice.org website** and **documentation**
  - Get faster to the first steps for users (coming soon)
  - All user documentation in one place (not in different wikis, READMEs, ...) (done)
  - Create a Community section to better communicate the size and contributions of the community, provide contribution guides (done)
  - Getting preCICE: first choose the target system, then get instructions for the specific system (coming soon)
- Provide a reference **virtual machine image** with preCICE already installed (done)
- Keep investing on **Spack**
- Extend **documentation on "How to write an adapter"** (e.g. for mesh generation and moving meshes) (done (page 222))
- Develop a **tutorial on electromagnetics**
- Allow **solver-based data mapping** to support higher order shape functions
- Create videos and upload them on **YouTube**
  - Already created a YouTube channel
  - Create video tutorials
- Organize a **preCICE workshop again in 2021** (register now (page 0))
  - Offer again an optional (potentially longer) introductory course on the first day (e.g. Monday)
  - Add an overview talk on documentation ("Where is what") and community ("How to become a good user")
  - Start main part with an evening event (e.g. Monday dinner)
  - More presentations (and training) from users (open call)
  - More presentations on new and future features
  - Offer again optional hands-on user support to close the workshop

Mid-future (preCICE 2.x or later)

- macOS support (coming in v2.2)
- `watch-integral` (coming in v2.2)
- Contiguous mapping
- Nearest-Projection mapping for quad meshes (done in v2.1)
- RBF mapping without PETSc (done in v2.1)
- Non-mesh-related global data exchange
- Improved error messages (done in v2.1)
- Windows support
- Brute-force re-initialization
- Splitting interface into patches
- Test more platforms in CI
- Two-level initialization enabled by default (coming in v2.2)
    - Currently, we perform the mesh initialization in preCICE in a gather-scatter approach: The communicated mesh is gathered on one side and scattered on the other. This limits the size of the coupling mesh and the scalability (for very large cases). We plan to replace this technique with a two-level approach: Exchanging bounding boxes first and do a parallel mesh initialization afterwards. Ultimately, this will also allow to handle dynamically changing meshes efficiently. This feature was introduced in preCICE v2.0, but is currently switched off by default.

Long-term (preCICE 3.x or later)

- **Consistent time interpolation**, to correctly treat multiscale scenarios with large differences in the respective timestep size of the participating solvers and higher order time stepping schemes. Currently a loss of accuracy and stability can be observed. (needs API changes)
- **3D-1D and 3D-2D data mapping**, e.g. to couple a 3D fluid solver with a 1D model.
- Support for **dynamic adaptive meshes**
- Full support for Volume coupling
- Solver-based data mapping to support higher order shape functions

There are even more features coming, stay tuned!

Adapter-related plans

- **Fluid-fluid module for the OpenFOAM adapter.** This will allow to couple different fluid solvers with each other. (done)
- Develop an adapter for **Elmer**. (doing)

Output files

Summary: During runtime, preCICE writes different output files. On this page, we give an overview of these files and their content.

If the participant's name is `MySolver`, preCICE creates the following files:

precice-MySolver-iterations.log

Information per time window with number of coupling iterations etc. (only for implicit coupling). In case you use a quasi-Newton acceleration, this file also contains information on the state of the quasi-Newton system.

An example file:

```
TimeWindow TotalIterations Iterations Convergence QNColumns DeletedQNColumns DroppedQNColumns 1 6 6
1 5 0 0 2 9 3 1 7 0 0 3 12 3 1 9 0 0 4 14 2 1 10 0 0 5 16 2 1 11 0 0 6 18 2 1 7 0 5 7 20 2 1 6 0 2
```

- `TimeWindow` is the time window counter.
- `TotalIterations` is the total (summed up) number of coupling iterations.
- `Iterations` is the number of iterations preCICE used in each time window.
- `Convergence` indicates whether the coupling converged (`1`) or not (`0`) in each time window.
- `QNColumns` gives the amount of columns in the tall-and-skinny matrices V and W after convergence.
- `DeletedQNColumns` gives the amount of columns that were filtered out during this time window (due to a QR filter). In this example no columns were filtered out.
- `DroppedQNColumns` gives the amount of columns that went out of scope during this time window (due to `max-iterations` or `time-windows-reused`). Here, for example, 5 columns went out of scope during the 6th time window.

Further reading: quasi-Newton configuration (page 74).

precice-MySolver-convergence.log

Information per iteration with current residuals (only for `second` participant in an implicit coupling).

An example file:

```
TimeWindow Iteration ResRel(Temperature) ResRel(Heat-Flux) 1 1 1.0000000000000000 1.000000000000000
0 1 2 0.0009551938284061 0.4856546284783871 1 3 0.0008506916349598 0.0211064920997584 1 4 0.0000145
046004076 0.0020145518273713 1 5 0.0000016534786952 0.0002524364641765 1 6 0.0000002133839605 0.000
0423773334943 2 1 0.0008862176730224 0.0547823667891377 2 2 0.0000844808395569 0.0002313890428748 2
3 0.0000001593921083 0.0000214061446449 ...
```

- `TimeWindow` is the time window counter.
- `Iteration` is the coupling iteration counter within each time window. So, in the first time window, 6 iterations were necessary to converge, in the second time window 3.
- And then two convergence measure are defined in the example. Two relative ones – hence the `...Rel(...)`. The two columns `ResRel(Temperature)` and `RelRel(Force)` give the relative residual for temperature and heat flux, respectively, at the start of each iteration.

precice-MySolver-events.json

Recorded events with timestamps. See page on performance analysis (page 0).

precice-MySolver-events-summary.log

Summary of all events timings. See page on performance analysis (page 0).

precice-postProcessingInfo.log

Advanced information on the numerical performance of the Quasi-Newton coupling (if used and enabled)

In preCICE v1.3.0 and earlier, instead of `precice-MySolver-events.json`, two performance output files were used: `precice-MySolver-events.log` and `precice-MySolver-eventTimings.log`.

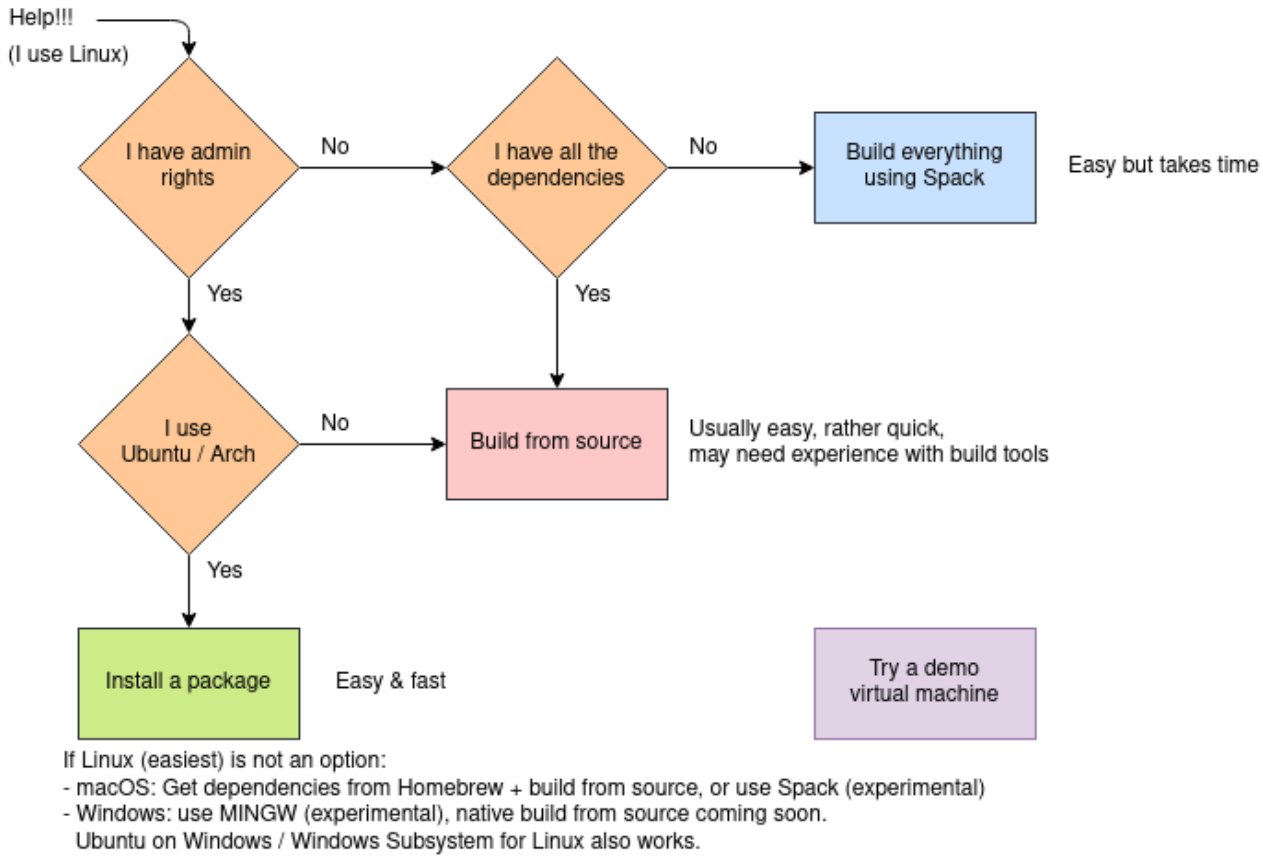
In preCICE v1.2.0 and earlier, slightly different names were used: `iterations-MySolver.txt`, `convergence-MySolver.txt`, `Events-MySolver.log`, `EventTimings-MySolver.log`, and `postProcessingInfo.txt`.

Installing preCICE

Summary: You always need to install the preCICE library and you have a few ways to do this: using a binary package, building from source manually, or building using Spack. You may additionally need to install bindings for Python, Fortran, or Matlab separately.

Installing the core library

The aim of this section is to help you to install preCICE on your system. Depending on your system and your requirements, this process may vary greatly in complexity. To find the right method for you, follow this decision graph, or simply read on!



Note that preCICE is much more than the core library. To find out which library, bindings, adapters, and tutorials versions work together, have a look at the preCICE distribution (page 61).

Operating systems

Linux

Linux is the easiest option, see figure above. We provide binary packages (page 16) for Ubuntu and an AUR package for Arch Linux and Manjaro. If a binary package is not enough for you, keep reading.

macOS

The recommended way is to use Homebrew to install the preCICE dependencies and then compile preCICE from source (page 20). You can alternatively build preCICE using Spack.

Microsoft Windows

We are currently working on native builds of preCICE on Windows. Until then, you can Ubuntu on Windows via the Windows subsystem for Linux (WSL). You can then follow all the instructions for Ubuntu.

- In case you want to use "Ubuntu on Windows", note the following: (click to reveal)
  - You first need to enable WSL. Both WSL 1 and 2 are fine. Simplest option: in your system settings, find the menu `Turn Windows features on or off` and activate WSL there.
  - Whenever you run a coupled simulation, you will get a warning from the Windows firewall. This is because preCICE uses local network addresses to communicate. Give permission to use your network.
  - Whenever you need to start a second terminal, you can just start a second “Ubuntu on Windows” window.
  - In WSL 1, it is complicated to start applications with a graphical user interface, such as ParaView (to visualize your results). Instead, you can install ParaView on Windows and access your files in `\\wsl$\Ubuntu\home`.

Alternatively, you can get preCICE built with MinGW from MSYS2 (package maintained by the community).

Use cases

Are there packages available for my system?

Check our packages (page 16) to see if there are binary packages available for your system. If they are available, install them and you are done!

Are you not allowed to install packages? Do you need to build preCICE in multiple variants and configurations?

Maybe you want to compare how preCICE performs when built with different compilers, MPI versions or dependency versions. If this is the case, strongly consider using the preCICE Spack package (page 17). Once set up, this will simplify your work tremendously.

Do you need to build the debug version of preCICE?

The debug version of preCICE provides a lot of additional debug information and may be necessary for isolating bugs and understanding error messages. If your system provides packages for all required dependencies (page 0), installing from source (page 0) is the easiest way of installing preCICE. If there are packages missing, things get complicated. At this point it is wiser to invest your time in setting up Spack and install precice using spack (page 0) than attempting to install everything by yourself.

Do you want to hack preCICE?

Build preCICE from source (page 0) is the way to go here. In addition to building from source, Spack (page 0) is useful for building and testing with various dependency version in-place.

Do you just want to play around?

We have a sandbox for your first adventures: a demo virtual machine (page 58), with preCICE and most adapters and related tools already installed. Perfect if you just want a system to experiment on in the beginning and then throw it away.

Nothing of the above grabs your attention?

If your system provides packages for all required dependencies (page 0), installing from source (page 0) is the easiest way of installing preCICE. If there are packages missing, things get complicated. At this point it is wiser to invest your time in setting up Spack and install precice using spack (page 0) than attempting to install everything by yourself.

Installing language bindings

preCICE offers further language bindings. Please refer to the following pages for installation instructions:

- Python (page 0)
- Fortran (page 0)
- Matlab (page 0)

System packages

For some systems, preCICE is available in form of a pre-build package or a package recipe. This section lists systems and instructions on how to install these packages.

Ubuntu

You can download version-specific Ubuntu (Debian) packages from each GitHub release. To install, simply open it in your software center.

Alternatively, download & install it from the command line. For **Ubuntu 20.04 (focal)**:

```
wget https://github.com/precice/precice/releases/download/v2.2.1/libprecice2_2.2.1_focal.deb sudo a
pt install ./libprecice2_2.2.1_focal.deb
```

We support the latest two Ubuntu LTS versions, as well as the latest normal Ubuntu release. Change `focal` to `hirsute` for 21.04, `groovy` for 20.10, or to `bionic` for 18.04.

Is a newer preCICE release out and we have not yet updated the above links? Please edit this page.

Arch Linux / Manjaro

We maintain a package in the Arch User Repository. Please have a look at the official AUR wiki page to find out how to install it.

Something else

For other systems you need to either use Spack (page 17) or build from source (page 20).

Community efforts

These packages are maintained by the preCICE community and may be occasionally outdated or not fully working. However, we appreciate the effort and you may be able to contribute to them.

- MSYS2 (for Windows, built with MinGW), thread on our forum
- Nix / NixOS
- EasyBuild

Using Spack

Summary: Get and use Spack to easily build preCICE and all its dependencies from source on your Linux/macOS laptop or local supercomputer, without any root access.

What is Spack?

Spack is a

> multi-platform package manager that builds and installs multiple versions and configurations of software. It works on Linux, macOS, and many supercomputers. Spack is non-destructive: installing a new version of a package does not break existing installations, so many configurations of the same package can coexist.

It also has amazing documentation!

Why Spack?

You can install Spack locally, without root permissions and in an environment that will not affect the rest of your system. Meanwhile, it allows to be chained to installed variants provided by your system administrators.

It builds preCICE and all its dependencies from source and allows you to load the installation into your running environment when needed. It also allows you to build multiple versions and variants of preCICE, which then coexist on your system. Spack also generates module files which can be useful on clusters.

A few hints to get you started:

- `spack info precice` displays the package info, versions, variants, and dependencies.
- `spack spec precice` displays all the dependencies that will be built.
- Want to use a system-installed compiler (e.g. Intel)? Try `spack compiler find`.
- Do you need a specific compiler version? You can build `gcc` and `llvm` from source and use them to compile your software.
- Want to build something special? If you ran `spack install precice@develop%gcc@7.3.0 ^openmpi@3.1.2 build_type=RelWithDebInfo`, this would install the `develop` version of preCICE, with the compiler GCC 7.3.0, OpenMPI version 3.1.2 and in `RelWithDebInfo` mode.
- You can even edit the package/recipe using `spack edit precice`.
- For more advanced usage, you can create your own package repository and use it to build your software.

Where are the packages installed? Just inside the `spack/` directory! Deleting this directory will remove everything.

Setting up Spack

Get and configure Spack on your laptop / supercomputer (no sudo required!):

```
git clone -b develop https://github.com/spack/spack.git source spack/share/spack/setup-env.sh # May be put this in your ~/.bashrc
```

Installing preCICE

To install the latest release of preCICE run:

```
spack install precice ^boost@1.74.0
```

That's it! Spack will now automatically get and build preCICE and all of its dependencies from source. This may take a while, but you don't need to do anything else.

Note: preCICE depends on Boost, which often introduces breaking changes that affect preCICE. We support newer Boost versions as soon as possible in patch releases. Here, we recommend the latest known compatible Boost version only to avoid such potential conflicts. Feel free to try the very latest by ommitting this option: `spack install precice` (but keep an eye on for Boost-related compilation errors and let us know in that case).

You just installed the latest release of precice with the default configuration under `$SPACK_ROOT/opt/spack/<system-name>/<compiler-name>/`. To see all the installed variants of precice that Spack knows, run the following:

```
spack find precice
```

To load the preCICE module, run:

```
spack load precice
```

You can now use preCICE normally and build any adapter following their respective instructions.

If you want to uninstall preCICE, `spack uninstall precice` or delete the complete `spack/` directory to remove everything.

Installing the python bindings

To install the python bindings (page 43) using Spack, run the following:

```
spack install py-pyprecice@2.2.0.2
```

Then to use the python bindings:

```
spack load py-pyprecice@2.2.0.2
```

Advanced tips

Use dependencies from your system

You can instruct Spack to recognize specific dependencies that are already installed on your system, by modifying your preferences in `~/.spack/packages.yaml`.

Tip: If this is the first time you set preferences, the file might not exist and you have to create it yourself

For example, to specify a locally installed MPI version, you could write:

```
packages: openmpi: paths: openmpi@3.1.2: /opt/local buildable: False
```

Here we specify that a local install of OpenMPI version 3.1.2 exists in `/opt/local`. The `buildable` flag specifies that Spack is allowed to look for and build newer versions of the package if they exist instead of using the locally available one. Here we set it to `false` to prevent Spack from trying to build a newer version and add unnecessary installation time.

Install preCICE without non-essential dependency extensions

You might want to opt out some default install options for some dependencies of preCICE as they can cause conflicts. Specifically, extensions of Eigen and Boost can cause errors.

To only install the essential boost libraries that are used by preCICE, you can strip away some default options of the Eigen and Boost packages:

```
spack install precice ^boost@1.65.1 -atomic -chrono -date_time -exception -graph -iostreams -locale
-math -random -regex -serialization -signals -timer -wave ^eigen@3.3.1 -fftw -metis -mpfr -scotch
-suitesparse ^openmpi@3.1.2
```

Note that, here, we install preCICE specifically with Boost 1.65.1 and Eigen 3.3.1. We also demand OpenMPI version 3.1.2 as this allows Spack to use the local OpenMPI install we specified in the example `packages.yaml` above. This is not necessary: feel free to use any other OpenMPI version or just fully omit the `^openmpi` argument to let Spack decide.

I need more help with Spack

Look first for topics with the `spack` tag on our forum on Discourse. If you don't find anything, we will be happy to help you there!

Building from source - Preparation

Which version to build

You decided to build preCICE from source, thus you most likely require a specific configuration.

preCICE builds as a fully-featured library by default, but you can turn off some features. This is the way to go if you run into issues with an unnecessary dependency.

These features include:

- Support for MPI communication.
- Support for radial-basis function mappings based on PETSc. This requires MPI communication to be enabled.
- Support for user-defined python actions.

We recommend to leave all features enabled unless you have a good reason to disable them.

Next is the type of the build which defaults to debug:

- A **debug** build enables some logging facilities, which can give you a deep insight into preCICE.
  This is useful to debug and understand what happens during API calls. This build type is far slower than release builds for numerous reasons and not suitable for running large examples.
- A **release** build is an optimized build of the preCICE library, which makes it the preferred version for running large simulations.
  The version offers limited logging support: debug and trace log output is not available.
- A **release with debug info** build allows to debug the internals of preCICE only. Similar to the release build, it does not support neither debug nor trace logging.

At this point, you should have decided on which build-type to use and which features to disable.

The source code

Download and unpack the `Source Code` of the latest release of preCICE and unpack the content to a directory. Then open a terminal in the resulting folder.

To download and extract a version directly from the terminal, please execute the following:

```
wget https://github.com/precice/precice/archive/v2.2.1.tar.gz tar -xzvf v2.2.1.tar.gz cd precic
e-2.2.1
```

Installation prefix

The next step is to decide where to install preCICE to. This directory is called the installation prefix and will later contain the folders `lib` and `include` after installation. System-wide prefixes require root permissions and may lead to issues in the long run, however, they often do not require setting up additional variables. User-wide prefixes are located in the home directory of the user. These prefixes do not conflict with the system libraries and do not require special permissions. Using such prefixes is generally required when working on clusters.

Using a user-wide prefix such as `~/software/precice` is the recommended choice.

Common system-wide prefixes are:

- `/usr/local` which does not collide with package managers and is picked up by most linkers (depends on each system).

- `/opt/precice` which is often used for system-wide installation of optional software. Choosing this prefix requires setting additional variables, which is why we generally don't recommend using it.

Common user-wide prefixes are:

- `~/software/precice` which allows to install preCICE in an isolated directory. This requires setting some additional variables, but saves a lot of headache.
- `~/software` same as above but preCICE will share the prefix with other software.

In case you choose a user-wise prefix you need to extend some additional environment variables in your `~/.bashrc`:

Replace `<prefix>` with your selected prefix

```
PRECICE_PREFIX=~/software/prefix # set this to your selected prefix export LD_LIBRARY_PATH=$PRECIC
E_PREFIX/lib:$LD_LIBRARY_PATH export CPATH=$PRECICE_PREFIX/include:$CPATH # Enable detection with p
kg-config and CMake export PKG_CONFIG_PATH=$PRECICE_PREFIX/lib/pkgconfig:$PKG_CONFIG_PATH export CM
AKE_PREFIX_PATH=$PRECICE_PREFIX:$CMAKE_PREFIX_PATH
```

After adding these variables, please start a new session (open a new terminal or logout and login again).

Note: On debian-based distributions, you can also build preCICE as a debian package and install it using the package manager. Read more (page 0)

The next step

In the next step we will install all required dependencies (page 0).

Building from source - Dependencies

Summary: This page describes the dependencies used by preCICE, how to install them on various systems and how to build them.

How to use this page?

Start by checking if there is a guide for your system (page 26). It will include all required steps to get preCICE ready to build.

If there is no guide for your system, find out if there are suitable system packages for the dependencies. Then use the dependencies (page 22) section to install all missing dependencies from source.

After all dependencies are ready to use, proceed with configuring preCICE (page 0).

Dependencies

This section lists all dependencies alongside required versions and steps on how to install them from source. Meaning, installing dependencies based on the steps in this section should be the *last resort* for normal users. Prefer to follow the system guides (page 26) and only install custom versions if you have a reason to do so.

Overview

Required dependencies

- C++ compiler (page 22) (with support for C++14, e.g. GCC version >= 5)
- CMake (page 23) (version >= 3.10.1)
- Eigen (page 23)
- Boost (page 23) (version >= 1.65.1)
- libxml2 (page 24)

Optional dependencies

- MPI (page 25)
- PETSc (page 25) (version >= 3.12)
- Python (page 25) (with NumPy)

C++ compiler

preCICE requires a C++ compiler with full C++14 support. The following table lists the minimal requirement for compiler versions:

Toolchain	Minimal Version	Note
GCC	`5`	
Intel	`17`	also requires GCC `5`
Cray	`8.6`	also requires GCC `5`
Clang	`3.4`	

MSVC	`19.10`	*For future reference*

If you are using Debian/Ubuntu, the `build-essential` package will install everything needed.

When compiling with MPI enabled (the default) and using your MPI compiler wrapper as compiler, then it needs to use a suitable compiler. For example, check if the `mpicxx --version` reports a compatible compiler version. Check the section on MPI (page 25) for more information.j

CMake

preCICE required the build system CMake at a minimal version of `3.10`. You can check your CMake version using `cmake --version`.

Depending on the versions of CMake and Boost, CMake may not find all libraries in boost and display warnings when configuring preCICE. This can be safely ignored as preCICE does not use problematic libraries. Fixing this requires to upgrade CMake..

Download CMake binaries

Download the official binaries for your platform and extract them into a folder. Then extend the path environment variable by executing the following:

```
export PATH=$PATH:/path/to/extracted/location/version/bin cmake --version
```

This should now display the version of the latest release. If the version is correct, you can make this change persistent by appending the above export statement to your `.bashrc` or similar.

Eigen

preCICE uses Eigen for linear algebra computations and for a version of RBF mappings which does not require PETSc.

Download the Eigen headers

Eigen is a header-only library, i.e. it is compiled into preCICE and does not require linkage. Download the sources from their latest release and extract them to some location. The folder of your choice should now contain a folder called `eigen-x.y.z` for version `x.y.z`. Set the environment variable `Eigen3_ROOT` to the `eigen-x.y.z` folder by adding this to your `~.bashrc`.

```
export Eigen3_ROOT=/path/to/eigen/eigen-x.y.z
```

Boost

preCICE uses Boost for several features and requires version 1.65.1 or higher. While Boost 1.67 or newer also works with preCICE, it may complicate how you install adapters that use yaml-cpp. Note that users have experienced problems building Boost 1.68 and 1.69 with some compilers.

Note: Boost 1.75.0 is not supported before preCICE 2.2.0. Similarly, Boost 1.73.0 is not supported before preCICE 2.1.0.

You might save some time and space by installing only the necessary libraries:

- `boost_log`
- `boost_log_setup`
- `boost_thread`
- `boost_system`

- `boost_filesystem`
- `boost_program_options`
- `boost_unit_test_framework`

These libraries may also depend on other Boost libraries. Make sure that these get installed, too.

The following header-only Boost libraries are also needed: `vmd`, `geometry`, `signals2`, `container`, `ranges`.

Build boost from source

1. Download and extract Boost into any directory. Switch to that directory.
2. Prepare the installation, selecting only the libraries that need to be built (this does not affect the header-only libraries). Select a prefix to install Boost to. This will later contain the directories `include` and `lib`. On systems using modules, we recommend to specify the toolset manually by additionally passing `--with-toolset=gcc` (or `intel`).

   Now run with the prefix of your choice:

   ```
   ./bootstrap.sh --with-libraries=log,thread,system,filesystem,program_options,test --prefix=<prefix>
   ```

3. Build and install the libraries. Depending on your choice, you may need root access.

   ```
   ./b2 install # user has write access to the prefix sudo ./b2 install # user does not have sufficient permissions
   ```

   The directory you chose as prefix now contains libraries in `<prefix>/lib` and the all the Boost headers in `<prefix>/include`. You may now safely remove the boost directory from step 1.

4. If you selected `/usr/local` as prefix, update the dynamic linker's run-time bindings:

   ```
   sudo ldconfig
   ```

5. If you did not select `/usr/local` as prefix, you need to make the boost installation visible to the linker and compiler. Add the following to your `~/.bashrc`:

```
export BOOST_ROOT=<prefix> export LIBRARY_PATH=$BOOST_ROOT/lib:$LIBRARY_PATH export LD_LIBRARY_PATH=$BOOST_ROOT/lib:$LD_LIBRARY_PATH export CPLUS_INCLUDE_PATH=$BOOST_ROOT/include:$CPLUS_INCLUDE_PATH
```

For more information, please refer to the "Getting Started" instructions of Boost.

libxml2

preCICE uses libxml2 for parsing the configuration file.

> **Note:** libxml2 is available on close to any system you can imagine.
> Please double check if there are no system packages before attempting to build this dependency from source.

Install libxml2 from source

1. Download the latest release of libxml.
2. Extract the sources to a location of your choice.

3. Choose a directory to install the library to and use it as `<prefix>`.
4. Build and install the library

```
./autogen --prefix=<prefix> make make install
```

5. If you did not select `/usr/local` as prefix, you need to make the installation visible to the linker and compiler. Add the following to your `~/.bashrc` replacing prefix with the chosen directory:

```
export LIBRARY_PATH=<prefix>/lib:$LIBRARY_PATH export LD_LIBRARY_PATH=<prefix>/lib:$LD_LIBRARY_PATH
export CPLUS_INCLUDE_PATH=<prefix>/include:$CPLUS_INCLUDE_PATH
```

PETSc

PETSc is used for RBF mappings and is highly recommended for large cases. For small/medium-size cases, preCICE can still do an RBF mapping in parallel without PETSc. If you don't need this feature, you may specify `-DPRECICE_PETScMapping=off` when building preCICE.

We require at least version 3.12. For preCICE versions earlier than v2.1.0, PETSc version between 3.6 and 3.12 might still work, but needs to be built with 64bit index sizes. In particular on Ubuntu 18.04, we require at least 3.12.

Build PETSc from source

If you prefer to install the most recent version from source, do the following:

1. Download it or get the repository using `git clone -b maint https://bitbucket.org/petsc/petsc petsc`
2. Change into that directory and compile with or without debugging: `./configure --with-debugging=0` (disable debugging for optimal performance)
3. Use the `make` command as the configure script proposes, e.g. `make PETSC_DIR=/path/to/petsc PETSC_ARCH=arch-linux2-c-opt all` Further documentation see the PETSc installation documentation.
4. Usage: You will need to add PETSc to your dynamic linker search path (`LD_LIBRARY_PATH` on Linux or `DYLD_LIBRARY_PATH` on macOS). You may also need to set the `$PETSC_ARCH`.

Finally, in some cases you may need to have PETSc in your `CPATH`, `LIBRARY_PATH`, or `PYTHONPATH`. Here is an example:

```
export PETSC_DIR=/path/to/petsc export PETSC_ARCH=arch-linux2-c-debug export LD_LIBRARY_PATH=$PETS
C_DIR/$PETSC_ARCH/lib:$LD_LIBRARY_PATH
```

Python

You only need Python if you want to use the Python action interface (only used for rare applications). If you don't need this feature, you may specify `-DPRECICE_PythonActions=off`. In particular, you don't need to build with Python if you only want to use the preCICE Python bindings (page 43).

You probably already have Python installed. However, in order to use the Python interface, you also need to install NumPy and the header files for Python and NumPy. On Debian/Ubuntu, install the packages `python3-numpy` and `python3-dev`.

MPI

You can build preCICE without MPI in case of compatibility issues with a certain solver (e.g. a closed source solver with a binary-distributed MPI version, or when running on Windows). To do so, use `-DPRECICE_MPICommunication=OFF` when building with CMake. In such a case, you can still use TCP/IP sockets instead. This might, however, result in lower performance and is, therefore, not recommended if not necessary.

Please note that OpenMPI does not currently fully support the MPI ports functionality [citation needed]. In case you link to OpenMPI, you cannot use MPI for the m2n communication of preCICE. With preCICE versions earlier than 2.1.0, the tests for MPI Ports will fail.

Keep in mind that already PETSc (page 25) should have installed MPI.

⚠ **Important:** Make sure that PETSc, preCICE, and your solvers are all compiled with the same MPI version!

System guides

If you want build preCICE on your own computer and you are using one of the following Linux distributions, we provide a summary here to quickly install everything you need. If everything works, you may ignore the rest of this page.

Other modern versions of popular Linux distributions are also perfectly compatible, here we just list a few popular options. Since our users have tried preCICE on various distributions, you may as well ask on our forum for any questions.

Ubuntu 20.04 LTS Focal Fossa

With every release, we also ship binary packages for Ubuntu 20.04. However, if you still want to build from source, everything is available through the distribution's repositories:

```
sudo apt update && \ sudo apt install build-essential cmake libeigen3-dev libxml2-dev libboost-all-dev petsc-dev python3-dev python3-numpy
```

The same instructions apply for later Ubuntu releases.

Ubuntu 18.04 Bionic Beaver

With every release, we also ship binary packages for Ubuntu 18.04. However, if you still want to build from source, almost everything is available through the distribution's repositories:

```
sudo apt update && \ sudo apt install build-essential cmake libeigen3-dev libxml2-dev libboost-all-dev python3-dev python3-numpy
```

If you don't plan to use RBF mappings in large parallel cases you can continue without installing PETSc and build with `-DPRECICE_PETScMapping=OFF`. If you need PETSc, follow the steps in the PETSc (page 25) section and you are done.

Ubuntu 16.04 Xenial Xerus

In Ubuntu 16.04, only a fraction of packages is available through the distribution's repositories. Further packages needs to be build from source. First install the available packages:

```
sudo apt update && \ sudo apt install build-essential g++-5 libxml2-dev python3-dev python3-numpy
```

Next, you need to install CMake (page 23), Eigen (page 23) and boost (page 23) as descibed in their respective sections.

If you don't plan to use RBF mappings in large parallel cases you can continue without installing PETSc and build with `-DPRECICE_PETScMapping=OFF` (page 0). If you need PETSc, follow the steps in the PETSc (page 25) section and you are done.

ℹ **Note:** The repositories contain a package `libeigen3-dev`, however, unsing it results in issues with nearest-projection mapping.

Debian 11 Bullseye

Everything is available from the distribution's repositories:

```
su apt update && \ apt install build-essential cmake libeigen3-dev libxml2-dev libboost-all-dev pet
sc-dev python3-dev python3-numpy
```

Debian 10 Buster

In Debian 10.5, almost everything is available through the distribution's repositories:

```
su apt update && \ apt install build-essential cmake libeigen3-dev libxml2-dev libboost-all-dev pyt
hon3-dev python3-numpy
```

If you don't plan to use RBF mappings in large parallel cases you can continue without installing PETSc and build with `-DPRECICE_PETScMapping=OFF`. If you need PETSc, follow the steps in the PETSc (page 25) section and you are done.

Fedora 34

In Fedora 33 and 34, everything is available through the distribution's repositories:

```
sudo dnf update sudo dnf install gcc-c++ cmake libxml2-devel boost-devel openmpi-devel petsc-openmp
i-devel eigen3-devel python3-devel
```

Afterwards, start a new terminal, to make MPI discoverable (read more about MPI on Fedora). Before configuring & building preCICE, load MPI using the module:

```
module load mpi/openmpi-x86_64
```

Note: In case you use the docker image of fedora, you need to install the support for environment modules first: `sudo dnf install environment-modules`

If you don't plan to use RBF mappings in large parallel cases you can continue without installing PETSc and build with `-DPRECICE_PETScMapping=OFF`. You may need this with older preCICE and Fedora versions (e.g. preCICE v2.1 on Fedora 32 or earlier, see a related issue.

CentOS 8

(The same instructions apply also to Rocky Linux 8)

This system requires to install some tools in a fixed order.

1. First, make sure that a few common dependencies are installed. You need to enable the PowerTools repository (for Eigen) and to install the Development Tools group (compilers, Git, make, pkg-config, ...).

   ```
   sudo dnf update sudo dnf install dnf-plugins-core sudo dnf groupinstall "Development Tool
   s" sudo dnf config-manager --set-enabled powertools sudo dnf update
   ```

   Note that, instead of `dnf`, you can also type `yum` with the same options.

2. Then, install the available preCICE dependencies:

```
sudo dnf install cmake libxml2-devel boost-devel openmpi-devel eigen3-devel python3-devel
pip3 install --user numpy
```

3. Before configuring & building preCICE, load MPI:

```
module load mpi/openmpi-x86_64
```

4. Unfortunately, the PETSc package (`petsc-openmpi-devel`) in this distribution is too old. If you don't plan to use RBF mappings in large parallel cases you can continue without installing PETSc and build with `-DPRECICE_PETScMapping=OFF` . If you need PETSc, follow the steps in the PETSc (page 25) section and you are done.

CentOS 7

This system requires to install some tools in a fixed order.

1. First install the group 'Development Tools'.

```
sudo yum groupinstall 'Development Tools' sudo yum update
```

2. Then install available dependencies from the repositories:

```
sudo yum install cmake3 libxml2-devel eigen3 openmpi-devel python3-devel boost169-devel
```

3. Then add the following to your `~./bashrc` :

```
export PATH=/usr/lib64/openmpi/bin:$PATH export CC=/opt/rh/devtoolset-7/root/usr/bin/gcc e
xport BOOST_LIBRARYDIR=/usr/lib64/boost169/ export BOOST_INCLUDEDIR=/usr/include/boost169/
```

4. Then install install a newer version of gcc using a software development package:

```
sudo yum install centos-release-scl sudo yum install devtoolset-7
```

To enable the new gcc compiler in a terminal:

```
scl enable devtoolset-7 bash
```

⚠ **Important:** Use `cmake3` instead of `cmake` to configure preCICE!

OpenSUSE Leap 15.2

In OpenSUSE Leap 15.2, things are a bit more complicated (please contribute in this section). Get the basic dependencies:

```
sudo zypper refresh sudo zypper install gcc-c++ make cmake libxml2-devel \ libboost_log1_66_0-devel
libboost_thread1_66_0-devel libboost_system1_66_0-devel libboost_filesystem1_66_0-devel libboost_pr
ogram_options1_66_0-devel libboost_test1_66_0-devel \ eigen3-devel python3-devel
```

You may need to set the Eigen location when configuring preCICE:

```
cmake -DEIGEN3_INCLUDE_DIR=/usr/include/eigen3 <options as usual>
```

If you don't already have a fitting combination of MPI and PETSc (not shown here), disable the respective features when configuring preCICE:

```
cmake -DPRECICE_MPICommunication=OFF -DPRECICE_PETScMapping=OFF <options as usual>
```

Arch Linux

(The same applies to Manjaro and other derived distributions)

Good news: preCICE is already on AUR, so you can directly use or modify the respective `PKGBUILD`.

macOS Catalina 10.15

First, `XCode Command Line Tools` should be installed from Apple Developer page or from XCode application.

Then, all the dependencies can be installed using a package manager such as Homebrew or MacPorts:

```
brew install cmake eigen libxml2 boost petsc openmpi python3 numpy
```

or

```
port install cmake eigen3 libxml2 boost petsc openmpi python3 numpy
```

Building from source - Configuration

preCICE uses CMake to configure and build the library. After this step, preCICE is ready to be built (page 0).

Build directory

CMake keeps track of the source and the build directory separately. This allows to cleanly create multiple build configurations for a single source directory.

Please create a build directory inside the preCICE source directory as follows:

```
cd precice-2.2.1 # Enter the preCICE source directory mkdir build cd build
```

Options

Now it is time to configure preCICE with the decisions taken in the preparation steps (page 0). First, make sure that you changed into the `build/` directory.

If you need to configure a debug build with all default settings, simply run:

```
cmake -DBUILD_SHARED_LIBS=ON ..
```

As you can see, you can pass variables to cmake using the syntax `-DNAME=VALUE`. The following table lists the most important options to pass to CMake.

Assemble your CMake command and run it to configure preCICE.

This example builds the release version of preCICE with the PETSc mapping and the user-defined python actions off, which will be installed in the prefix `~/software/precice`.

```
cmake -DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=~/software/precice
-DPRECICE_PETScMapping=OFF -DPRECICE_PythonActions=OFF ..
```

Option	Type	Default	Description
BUILD_SHARED_LIBS	Boolean	OFF	Build as a shared library.
CMAKE_BUILD_TYPE	String	Debug	Choose Debug, Release, or RelWithDebInfo.
CMAKE_INSTALL_PREFIX	Path	/usr/local	The prefix used in the installation step.
PRECICE_MPICommunication	Boolean	ON	Build with MPI.
MPI_CXX_COMPILER	Path		MPI compiler wrapper to use for detection.
PRECICE_PETScMapping	Boolean	ON	Build with PETSc (for MPI-parallel RBF mapping), requires PRECICE_MPICommunication=ON.
PRECICE_PythonActions	Boolean	ON	Build support for python actions.
PYTHON_EXECUTABLE	Path		Path to the python interpreter to use.

Option	Type	Default	Description
BUILD_TESTING	Boolean	ON	Build and register the tests.
PRECICE_InstallTest	Boolean	OFF	Install `testprecice` and test configuration files.
PRECICE_Packages	Boolean	ON	Enable package configuration.
PRECICE_ENABLE_C	Boolean	ON	Enable the native C bindings.
PRECICE_ENABLE_FORTRAN	Boolean	ON	Enable the native Fortran bindings.
PRECICE_ALWAYS_VALIDATE_LIBS	Boolean	OFF	Force CMake to always validate required libraries.
PRECICE_TEST_TIMEOUT_LONG	Integer	180	Timeout for big test suites
PRECICE_TEST_TIMEOUT_SHORT	Integer	20	Timout for small test suites
PRECICE_CTEST_MPI_FLAGS	String		Additional flags to pass to `mpiexec` when running the tests.

The next step

preCICE is now configured and the build system has been generated. The next step covers how to build preCICE (page 0).

Building from source - Building

To build preCICE, simply run `make` in the build directory.

You can also build in parallel using all available logical cores using `make -j $(nproc)`. In that case, remember that the more threads you use, the more main memory will be needed.

The next step

As a next step, please run the tests (page 0) to ensure everything works as expected.

If you feel lucky, you can skip straight to installing preCICE (page 0).

Building from source - Testing

To test preCICE after building, run `ctest` inside the build directory.

This will execute 3 types of tests:

- Component-wise unit tests
- Integration tests
- Compilation and run tests based on example programs

For technical reasons, unit and integration tests require preCICE to be compiled with **MPI enabled**.

To display log output for tests use `ctest -VV` or `ctest --output-on-failure`. To change the log level of the output, set the environment variable `export BOOST_TEST_LOG_LEVEL=all|test_suite|warning`.

Please note that debug and trace logs require preCICE to be built using `-DCMAKE_BUILD_TYPE=Debug`.

The next step

The next step concludes the installation guide by installing preCICE (page 0).

Building from source - Installation

It is time to install preCICE into the installation prefix chosen during preparation (page 0) and used during tha configuration with CMake (page 0).

To install preCICE run `make install`.

If the chosen prefix points to a system directory, you may have to run `sudo make install`.

Testing your installation

To test your installation please run `make test_install`. This will attempt to build our C++ example program against the **installed version** of the library. This is commonly known as *the smoke test*.

Next steps

This concludes the preCICE installation and you should have a working installation of preCICE on your system.

To use preCICE in your project, see the page Linking to preCICE (page 0).

Building from source - Advanced

Debian packages

Note: You may prefer to directly use the provided packages attached to our releases.

To generate Debian packages, make sure to set the following variables:

```
cmake -DCMAKE_INSTALL_PREFIX=/usr -DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_TYPE=Release .. make -j $(np
roc) package
```

The directory should now contain a `.deb` package and the corresponding checksum file. You can install this using your package manager (to be able to remove properly): `sudo apt install ./libprecice2.2.1.deb`

In case you want to remove this package, use your package manager: `sudo apt purge libprecice2.2.1`.

Static library

To build preCICE as a static library, you can set `-DBUILD_SHARED_LIBS=OFF` or simply omit the option.

This is not recommended or supported by the preCICE developers! You may contribute here (better support is coming soon).

Disabling native bindings

The library provides native bindings for C and Fortran. They are called native as they are compiled into the resulting library. If you know what you are doing, you can disable them by specifying `-DPRECICE_ENABLE_C=OFF`, or `-DPRECICE_ENABLE_FORTRAN=OFF`.

We highly discourage you to do this, as the resulting binaries will not be compatible with C or Fortran adapters!

Overriding dependencies

BOOST

- `BOOST_ROOT` as described in the CMake documentation

Eigen3

- `EIGEN3_INCLUDE_DIR` being the root of the repository.

LibXML2

- `LIBXML2_LIBRARIES` and `LIBXML2_INCLUDE_DIRS`

JSON

- `JSON_INCLUDE_DIR` this expects the scoped include to work `#include <nlohmann/json.hpp`

Prettyprint

- `PRETTYPRINT_INCLUDE_DIR` this expects the scoped include to work `#include <prettyprint/prettyprint.hpp`

PETSc

- Environment variables `PETSC_DIR` and `PETSC_ARCH`.

Python

- `PYTHON_LIBRARY`, `PYTHON_INCLUDE_DIR`, `NumPy_INCLUDE_DIR` (the two latter are often identical)

MPI - Build preCICE using non-default MPI implementation

- Set `CXX` to the compiler wrapper if you want to be sure that the right installation is picked.

For using a non-default MPI implementation one can steer the CMake MPI discovery by setting the variable `MPI_CXX_COMPILER` to the path to the `mpicxx` compiler wrapper shipped with your MPI distribution of choice.

Example - building with MPICH:

```
cmake -DBUILD_SHARED_LIBS=ON -DPRECICE_MPICommunication=ON -DMPI_CXX_COMPILER=/usr/bin/mpicxx.mpich
..
```

Building from source - Troubleshooting

Troubleshooting

Finding Boost

- Boost versions prior to 1.70.0 use the `FindBoost` module. For custom install prefixes, simply set the `BOOST_ROOT=/path/to/prefix` CMake option or environment variable.
- Boost version 1.70.0 and later ship with their own config module, which you can find in `<prefix>/lib/cmake/Boost-x.xx.x/`. To detect it in custom prefixes, set the `Boost_DIR=<prefix>/lib/cmake/Boost-x.xx.x/`. Have a look at `<prefix>/lib/cmake/Boost-x.xx.x/BoostConfig.cmake` for additional options.

Finding Python and NumPy

The NumPy detection is directly connected to the detected python interpreter. The easiest solution to force CMake to use a python installation is to set the CMake Variable `PYTHON_EXECUTABLE` when configuring preCICE for the first time. This is also the method of choice when using a virtual environment or `pyenv`.

Example:

```
cmake -DPRECICE_PythonActions=ON -DPYTHON_EXECUTABLE=/usr/bin/python3.8 .
```

PETSc could not be found

Note: PETSc is an optional dependency, only needed for parallel RBF mapping, which you can switch off. Since preCICE v2.1.0, single-node-parallel RBF mapping is also possible without PETSc.

There are multiple problems than can lead to FindPETSc failing:

1. `PETSC_DIR` and `PETSC_ARCH` not set, or set incorrectly.
   In this case, FindPETSc fails **before** running tests.
2. *Pre 1.5.0:* Compiler CXX not set to the compiler wrapper provided by your MPI distribution.
   In this case, FindPETSc fails **after** running tests.

Find more regarding PETSc-related issues on our forum.

Tests fail

The end of your log output looks like this:

```
-- petsc_lib_dir /.../.../.../petsc/arch-linux2-c-debug/lib -- Recognized PETSc install with single library for all packages -- Performing Test MULTIPASS_TEST_1_petsc_works_minimal -- Performing Test MULTIPASS_TEST_1_petsc_works_minimal - Failed -- Performing Test MULTIPASS_TEST_2_petsc_works_allincludes -- Performing Test MULTIPASS_TEST_2_petsc_works_allincludes - Failed -- Performing Test MULTIPASS_TEST_3_petsc_works_alllibraries -- Performing Test MULTIPASS_TEST_3_petsc_works_alllibraries - Failed -- Performing Test MULTIPASS_TEST_4_petsc_works_all -- Performing Test MULTIPASS_TEST_4_petsc_works_all - Failed -- PETSc could not be used, maybe the install is broken. CMake Error at /.../.../.../.../cmake/share/cmake-3.10/Modules/FindPackageHandleStandardArgs.cmake:137 (message): PETSc could not be found. Be sure to set PETSC_DIR and PETSC_ARCH. (missing: PETSC_EXECUTABLE_RUNS) (found suitable version "3.10.2", minimum required is "3.12") Call Stack (most recent call first): /.../.../.../.../cmake/share/cmake-3.10/Modules/FindPackageHandleStandardArgs.cmake:378 (_FPHSA_FAILURE_MESSAGE) tools/cmake-modules/FindPETSc.cmake:343 (find_package_handle_standard_args) CMakeLists.txt:60 (find_package) -- Configuring incomplete, errors occurred! See also "/.../.../CMakeFiles/CMakeOutput.log". See also "/.../.../CMakeFiles/CMakeError.log".
```

In this case the library, includes, etc. were found, however, the tests do not compile. Check that the compiler is set to the compiler wrapper provided by your MPI distribution (e.g. with `CXX=mpicxx cmake [options]`). You may need to delete and recreate the `build` directory. For further information check the log file `./CMakeFiles/CMakeError.log` for the error thrown by the compiler invocation.

This is fixed in preCICE v1.5.0. Please let us know if you still get this error.

No tests run

The end of your log output looks like this:

```
CMake Error at /.../.../.../.../cmake/share/cmake-3.10/Modules/FindPackageHandleStandardArgs.cmake:137 (message): PETSc could not be found. Be sure to set PETSC_DIR and PETSC_ARCH. (missing: PETSC_INCLUDES PETSC_LIBRARIES PETSC_EXECUTABLE_RUNS) (Required is at least version "3.12") Call Stack (most recent call first): /.../.../.../.../cmake/share/cmake-3.10/Modules/FindPackageHandleStandardArgs.cmake:378 (_FPHSA_FAILURE_MESSAGE) tools/cmake-modules/FindPETSc.cmake:343 (find_package_handle_standard_args) CMakeLists.txt:60 (find_package) -- Configuring incomplete, errors occurred! See also "/.../.../CMakeFiles/CMakeOutput.log". See also "/.../.../CMakeFiles/CMakeError.log".
```

In this case, the FindPETSc module cannot locate PETSc.

- Check the values of `PETSC_DIR` and `PETSC_ARCH`.
- Make sure `ls $PETSC_DIR/$PETSC_ARCH/include` does not result in an error.

Notes on CMake

General information on CMake

- CMake is a tool for build system configuration.
  It generates a chosen build system in the directory it was executed in. The build system to generate can be specified using the `-G` flag which defaults to `Makefile`. A list of all supported generators (e.g. for IDEs) can be found here.
- Use the console tool `ccmake` or the gui `cmake-gui` for a more comfortable configuration.
- The generated build system automatically reconfigures cmake when necessary.
- It respects your environment variables `CXX`, `CXX_FLAGS`, ... during configuration.
  Please use them to set your compiler and warning level of choice.
- Invoke `cmake` with `-DVariable=Value` to set the cmake-internal variable `Variable` to `Value`.
  A complete list of variables recognised by cmake can be found here.
- Use `-DCMAKE_BUILD_TYPE` to specify what type of build you would like `Debug, Release, RelWithDebInfo, MinSizeRel`
  CMake will select appropriate flag for you (e.g. `-g`, `-O2`). Specifying no `BUILD_TYPE` results in an un-optimised non-debug build.
- The generated build system *knows* where the source directory is. It is thus possible to have multiple configurations using the same (e.g. `build/debug/`, `build/release/`)
- Use `-DBUILD_SHARED_LIBS=ON` to build shared libraries.
- To use `ccache` or `distcc` with cmake please set the variable CXX_COMPILER_LAUNCHER.
- Use `-DCMAKE_INSTALL_PREFIX=/path/to/dir/` to specify the install prefix. Default is `/usr/local` on unix. Read more

Linking to preCICE

Linking against preCICE requires to pass two set of flags to two programs. The compiler requires information on where to find the preCICE headers. The linker requires the name of the library as well as its location.

CMake

This is the preferred method of linking to preCICE. preCICE provides a precice-Config file which contains all required information to the build system.

Linking to preCICE from a CMake project is simple. Use `find_package(precice)` and link `precice::precice` to your target:

```
find_package(precice REQUIRED CONFIG) add_executable(myTarget main.cpp) target_link_libraries(myTar
get PRIVATE precice::precice)
```

This works out-of-the-box if you installed preCICE from provided packages or manually to `/usr/local`. In case preCICE was installed to another directory, CMake needs to know which directory to look into.

The simplest solution is to explicitly pass the location of the config file to CMake using `-Dprecice_DIR=<prefix>/lib/cmake/precice`. This directory can also point to the build directory of preCICE. This allows to use preCICE without explicitly installing it.

An alternative is to tell CMake to consider an additional install prefix by passing the following to CMake `-DCMAKE_PREFIX_PATH=<prefix>`.

Note: Static linking is not recommended nor supported by the preCICE developers!
Static linking in CMake requires you to provide all transitive dependencies of the preCICE, which includes private dependencies! Meaning that you have to find and provide the requested targets in your `CMakeLists.txt`. You may contribute here

Autotools

Linking to preCICE from a GNU Autotools project is similarly simple. Just use the following in your `configure.ac` file:

```
PKG_CHECK_MODULES(preCICE, libprecice)
```

This will extract the `CFLAGS` and `LIBS` into `preCICE_CFLAGS` and `preCICE_LIBS`, which you can then use in your `Makefile.am` as:

```
my_cxx_flags += @preCICE_CFLAGS@ my_ldadd += @preCICE_LIBS@
```

Make and scripts in general

The recommended way to link preCICE to another project is by embedding pkg-config commands into a building script/Makefile to extract the necessary flags from the generated `liprecice.pc` file.

Use the following two commands to fetch necessary flags:

```
pkg-config --cflags libprecice pkg-config --libs libprecice
```

These two commands should return (if the paths are not already known by the system):

```
-I/path/to/include -L/path/to/lib -lprecice
```

You can use backticks to evaluate a command and use its result in your shell script, for example:

```
CFLAGS = `pkg-config --cflags libprecice`
```

The syntax to do the same in a Makefile is:

```
CFLAGS = $(shell pkg-config --cflags libprecice)
```

If you built preCICE and installed it into a custom prefix, e.g. `~/software/`, you need to set the following environment variable first:

```
export PKG_CONFIG_PATH="~/software/lib/pkgconfig"
```

Now you can use `pkg-config` to extract the necessary flags.

Troubleshooting

`SolverInterface.hpp` cannot be found

There are two reasons you may be getting this error:

- `pkg-config` could not find a `libprecice.pc` file (keep reading)
- you are including the file as `SolverInterface.hpp` and not as `precice/SolverInterface.hpp`

`SolverInterfaceC.h` cannot be found

If you are using the C bindings, please note that they are now installed in `[prefix]/include/precice/`, alongside the C++ headers. If your code includes e.g. `precice/bindings/c/SolverInterfaceC.h`, please update this to `precice/SolverInterfaceC.h`.

libprecice cannot be found (during building)

If you installed preCICE in a path not known by your compiler/linker, pkg-config will try to locate the file `libprecice.pc` and extract the necessary information. However, pkg-config only looks in specific places (usually in `/usr/lib/pkgconfig`, but *not* e.g. in `/usr/local/lib/pkgconfig`). Before building, you need to set the path where pkg-config can find this file, e.g. with:

```
PKG_CONFIG_PATH=/path/to/lib/pkgconfig [make or anything else]
```

libprecice cannot be found (at runtime)

If you built preCICE (as a shared library) in a non-standard path, pkg-config only helps during building. At runtime, libprecice will not be discoverable, unless you e.g. include this path in your `LD_LIBRARY_PATH`.

Depending on the configuration of `ld` it might look by default into `/usr/local/lib` or not. This might lead to linking problems and can be either solved by adding `/usr/local/lib` to the `LD_LIBRARY_PATH` or changing the configuration o `ld`.

Fortran bindings

This section lists the various language bindings of preCICE and describes how to fetch and install them.

Fortran bindings

These languages are natively supported by preCICE. For more details please read linking to preCICE (page 40).

Fortran module

The fortran module can be found here.

Python bindings

Summary: Use pip3 install –user pyprecice to install the python language bindings from PyPI

The versioning scheme

The versioning scheme of the python bindings is the preCICE version with the additional version of the python bindings.

Example: version `1` of the python bindings for preCICE version `2.2.0` is `2.2.0.1`

Installation

The python bindings for preCICE are published on PyPI with the package `pyprecice`. You can use your python package manager for installing the language bindings. For example, `pip3 install --user pyprecice`. Note that preCICE and MPI have to be installed on your system.

Usage

The usage of the python language bindings for preCICE is very similar to the C++ API. Therefore, please refer to our section on coupling your code for getting started. Some important differences:

- Call `import precice` at the beginning of your script.
- The object `precice.Interface` is the main access point to the preCICE API.
- We try to follow PEP8 with respect to function and class names. Meaning: `write_block_scalar_data`, not `writeBlockScalarData`, since this is a function call.
- Please use `numpy` arrays, if this seems appropriate. For scalar data a 1D-`numpy` with `size` entries should be used; for vector data a 2D-`numpy` array with `size x dimensions` entries should be used. This allows us to drop the `size` argument some functions calls. Meaning: not `writeBlockScalarData (int dataID, int size, int* vertexIDs, double* values)`, but `write_block_scalar_data(dataID, vertexIDs, values)`.

☑ **Tip:** You can use Python's `help()` function for getting detailed usage information. Example: Open a python3 shell, `import precice`, `help(precice.Interface)` or `help(precice.Interface.write_block_scalar_data)`

More details & troubleshooting

The python package and detailed documentation for the python bindings can be found here.

Matlab bindings

Summary: Clone the repository precice/matlab-bindings and run the installation script with Matlab

Matlab

The MATLAB bindings for preCICE can be found on github. For installation, please clone the repository via `git clone https://github.com/precice/matlab-bindings.git`. Then run the provided installation script `compile_matlab_bindings_for_precice.m` from MATLAB.

For troubleshooting, please refer to the `README.md`. Note that preCICE has to be installed on your system.

Special systems

This page contains instructions for building preCICE on special systems, being clusters and supercomputers.

The systems in the archived section (page 54) are no longer operational. The instructions may still be valuable for unlisted systems.

Note: We encourage users to actively contribute to this page! If your system is not listed, please feel encouraged to add instructions for it!

Active systems

HAWK (HPE Apollo/AMD, Stuttgart)

Building

The following steps explain how to install preCICE on HAWK with PETSc and MPI using the system standard HPE MPI implementation:

(1) Download Eigen and copy it to HAWK. Afterwards export the `EIGEN3_ROOT`, e.g.,

```
export EIGEN3_ROOT="$HOME/precice/eigen"
```

(2) Load available modules:

```
module load cmake boost petsc/<VERSION>-int32-shared
```

Note: libxml2 is part of the `-devel` packages, which are loaded by default on the login nodes. The compute nodes run in a diskless mode in order to save RAM. Therefore, make sure to use the login nodes for building purposes.

(3) Build preCICE. For PETSc, the library path and include path need to be defined explicitly:

```
cmake -DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX="my/install/prefix" -D
PRECICE_PETScMapping=ON -DPETSc_INCLUDE_DIRS="$PETSC_DIR/include" -DPETSc_LIBRARIES="$PETSC_DIR/li
b/libpetsc.so" -DPRECICE_PythonActions=OFF /path/to/precice/source make install -j 16
```

Usually, both variables, `PETSc_LIBRARIES` and `PETSc_INCLUDE_DIRS` are supposed to be found by `cmake`. This detection mechanism fails on Hawk and therefore we have to specify these variables on the command line. The reason for the detection mechanism to fail is unclear. It might be causes by our PETSc detection mechanism or might be an issue with the cluster. If you find a more native way to use the PETSc installation provided on Hawk, please update this documentation. The PETSc module, where this issue occurred, was `petsc/3.12.2-int32-shared`.

Note: In order to run the tests, you need to enable spawn capable MPI runs by specifying the global MPI universe size. This can be done by configuring `cmake` with the additional argument `-D MPIEXEC_NUMPROC_FLAG="-up;4;-np"` (environment variables can be exported as an alternative). All tests apart from the parallel integration test (which probably fails due to improper network specification) should pass afterwards. In order to run MPI spawn capable simulations (required for IQN-IMVJ, but not IQN-ILS) you need to specify the global MPI universe size using the `-up` flag as well, e.g., `mpirun -up 64 -np 32 ./my_solver arg1`

Running on a single node

Simulations on a single node are possible, but you explicitly need to specify the hardware. Otherwise, the MPI jobs are executed on the same cores, which will slow down the whole simulation due to migration significantly. In order to run the a coupled simulation on a single node with 8 ranks, use the following command:

```
mpirun -np 4 omplace -nt 1 ./exec1 args & mpirun -np 4 omplace -b 4 -nt 1 ./exec2 args2
```

The `nt` argument specifies the number of threads each rank uses. Since we don't want to use multi-threading, we select just a single thread per core. The argument option `-b` specifies the starting CPU number for the effective CPU list, so that we shift the starting number of CPU list in the second participant by the cores employed for the first participant. In our case we want to use 4 ranks/cores for each participant. There are further options to specify the hardware. Have a look at `omplace` using `man omplace` or the hardware pinning documentation for more information.

Notes on deal.II

`METIS` is preinstalled and can be loaded via the module system. In case that the preCICE modules above are loaded, `METIS` will already be loaded as a dependency of PETSc. However, in order to install deal.II with `METIS` support, you additionally need to enable a support for `LAPACK` (`DEAL_II_WITH_LAPACK=ON`) in the deal.II installation. In order to use `LAPACK` on Hawk, you can load the module `libflame`.

Additional dependencies of deal.II, such as `TRILINOS`, are available through module system and can be loaded accordingly. You can get obtain the a full list of preinstalled software on Hawk using the command `module avail`.

Notes on OpenFOAM

OpenFOAM is available on the system. You may want to call `module avail openfoam` for a complete overview of preinstalled OpenFOAM versions.

SuperMUC-NG (Lenovo/Intel, Munich)

Login: LRZ page

Available Modules

The LRZ provides a precice modules since 28. June 2021. These are built with PETSc as well as MPI using both GCC and the Intel compiler.

To display all precice modules:

```
module avail precice
```

Load using:

```
module load precice
```

Building

(1) Download Eigen and copy it to SuperMUC. Put in your `.bashrc`.

```
export EIGEN3_ROOT="$HOME/Software/eigen3"
```

(2) Download latest boost, copy it to SuperMUC and build yourself:

```
./bootstrap.sh --with-libraries=log,thread,system,filesystem,program_options,test --prefix=$HOME/Software/boost-install ./b2 install
```

Then, in your `.bashrc`:

```
export BOOST_ROOT=$HOME/Software/boost-install export LIBRARY_PATH=$BOOST_ROOT/lib:$LIBRARY_PATH export LD_LIBRARY_PATH=$BOOST_ROOT/lib:$LD_LIBRARY_PATH export CPLUS_INCLUDE_PATH=$BOOST_ROOT/include:$CPLUS_INCLUDE_PATH
```

(3) Some further modules you need:

```
module load cmake module load gcc
```

(4) Build

```
CXX=mpicxx cmake -DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_TYPE=Debug -DPETSC=OFF -DPYTHON=OFF
```

(5) Run tests. Create a `job.cmd` with

```
#!/bin/bash #SBATCH --time=00:10:00 #SBATCH -J tests #Output and error (also --output, --error): #SBATCH -o ./%x.%j.out #SBATCH -e ./%x.%j.err #Initial working directory (also --chdir): #SBATCH -D ./ #SBATCH --exclusive #SBATCH --partition=test #SBATCH --nodes=1 #SBATCH --ntasks-per-node=48 #SBATCH --account=pn56se #SBATCH --get-user-env module load slurm_setup rm -rf tests mkdir tests cd tests mpiexec -np 4 ../../Software/precice-1.6.1/build/debug/testprecice --log_level=test_suite --run_test="\!@MPI_Ports"
```

Notes on OpenFOAM

To get OpenFOAM and the OpenFOAM adapter to work, some hacks are needed.

(1) **OpenFOAM**: None of the OpenFOAM modules seem to work, but you can directly:

```
source /lrz/sys/applications/OpenFOAM/OpenFOAM-v1812+.impi.gcc/OpenFOAM-v1812/etc/bashrc_impi.gcc
```

Afterwards, you might need to reload

```
module load intel/19.0
```

and you also need to change

```
module swap mpi.intel/2019_gcc mpi.intel/2018_gcc
```

(2) **yaml-cpp**

Copy yaml-cpp to SuperMUC, 0.6.3 seems to work.

- From `yaml-cpp-yaml-cpp-0.6.3` path: `mkdir build` and `cd build`
- `CXX=gcc CC=gcc cmake -DYAML_BUILD_SHARED_LIBS=ON ..`
- `make yaml-cpp`
- and add to your `.bashrc`

```
export CPLUS_INCLUDE_PATH="$HOME/Software/yaml-cpp-yaml-cpp-0.6.3/include:${CPLUS_INCLUDE_PATH}" ex
port LD_LIBRARY_PATH="$HOME/Software/yaml-cpp-yaml-cpp-0.6.3/build:${LD_LIBRARY_PATH}"
```

(3) Build the **OpenFOAM adapter**

Notes on SWAK

- `ln -s swakConfiguration.automatic swakConfiguration`
- `export WM_NCOMPPROCS=16`
- `module load python/3.6_intel`
- download bison (`wget http://ftp.gnu.org/gnu/bison/bison-3.0.4.tar.gz`) and copy it to `privateRequirements/sources`

Job script for Ateles

```
#!/bin/bash #SBATCH --time=00:30:00 #SBATCH -J <name of your job> #Output and error (also --output,
--error): #SBATCH -o ./%x.%j.out #SBATCH -e ./%x.%j.err #Initial working directory (also --chdir):
#SBATCH -D ./ #SBATCH --mail-type=END #SBATCH --mail-user= <your email address> #SBATCH --exclusive
#SBATCH --partition=test #SBATCH --nodes=11 #SBATCH --ntasks-per-node=48 #SBATCH --account= <projec
t name> #SBATCH --ear=off module load slurm_setup module unload devEnv/Intel/2019 module load devEn
v/GCC export FC=mpif90 export CC=mpicc BOOST_ROOT=$HOME/lib_precice/boost export PRECICE_ROOT=$HOM
E/lib_precice/precice export LD_LIBRARY_PATH=$PRECICE_ROOT/build/last:$LD_LIBRARY_PATH export LIBRA
RY_PATH=$PRECICE_ROOT/build/last:$LIBRARY_PATH export LIBRARY_PATH=$BOOST_ROOT/lib:$LIBRARY_PATH ex
port LD_LIBRARY_PATH=$BOOST_ROOT/lib:$LD_LIBRARY_PATH export CPLUS_INCLUDE_PATH=$BOOST_ROOT/includ
e:$CPLUS_INCLUDE_PATH #UPDATE intervalk in seconds DELAY="60" rm -f simultan.machines rm -f *hosts
rm -fr .*address
```

```
echo "tpn: ${SLURM_TASKS_PER_NODE%%(*}" for i in `scontrol show hostname $SLURM_JOB_NODELIST`; do f
or j in $(seq 1 ${SLURM_TASKS_PER_NODE%%(*}); do echo $i >> simultan.machines; done done #### CAUTI
ON: NO NODE SHARING BETWEEN PARTICIPANTS IS ALLOWED! #### L1=1 L2=432 sed -n -e "${L1},${L2}p" ./si
multan.machines > dom1.hosts mpiexec -np 432 -hostfile dom1.hosts <Path to executable> <File to be
executed> &> <Output file> & PID1=$! echo $PID1 L1=433 L2=480 sed -n -e "${L1},${L2}p" ./simultan.m
achines > dom2.hosts mpiexec -np 37 -hostfile dom2.hosts <Path to executable> <File to be executed>
&> <Output file> & PID2=$! echo $PID2 L1=481 L2=528 sed -n -e "${L1},${L2}p" ./simultan.machines >
dom3.hosts mpiexec -np 2 -hostfile dom3.hosts <Path to executable> <File to be executed> &> <Output
file> & PID3=$! echo $PID3 check_and_kill() { ps $1 if (( $? )); then kill $2 return 1 else return
0 fi } while [[ 1 ]] do check_and_kill "$PID1" "$PID2" || exit check_and_kill "$PID2" "$PID1" || ex
it sleep $DELAY done
```

CooLMUC (LRZ Linux Cluster, Munich)

Get preCICE

You can use preCICE on the LRZ Linux Cluster (here CooLMUC2) by building it from source or use the provided module (since June 2021).

Use the preCICE module

Make sure that the module `spack/21.1.1` (or newer) is loaded. Checking via `module list` should give you an output similar to:

```
Currently Loaded Modulefiles: 1) admin/1.0 2) tempdir/1.0 3) lrz/1.0 4) spack/21.1.1
```

If `spack/21.1.1` is not loaded. Run `module load spack/21.1.1` first.

`module av precice` shows you the available preCICE modules. You can load preCICE by running `module load precice/2.2.0-gcc8-impi` or `module load precice/2.2.0-intel19-impi`. Make sure to also load the required compiler and MPI. E.g.:

```
module load gcc/8 intel-mpi/2019-gcc # we need the gcc compiler for FEniCS module load precice/
2.2.0-gcc8-impi
```

This gives on `module list`:

```
Currently Loaded Modulefiles: 1) admin/1.0 2) tempdir/1.0 3) lrz/1.0 4) spack/21.1.1 5) gcc/8.4.0
6) intel-mpi/2019-gcc 7) precice/2.2.0-gcc8-impi
```

Note: If you want to use FEniCS (see below), please stick to GCC from the very beginning.

Building with CMake

Warning: This page needs updates for preCICE v2 and the module system rolled out on CooLMUC in June 2021

If you load modules for any preCICE related installation, make sure the used MPI versions are consistent. This is also relevant for any solver you want to couple with preCICE. Therefore, it might be helpful to have a look in your solvers module installation before you start compiling preCICE. You can use `module show` to get information about specific modules.

since June 2021 most dependencies below (PETSc, Python, Boost) are available through the module system. Feel free to use these modules, if you want to build preCICE from source and update this section.

Basic building (without PETSc or Python)

Most of the necessary dependencies for a basic building are available via modules. We use here `mpi.intel/2018_gcc` for the MPI dependency as an example, since we later load an OpenFOAM module, which needs this MPI version.

```
module load gcc/7 module load mpi.intel/2018_gcc module load boost/1.68.0 # Read below if you need
yaml-cpp module load cmake/3.12.1
```

Before running the command `module load mpi.intel/2018_gcc` the user has to run `module unload mpi.intel` to unload the preloaded mpi version. Steps for the Eigen dependency are described in the wiki page for SuperMUC (page 0). Afterwards, follow the usual building instructions for CMake:

```
mkdir build && cd build cmake -DBUILD_SHARED_LIBS=ON -DPETSC=OFF -DPYTHON=OFF -DCMAKE_INSTALL_PREFI
X=/path/to/precice/installation -DCMAKE_BUILD_TYPE=RelWithDebInfo .. make -j 12 make install
```

After installing, make sure you add the preCICE installation paths to your `.bashrc`, so that other programs can find it:

```
export PRECICE_ROOT="path/to/precice_install" export PKG_CONFIG_PATH="path/to/precice_install/lib/p
kgconfig:${PKG_CONFIG_PATH}" export CPLUS_INCLUDE_PATH="path/to/precice_install/include:${CPLUS_INC
LUDE_PATH}" export LD_LIBRARY_PATH="path/to/precice_install/lib:${LD_LIBRARY_PATH}"
```

Boost and yaml-cpp

If you want to install a solver/adapter which depends on **yaml-cpp** (e.g. OpenFOAM adapter or CalculiX adapter), its compilation will probably lead to linking errors for yaml-cpp versions >= 0.6. Since a yaml-cpp < 0.6 requires boost < 1.67 and preCICE needs at least a boost version >= 1.65.1, we need to compile Boost from source. Therefore, download the desired (in your case 1.65.1) boost version from the boost version history and copy it to the cluster.

```
tar -xzvf boost_1_65_1.tar.gz cd boost_1_65_1 ./bootstrap.sh --with-libraries=log,thread,system,fil
esystem,program_options,test --prefix=/path/to/installation/target ./b2 install
```

This installs Boost in your prefix directory. You need to add the `prefix/lib` path to your `LD_LIBRARY_PATH` and your `prefix/include` path to your `CPLUS_INCLUDE_PATH`. Additionally, set the `BOOST_ROOT` according to your prefix. If the boost installation is done in a separate folder, result might look like:

```
export LIBRARY_PATH="path/to/boost_install/lib:${LIBRARY_PATH}" export LD_LIBRARY_PATH="path/to/boo
st_install/lib:${LD_LIBRARY_PATH}" export CPLUS_INCLUDE_PATH="path/to/boost_install/include:${CPLU
S_INCLUDE_PATH}" export BOOST_ROOT='path/to/boost_install'
```

Then, follow the description above (without loading the boost module).

You can also try not installing Boost, but directly using the `path/to/boost_source/libs` and `path/to/boost_source/boost` directories instead.

PETSc

There are some available versions of PETSc. You might want to pick one of them and install preCICE. In our case, the available versions are unfortunately not compatible with our (above) chosen MPI version and the compilation fails. Hence, we install our own PETSc version:

```
git clone -b maint https://bitbucket.org/petsc/petsc petsc
```

PETSc depends on BLAS and LAPACK. You could either download the LAPACK tar ball, which includes also BLAS from their webpage or you let PETSc download and compile it automatically, which is shown below.

The PETSc `configure` script will fail on the login nodes, (probably) since MPI is disabled. Hence, you need to start an interactive job, before you run the script. Details on how to do this can be found on the LRZ documentation.

If you login through lxlogin5 (CooLMUC2), you could do

```
salloc --ntasks=12
```

As mentioned above, we want to use `mpi.intel/2018_gcc`. You may get an error message if you run the configuration script without specifying the `mpi-dir`. If you use another version, you can look it up with `module show` under `I_MPI_ROOT`. Then, run the `configure` script and follow the instructions:

```
./configure --with-mpi-dir=/lrz/sys/intel/studio2018_p4/impi/2018.4.274 --download-fblaslapack=1
```

You may additionally specify a `--prefix` for the target directory. Then, you just need to set `PETSC_DIR= prefix`. If you don't specify it, you need to set `PETSC_DIR=/path/to/petsc` and `PETSC_ARCH`. Additionally, export your paths:

```
export LD_LIBRARY_PATH=$PETSC_DIR/$PETSC_ARCH/lib:$LD_LIBRARY_PATH export CPATH=$PETSC_DIR/includ
e:$PETSC_DIR/$PETSC_ARCH/include:$CPATH export LIBRARY_PATH=$PETSC_DIR/$PETSC_ARCH/lib:$LIBRARY_PAT
H export PYTHONPATH=$PETSC_DIR/$PETSC_ARCH/lib
```

Afterwards, you could follow the usual building instructions:

```
mkdir build && cd build CC=mpicc CXX=mpicxx cmake -DPETSC=ON -DPYTHON=OFF -DCMAKE_INSTALL_PREFIX=/p
ath/to/precice/installation -DCMAKE_BUILD_TYPE=Release ../.. make -j 12 make install
```

Run tests

If you are using the preCICE module

Testing the module is not necessary. You can still clone the preCICE repository and run the solverdummies, if you want to make sure:

```
git clone https://github.com/precice/precice.git cd precice/examples/solverdummies/cpp/ cmake . mak
e salloc --ntasks=1 # needed due to MPI ./solverdummy ../precice-config.xml SolverOne MeshOne & ./s
olverdummy ../precice-config.xml SolverTwo MeshTwo
```

If preCICE was build from source

Since the preCICE tests also need MPI, you need to start an interactive job as described above:

```
salloc --ntasks=12 cd $SCRATCH/precice/build ctest
```

Don't forget to source your paths and modules, if you don't specify them in your `.bashrc`.

Another option is the usage of a jobscript. An example might look like this:

```
#!/bin/bash #SBATCH -o $SCRATCH/clusteroutput.out #SBATCH -D $SCRATCH #SBATCH -J precice_tests #SBA
TCH --get-user-env #SBATCH --clusters=mpp2 #SBATCH --ntasks=28 #SBATCH --mail-type=end #SBATCH --ma
il-user=examplemail@domain.de #SBATCH --export=NONE #SBATCH --time=08:00:00 source /etc/profile.d/m
odules.sh cd $SCRATCH/ source modules.txt cd $SCRATCH/precice/build ctest
```

Run simulations

For running coupled simulations the user can launch both the solvers from a single job script, for example:

```
mpirun -np 4 ${Solver1} -parallel -case ${Participant1} > ${Participant1}.log 2>&1 & mpirun -np 4
${Solver2} -parallel -case ${Participant2} > ${Participant2}.log 2>&1 &
```

Alternatively the user can write a `Allrun_parallel` then the script can be directly launched from the job script.

More information about running parallel jobs on this cluster can be found on the LRZ documentation.

Start the job with `sbatch name_of_jobscript.job`.

Installing the Python bindings for Python 3 (with conda)

Preparing an environment

We will use conda for all python-related dependencies. Start with

```
module load anaconda3/2019.10
```

Now create an environment (here named `pyprecice`)

```
conda create -n pyprecice
```

If you are using conda the first time, then `$ conda activate pyprecice` might not work. Run `conda init bash`. Exit session end enter it again. Try again:

```
(base) $ conda activate pyprecice (pyprecice) $
```

The brackets before the `$` indicate the active environment.

Installing the Python bindings

We first activate the environment and install some dependencies via conda:

```
(base) $ conda activate pyprecice (pyprecice) $ git clone https://github.com/precice/python-binding
s.git (pyprecice) $ cd python-bindings (pyprecice) $ git checkout v2.2.0.2 # if you want to use a r
elease and not the develop version (pyprecice) $ conda install cython numpy mpi4py
```

Then install the bindings:

```
(pyprecice) $ python setup.py install
```

Testing

Again, you can test your installation by running the solverdummy:

```
(pyprecice) $ salloc --ntasks=1 (base) $ conda activate pyprecice (pyprecice) $ cd solverdummy (pyp
recice) $ python3 solverdummy.py precice-config.xml SolverOne MeshOne & python3 solverdummy.py prec
ice-config.xml SolverTwo MeshTwo
```

Note: after `salloc` you have to switch to the correct environment!

Installing FEniCS and fenicsprecice

Picking the right compiler and mpi implementation

Since FEniCS only support GCC, we will have to first unload the intel compiler and load gcc:

```
module unload intel-mpi/2019-intel intel/19.0.5 module load gcc/8 intel-mpi/2019-gcc module load pr
ecice/2.2.0-gcc8-impi
```

Install FEniCS

We will again use conda and continue using the environment `pyprecice` from above:

```
(base) $ conda activate pyprecice (pyprecice) $ conda install -c conda-forge fenics
```

You can do a quick test:

```
(pyprecice) $ python Python 3.7.10 (default, Jun 4 2021, 14:48:32) [GCC 7.5.0] :: Anaconda, Inc. on linux Type "help", "copyright", "credits" or "license" for more information. >>> import fenics >>> fenics.Expression("x[0] + x[1]", degree=0)
```

You might run into an error similar to this one:

```
In file included from /dss/dsshome1/lxc0E/ga25zih2/.conda/envs/fenicsproject/include/eigen3/Eigen/Core:96, from /dss/dsshome1/lxc0E/ga25zih2/.conda/envs/fenicsproject/include/eigen3/Eigen/Dense:1, from /dss/dsshome1/lxc0E/ga25zih2/.conda/envs/fenicsproject/include/dolfin/function/Expression.h:26, from /gpfs/scratch/pr63so/ga25zih2/ga25zih2/tmpdtucmkcr/dolfin_expression_523698ac7e42b5ce64e60789704de9c6.cpp:13: /dss/dsshome1/lrz/sys/spack/release/21.1.1/opt/x86_64/intel/19.0.5-gcc-uglchea/include/complex:305:20: note: field 'std::complex<double>::_ComplexT std::complex<double>::_M_value' can be accessed via 'constexpr std::complex<double>::_ComplexT std::complex<double>::__rep() const' 305 | return __x._M_value / __y;
```

Make sure to use `gcc`, not the intel compiler. Check via `module list`. If necessary `module unload intel...` and `module load gcc...`.

Install fenicsprecice

We will build fenicsprecice from source:

```
(base) $ conda activate pyprecice (pyprecice) $ git clone https://github.com/precice/fenics-adapter.git (pyprecice) $ cd fenics-adapter (pyprecice) $ git checkout v1.1.0 (pyprecice) $ python3 setup.py install
```

For testing, please clone the tutorials and try to run them:

```
(pyprecice) $ git clone https://github.com/precice/tutorials.git (pyprecice) $ cd tutorials (pyprecice) $ git checkout v202104.1.1 (pyprecice) $ cd tutorials/partitioned-heat-conduction/fenics (pyprecice) $ salloc --ntasks=1 (base) $ conda activate pyprecice (pyprecice) $ ./run.sh -d & ./run.sh -n
```

Quick-path to the tutorials:

Run this, if you log in and everything has already been prepared as described above:

```
module unload intel-mpi/2019-intel intel-mkl/2019 intel/19.0.5 module load gcc/8 intel-mpi/2019-gcc precice/2.2.0-gcc8-impi source activate pyprecice
```

Cartesius (Dutch national supercomputer)

modules and environment

```
module load 2020 module load CMake/3.16.4-GCCcore-9.3.0 PETSc/3.12.4-foss-2020a-Python-3.8.2 Eigen/3.3.9-GCCcore-9.3.0 ScaLAPACK/2.1.0-gompi-2020a
```

After loading these modules you can proceed with the `cmake` build steps for preCICE.

For python bindings,

```
CPATH=<PRECICE_DIR>/include/ pip install pyprecice
```

Replace `PRECICE_DIR` with the installation prefix used for preCICE. Also, make sure that preCICE libraries locations are in `LD_LIBRARY_PATH` and `LIBRARY_PATH`.

Archived systems

Hazel Hen (Cray/Intel, Stuttgart)

Warning: This page needs updates for preCICE v2.

Building on Hazel Hen

Modules on Hazel Hen

```
module swap PrgEnv-cray PrgEnv-gnu module load cray-python/3.6.1.1 module load cray-petsc # if dist
ributed RBF are used. module load tools/boost/1.66.0
```

Other dependencies

PETSc on Hazel Hen

Just load the module and use `petsc=on` for compilation: `module load cray-petsc`

Compilation

You **must** use `platform=hazelhen` for compilation.

```
scons petsc=on python=off compiler=CC platform=hazelhen
```

Executing

aprun

`aprun` is used instead of `mpirun`. However, you can execute only one `aprun` per node, i.e. for two aprun calls you must reserve at least two nodes.

Network Interface

Use `ipogif0` for socket communication.

SuperMUC (Lenovo/Intel, Munich)

Warning: This page needs updates for preCICE v2.

:information_source: SuperMUC was shut down in 2019. This page may still be useful for other clusters. See also the instructions for SuperMUC-NG (page 0).

Building with CMake

Build

Building preCICE on SuperMUC or other LRZ systems is very similar to building it locally. The main differences are that we can easily get most of the dependencies through the module system.

Basic building (without Python)

You may build preCICE without PETSc or Python and still use most of its features. See also the general build instructions (page 0).

(1) Load some modules (or directly put them in your `.bashrc`)

```
module load gcc/6 module swap mpi.ibm mpi.intel/5.1 module load petsc/3.8 module load boost/1.65_gc
c export BOOST_ROOT=$BOOST_BASE
```

(2) Get Eigen: Download the latest version of Eigen and copy it to SuperMUC. Specify the path in your `.bashrc`, e.g.

```
export EIGEN3_ROOT="$HOME/eigen3"
```

- Don't only get the `eigen/Eigen` folder, copy the complete archive.
- Don't copy it to your `precice/src`, copy it to a separate directory (e.g. `eigen3`)

(3) Get cmake. The newest version on SuperMUC (3.8) is, unfortunately, too old. We need >= 3.10.2. Download the latest stable version and copy it to SuperMUC. Then, e.g.

```
tar -xzvf cmake-3.13.4.tar.gz ./bootstrap make -j28 export PATH=$PATH:$HOME/cmake-3.13.4/bin
```

(4) Build preCICE. In the root directory of preCICE:

```
mkdir build/no_python cd build/no_python CXX=mpicxx cmake -DPYTHON=OFF -DCMAKE_BUILD_TYPE=Release
../.. make -j28
```

Without PETSc

```
mkdir build/no_petsc_no_python cd build/no_petsc_no_python CXX=mpicxx cmake -DPETSC=OFF -DPYTHON=OF
F -DCMAKE_BUILD_TYPE=Release ../.. make -j28
```

Python

So far, we did not get Python to work. Please let us know if you do.

Run tests

Use the job system, to run the tests. Get a standard job script and run:

```
#!/bin/bash #@ wall_clock_limit = 00:30:00 #@ job_type = MPICH #@ job_name = Tests #@ class = test
#@ island_count = 1,1 #@ network.MPI = sn_all,not_shared,us #@ node = 1 #@ tasks_per_node = 28 #@ o
utput = $(job_name).out #@ error = $(job_name).err #@ initialdir = $(home)/precice/build/no_python
#@ energy_policy_tag = my_energy_tag #@ minimize_time_to_solution = yes #@ queue . /etc/profile .
/etc/profile.d/modules.sh ctest
```

Run simulations

When using socket communication on SuperMUC (as well as other LRZ clusters), it is important to specify in the preCICE configuration file that the communication should happen through the Infiniband network (`network="ib0"`), instead of the local network (`network="lo"`).

MAC Cluster (various architectures, Munich)

Warning: This page needs updates for preCICE v2.

:information_source: The MAC Cluster was shut down in 2018. However, these instructions may also be useful for users of other HPC systems.

General Information

Read first some information about the MAC-Cluster and about Running parallel jobs with SLURM.

You may allocate an interactive shell like this:

```
salloc --partition=snb --ntasks=1 --cpus-per-task=32
```

Then you can run your executable in the interactive shell: e.g.

```
mpiexec.hydra -ppn 1 -n 1 ./executable parameters ...
```

Note that each node has 16 physical resp. 32 virtual cores. This means that the following combinations should be applied for scaling tests (to always reserve full nodes): (salloc ntasks, salloc cpus-per-task, mpiexec ppn, mpiexec n): (1,32,1,1), (2,16,2,2), (4,8,4,4), (8,4,8,8), (16,2,16,16), (32,2,16,32), (64,2,16,64), ...

Building preCICE

To build preCICE on the MAC Cluster you may follow the same instructions for building for SuperMUC (page 0). In SCons you need to set `platform=supermuc` also in the case of the MAC Cluster. Note that, in contrast to SuperMUC, you can access GitHub from the MAC Cluster.

You should build preCICE on the login node of the MAC Cluster partition that you are going to use.

Running the tests

In order to run the tests, the same instructions as for SuperMUC (page 0) apply. After you load the correct modules, you may execute the tests in a compute node, from your `PRECICE_ROOT` directory:

```
salloc --partition=snb --ntasks=1 --cpus-per-task=32 srun ./tools/compileAndTest.py -t exit
```

Note: Before preCICE v1.1.0, the `-t` option was named `-b` .

MareNostrum (Lenovo/Intel, Barcelona)

Warning: This page needs updates for preCICE v2.

Build

See also the general build instructions (page 0).

- To be put in your `.bashrc` :

```
module load python #needed by scons module unload intel module load gcc/4.8.2 module switch openmpi
impi export PRECICE_BOOST_ROOT=$HOME/software/boost_1_53_0
```

- copy boost_1_53_0 to `software` (no installation needed!)

- copy `Eigen` to `src` (just as usual)
- `scons petsc=off python=off compiler=mpicxx build=release platform=supermuc -j32`

ALYA

- you have to further `module load intel` (but only once preCICE is compiled)
- configure ALYA with `-L[PathToPreCICE]/build/last -lprecice -lstdc++ -lrt`
- for running: also put `module load intel` in your jobscript
- use `network="ib0"` for sockets communication beyond one node

Max Planck Computing and Data Facility (MPCDF) Cobra cluster

Installing dependencies

Eigen3

See the Eigen dependency section as Eigen is not available as a module on this cluster.

Installing preCICE

On the Cobra cluster, you can easily install preCICE from source. Clone the repository or copy the code to the cluster, set the installation prefix paths as shown in this section, and then run the following commands:

```
module purge module load gcc/9 impi/2019.7 cmake/3.18 petsc-real boost/1.74 module list rm -rf buil
d mkdir -p build && cd build cmake -DBUILD_SHARED_LIBS=ON -DMPI_CXX_COMPILER=mpigcc -DCMAKE_BUILD_T
YPE=Debug -DPRECICE_PythonActions=OFF .. make -j
```

Demo Virtual Machine

Summary: A sandbox to try preCICE and all the adapters without having to install them on your system.

Do I need this?

You probably only want to use this if you are very new to preCICE and want to learn, for example during our preCICE Workshops or other conferences (page 0) where we may be present with a training session.

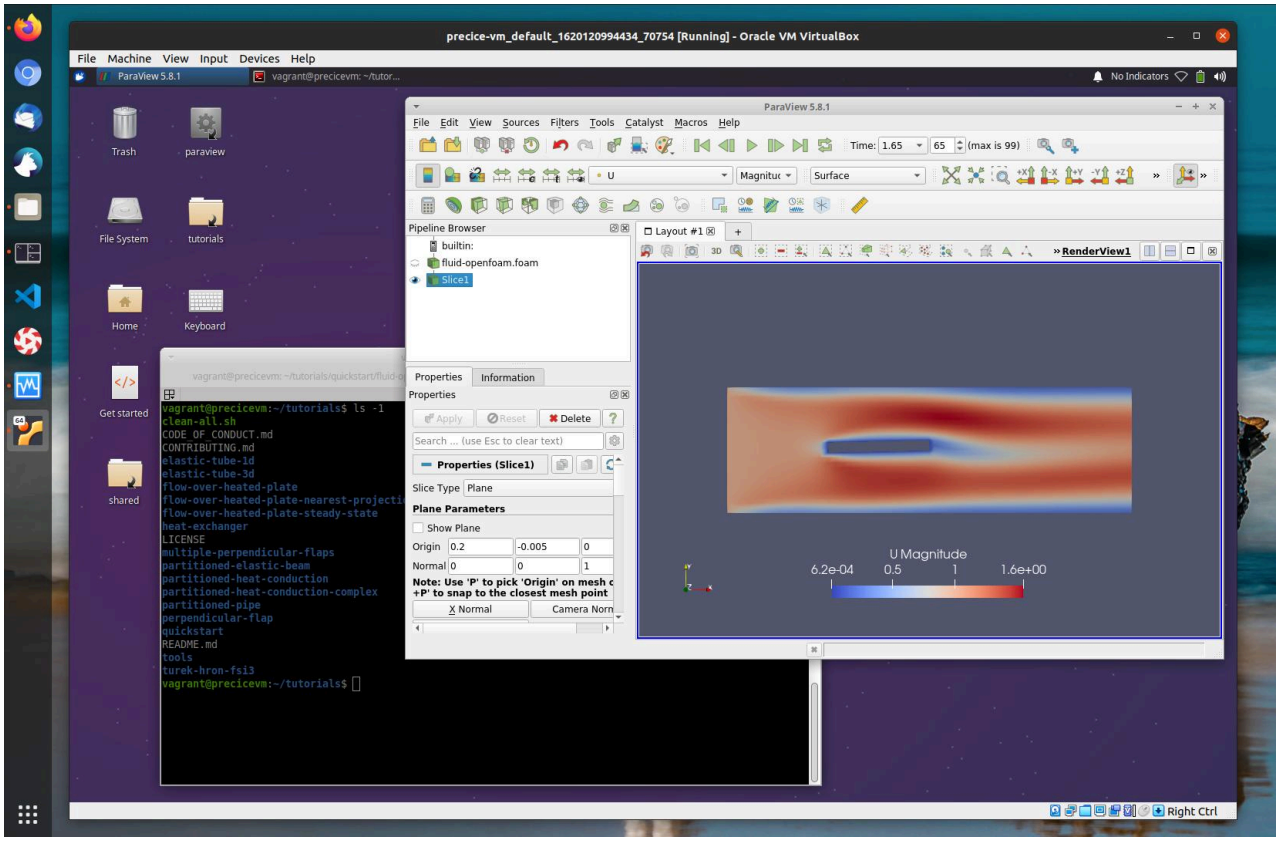
After trying this out for a few days, you probably want to just delete it and install only the components you need directly on your target system.

What is this?

This is a Vagrant box, essentially a VirtualBox virtual machine image, with additional automation to make it easier for you to use and for us to maintain.

After installing this on any operating system, you will be able to start a virtual machine with Linux and a lightweight graphical interface. You can do anything you like in there, without breaking anything. This will download a very large file (~5GB), will occupy significant storage space (~15GB), and will reserve 2GB of main memory while running, but you can easily delete it when you don't need it anymore.

This contains all the solvers and adapters used in our tutorials, already built and configured for you to enjoy.



See what is included in detail.

How to use this?

You can use this on any mainstream operating system (Windows/macOS/Linux), but it is necessary that your CPU supports virtualization (most systems nowadays do) and that this is enabled in your BIOS/UEFI settings. Note once more that this will consume a significant amount of storage and main memory space.

1. Install Vagrant **and** VirtualBox (6.0 or later).
2. Start your terminal / command prompt and go to an empty directory.
3. Run `vagrant init precice/precice-vm` to prepare the directory.
4. Run `vagrant up` to download the box (~4GB) and start the system.
5. Open VirtualBox: A new virtual machine should be running. Double-click to open its window. The login password in `vagrant`.

Alternatively, you can connect to the vm while being able to use graphical applications using `vagrant ssh -- -Y`.

After logging in, start a terminal (e.g. terminator) from the applications menu. On the Desktop (`cd ~/Desktop/`) you can find the basic solverdummy examples, as well as the tutorials.

You can turn off the system normally from the GUI and start it again with `vagrant up`.

What else may I want to do?

Sharing files and clipboard

Vagrant gives access to the same directory where you downloaded the box into. If you add any files there from your host system, you will be able to see them in `~/Desktop/shared` or in `/vagrant/`.

You can enable copy-pasting text by clicking in the VirtualBox menu bar at `Devices > Shared Clipboard > Bidirectional`. The VirtualBox Guest Additions that enable this are already installed.

Even though you can directly start the VM again by clicking on it in VirtualBox, it is important that you start it with `vagrant up` to set up these features.

Changing the keyboard layout

The default keyboard layout is US English (QWERTY). Change this clicking on the `Keyboard` link on the Desktop, removing the already added en-us layout, and adding yours.

Installing additional software

You can install additional software using `sudo apt install <package>`, without any password.

In terms of editors, gedit, vim, and nano are already installed. If you need a more advanced editor with a GUI, you can install VSCode by running `~/install-vscode.sh`. If you double-click on it, it will run silently. Wait for a bit and you will then find it under a new category `Development` in the applications menu.

Updating the system

preCICE, the tutorials, and all adapters are installed from their Git repositories in the home directory, using their main/master branches. You can do a `git pull` at any time to get the latest state of each package.

You can also update the complete box, but this will delete the previous one and you will lose any changes.

Deleting everything

To go back to the state before trying this, run `vagrant destroy`, `vagrant box remove precice/precice-vm`, and uninstall Vagrant and VirtualBox.

I found an issue

Please report any technical issues on the vm repository on GitHub. Should we definitely include some package you love? Let us know! For general support, please refer to our community channels (page 0).

The preCICE distribution

Summary: A frozen state of component versions that work together.

What is the preCICE distribution?

preCICE is much more than the core library: it is a larger ecosystem that includes language bindings, adapters for popular solvers, tutorials, and more. We know that it can be difficult to figure out which versions to install, therefore we will be publishing here lists of known-to-work versions.

v2104.0

This is the first preCICE distribution version, coming after the restructuring of our tutorials.

It comprises of the following components:

- preCICE: v2.2.0
- Python bindings: v2.2.0.2
- Fortran module: commit 9826f27
- Matlab bindings: v2.2.0.1
- OpenFOAM adapter: v1.0.0
- deal.II adapter: commit 685508e
- FEniCS adapter: v1.1.0
- CalculiX adapter: commit 9fefcef
- SU2 adapter: commit ab84387
- code_aster adapter: commit ce995e0
- Tutorials: v202104.1.1
- vm: v202104.1.0

Configuration overview

Summary: preCICE needs to be configured at runtime via an `xml` file, typically named `precice-config.xml`. Here, you specify which solvers participate in the coupled simulation, which coupling data values they exchange, which numerical methods are used for the data mapping and the fixed-point acceleration and many other things. On this page, we give you an overview of the complete configuration section of the documentation.

You are new to preCICE and want to learn how the configuration works?

Have first a look at the introduction page (page 64). Here, we explain in which basic sections the configuration is structured and how the different sections are connected. Afterwards you can study the details of the main parts:

- Mapping configuration (page 67)
- Communication configuration (page 70)
- Coupling scheme configuration (page 71)
- Acceleration configuration (page 73)
- Mesh exchange example (page 0)

And some optional advanced parts:

- Logging configuration (page 79)
- Exports configuration (page 82)
- Action configuration (page 83)
- Watchpoint configuration (page 87)
- Multi coupling configuration (page 78)

You are already familiar with the preCICE configuration, but you don't remember how a certain option was called?

Then you should look at the configuration reference (page 89). Also simply try the search here on top. The configuration reference is up to date with the last release of preCICE. If you need an older version, you can always generate this documentation yourself:

```
./binprecice md > reference.md
```

There is also an `xml` variant of the reference. Just call `binprecice` without arguments to see all options.

You want to visualize your configuration file?

Visualizing the configuration file is a good way to spot mistakes, but also to learn how the configuration is structured. Do not forget to try out the configuration visualizer (page 162).

You want to port your configuration file from preCICE v1.x to v2.x?

There is a seperate page with all steps required for porting (page 251).

Note: The parsing of floating point numbers in the configuration files depends on your system locale. If you get errors emitted by `xml::XMLAttribute`, then please set the locale to `export LANG=en_US.UTF-8`.

Introduction to configuration

> **Summary:** The preCICE configuration file is structured in several sections. It is important to understand what the section are and how they are connected. On this page, we explain you that.

The configuration consists, in general, of the following five parts:

```
<precice-configuration> <solver-interface dimensions="3"> <data .../> <mesh .../> <participant
.../> <m2n .../> <coupling-scheme .../> </solver-interface> </precice-configuration>
```

Note: On this page, you also find references to the preCICE API. If you are only using (and not developing) an adapter, don't panic: you can use these references to get a better understanding, but you don't need to change anything in your adapter.

0. Dimensions

The value `dimensions` needs to match the physical dimension of your simulation, i.e. the number of coordinates a vertex has in `setMeshVertex`, etc. Some solvers only support 3D simulation, such as OpenFOAM or CalculiX. In this case the adapter maps from 3D to 2D if the preCICE dimension is 2D. This, of course, only works if you simulate a quasi-2D scenario with one layer of cells in z direction.

1. Coupling data

You need to define which data values the coupled solvers want to exchange, e.g. displacements, forces, velocities, or temperature.

```
<data:scalar name="Temperature"/> <data:vector name="Forces"/>
```

Once you have defined these fields, you can use the preCICE API to access them:

```
int temperatureID = precice.getDataID("Temperature", meshID);
```

2. Coupling meshes

Next, you can define the interface coupling meshes.

```
<mesh name="MyMesh1"> <use-data name="Temperature"/> <use-data name="Forces"/> </mesh>
```

With the preCICE API, you get an ID for each mesh:

```
int meshID = precice.getMeshID("MyMesh1");
```

3. Coupling participants

Each solver that participates in the coupled simulation needs a participant definition. You need to define at least two participants.

```
<participant name="MySolver1"> <use-mesh name="MyMesh1" provide="yes"/> <read-data name="Temperatur
e" mesh="MyMesh1"/> <write-data name="Forces" mesh="MyMesh1"/> ... </participant>
```

The name of the participant has to coincide with the name you give when creating the preCICE interface object in the adapter:

```
precice::SolverInterface precice("MySolver1",rank,size);
```

The participant `provides` the mesh. This means that you have to define the coordinates:

```
precice.setMeshVertices(meshID, vertexSize, coords, vertexIDs);
```

The other option is to receive the mesh coordinates from another participant (who defines them):

```
<use-mesh name="MyMesh2" from="MySolver2"/>
```

If a participant uses at least two meshes, you can define a data mapping between both:

```
<mapping:nearest-neighbor direction="read" from="MyMesh2" to="MyMesh1" constraint="consistent"/>
```

`nearest-neighbor` means that the nearest-neighbor mapping method is used to map data from `MyMesh1` to `MyMesh2`.

Read more about the mapping configuration (page 67).

4. Communication

If two participants should exchange data, they need a communication channel.

```
<m2n:sockets from="MySolver1" to="MySolver2" />
```

Read more about the communication configuration (page 70).

5. Coupling scheme

At last, you need to define how the two participants exchange data. If you want an explicit coupling scheme (no coupling subiterations), you can use:

```
<coupling-scheme:parallel-explicit> <participants first="MySolver1" second="MySolver2"/> <max-time
value="1.0"/> <time-window-size value="1e-2"/> <exchange data="Forces" mesh="MyMesh2" from="MySolve
r1" to="MySolver2"/> <exchange data="Temperature" mesh="MyMesh2" from="MySolver2" to="MySolver1"/>
</coupling-scheme:parallel-explicit>
```

`parallel` means here that both solver run at the same time. In this case, who is `first` and `second` only plays a minor role. `max-time` is the complete simulation time. After this time,

```
precice.isCouplingOngoing()
```

will return `false`. The `time-window-size`, is the coupling time window (= coupling time step) length. This means if a solver uses a smaller time step size, he subcycles, i.e. needs more smaller time steps until data is exchanged.

Both participants need to `use` the mesh over which data is `exchanged` (here `MyMesh2`).

For implicit coupling, i.e. both solver subiterate in every time window until convergence, the configuration looks a bit more complicated.

Read more about the coupling scheme configuration (page 71).

☑ **Tip:** Visualizing the configuration helps a lot in understanding the connections between these five parts. Do not forget to try out the configuration visualizer (page 162).

Mapping configuration

Summary: When coupling two participants at a common coupling interface, in general, the two surface meshes do not match. Therefore, preCICE provides data mapping methods to map coupling data from one mesh to the other. On this page, we explain how to configure such data mapping methods.

Basics

A participant needs to `use` at least two meshes to define a mapping between both:

```
<participant name="MySolver1"> <use-mesh name="MyMesh1" provide="yes"/> <use-mesh name="MyMesh2" fr
om="MySolver2"/> <write-data name="Forces" mesh="MyMesh1"/> <read-data name="Temperature" mesh="MyM
esh1"/> <mapping:nearest-neighbor direction="read" from="MyMesh2" to="MyMesh1" constraint="consiste
nt"/> <mapping:nearest-neighbor direction="write" from="MyMesh1" to="MyMesh2" constraint="conservat
ive"/> </participant>
```

Mappings can be defined in two `directions`, either `read` or `write`:

- `read` mappings are executed *before* you can read data from the mesh. In the example above, `Temperature` is received on `MyMesh2`, then it is mapped from `MyMesh2` to `MyMesh1` and, finally, read from `MyMesh1`.
- `write` mappings are executed *after* you have written data.

Furthermore, mappings come int two types of `constraint`: `consistent` and `conservative`:

- `conservative` mapping: Mapping between different meshes (example, from a fine to a coarse grid), the value at a coarse node is computed as an aggregation of the corresponding fine nodes, such that the total coupling value (in our example `Forces`) on the coarse and fine mesh is the same. This is required for quantities that are absolute (extensive quantities, such as force, mass, etc.). An example for a nearest-neighbor mapping could look like this:

```
f=2 f=1 f=2 f=1 f=1 ------+------+------+------+------+------ \ | / | / -------------+-------------
+------------- f=(2+1+2) f=(1+1)
```

- `consistent` mapping: For quantities that are normalized (intensive quantities; `Temperature` in our example, or pressure, which is force *per unit area*), we need a consistent mapping. This means that the value at coarse nodes is the same as the value at the corresponding fine node. Again, an example for a nearest-neighbor mapping could look like this:

```
T=2 T=1 T=2 T=1 T=1 ------+------+------+------+------+------ | | -------------+-------------
+------------- T=1 T=1
```

For a sequential participant, any combination of `read` / `write` - `consistent/conservative` is valid. For a parallel participant (i.e. a `master` tag is defined), only `read` - `consistent` and `write` - `conservative` is possible. More details are given further below (page 69).

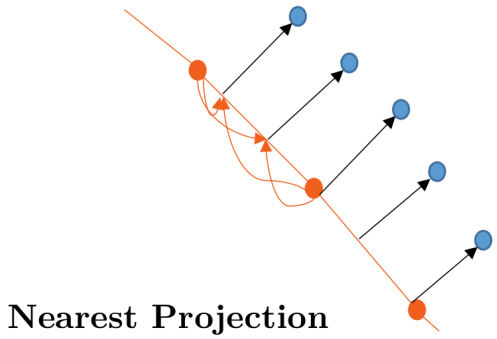
Furthermore, mappings have an optional parameter `timing`, it can be:

- `initial` (the default): The mapping is only computed once, the first time it is used. This is sufficient for stationary meshes (also including the reference mesh in an Lagrangian or an ALE description).

- `onadvance`: The mapping is newly computed for every mapping of coupling data. This can be expensive and is only recommend if you know exactly why you want to do this.
- `ondemand`: Data is not mapped in `initialize`, `initializeData`, or `advance`, but only if steered manually through `mapReadDataTo` resp. `mapWriteDataFrom`. Only use this if you are sure that your adapter uses theses methods.

Concerning mapping methods, preCICE offers three variants:

- `nearest-neighbor`: A first-order method, which is fast, easy to use, but, of course, has numerical deficiencies.
- `nearest-projection`: A (mostly) second-order method, which first projects onto mesh elements and, then, uses linear interpolation within each element (compare the figure below). The method is still relatively fast and numerically clearly superior to `nearest-neighbor`. The downside is that mesh connectivity information needs to be defined, i.e. in the adapter, the participant needs to tell preCICE about edges in 2D and edges, triangles, or quads (see issue) in 3D. On the mesh connectivity page (page 239), we explain how to do that. If no connectivity information is provided, `nearest-projection` falls back to an (expensive) `nearest-neighbor` mapping.
- Radial-basis function mapping. Here, the configuration is more involved, so keep reading.



Radial-basis function mapping

Radial basis function mapping computes a global interpolant on one mesh, which is then evaluated at the other mesh. The global interpolant is formed by a linear combination of radially-symmetric basis functions centered on each vertex, enriched by one global linear polynomial. For more information, please refer, e.g., to Florian's thesis (pages 37 ff.) or to this paper and the reference therein.

To compute the interpolant, a linear equation system needs to be solved in every mapping of data. We use either:

- the external library Eigen and a QR decomposition, or
- the external library PETSc and a GMRES solver.

For small/medium size problems, the QR decomposition is enough and you don't need to install anything else. However, this follows a gather-scatter approach, which limits the scalability. For large problems, the GMRES solver performs better than the QR decomposition. For this, you need to build preCICE with PETSc. If you built with PETSc, the default is always GMRES. If you still want to use the QR decomposition, you can use the option `use-qr-decomposition`.

Radial basis function mapping also behaves as a second-order method just as `nearest-projection`, but without the need to define connectivity information. The downside is that it is normally more expensive to compute and that it shows numerical problems for large or highly irregular meshes.

The configuration might look like this:

```
<mapping:rbf-thin-plate-splines direction="read" from="MyMesh2" to="MyMesh1" constraint="consisten
t"/>
```

`thin-plate-splines` is the type of basis function used. preCICE offers basis function with global and local support:

- Basis function with global support (such as `thin-plate-splines`) are easier to configure as no further parameter needs to be set. For larger meshes, however, such functions lead to performance issues in terms of algorithmic complexity, numerical condition, and scalability.
- Basis functions with local support need either the definition of a `support-radius` (such as for `rbf-compact-tps-c2`) or a `shape-parameter` (such as for `gaussian`). To have a good trade-off between accuracy and efficiency, the support of each basis function should cover three to five vertices in every direction. You can use the tool rbfShape.py to get a good estimate of `shape-parameter`.

For a complete overview of all basis function, refer to this paper, page 5.

The interpolation problem might not be well-defined if you map along an axis-symmetric surface. This means, preCICE tries to compute, for example, a 3D interpolant out of 2D information. If so, preCICE throws an error `RBF linear system has not converged` or `Interpolation matrix C is not invertible`. In this case, you can restrict the interpolation problem by ignoring certain coordinates, e.g. `x-dead="true"` to ignore the x coordinate.

Note: All data mappings are executed during `advance` and not in `readBlockVectorData` etc., cf. the section on how to couple your own code (page 222).

Restrictions for parallel participants

As stated above, for parallel participants only `read` - `consistent` and `write` - `conservative` are valid combinations. If want to find out why, have a look at Benjamin's thesis, page 85. But what to do if you want a `write` - `consistent` mapping? The trick is to move the mapping to the other participant, then `write` becomes `read`:

- Move the mapping, adjust `write` to `read`
- Be sure that the other participant also uses both meshes. Probably you need an additional `<use-mesh name="MyMesh1" from="MySolver1"/>`. This means another mesh is communicated at initialization, which can increase initialization time.
- Last, be sure to update the `exchange` tags in the coupling scheme, compare the coupling scheme configuration (page 0) (e.g. change which mesh is used for the exchange and acceleration)

After applying these changes, you can use the preCICE Config Visualizer to visually validate your updated configuration file.

Maybe an example helps. You find one in the preCICE Forum.

Communication configuration

Summary: A very basic ingredient to coupling is communication. The participants you want to couple need to be able to communicate data. On this page, we explain how communication between participants can be configured.

The m2n tag

For each two participants that should exchange data, you have to define an m2n communication, for example like this:

```
<m2n:sockets from="MySolver1" to="MySolver2" exchange-directory="../"/>
```

This establishes an `m2n` (i.e. parallel, from the M processes of the one participant to the N processes of the other) communication channel based on TCP/IP `sockets` between `MySolver1` and `MySolver2`.

For certain systems, you need to specify the network over which the TCP/IP sockets get connected: `network="..."`. It defaults to `"lo"`. For some clusters, you could use the infiniband, e.g. `"ib0"`. macOS is also a special case (page 0).

The alternative to TCP/IP sockets is MPI ports (an MPI 2.0 feature):

```
<m2n:mpi .../>
```

As the ports functionality is not a highly used feature of MPI, it has robustness issues for several MPI implementations (for OpenMPI, for example (page 0)). In principle, MPI gives you faster communication roughly by a factor of 10, but, for most applications, you will not feel any difference as both are very fast. We recommend using `sockets`.

Which participant is `from` and which one is `to` makes almost no difference and cannot lead to deadlock. Only for massively parallel runs, it can make a performance difference at initialization. For such cases, ask us for advice (page 0).

The `exchange-directory` should point to the same location for both participants. We use this location to exchange hidden files with initial connection information. It defaults to `"."`, i.e. both participants need to be started in the same folder. We give some best practices on how to arrange your folder structure and start the coupled solvers here (page 0).

⚠ **Important:** If you face any problems with establishing the communication, have a look here (page 0).

Advanced: the master tag

If you build preCICE without MPI (and **only** in this case) you might also need to change the communication preCICE uses to communicate between ranks of a single parallel participant. You can specify to use TCP/IP sockets with:

```
<participant name="MySolver1"> ... <master:sockets/> ... </participant>
```

Coupling scheme configuration

Summary: The coupling scheme is the centerpiece of the preCICE configuration. It describes the logical execution order of two or more participants. On this page, we explain how to couple two participants.

A coupling scheme can be either serial or parallel and either explicit or implicit. Serial refers to the staggered execution of one participant after the other. Parallel, on the other hand, refers to the simultaneous execution of both participants. With an explicit scheme, both participants are only executed once per time window. With an implicit scheme, the participants are executed multiple times until convergence.

For coupling more than two participants, please see the page on multi coupling (page 78).

Explicit coupling schemes

For a serial-explicit coupling scheme, your configuration could look like this:

```
<coupling-scheme:serial-explicit> <participants first="MySolver1" second="MySolver2"/> <max-time-windows value="20"/> <time-window-size value="1e-3"/> <exchange data="Forces" mesh="MyMesh2" from="MySolver1" to="MySolver2"/> <exchange data="Temperature" mesh="MyMesh2" from="MySolver2" to="MySolver1"/> </coupling-scheme:serial-explicit>
```

With `participants` tag, you define which participants are coupled in this scheme. For a serial scheme, the `first` participant is computed before the `second` one. For a parallel-explicit scheme, simply write:

```
<coupling-scheme:parallel-explicit> <participants first="MySolver1" second="MySolver2"/> ... </coupling-scheme:parallel-explicit>
```

Now, the `first` and the `second` participant are executed at the same time. Actually, it makes no real differences who is who here (it will make a difference for implicit coupling, but that is described further down).

With `max-time-windows`, you say how many coupling time windows you want to simulate. Alternatively, you can use:

```
<max-time value="1.0"/>
```

Afterwards,

```
precice.isCouplingOngoing()
```

will return false and `precice.finalize()` should be called (compare with step 5 of the couple-your-code section (page 235)).

With `time-window-size`, you can define the coupling time window (=coupling time step) size. If a participant uses a smaller one, it will subcycle until this *window* size is reached. Find more details also in step 5 of the couple-your-code section (page 232).

Finally, with `exchange`, you need to define which data values should be exchanged within this coupling scheme:

```
<exchange data="Forces" mesh="MyMesh2" from="MySolver1" to="MySolver2"/>
```

`mesh` needs to be a mesh that both participant `use`, typically one participant provides the mesh and the other receives it, as we explained on the introduction page (page 64). If this still confuses you have a look at the mesh exchange example (page 76).

Implicit coupling schemes

For implicit coupling, you need to specify several additional options:

```
<coupling-scheme:parallel-implicit> <participants first="MySolver1" second="MySolver2"/> ... <excha
nge data="Temperature" mesh="MyMesh2" from="MySolver2" to="MySolver1"/> <max-iterations value="10
0"/> <relative-convergence-measure limit="1e-4" data="Displacements" mesh="MyMesh2"/> <relative-con
vergence-measure limit="1e-4" data="Forces" mesh="MyMesh2"/> <acceleration:IQN-ILS> ... </accelerat
ion:IQN-ILS> </coupling-scheme:parallel-implicit>
```

To control the number of sub-iterations within an implicit coupling loop, you can specify the maximum number of iterations, `max-iterations` and you can specify one or several **convergence measures**:

- `relative-convergence-measure` for a relative criterion
- `absolute-convergence-measure` for an absolute criterion
- `min-iteration-convergence-measure` to require a minimum of iterations
  If multiple convergence measure are combined they all need to be fulfilled to go to the next time window. Alternatively, you can specify `suffices="yes"` within any convergence measure. The data used for a convergence measure needs to be exchanged within the coupling-scheme (tag `exchange`).

Each convergence measure prints its current state as INFO logging in every coupling iteration (how to configure the logging (page 79)). For example for a `relative-convergence-measure`:

```
relative convergence measure: relative two-norm diff = 2.6023e-05, limit = 1e-05, normalization =
0.00100051, conv = false
```

- `relative two-norm diff` is the relative coupling residual ||H(x^k) - x^k||_2 / ||x^k||_2 for a fixed-point equation H.
- `limit` is the convergence limit specified in the configuration.
- `normalization` is the normalization factor ||x^k||_2.

Most important for implicit coupling is to use a **acceleration scheme**, i.e. to let preCICE modify the exchanged data. We give more details on the acceleration configuration page (page 73). For numerical reasons, you should always use a acceleration for implicit coupling. Otherwise, an implicit coupling has no benefit over an explicit coupling. You can only define one acceleration per coupling scheme.

Additionally, you can speed up an implicit coupling by using an extrapolated value from previous time windows as initial guess, `<extrapolation-order value="2"/>`. This tag is optional and requires some trial-and-error tuning as extrapolation does not always result in fewer iterations. Use with care!

For implicit coupling, the tags `first` and `second` do not only determine the order of execution (for serial coupling), but they also determine where preCICE computes the convergence measures and the acceleration: Both are executed on the `second` participant.

Besides `parallel-implicit`, you can also use a `serial-implicit` coupling. However, for performance reasons, we recommend to use `parallel-implicit`. To explain this is beyond the scope of this documentation. We refer, instead, to the respective publications (page 8).

Did you know, you can also inspect the number of iterations and the residuals through log files? Have a look at the output files description (page 0).

Acceleration configuration

Summary: Mathematically, implicit coupling schemes lead to fixed-point equations at the coupling interface. A pure implicit coupling without acceleration corresponds to a simple fixed-point iteration, which still has the same stability issues as an explicit coupling. We need acceleration techniques to stabilize and accelerate the fixed-point iteration.

To find out more about the mathematical background, please refer, for example, to this paper.

In preCICE, three different types of `acceleration` can be configured: `constant` (constant under-relaxation), `aitken` (adaptive under-relaxation), and various quasi-Newton variants (`IQN-ILS` aka. Anderson acceleration, `IQN-IMVJ` aka. generalized Broyden). Before looking at the details, we need to understand which data gets modified when.

Coupling data and primary data

All data communicated within a coupling scheme needs to be configured through `exchange` tags. Let's call these data fields **coupling data**.

```
<coupling-scheme:serial-implicit> <participants first="FluidSolver" second="StuctureSolver"/> <exch
ange data="Displacements" mesh="StructureMesh" from="StuctureSolver" to="FluidSolver"/> <exchange d
ata="Forces" mesh="StructureMesh" from="FluidSolver" to="StuctureSolver"/> ... <acceleration:...>
<data name="Displacements" mesh="StructureMesh"/> ... </acceleration:...> </coupling-scheme:serial-
implicit>
```

The acceleration modifies coupling data in `advance()`. This means, what you write to preCICE on the one participant is not the same with what you read on the other participant. The data values are stabilized (or "accelerated") instead. This happens also by using values from previous iterations. Simply think of a linear combination of previous iterations.

- For a **parallel coupling**, all coupling data is post-processed the same way. This means all coupling data use the same coefficients for the linear combination.
- For a **serial coupling** only coupling that is exchanged from the `second` to the `first` participant is post-processed. Coupling data exchanged in the other direction (from `first` to `second`) is not modified.

Let's look at an example: For fluid-structure interaction, if we first execute the fluid solver with given interface displacements followed by the structure solver taking forces from the fluid solver and computing new interface displacements, (only) the displacements are post-processed in case of serial coupling. For parallel coupling, both displacements and forces are post-processed.

Next, we have to configure based on which data the acceleration computes, i.e. how the coefficients in the linear combinations get computed. These data fields are defined within the acceleration as `data` tags (such as `Displacements` in the code example above). Let's call these data fields **primary data**. (Just for completeness: All coupling data that gets post-processed and that is not primary data, is called "secondary data".)

- For **serial coupling**, you can only configure one primary data field, which should correspond to a coupling data field that is exchanged from the `second` to the `first` participant. In the FSI example, the `Displacements` .
- For **parallel coupling**, an arbitrary number of primary data can be configured. For numerical performance reasons, you should define at least one coupling data field of each direction (one from `second` to `first` , one from `first` to `second`). In the FSI example, configure `Displacements` and `Forces` .

Now, we know the difference between coupling data and primary data. Next, we have a look on how we actually configure the type of acceleration.

Constant under-relaxation

```
<acceleration:constant> <relaxation value="0.5"/> </acceleration:constant>
```

The configuration for constant under-relaxation is straight-forward. The only parameter to be configured is the under-relaxation factor `relaxation`. In particular, the configuration of primary data is not necessary as the relaxation parameter stays constant, i.e. the linear combination has fixed coefficients (`relaxation` for the current iteration, `1-relaxation` for the previous iteration).

Constant under-relaxation with a factor of 0.5 can be a good choice for e.g. turbulent FSI at a high Reynolds number.

Dynamic Aitken under-relaxation

```
<acceleration:aitken> <data name="Displacements" mesh="StructureMesh"/> <initial-relaxation valu
e="0.1"/> </acceleration:aitken>
```

Aitken under-relaxation adapts the under-relaxation factor in every iteration based on current residuals of the defined primary data and we only need to define an initial relaxation factor `initial-relaxation`. A value of 0.1 is a robust initial-relaxation factor.

Aitken under-relaxation can be a good choice for strong interaction with a fluid solvers that does not fully converge in every iteration or for compressible fluid solvers. For most cases, however, it is beneficial to change to a quasi-Newton scheme. Using Aitken under-relaxation is generally not recommended, if a parallel coupling scheme is used (refer to table 1 in this paper, where *Vec-Aitken* refers to our implementation of Aitken under-relaxation for parallel coupling schemes).

Quasi-Newton schemes

```
<acceleration:IQN-ILS> <data name="Displacements" mesh="StructureMesh"/> <data name="Forces" mes
h="StructureMesh"/> <preconditioner type="residual-sum"/> <filter type="QR2" limit="1e-3"/> <initia
l-relaxation value="0.1"/> <max-used-iterations value="100"/> <time-windows-reused value="20"/> </a
cceleration:IQN-ILS>
```

For quasi-Newton, the configuration is more involved and requires some attention to achieve good performance. In the following, we list the options and parameters to be chosen and give hints on good combinations of choices:

- Pick a quasi-Newton variant from the following choices: `IQN-ILS` (aka. Anderson acceleration), `IQN-IMVJ` (aka. generalized Broyden). `IQN-ILS` is simpler to start with.
- If parallel coupling is used and, thus, several primary data fields are configured, an equal weighting between them has to be ensured. This is done by defining a `preconditioner`. As `type`, we recommend to use `"residual-sum"` .
- To ensure linear independence of columns in the multi-secant system for Jacobian estimation, a `filter` can be used. The `type` can be chosen as `QR1` or `QR2`. In addition to `type`, a threshold for linear dependency, `limit` has to be defined. In most cases, the filter efficiency is not very sensitive with respect to the `limit`. We recommend to start with a `limit` of 1e-3 or 1e-2 and `QR2` (For `QR1` 1e-6 or 1e-5 is a good choice). A filter should be used with all quasi-Newton variants. If the respective line of the configuration file is omitted, no filter is applied. To find our more, you can have a look at this paper.
- In the first iteration, quasi-Newton methods don't provide an estimate for the Jacobian yet. Thus, the first

iteration is an under-relaxed fixed-point iteration, for which we have to define the parameter `initial-relaxation`. 0.1 is a robust choice. Too small values can render the information from the first iteration too coarse for the calculation of a good Jacobian estimate. Too large values might lead to stability problems.

- The parameter `max-used-iterations` specifies the maximum number of previous iterations used to generate the data basis for Jacobian estimation. In particular for small simulations with only few degrees for freedom, this is an important parameter. It should be chosen to be smaller than half of the total number of degrees of freedom at the interface. For large-scale simulations 100 is a robust choice.
- The parameter `time-windows-reused` also limits the number of previous iterations, but in a per-time-window fashion. Iterations from older time windows than `time-windows-reused` are dropped. Note that, as we don't know the number of iterations per time window a priori, this is not equivalent to setting max-used-iterations. For `IQN-IMVJ`, this parameter can be set to 0 as information from past time windows is implicitly used in the modified Jacobian norm minimization. For `IQN-ILS`, this parameter is an important tuning parameter, in particular if no filtering or filtering with a very low threshold is used. The optimum highly depends on the application, the used solvers and also the grid resolution. We recommend to choose a rather large value (10-30) and combine it with effective filtering (e.g., `QR2` with limit 1e-2) as a starting point for further optimization. With increasing degree of non-linearity of the considered application, the optimal value for `time-windows-reused` is expected to decrease.

Quasi-Newton acceleration is a good choice for strong interactions. Please note that a necessary prerequisite for convergence of the implicit coupling loop is the proper convergence of each participant internally. Inner convergence measure (e.g. of the fluid solver) should be two orders of magnitude stricter than the coupling convergence-measure to achieve good performance with quasi-Newton.

Mesh exchange example

Summary: If you struggle with which mesh you should use where in the configuration and whether a mapping is read or write, you might find this example helpful.

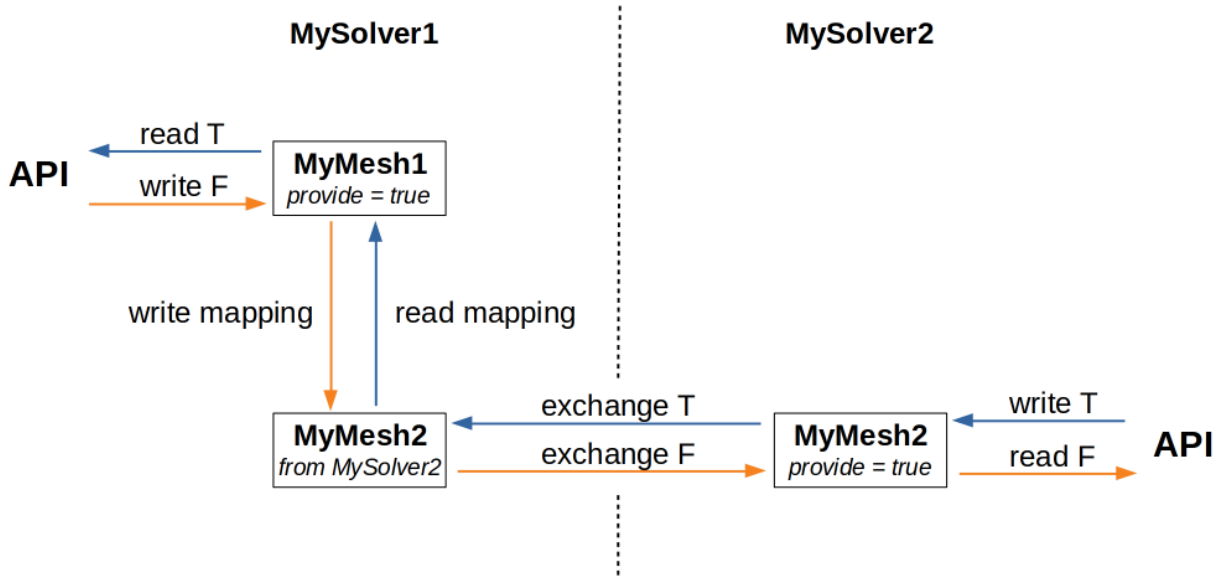
People that are new to preCICE typically struggle with the same things in the configuration:

- What does it mean that a mapping is a “write” mapping?
- Which mesh should be received “from” another participant?
- Which mesh should be mentioned in the `exchange` tag?

Example configuration

All this sounds complicated at first, but is relatively clear once you draw the right picture. Let’s do this here exemplary for the following configuration file:

```
... <participant name="MySolver1"> <use-mesh name="MyMesh1" provide="yes"/> <use-mesh name="MyMesh
2" from="MySolver2"/> <read-data name="Temperature" mesh="MyMesh1"/> <write-data name="Forces" mes
h="MyMesh1"/> <mapping:nearest-neighbor direction="read" from="MyMesh2" to="MyMesh1" constraint="co
nsistent"/> <mapping:nearest-neighbor direction="write" from="MyMesh1" to="MyMesh2" constraint="con
servative"/> ... </participant> <participant name="MySolver2"> <use-mesh name="MyMesh2" provide="ye
s"/> <read-data name="Forces" mesh="MyMesh2"/> <write-data name="Temperature" mesh="MyMesh2"/> ...
</participant> <coupling-scheme:serial-explicit> <participants first="MySolver1" second="MySolver
2"/> <exchange data="Forces" mesh="MyMesh2" from="MySolver1" to="MySolver2"/> <exchange data="Tempe
rature" mesh="MyMesh2" from="MySolver2" to="MySolver1"/> ... </coupling-scheme:serial-explicit>
```



Why do we make all this so complicated?

We want to give the user as much freedom as possible to adjust the setup to her specific needs. Typical constraints / wishes are:

- Communication of coupling data on the coarser mesh
- Computation of the quasi-Newton acceleration on the coarser mesh (typically more robust)
- Restriction of the mapping in parallel to “read-consistent” and “write-conservative” (more information on the mapping configuration page (page 69)

Multi coupling configuration

Summary: If you want to couple more than two participants, there are two options: You can combine multiple normal coupling schemes (composition) or you can use a fully-implicit multi-coupling scheme. On this page, we explain both options.

Composition of bi-coupling schemes

To combine multiple coupling schemes, simply add them one after the other in the configuration:

```
<coupling-scheme:parallel-explicit> <participants first="MySolver1" second="MySolver2"/> ... </coup
ling-scheme:parallel-explicit> <coupling-scheme:parallel-explicit> <participants first="MySolver1"
second="MySolver3"/> ... </coupling-scheme:parallel-explicit>
```

For this example, all three participants are executed in parallel to one another, whereas `MySolver1` exchanges data with `MySolver2` and `MySolver3`, but not the latter two with each other. To also get an interaction between `MySolver2` and `MySolver3`, simply add a third coupling scheme.

You can probably imagine that you can do very strange combinations, where most of them have only limited practical relevance. To find out more, you can read Section 4.1.5 in Bernhard's thesis. Numerically, it only makes sense to either only have explicit schemes or to combine one implicit scheme with several explicit ones. To find out more, you can read this paper. If you want to resolve more than one strong interaction, you need a fully-implicit multi-coupling.

Fully-implicit multi-coupling

In a fully-implicit multi-coupling, an arbitrary number of solvers are executed in parallel to each other in an implicit fashion.

```
<coupling-scheme:multi> <participant name="MySolver1" control="yes"/> <participant name="MySolver2"
/> <participant name="MySolver3" /> ... </coupling-scheme:multi>
```

Exactly one participants needs to have `control`. preCICE computes the convergence measures and the acceleration on this participant. Be careful: this participant needs to have `m2n` connections to all other participants and the `exchange` tags needs to be properly configured.

All other tags are similar to a normal implicit coupling (page 72).

To find out more about multi coupling, you can also read Section 3.8 in Benjamin's thesis.

Logging configuration

Summary: By default, preCICE provides a meaningful logging output to stdout. In case you want to modify the default logging, this page describes how to do this.

Introduction

Logging in preCICE is based on boost.log.

For debug logging, you need to build preCICE in debug mode (page 30). Please note that the Debian packages are not built in debug mode.

In principle, to modify the logging, you simply define your own logging. This is done in the preCICE configuration file. We start here with a dummy example. Further below, you can find useful examples for certain use cases:

```
<precice-configuration> <log enabled="true"> <sink type="stream" output="stdout" format="Hello %Mes
sage%" filter="%Severity% > debug" enabled="true" /> <sink type="file" output="debug.log" filter=""
enabled="false" /> </log> <solver-interface dimensions="3"> ...
```

This configures two sinks: the first one logs to stdout, uses a somehow absurd logging format and filters, so that the messages with a severity higher than debug are printed. The other, disabled one, uses an empty filter, thus printing all messages and writes them to a file. Beware: because of trace messages this file might become huge.

`<log>` has one attribute:

- `enabled` can be used to completely disable logging. It defaults to `true`.

Each sink has these attributes:

- `type` can be `stream` or `file`
- `output` can be `stdout` or `stdin` if `type=stream` or a filename if `type=file`
- `format` is some boost.log format string.
- `filter` is a boost.log filter string. The default filter string is `%Severity% > debug`
- `enabled` is a boolean value. It can be one of `0, 1, yes, no, true, false` Note that if all sinks are disabled, the default sink is used. Use `<log enabled="false">` to completely disable logging.

The `<log>` tag is optional. If it is ommitted, default values are used. `type` and `output` are mandatory, all others attributes are optional.

log.conf

Logging can also be configured using a file `log.conf` in the current working directory. This is the way to configure logging when you run tests via `testprecice`.

```
[Debug] Filter = (%Severity% > debug) or (%Module% contains "PetRadialBasisFctMapping") Format = %M
essage% [EverythingToFile] Filter = Type = file output = precice.log
```

The `[SectionHeaders]` are just for distingushing the sections, the names are not used.

Attributes

Attributes available to the filter and the formatter are:

Attribute	Description
`Severity`	Severity, can be `trace`, `debug`, `info`, `warn`, `error`
`File`	The absolute path to the file at the log location.
`Line`	The line number of the log location.
`Function`	The function at the log location.
`Module`	The module at the log location. This is mostly the class holding the logger.
`Rank`	The MPI rank producing the log
`Participant`	The name of the current participant, e.g., `Fluid`

Examples

- All info messages, but also trace and debug logging for the interface of preCICE. This logging is very useful if you want to find out if the coupled simulation crashes in preCICE or in your solver.

```
<log> <sink type="stream" output="stdout" filter= "(%Severity% > debug) or (%Severity% >= trace and %Module% contains SolverInterfaceImpl)" enabled="true" /> </log>
```

- The standard preCICE info output, but in a more compact format. This can be useful if preCICE works fine and you simply want to focus on your solver's output.

```
<log> <sink type="stream" output="stdout" filter= "%Severity% > debug and %Rank% = 0" format="preCICE: %ColorizedSeverity% %Message%" enabled="true" /> </log>
```

- To debug where initialization hangs:

```
<log> <sink type="stream" output="stdout" filter= "(%Severity% > debug) or (%Severity% >= debug and %Module% contains SolverInterfaceImpl) or (%Severity% >= debug and %Module% contains partition) or (%Severity% >= debug and %Module% contains PointToPointCommunication)" enabled="true" /> </log>
```

- You want to look at your output in an editor and the colors (ANSI escape codes) destroy the formatting.

```
<log> <sink type="file" output="precice.log" filter= "%Severity% > debug and %Rank% = 0" format="(%Rank%) [%Module%]:%Line% in %Function%: %Severity% %Message%" enabled="true" /> </log>
```

- You develop in preCICE and want also trace output for `PetRadialBasisFctMapping`.

```
<log> <sink type="stream" output="stdout" filter= "(%Severity% > debug and %Rank% = 0) or (%Severity% >= trace and %Module% contains PetRadialBasisFctMapping)" enabled="true" /> </log>
```

- Filter according to participant and put the messages into different files.

```
<log> <sink type="file" output="debug_Fluid.log" filter="%Participant% = Fluid" /> <sink type="fil
e" output="debug_Solid.log" filter="%Participant% = Solid" /> </log>
```

Note, that the files `debug_Fluid.log` and `debug_Solid.log` are created in any case, so you may end up with empty files if everything has been filtered out.

Exports configuration

Summary: You can export your coupling meshes to vtk. This is a great feature for debugging. On this page, we explain how.

Well, it's easy. Just write ...

```
<participant name="MySolver1"> <use-mesh name="MyMesh1" provide="yes"/> ... <export:vtk director
y="preCICE-output" /> ... </participant>
```

With `directory`, you can give an extra folder to write the VTK files to, relative to the location from where you start the participant. preCICE writes output for every `use-mesh` separately.

Of course, this is only the data at the coupling surface. So the main purpose of this feature is to debug, not to analyze physical results.

Optional parameters:

- `every-n-time-windows="{integer}"`: Use this if you want to output only every x timesteps
- `every-iteration="true"`: Use this if you want output for every coupling iteration (for an implicit coupling scheme)

Visualization with ParaView

If you have not defined edges or triangles, the VTK files will only contain point data. You can visualize them in ParaView using either of:

- Glyphs - Note that more recent paraView version use as default representation 'arrows', which might be perpendicular in the 2D plane and therefore not visible by default. You might need to switch the representation.
- A `Delaunay 2D` filter to get a surface from the points. If your coupling surface is not XY-aligned, use the `best fitting plane` setting of the filter. If `Delaunay 2D` is not able to reconstruct a meaningful surface (i.e. in the case of a thin flap), `Delaunay 3D` may give a meaningful volume.

Action configurations

Summary: Sometimes, coupled solvers provide just not quite the data that you need to couple. For instance, a fluid solver provides stresses at the coupling boundary, whereas a solid solver requires forces. In this case, you can use so-called coupling actions to modify coupling data at runtime. These coupling actions are essentially a set of functionalities that have access to coupling meshes and the corresponding data values. On this page, we explain how you can use them.

There are two types of coupling actions: pre-implemented ones and user-defined ones. For the latter, you can access coupling meshes through a Python callback interface.

Basics and pre-implemented actions

```
<participant name="MySolver1"> <use-mesh name="MyMesh1" provide="yes"/> <write-data name="Stresses"
mesh="MyMesh1"/> ... <action:multiply-by-area mesh="MyMesh1" timing="write-mapping-post"> <target-d
ata name="Stresses"/> </action:multiply-by-area> ... </participant>
```

This example multiplies the stresses values by the respective element area, transforming stresses into forces. Please note that for this specific action, mesh connectivity information needs to be provided. (edges, triangles, etc. through `setMeshEdge` or similar API functions (page 239).

`timing` defines *when* the action is executed. Options are:

- `write-mapping-prior` and `write-mapping-post`: directly before or after each time the write mappings are applied.
- `read-mapping-prior` and `read-mapping-post`: directly before or after each time the read mappings are applied.
- `on-time-window-complete-post`: after the coupling in a complete time window has converged, after `read` data is mapped.

•Older (preCICE version < 2.1.0) timings that are deprecated and revert to one of the above options: (click for details)

- `regular-prior`: In every `advance` call (also for subcycling) and in `initializeData`, after `write` data is mapped, but *before* data might be sent. (*v2.1 or later: reverts to* `write-mapping-prior`)
- `regular-post`: In every `advance` call (also for subcycling), in `initializeData` and in `initialize`, before `read` data is mapped, but *after* data might be received and after acceleration. (*v2.1 or later: reverts to* `read-mapping-prior`)
- `on-exchange-prior`: Only in those `advance` calls which lead to data exchange (and in `initializeData`), after `write` data is mapped, but *before* data might be sent. (*v2.1 or later: reverts to* `write-mapping-post`)
- `on-exchange-post`: Only in those `advance` calls which lead to data exchange (and in `initializeData` and `initialize`), before `read` data is mapped, but *after* data might be received. (*v2.1 or later: reverts to* `read-mapping-prior`)

Pre-implemented actions are:

- `multiply-by-area` / `divide-by-area`: Modify coupling data by mesh area

- `scale-by-computed-dt-ratio` / `scale-by-computed-dt-part-ratio` / `scale-by-dt`: Modify coupling data by timestep size
- `compute-curvature`: Compute curvature values at vertices
- `summation`: Sum up the data from source participants and write to target participant

Note: All target and source data used in actions require `<read-data ... />` or `<write-data ... />` tags.

For more details, please refer to the XML reference (page 89).

Python callback interface

Other than the pre-implemented coupling actions, preCICE also provides a callback interface for Python scripts to execute coupling actions. To use this feature, you need to build preCICE with python support (page 30).

Note: The primary purpose of the python interface is prototyping. If you need a native version of the action, please contact us on GitHub to develop and possibly integrate it into the project.

We show an example for the 1D elastic tube (page 0):

```
<participant name="Solid"> <use-mesh name="Solid-Nodes-Mesh" provide="yes"/> <use-mesh name="Fluid-Nodes-Mesh" from "Fluid" /> <write-data name="CrossSectionLength" mesh="Solid-Nodes-Mesh" /> <read-data name="Pressure" mesh="Solid-Nodes-Mesh" /> <action:python mesh="Solid-Nodes-Mesh" timing="read-mapping-prior"> <path name="<PATH_TO_PYTHON_ACTION_SCRIPT>"/> <module name="<PYTHON_SCRIPT_NAME.PY>"/> <source-data name="Pressure"/> <target-data name="Pressure"/> </action:python> </participant>
```

The callback interface consists of the following three (optional) functions:

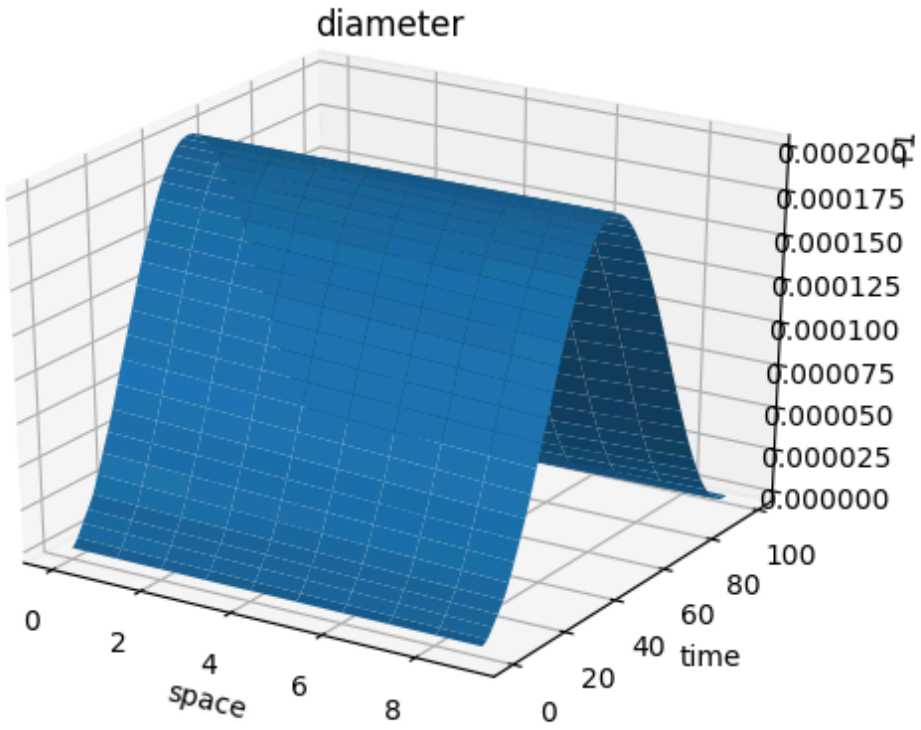
```
performAction(time, sourceData, targetData) vertexCallback(id, coords, normal) postAction()
```

`performAction` gives access to the coupling value arrays. You can store these values in global variables to grant access to the other two functions.

`vertexCallback` gives access to the geometric data of each vertex. This function is called successively for every vertex of the specified coupling mesh and you can use the corresponding geometric data.

`postAction` is called at the final step. You can perform any finalizing code after deriving information from the vertices, if wished.

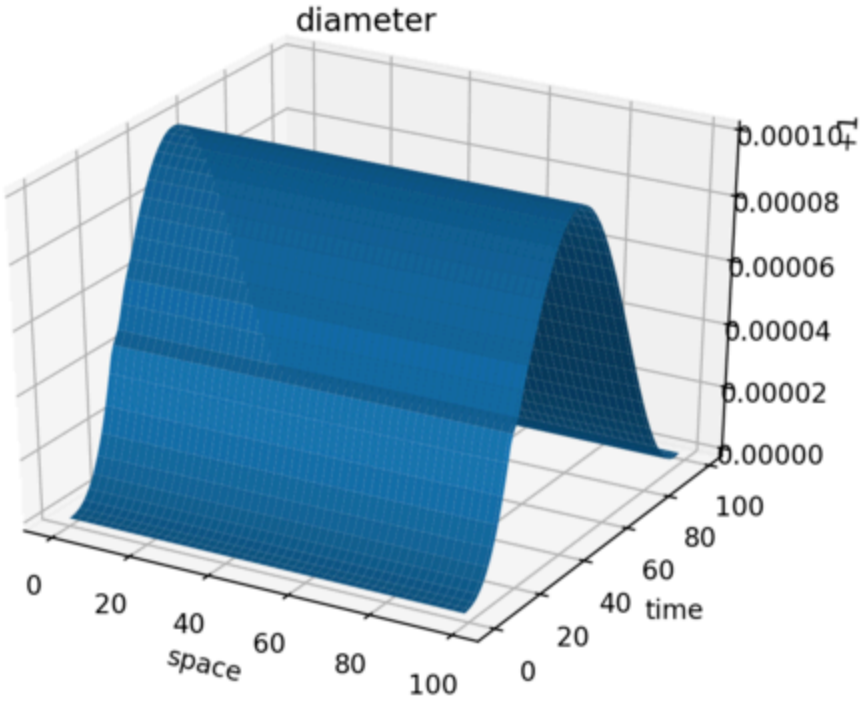
Without the Python action, the 1D elastic tube gives the following results:



Now, we want to ramp up the pressure values written by the fluid solver over time. A feature often needed to get a stable coupled simulation.

```
mySourceData = 0 myTargetData = 0 def performAction(time, dt, sourceData, targetData): # This funct
ion is called first at configured timing. It can be omitted, if not # needed. Its parameters are ti
me, timestep size, the source data, followed by the target data. # Source and target data can be om
itted (selectively or both) by not mentioning # them in the preCICE XML configuration. global mySou
rceData global myTargetData mySourceData = sourceData # store (reference to) sourceData for later u
se myTargetData = targetData # store (reference to) targetData for later use timeThreshold = 0.2 #
Ramp up the pressure values until this point in time if time < timeThreshold: for i in range(myTarg
etData.size): # Ramp up pressure value myTargetData[i] = (time / timeThreshold) * mySourceData[i] e
lse: for i in range(myTargetData.size): # Assign the computed physical pressure values myTargetDat
a[i] = mySourceData[i] def vertexCallback(id, coords, normal): # This function is called for every
vertex in the configured mesh. It is called # after performAction, and can also be omitted. # Usage
example: global mySourceData # Make global data set in performAction visible global myTargetData #
Example usage, add data to vertex coords: # myTargetData[id] += coords[0] + mySourceData[id] def po
stAction(): # This function is called at last, if not omitted. global mySourceData # Make global da
ta set in performAction visible global myTargetData # Do something ...
```

With the Python action, you should now get the following results. Note the lower maximum diameter and the change at `t=0.2` (`t=20` in the graph).


diameter
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
0
20
40
60
80
100
space
0
20
40
60
80
100
time

Watchpoint configuration

Summary: With a watch point, you can track the coupling data values at a certain position over time. This is very handy for applications such as the Turek and Hron FSI3 benchmark where you want to analyze the movement of the tip of a flexible plate.

```
<participant name="MySolver1"> <use-mesh name="MyMesh1" provide="yes"/> ... <watch-point mesh="MyMesh1" name="MyWatchPoint" coordinate="0.6; 0.2"/> ... </participant>
```

This will create a logging file `precice-MySolver1-watchpoint-MyWatchPoint.log` with one row per timestep.

- Only a participant that provides the respective mesh can set a watchpoint on that mesh.
- You can freely choose the name `MyWatchPoint`.
- Please note the format of `coordinate`. Here, values at (x,y)=(0.6,0.2) are tracked. The dimensions need to match the overall preCICE `dimensions` in the `solver-interface` tag, cf. the configuration overview (page 0).
- If (0.6, 0.2) is not explicitly a vertex of `MyMesh1`, the nearest neighbor is chosen (resp. nearest projection if mesh connectivity is defined, cf. the mapping configuration (page 0)).
- The dimensions of the watchpoint need to match the dimensions of the interface (2D vs. 3D).

Watch integral configuration

Summary: With a watch integral, you can track the transient change of integral values of coupling data over complete coupling meshes. This is especially useful when you want to track conserved quantities of your simulation, such as flow rate or the overall force acting on a geometry.

⚠ **Important:** This feature is only available for preCICE versions >= v2.2

```
<participant name="MySolver1"> <use-mesh name="MyMesh1" provide="yes"/> ... <watch-integral mesh="MyMesh1" name="MyWatchIntegral" scale-with-connectivity="yes"/> ... </participant>
```

This creates a log file `precice-MySolver1-watchintegral-MyWatchIntegral.log` with one row per time window and integral values of coupling data per column. If `scale-with-connectivity` is set to `yes` and there is connectivity information defined (find out more about mesh connectivity in step 8 of the couple-your-code section (page 239)), surface area is included as an additional column.

- Only a participant that provides a mesh can set a watch integral on this mesh.
- You can freely choose the name `MyWatchIntegral`.
- There are two ways to calculate integral data:
    - **Calculate with scaling:** While calculating the integral values, area weighted sum of vertex data is calculated. This approach is useful when your data is not yet associated to any cell size such as a flow rate or a displacement field. In case your data is already associated with a cell size (e.g. a force acting on a cell face), this option is usually not required. Data, where a scaling is reasonable, is usually mapped `consistent` between participants. This option requires mesh connectivity information (edges for 2D, faces for 3D) and `scale-with-connectivity` option is set to `yes`. For 2D, edge lengths are used while for 3D face areas are used for scaling.
    - **Calculate without scaling:** The coupling data is summed up over all vertices on the coupling mesh. This is useful when your coupling data is already associated to a certain cell size (e.g. a force acting on a cell face) of your coupling mesh since no additional weighting of the coupling data is applied. Data, where a scaling is not reasonable, is usually mapped `conservative` between participants. If there is no mesh connectivity information provided, no scaling is performed to calculate the integral. If there is mesh connectivity information, you can switch of the scaling by setting `scale-with-connectivity` option to `no`.
- Some important points for the interpretation of the integral data:
    - Integral calculation is based on weighted sum of vertex data and does not distinguish between conservative and consistent data. For example, watch integral is useful for calculating the total flow rate over the interface, or total force which are both conserved variables. However, using watch integral for stress data would not be useful since summing up total stress has no physical interpretation.
    - Scaling of the data is going to be always based on the connectivity information of the mesh given in configuration. For example, in the given configuration file, `MyWatchIntegral` is defined on mesh `MyMesh1`. Each of the coupling data used by `Mesh1` is scaled with the connectivity information of `Mesh1`
    - If your solver uses a Lagrangian or ALE description please note that the scaling is based on the initial reference vertex coordinates of your geometry.

XML reference

> **Summary:** On this page you find the complete configuration references of preCICE API.

Warning: This page is not for learning how the preCICE configuration works (better start here (page 0)). It's the right place if you already know what you are looking for.

Tip: Also try the search bar at the top of the website.

Note: You can generate the reference yourself:

```
./binprecice md > reference.md
```

precice-configuration

Main tag containing preCICE configuration.

Example:

```
<precice-configuration sync-mode="0"> <log enabled="1"> ... </log> <solver-interface dimension
s="2"> ... </solver-interface> </precice-configuration>
```

Attribute	Type	Description	Default	Options
sync-mode	boolean	sync-mode enabled additional inter- and intra-participant synchro-nizations	`0`	none

Valid Subtags:

- log (page 89) `0..1`
- solver-interface (page 90) `1`

log

Configures logging sinks based on Boost log.

Example:

```
<log enabled="1"> <sink filter="(%Severity% > debug) and not ((%Severity% = info) and (%Rank% !=
0))" format="(%Rank%) %TimeStamp(format="%H:%M:%S")% [%Module%]:%Line% in %Function%: %ColorizedSev
erity%%Message%" output="stdout" type="stream" enabled="1"/> </log>
```

Attribute	Type	Description	Default	Options
enabled	boolean	Enables logging	`1`	none

Valid Subtags:

- sink (page 90) `0..*`

sink

Contains the configuration of a single log sink, which allows fine grained control of what to log where. Available attributes in filter and format strings are `%Severity%`, `%ColorizedSeverity%`, `%File%`, `%Line%`, `%Function%`, `%Module%`, `%Rank%`, and `%Participant%`

Example:

```
<sink filter="(%Severity% > debug) and not ((%Severity% = info) and (%Rank% != 0))" format="(%Rank%) %TimeStamp(format="%H:%M:%S")% [%Module%]:%Line% in %Function%: %ColorizedSeverity%%Message%" output="stdout" type="stream" enabled="1"/>
```

Attribute	Type	Description	Default	Options
filter	string	Boost Log Filter String	`(%Severity% > debug) and not ((%Severity% = info) and (%Rank% != 0))`	none
format	string	Boost Log Format String	`(%Rank%) %TimeStamp(format="%H:%M:%S")% [%Module%]:%Line% in %Function%: %ColorizedSeverity%%Message%`	none
output	string	Depends on the type of the sink. For streams, this can be stdout or stderr. For files, this is the filename.	`stdout`	none
type	string	The type of sink.	`stream`	`stream`, `file`
enabled	boolean	Enables the sink	`1`	none

solver-interface

Configuration of simulation relevant features.

Example:

```
<solver-interface dimensions="2"> <data:scalar name="{string}"/> <mesh name="{string}" flip-normals="0"> ... </mesh> <m2n:sockets port="0" exchange-directory="" from="{string}" network="lo" to="{string}" enforce-gather-scatter="0" use-two-level-initialization="0"/> <participant name="{string}"> ... </participant> <coupling-scheme:serial-explicit> ... </coupling-scheme:serial-explicit> </solver-interface>
```

Attribute	Type	Description	Default	Options
dimensions	integer	Determines the spatial dimensionality of the configuration	`2`	`2`, `3`

Valid Subtags:

- mesh (page 91) `1..*`
- participant (page 94) `1..*`
- coupling-scheme
  - serial-explicit (page 116) `0..*`
  - parallel-explicit (page 118) `0..*`
  - serial-implicit (page 120) `0..*`
  - parallel-implicit (page 133) `0..*`
  - multi (page 147) `0..*`
- data
  - scalar (page 91) `0..*`
  - vector (page 91) `0..*`
- m2n
  - sockets (page 92) `0..*`
  - mpi (page 93) `0..*`
  - mpi-singleports (page 93) `0..*`

data:scalar

Defines a scalar data set to be assigned to meshes.

Example:

```
<data:scalar name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Unique name for the data set.	*none*	none

data:vector

Defines a vector data set to be assigned to meshes. The number of components of each data entry depends on the spatial dimensions set in tag .

Example:

```
<data:vector name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Unique name for the data set.	*none*	none

mesh

Surface mesh consisting of vertices and (optional) of edges and triangles (only in 3D). The vertices of a mesh can carry data, configured by tag . The mesh coordinates have to be defined by a participant (see tag).

Example:

```
<mesh name="{string}" flip-normals="0"> <use-data name="{string}"/> </mesh>
```

Attribute	Type	Description	Default	Options
name	string	Unique name for the mesh.	*none*	none
flip-normals	boolean	Flips mesh normal vector directions.	`0`	none

Valid Subtags:

- use-data (page 92) `0..*`

use-data

Assigns a before defined data set (see tag) to the mesh.

Example:

```
<use-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data set.	*none*	none

m2n:sockets

Communication via Sockets.

Example:

```
<m2n:sockets port="0" exchange-directory="" from="{string}" network="lo" to="{string}" enforce-gath
er-scatter="0" use-two-level-initialization="0"/>
```

Attribute	Type	Description	Default	Options
port	integer	Port number (16-bit unsigned integer) to be used for socket communication. The default is “0”, what means that the OS will dynamically search for a free port (if at least one exists) and bind it automatically.	`0`	none
exchange-directory	string	Directory where connection information is exchanged. By default, the directory of startup is chosen, and both solvers have to be started in the same directory.	``	none
from	string	First participant name involved in communication. For performance reasons, we recommend to use the participant with less ranks at the coupling interface as “from” in the m2n communication.	*none*	none

Attribute	Type	Description	Default	Options
network	string	Interface name to be used for socket communiation. Default is the cannonical name of the loopback interface of your platform. Might be different on supercomputing systems, e.g. “ib0” for the InfiniBand on SuperMUC.	`lo`	none
to	string	Second participant name involved in communication.	*none*	none
enforce-gather-scatter	boolean	Enforce the distributed communication to a gather-scatter scheme. Only recommended for trouble shooting.	`0`	none
use-two-level-initialization	boolean	Use a two-level initialization scheme. Recommended for large parallel runs (>5000 MPI ranks).	`0`	none

m2n:mpi

Communication via MPI with startup in separated communication spaces, using multiple communicators.

Example:

```
<m2n:mpi exchange-directory="" from="{string}" to="{string}" enforce-gather-scatter="0" use-two-lev
el-initialization="0"/>
```

Attribute	Type	Description	Default	Options
exchange-directory	string	Directory where connection information is exchanged. By default, the directory of startup is chosen, and both solvers have to be started in the same directory.	``	none
from	string	First participant name involved in communication. For performance reasons, we recommend to use the participant with less ranks at the coupling interface as “from” in the m2n communication.	*none*	none
to	string	Second participant name involved in communication.	*none*	none
enforce-gather-scatter	boolean	Enforce the distributed communication to a gather-scatter scheme. Only recommended for trouble shooting.	`0`	none
use-two-level-initialization	boolean	Use a two-level initialization scheme. Recommended for large parallel runs (>5000 MPI ranks).	`0`	none

m2n:mpi-singleports

Communication via MPI with startup in separated communication spaces, using a single communicator

Example:

```
<m2n:mpi-singleports exchange-directory="" from="{string}" to="{string}" enforce-gather-scatter="0"
use-two-level-initialization="0"/>
```

Attribute	Type	Description	Default	Options
exchange-directory	string	Directory where connection information is exchanged. By default, the directory of startup is chosen, and both solvers have to be started in the same directory.	``	none
from	string	First participant name involved in communication. For performance reasons, we recommend to use the participant with less ranks at the coupling interface as “from” in the m2n communication.	*none*	none
to	string	Second participant name involved in communication.	*none*	none
enforce-gather-scatter	boolean	Enforce the distributed communication to a gather-scatter scheme. Only recommended for trouble shooting.	`0`	none
use-two-level-initialization	boolean	Use a two-level initialization scheme. Recommended for large parallel runs (>5000 MPI ranks).	`0`	none

participant

Represents one solver using preCICE. At least two participants have to be defined.

Example:

```
<participant name="{string}"> <write-data mesh="{string}" name="{string}"/> <read-data mesh="{strin
g}" name="{string}"/> <mapping:rbf-thin-plate-splines solver-rtol="1e-09" constraint="{string}" dir
ection="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" t
o="{string}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/> <action:multiply-by-area m
esh="{string}" timing="{string}"> ... </action:multiply-by-area> <export:vtk every-n-time-window
s="1" directory="" every-iteration="0" normals="1"/> <watch-point mesh="{string}" name="{string}" c
oordinate="{vector}"/> <watch-integral mesh="{string}" name="{string}" scale-with-connectivity="{bo
olean}"/> <use-mesh safety-factor="0.5" from="" geometric-filter="on-slaves" name="{string}" provid
e="0"/> <master:sockets port="0" exchange-directory="" network="lo"/> </participant>
```

Attribute	Type	Description	Default	Options
name	string	Name of the participant. Has to match the name given on construction of the precice::SolverInterface object used by the participant.	*none*	none

Valid Subtags:

- write-data (page 95) `0..*`
- read-data (page 96) `0..*`
- watch-point (page 114) `0..*`
- watch-integral (page 114) `0..*`

- use-mesh (page 115) `0..*`
- action
  - multiply-by-area (page 106) `0..*`
  - divide-by-area (page 106) `0..*`
  - scale-by-computed-dt-ratio (page 107) `0..*`
  - scale-by-computed-dt-part-ratio (page 108) `0..*`
  - scale-by-dt (page 109) `0..*`
  - summation (page 110) `0..*`
  - compute-curvature (page 111) `0..*`
  - recorder (page 112) `0..*`
  - python (page 112) `0..*`
- export
  - vtk (page 113) `0..*`
- mapping
  - rbf-thin-plate-splines (page 96) `0..*`
  - rbf-multiquadrics (page 97) `0..*`
  - rbf-inverse-multiquadrics (page 98) `0..*`
  - rbf-volume-splines (page 99) `0..*`
  - rbf-gaussian (page 100) `0..*`
  - rbf-compact-tps-c2 (page 101) `0..*`
  - rbf-compact-polynomial-c0 (page 102) `0..*`
  - rbf-compact-polynomial-c6 (page 103) `0..*`
  - nearest-neighbor (page 104) `0..*`
  - nearest-projection (page 105) `0..*`
- master
  - sockets (page 115) `0..1`
  - mpi (page 116) `0..1`
  - mpi-single (page 116) `0..1`

write-data

Sets data to be written by the participant to preCICE. Data is defined by using the tag.

Example:

```
<write-data mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	Mesh the data belongs to. If data should be read/written to several meshes, this has to be specified separately for each mesh.	*none*	none
name	string	Name of the data.	*none*	none

read-data

Sets data to be read by the participant from preCICE. Data is defined by using the tag.

Example:

```
<read-data mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	Mesh the data belongs to. If data should be read/written to several meshes, this has to be specified separately for each mesh.	*none*	none
name	string	Name of the data.	*none*	none

mapping:rbf-thin-plate-splines

Global radial-basis-function mapping based on the thin plate splines.

Example:

```
<mapping:rbf-thin-plate-splines solver-rtol="1e-09" constraint="{string}" direction="{string}" fro
m="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{string}" use-qr-decom
position="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`

Attribute	Type	Description	Default	Options
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `onadvance` , `ondemand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:rbf-multiquadrics

Global radial-basis-function mapping based on the multiquadrics RBF.

Example:

```
<mapping:rbf-multiquadrics shape-parameter="{float}" solver-rtol="1e-09" constraint="{string}" direction="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{string}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
shape-parameter	float	Specific shape parameter for RBF basis function.	*none*	none
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`

Attribute	Type	Description	Default	Options
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `onadvance` , `ondemand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:rbf-inverse-multiquadrics

Global radial-basis-function mapping based on the inverse multiquadrics RBF.

Example:

```
<mapping:rbf-inverse-multiquadrics shape-parameter="{float}" solver-rtol="1e-09" constraint="{string}" direction="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{string}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
shape-parameter	float	Specific shape parameter for RBF basis function.	*none*	none
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`

Attribute	Type	Description	Default	Options
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `on-advance` , `on-demand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:rbf-volume-splines

Global radial-basis-function mapping based on the volume-splines RBF.

Example:

```
<mapping:rbf-volume-splines solver-rtol="1e-09" constraint="{string}" direction="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{string}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none

Attribute	Type	Description	Default	Options
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `on-advance` , `on-demand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:rbf-gaussian

Local radial-basis-function mapping based on the Gaussian RBF with a cut-off threshold.

Example:

```
<mapping:rbf-gaussian shape-parameter="{float}" solver-rtol="1e-09" constraint="{string}" directio
n="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{strin
g}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
shape-parameter	float	Specific shape parameter for RBF basis function.	*none*	none
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `on-advance` , `on-demand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:rbf-compact-tps-c2

Local radial-basis-function mapping based on the C2-polynomial RBF.

Example:

```
<mapping:rbf-compact-tps-c2 solver-rtol="1e-09" support-radius="{float}" constraint="{string}" direction="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{string}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none
support-radius	float	Support radius of each RBF basis function (global choice).	*none*	none
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `onadvance` , `ondemand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:rbf-compact-polynomial-c0

Local radial-basis-function mapping based on the C0-polynomial RBF.

Example:

```
<mapping:rbf-compact-polynomial-c0 solver-rtol="1e-09" support-radius="{float}" constraint="{string}" direction="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{string}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none
support-radius	float	Support radius of each RBF basis function (global choice).	*none*	none
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `onadvance` , `ondemand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:rbf-compact-polynomial-c6

Local radial-basis-function mapping based on the C6-polynomial RBF.

Example:

```
<mapping:rbf-compact-polynomial-c6 solver-rtol="1e-09" support-radius="{float}" constraint="{string}" direction="{string}" from="{string}" polynomial="separate" preallocation="tree" timing="initial" to="{string}" use-qr-decomposition="0" x-dead="0" y-dead="0" z-dead="0"/>
```

Attribute	Type	Description	Default	Options
solver-rtol	float	Solver relative tolerance for convergence	`1e-09`	none
support-radius	float	Support radius of each RBF basis function (global choice).	*none*	none
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
polynomial	string	Toggles use of the global polynomial	`separate`	`on` , `off` , `separate`
preallocation	string	Sets kind of preallocation for PETSc RBF implementation	`tree`	`estimate` , `compute` , `off` , `save` , `tree`
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `onadvance` , `ondemand`
to	string	The mesh to map the data to.	*none*	none
use-qr-decomposition	boolean	If set to true, QR decomposition is used to solve the RBF system	`0`	none
x-dead	boolean	If set to true, the x axis will be ignored for the mapping	`0`	none
y-dead	boolean	If set to true, the y axis will be ignored for the mapping	`0`	none
z-dead	boolean	If set to true, the z axis will be ignored for the mapping	`0`	none

mapping:nearest-neighbor

Nearest-neighbour mapping which uses a rstar-spacial index tree to index meshes and run nearest-neighbour queries.

Example:

```
<mapping:nearest-neighbor constraint="{string}" direction="{string}" from="{string}" timing="initial" to="{string}"/>
```

Attribute	Type	Description	Default	Options
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `on-advance` , `on-demand`
to	string	The mesh to map the data to.	*none*	none

mapping:nearest-projection

Nearest-projection mapping which uses a rstar-spacial index tree to index meshes and locate the nearest projections.

Example:

```
<mapping:nearest-projection constraint="{string}" direction="{string}" from="{string}" timing="initial" to="{string}"/>
```

Attribute	Type	Description	Default	Options
constraint	string	Use conservative to conserve the quantity of the data over the interface such as force or mass. Use consistent for normalized quantities such as temperature or pressure.	*none*	`conservative` , `consistent`
direction	string	Write mappings map written data prior to communication, thus in the same participant who writes the data. Read mappings map received data after communication, thus in the same participant who reads the data.	*none*	`write` , `read`
from	string	The mesh to map the data from.	*none*	none
timing	string	This allows to defer the mapping of the data to advance or to a manual call to mapReadDataTo and mapWriteDataFrom.	`initial`	`initial` , `on-advance` , `on-demand`
to	string	The mesh to map the data to.	*none*	none

action:multiply-by-area

Multiplies data values with mesh area associated to vertex holding the value.

Example:

```
<action:multiply-by-area mesh="{string}" timing="{string}"> <target-data name="{string}"/> </actio
n:multiply-by-area>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior` , `regular-post` , `on-exchange-prior` , `on-exchange-post` , `on-time-window-complete-post` , `write-mapping-prior` , `write-mapping-post` , `read-mapping-prior` , `read-mapping-post`

Valid Subtags:

- target-data (page 106) `1`

target-data

Data to read from and write to.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

action:divide-by-area

Divides data values by mesh area associated to vertex holding the value.

Example:

```
<action:divide-by-area mesh="{string}" timing="{string}"> <target-data name="{string}"/> </action:d
ivide-by-area>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none

Attribute	Type	Description	Default	Options
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior`, `regular-post`, `on-exchange-prior`, `on-exchange-post`, `on-time-window-complete-post`, `write-mapping-prior`, `write-mapping-post`, `read-mapping-prior`, `read-mapping-post`

Valid Subtags:

- target-data (page 107) `1`

target-data

Data to read from and write to.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

action:scale-by-computed-dt-ratio

Multiplies source data values by ratio of full dt / last computed dt, and writes the result into target data.

Example:

```
<action:scale-by-computed-dt-ratio mesh="{string}" timing="{string}"> <source-data name="{strin
g}"/> <target-data name="{string}"/> </action:scale-by-computed-dt-ratio>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior`, `regular-post`, `on-exchange-prior`, `on-exchange-post`, `on-time-window-complete-post`, `write-mapping-prior`, `write-mapping-post`, `read-mapping-prior`, `read-mapping-post`

Valid Subtags:

- source-data (page 107) `1`
- target-data (page 107) `1`

source-data

Single data to read from.

Example:

```
<source-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

target-data

Data to read from and write to.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

action:scale-by-computed-dt-part-ratio

Multiplies source data values by ratio of full dt / computed dt part, and writes the result into target data.

Example:

```
<action:scale-by-computed-dt-part-ratio mesh="{string}" timing="{string}"> <source-data name="{string}"/> <target-data name="{string}"/> </action:scale-by-computed-dt-part-ratio>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior`, `regular-post`, `on-exchange-prior`, `on-exchange-post`, `on-time-window-complete-post`, `write-mapping-prior`, `write-mapping-post`, `read-mapping-prior`, `read-mapping-post`

Valid Subtags:

- source-data (page 108) `1`
- target-data (page 107) `1`

source-data

Single data to read from.

Example:

```
<source-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

target-data

Data to read from and write to.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

action:scale-by-dt

Multiplies source data values by last computed dt, and writes the result into target data.

Example:

```
<action:scale-by-dt mesh="{string}" timing="{string}"> <source-data name="{string}"/> <target-data name="{string}"/> </action:scale-by-dt>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior` , `regular-post` , `on-exchange-prior` , `on-exchange-post` , `on-time-window-complete-post` , `write-mapping-prior` , `write-mapping-post` , `read-mapping-prior` , `read-mapping-post`

Valid Subtags:

- source-data (page 108) `1`
- target-data (page 107) `1`

source-data

Single data to read from.

Example:

```
<source-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

target-data

Data to read from and write to.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

action:summation

Sums up multiple source data values and writes the result into target data.

Example:

```
<action:summation mesh="{string}" timing="{string}"> <source-data name="{string}"/> <target-data name="{string}"/> </action:summation>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior`, `regular-post`, `on-exchange-prior`, `on-exchange-post`, `on-time-window-complete-post`, `write-mapping-prior`, `write-mapping-post`, `read-mapping-prior`, `read-mapping-post`

Valid Subtags:

- source-data (page 108) `1..*`
- target-data (page 107) `1`

source-data

Multiple data to read from.

Example:

```
<source-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

target-data

Data to read from and write to.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

action:compute-curvature

Computes curvature values at mesh vertices.

Example:

```
<action:compute-curvature mesh="{string}" timing="{string}"> <target-data name="{string}"/> </actio
n:compute-curvature>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior` , `regular-post` , `on-exchange-prior` , `on-exchange-post` , `on-time-window-complete-post` , `write-mapping-prior` , `write-mapping-post` , `read-mapping-prior` , `read-mapping-post`

Valid Subtags:

- target-data (page 107) `1`

target-data

Data to read from and write to.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

action:recorder

Records action invocations for testing purposes.

Example:

```
<action:recorder mesh="{string}" timing="{string}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior`, `regular-post`, `on-exchange-prior`, `on-exchange-post`, `on-time-window-complete-post`, `write-mapping-prior`, `write-mapping-post`, `read-mapping-prior`, `read-mapping-post`

action:python

Calls Python script to execute action. See preCICE file “src/action/PythonAction.py” for an overview.

Example:

```
<action:python mesh="{string}" timing="{string}"> <path name=""/> <module name="{string}"/> <sourc
e-data name="{string}"/> <target-data name="{string}"/> </action:python>
```

Attribute	Type	Description	Default	Options
mesh	string	Determines mesh used in action.	*none*	none
timing	string	Determines when (relative to advancing the coupling scheme) the action is executed.	*none*	`regular-prior`, `regular-post`, `on-exchange-prior`, `on-exchange-post`, `on-time-window-complete-post`, `write-mapping-prior`, `write-mapping-post`, `read-mapping-prior`, `read-mapping-post`

Valid Subtags:

- path (page 112) `0..1`
- module (page 113) `1`
- source-data (page 108) `0..1`
- target-data (page 107) `0..1`

path

Directory path to Python module, i.e. script file. If it doesn’t occur, the current path is used

Example:

```
<path name=""/>
```

Attribute	Type	Description	Default	Options
name	string	The path to the directory of the module.	``	none

module

Name of Python module, i.e. Python script file without file ending. The module name has to differ from existing (library) modules, otherwise, the existing module will be loaded instead of the user script.

Example:

```
<module name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

source-data

Source data to be read is handed to the Python module. Can be omitted, if only a target data is needed.

Example:

```
<source-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

target-data

Target data to be read and written to is handed to the Python module. Can be omitted, if only source data is needed.

Example:

```
<target-data name="{string}"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the data.	*none*	none

export:vtk

Exports meshes to VTK text files.

Example:

```
<export:vtk every-n-time-windows="1" directory="" every-iteration="0" normals="1"/>
```

Attribute	Type	Description	Default	Options
every-n-time-windows	integer	preCICE does an export every X time windows. Choose -1 for no exports.	1	none
directory	string	Directory to export the files to.	``	none
every-iteration	boolean	Exports in every coupling (sub)iteration. For debug purposes.	0	none
normals	boolean	If set to on/yes, mesh normals (if available) are added to the export.	1	none

watch-point

A watch point can be used to follow the transient changes of data and mesh vertex coordinates at a given point

Example:

```
<watch-point mesh="{string}" name="{string}" coordinate="{vector}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	Mesh to be watched.	*none*	none
name	string	Name of the watch point. Is taken in combination with the participant name to construct the filename the watch point data is written to.	*none*	none
coordinate	vector	The coordinates of the watch point. If the watch point is not put exactly on the mesh to observe, the closest projection of the point onto the mesh is considered instead, and values/coordinates are interpolated linearly to that point.	*none*	none

watch-integral

A watch integral can be used to follow the transient change of integral data and surface area for a given coupling mesh.

Example:

```
<watch-integral mesh="{string}" name="{string}" scale-with-connectivity="{boolean}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	Mesh to be watched.	*none*	none
name	string	Name of the watch integral. Is taken in combination with the participant name to construct the filename the watch integral data is written to.	*none*	none

Attribute	Type	Description	Default	Options
scale-with-connectivity	boolean	Whether the vertex data is scaled with the element area before summing up or not. In 2D, vertex data is scaled with the average length of neighboring edges. In 3D, vertex data is scaled with the average surface of neighboring triangles. If false, vertex data is directly summed up.	*none*	none

use-mesh

Makes a mesh (see tag available to a participant.

Example:

```
<use-mesh safety-factor="0.5" from="" geometric-filter="on-slaves" name="{string}" provide="0"/>
```

Attribute	Type	Description	Default	Options
safety-factor	float	If a mesh is received from another partipant (see tag), it needs to bedecomposed at the receiving participant. To speed up this process, a geometric filter (see tag), i.e. filtering by bounding boxes around the local mesh, can be used. This safety factor defines by which factor this local information is increased. An example: 0.5 means that the bounding box is 150% of its original size.	`0.5`	none
from	string	If a created mesh should be used by another solver, this attribute has to specify the creating participant's name. The creator has to use the attribute "provide" to signal he is providing the mesh geometry.	``	none
geometric-filter	string	If a mesh is received from another partipant (see tag), it needs to bedecomposed at the receiving participant. To speed up this process, a geometric filter, i.e. filtering by bounding boxes around the local mesh, can be used. Two different variants are implemented: a filter "on-master" strategy, which is beneficial for a huge mesh and a low number of processors, and a filter "on-slaves" strategy, which performs better for a very high number of processors. Both result in the same distribution (if the safety factor is sufficiently large). "on-master" is not supported if you use two-level initialization. For very asymmetric cases, the filter can also be switched off completely ("no-filter").	`on-slaves`	`on-master` , `on-slaves` , `no-filter`
name	string	Name of the mesh.	*none*	none
provide	boolean	If this attribute is set to "on", the participant has to create the mesh geometry before initializing preCICE.	`0`	none

master:sockets

A solver in parallel needs a communication between its ranks. By default, the participant's MPI_COM_WORLD is reused.Use this tag to use TCP/IP sockets instead.

Example:

```
<master:sockets port="0" exchange-directory="" network="lo"/>
```

Attribute	Type	Description	Default	Options
port	integer	Port number (16-bit unsigned integer) to be used for socket communiation. The default is “0”, what means that OS will dynamically search for a free port (if at least one exists) and bind it automatically.	`0`	none
exchange-directory	string	Directory where connection information is exchanged. By default, the directory of startup is chosen.	``	none
network	string	Interface name to be used for socket communiation. Default is the cannonical name of the loopback interface of your platform. Might be different on supercomputing systems, e.g. “ib0” for the InfiniBand on SuperMUC.	`lo`	none

master:mpi

A solver in parallel needs a communication between its ranks. By default, the participant’s MPI_COM_WORLD is reused.Use this tag to use MPI with separated communication spaces instead instead.

Example:

```
<master:mpi exchange-directory=""/>
```

Attribute	Type	Description	Default	Options
exchange-directory	string	Directory where connection information is exchanged. By default, the directory of startup is chosen.	``	none

master:mpi-single

A solver in parallel needs a communication between its ranks. By default (which is this option), the participant’s MPI_COM_WORLD is reused.This tag is only used to ensure backwards compatibility.

Example:

```
<master:mpi-single/>
```

coupling-scheme:serial-explicit

Explicit coupling scheme according to conventional serial staggered procedure (CSS).

Example:

```
<coupling-scheme:serial-explicit> <max-time value="{float}"/> <max-time-windows value="{integer}"/>
<time-window-size value="-1" valid-digits="10" method="fixed"/> <participants first="{string}" seco
nd="{string}"/> <exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initializ
e="0"/> </coupling-scheme:serial-explicit>
```

Valid Subtags:

- max-time (page 117) `0..1`

- max-time-windows (page 117) `0..1`
- time-window-size (page 117) `1`
- participants (page 117) `1`
- exchange (page 118) `1..*`

max-time

Defined the end of the simulation as total time.

Example:

```
<max-time value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	The value of the maximum simulation time.	*none*	none

max-time-windows

Defined the end of the simulation as a total count of time windows.

Example:

```
<max-time-windows value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum count of time windows.	*none*	none

time-window-size

Defines the size of the time window.

Example:

```
<time-window-size value="-1" valid-digits="10" method="fixed"/>
```

Attribute	Type	Description	Default	Options
value	float	The maximum time window size.	`-1`	none
valid-digits	integer	Precision to use when checking for end of time windows used this many digits. $\phi = 10^{-validDigits}$	`10`	none
method	string	The method used to determine the time window size. Use `fixed` to fix the time window size for the participants.	`fixed`	`fixed`, `first-participant`

participants

Defines the participants of the coupling scheme.

Example:

```
<participants first="{string}" second="{string}"/>
```

Attribute	Type	Description	Default	Options
first	string	First participant to run the solver.	*none*	none
second	string	Second participant to run the solver.	*none*	none

exchange

Defines the flow of data between meshes of participants.

Example:

```
<exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initialize="0"/>
```

Attribute	Type	Description	Default	Options
data	string	The data to exchange.	*none*	none
from	string	The participant sending the data.	*none*	none
mesh	string	The mesh which uses the data.	*none*	none
to	string	The participant receiving the data.	*none*	none
initialize	boolean	Should this data be initialized during initializeData?	`0`	none

coupling-scheme:parallel-explicit

Explicit coupling scheme according to conventional parallel staggered procedure (CPS).

Example:

```
<coupling-scheme:parallel-explicit> <max-time value="{float}"/> <max-time-windows value="{intege
r}"/> <time-window-size value="-1" valid-digits="10" method="fixed"/> <participants first="{strin
g}" second="{string}"/> <exchange data="{string}" from="{string}" mesh="{string}" to="{string}" ini
tialize="0"/> </coupling-scheme:parallel-explicit>
```

Valid Subtags:

- max-time (page 118) `0..1`
- max-time-windows (page 119) `0..1`
- time-window-size (page 119) `1`
- participants (page 119) `1`
- exchange (page 120) `1..*`

max-time

Defined the end of the simulation as total time.

Example:

```
<max-time value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	The value of the maximum simulation time.	*none*	none

max-time-windows

Defined the end of the simulation as a total count of time windows.

Example:

```
<max-time-windows value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum count of time windows.	*none*	none

time-window-size

Defines the size of the time window.

Example:

```
<time-window-size value="-1" valid-digits="10" method="fixed"/>
```

Attribute	Type	Description	Default	Options
value	float	The maximum time window size.	`-1`	none
valid-digits	integer	Precision to use when checking for end of time windows used this many digits. $\phi = 10^{-validDigits}$	`10`	none
method	string	The method used to determine the time window size. Use `fixed` to fix the time window size for the participants.	`fixed`	`fixed`

participants

Defines the participants of the coupling scheme.

Example:

```
<participants first="{string}" second="{string}"/>
```

Attribute	Type	Description	Default	Options
first	string	First participant to run the solver.	*none*	none

Attribute	Type	Description	Default	Options
second	string	Second participant to run the solver.	*none*	none

exchange

Defines the flow of data between meshes of participants.

Example:

```
<exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initialize="0"/>
```

Attribute	Type	Description	Default	Options
data	string	The data to exchange.	*none*	none
from	string	The participant sending the data.	*none*	none
mesh	string	The mesh which uses the data.	*none*	none
to	string	The participant receiving the data.	*none*	none
initialize	boolean	Should this data be initialized during initializeData?	`0`	none

coupling-scheme:serial-implicit

Implicit coupling scheme according to block Gauss-Seidel iterations (S-System). Improved implicit iterations are achieved by using a acceleration (recommended!).

Example:

```
<coupling-scheme:serial-implicit> <max-time value="{float}"/> <max-time-windows value="{integer}"/>
<time-window-size value="-1" valid-digits="10" method="fixed"/> <participants first="{string}" seco
nd="{string}"/> <exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initializ
e="0"/> <acceleration:constant> ... </acceleration:constant> <absolute-convergence-measure limi
t="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <relative-convergence-measure
limit="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <residual-relative-conver
gence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <min-iterat
ion-convergence-measure min-iterations="{integer}" data="{string}" mesh="{string}" strict="0" suffi
ces="0"/> <max-iterations value="{integer}"/> <extrapolation-order value="{integer}"/> </coupling-s
cheme:serial-implicit>
```

Valid Subtags:

- max-time (page 118) `0..1`
- max-time-windows (page 119) `0..1`
- time-window-size (page 119) `1`
- participants (page 119) `1`
- exchange (page 120) `1..*`
- absolute-convergence-measure (page 131) `0..*`

- relative-convergence-measure (page 131) `0..*`
- residual-relative-convergence-measure (page 132) `0..*`
- min-iteration-convergence-measure (page 132) `0..*`
- max-iterations (page 133) `1`
- extrapolation-order (page 133) `0..1`
- acceleration
  - constant (page 122) `0..1`
  - aitken (page 123) `0..1`
  - IQN-ILS (page 124) `0..1`
  - IQN-IMVJ (page 126) `0..1`
  - broyden (page 129) `0..1`

max-time

Defined the end of the simulation as total time.

Example:

```
<max-time value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	The value of the maximum simulation time.	*none*	none

max-time-windows

Defined the end of the simulation as a total count of time windows.

Example:

```
<max-time-windows value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum count of time windows.	*none*	none

time-window-size

Defines the size of the time window.

Example:

```
<time-window-size value="-1" valid-digits="10" method="fixed"/>
```

Attribute	Type	Description	Default	Options
value	float	The maximum time window size.	`-1`	none

Attribute	Type	Description	Default	Options
valid-digits	integer	Precision to use when checking for end of time windows used this many digits. $\phi = 10^{-validDigits}$	`10`	none
method	string	The method used to determine the time window size. Use `fixed` to fix the time window size for the participants.	`fixed`	`fixed` , `first-participant`

participants

Defines the participants of the coupling scheme.

Example:

```
<participants first="{string}" second="{string}"/>
```

Attribute	Type	Description	Default	Options
first	string	First participant to run the solver.	*none*	none
second	string	Second participant to run the solver.	*none*	none

exchange

Defines the flow of data between meshes of participants.

Example:

```
<exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initialize="0"/>
```

Attribute	Type	Description	Default	Options
data	string	The data to exchange.	*none*	none
from	string	The participant sending the data.	*none*	none
mesh	string	The mesh which uses the data.	*none*	none
to	string	The participant receiving the data.	*none*	none
initialize	boolean	Should this data be initialized during initializeData?	`0`	none

acceleration:constant

Accelerates coupling data with constant underrelaxation.

Example:

```
<acceleration:constant> <relaxation value="{float}"/> </acceleration:constant>
```

Valid Subtags:

- relaxation (page 123) `1`

relaxation

Example:

```
<relaxation value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	Constant relaxation factor.	*none*	none

acceleration:aitken

Accelerates coupling data with dynamic Aitken under-relaxation.

Example:

```
<acceleration:aitken> <initial-relaxation value="{float}"/> <data mesh="{string}" name="{string}"/>
</acceleration:aitken>
```

Valid Subtags:

- initial-relaxation (page 123) `1`
- data (page 123) `1..*`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

acceleration:IQN-ILS

Accelerates coupling data with the interface quasi-Newton inverse least-squares method.

Example:

```
<acceleration:IQN-ILS> <initial-relaxation value="{float}" enforce="0"/> <max-used-iterations valu
e="{integer}"/> <time-windows-reused value="{integer}"/> <data scaling="1" mesh="{string}" name="{s
tring}"/> <filter limit="1e-16" type="{string}"/> <preconditioner freeze-after="-1" type="{strin
g}"/> </acceleration:IQN-ILS>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- max-used-iterations (page 124) `1`
- time-windows-reused (page 124) `1`
- data (page 125) `1..*`
- filter (page 125) `0..1`
- preconditioner (page 126) `0..1`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none
enforce	boolean	Enforce initial relaxation in every time window.	`0`	none

max-used-iterations

Maximum number of columns used in low-rank approximation of Jacobian.

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

time-windows-reused

Number of past time windows from which columns are used to approximate Jacobian.

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of time windows.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	`1`	none
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

filter

Type of filtering technique that is used to maintain good conditioning in the least-squares system. Possible filters:

- `QR1-filter`: updateQR-dec with (relative) test $R(i,i) < \epsilon * \|R\|_F$
- `QR1_absolute-filter`: updateQR-dec with (absolute) test $R(i,i) < \epsilon$
- `QR2-filter`: en-block QR-dec with test $\|v_{\text{orth}}\|_2 < \epsilon * \|v\|_2$

Please note that a QR1 is based on Given's rotations whereas QR2 uses modified Gram-Schmidt. This can give different results even when no columns are filtered out.

Example:

```
<filter limit="1e-16" type="{string}"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit eps of the filter.	`1e-16`	none
type	string	Type of the filter.	*none*	`QR1`, `QR1-absolute`, `QR2`

preconditioner

To improve the performance of a parallel or a multi coupling schemes a preconditioner can be applied. A constant preconditioner scales every acceleration data by a constant value, which you can define as an attribute of data. A value preconditioner scales every acceleration data by the norm of the data in the previous time window. A residual preconditioner scales every acceleration data by the current residual. A residual-sum preconditioner scales every acceleration data by the sum of the residuals from the current time window.

Example:

```
<preconditioner freeze-after="-1" type="{string}"/>
```

Attribute	Type	Description	Default	Options
freeze-after	integer	After the given number of time steps, the preconditioner weights are frozen and the preconditioner acts like a constant preconditioner.	`-1`	none
type	string	The type of the preconditioner.	*none*	`constant` , `value` , `residual` , `residual-sum`

acceleration:IQN-IMVJ

Accelerates coupling data with the interface quasi-Newton inverse multi-vector Jacobian method.

Example:

```
<acceleration:IQN-IMVJ always-build-jacobian="0"> <initial-relaxation value="{float}" enforce="0"/>
<imvj-restart-mode truncation-threshold="0.0001" chunk-size="8" reused-time-windows-at-restart="8"
type="RS-SVD"/> <max-used-iterations value="{integer}"/> <time-windows-reused value="{integer}"/>
<data scaling="1" mesh="{string}" name="{string}"/> <filter limit="1e-16" type="{string}"/> <precon
ditioner freeze-after="-1" type="{string}"/> </acceleration:IQN-IMVJ>
```

Attribute	Type	Description	Default	Options
always-build-jacobian	boolean	If set to true, the IMVJ will set up the Jacobian matrix in each coupling iteration, which is inefficient. If set to false (or not set) the Jacobian is only build in the last iteration and the updates are computed using (relatively) cheap MATVEC products.	`0`	none

Valid Subtags:

- initial-relaxation (page 124) `1`
- imvj-restart-mode (page 127) `0..1`
- max-used-iterations (page 128) `1`
- time-windows-reused (page 128) `1`
- data (page 125) `1..*`
- filter (page 128) `0..1`

- preconditioner (page 129) `0..1`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none
enforce	boolean	Enforce initial relaxation in every time window.	`0`	none

imvj-restart-mode

Type of IMVJ restart mode that is used:

- `no-restart` : IMVJ runs in normal mode with explicit representation of Jacobian
- `RS-ZERO` : IMVJ runs in restart mode. After M time steps all Jacobain information is dropped, restart with no information
- `RS-LS` : IMVJ runs in restart mode. After M time steps a IQN-LS like approximation for the initial guess of the Jacobian is computed.
- `RS-SVD` : IMVJ runs in restart mode. After M time steps a truncated SVD of the Jacobian is updated.
- `RS-SLIDE` : IMVJ runs in sliding window restart mode.

Example:

```
<imvj-restart-mode truncation-threshold="0.0001" chunk-size="8" reused-time-windows-at-restart="8"
type="RS-SVD"/>
```

Attribute	Type	Description	Default	Options
truncation-threshold	float	If IMVJ restart-mode=RS-SVD, the truncation threshold for the updated SVD can be set.	`0.0001`	none
chunk-size	integer	Specifies the number of time steps M after which the IMVJ restarts, if run in restart-mode. Defaul value is M=8.	`8`	none
reused-time-windows-at-restart	integer	If IMVJ restart-mode=RS-LS, the number of reused time steps at restart can be specified.	`8`	none
type	string	Type of the restart mode.	`RS-SVD`	`no-restart` , `RS-0` , `RS-LS` , `RS-SVD` , `RS-SLIDE`

max-used-iterations

Maximum number of columns used in low-rank approximation of Jacobian.

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

time-windows-reused

Number of past time windows from which columns are used to approximate Jacobian.

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	1	none
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

filter

Type of filtering technique that is used to maintain good conditioning in the least-squares system. Possible filters:

- `QR1-filter`: updateQR-dec with (relative) test $R(i,i) < \epsilon * \|R\|_F$
- `QR1_absolute-filter`: updateQR-dec with (absolute) test $R(i,i) < \epsilon$
- `QR2-filter`: en-block QR-dec with test $\|v_{\text{orth}}\|_2 < \epsilon * \|v\|_2$

Please note that a QR1 is based on Given's rotations whereas QR2 uses modified Gram-Schmidt. This can give different results even when no columns are filtered out.

Example:

```
<filter limit="1e-16" type="{string}"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit eps of the filter.	`1e-16`	none
type	string	Type of the filter.	*none*	`QR1` , `QR1-absolute` , `QR2`

preconditioner

To improve the performance of a parallel or a multi coupling schemes a preconditioner can be applied. A constant preconditioner scales every acceleration data by a constant value, which you can define as an attribute of data.

- A value preconditioner scales every acceleration data by the norm of the data in the previous time window.
- A residual preconditioner scales every acceleration data by the current residual.
- A residual-sum preconditioner scales every acceleration data by the sum of the residuals from the current time window.

Example:

```
<preconditioner freeze-after="-1" type="{string}"/>
```

Attribute	Type	Description	Default	Options
freeze-after	integer	After the given number of time steps, the preconditioner weights are frozen and the preconditioner acts like a constant preconditioner.	`-1`	none
type	string	Type of the preconditioner.	*none*	`constant` , `value` , `residual` , `residual-sum`

acceleration:broyden

Accelerates coupling data with the (single-vector) Broyden method.

Example:

```
<acceleration:broyden> <initial-relaxation value="{float}" enforce="0"/> <max-used-iterations value="{integer}"/> <time-windows-reused value="{integer}"/> <data scaling="1" mesh="{string}" name="{string}"/> </acceleration:broyden>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- max-used-iterations (page 128) `1`

- time-windows-reused (page 128) `1`
- data (page 125) `1..*`

initial-relaxation

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float		*none*	none
enforce	boolean		`0`	none

max-used-iterations

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer		*none*	none

time-windows-reused

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer		*none*	none

data

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	`1`	none

Attribute	Type	Description	Default	Options
mesh	string		*none*	none
name	string		*none*	none

absolute-convergence-measure

Absolute convergence criterion based on the two-norm difference of data values between iterations.

$$\left\| H(x^k) - x^k \right\|_2 < \text{limit}$$

Example:

```
<absolute-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffice
s="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in ((0, 1]).	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

relative-convergence-measure

Relative convergence criterion based on the relative two-norm difference of data values between iterations.

$$\frac{\left\| H(x^k) - x^k \right\|_2}{\left\| H(x^k) \right\|_2} < \text{limit}$$

Example:

```
<relative-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffice
s="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in $(0, 1]$.	*none*	none
data	string	Data to be measured.	*none*	none

Attribute	Type	Description	Default	Options
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

residual-relative-convergence-measure

Residual relative convergence criterion based on the relative two-norm differences of data values between iterations.

$$\frac{\left\| H(x^k) - x^k \right\|_2}{\left\| H(x^{k-1}) - x^{k-1} \right\|_2} < \text{limit}$$

Example:

```
<residual-relative-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" s
uffices="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in ((0, 1]).	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

min-iteration-convergence-measure

Convergence criterion used to ensure a miminimal amount of iterations. Specifying a mesh and data is required for technical reasons and does not influence the measure.

Example:

```
<min-iteration-convergence-measure min-iterations="{integer}" data="{string}" mesh="{string}" stric
t="0" suffices="0"/>
```

Attribute	Type	Description	Default	Options
min-iterations	integer	The minimal amount of iterations.	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

max-iterations

Allows to specify a maximum amount of iterations per time window.

Example:

```
<max-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum value of iterations.	*none*	none

extrapolation-order

Sets order of predictor of interface values for first participant.

Example:

```
<extrapolation-order value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The extrapolation order to use.	*none*	none

coupling-scheme:parallel-implicit

Parallel Implicit coupling scheme according to block Jacobi iterations (V-System). Improved implicit iterations are achieved by using a acceleration (recommended!).

Example:

```
<coupling-scheme:parallel-implicit> <max-time value="{float}"/> <max-time-windows value="{intege
r}"/> <time-window-size value="-1" valid-digits="10" method="fixed"/> <participants first="{strin
g}" second="{string}"/> <exchange data="{string}" from="{string}" mesh="{string}" to="{string}" ini
tialize="0"/> <acceleration:constant> ... </acceleration:constant> <absolute-convergence-measure li
mit="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <relative-convergence-measu
re limit="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <residual-relative-con
vergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <min-ite
ration-convergence-measure min-iterations="{integer}" data="{string}" mesh="{string}" strict="0" su
ffices="0"/> <max-iterations value="{integer}"/> <extrapolation-order value="{integer}"/> </couplin
g-scheme:parallel-implicit>
```

Valid Subtags:

- max-time (page 118) `0..1`
- max-time-windows (page 119) `0..1`
- time-window-size (page 119) `1`
- participants (page 119) `1`
- exchange (page 120) `1..*`
- absolute-convergence-measure (page 144) `0..*`
- relative-convergence-measure (page 145) `0..*`
- residual-relative-convergence-measure (page 145) `0..*`
- min-iteration-convergence-measure (page 146) `0..*`
- max-iterations (page 146) `1`
- extrapolation-order (page 146) `0..1`
- acceleration
    - constant (page 136) `0..1`
    - aitken (page 136) `0..1`
    - IQN-ILS (page 137) `0..1`
    - IQN-IMVJ (page 139) `0..1`
    - broyden (page 143) `0..1`

max-time

Defined the end of the simulation as total time.

Example:

```
<max-time value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	The value of the maximum simulation time.	*none*	none

max-time-windows

Defined the end of the simulation as a total count of time windows.

Example:

```
<max-time-windows value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum count of time windows.	*none*	none

time-window-size

Defines the size of the time window.

Example:

```
<time-window-size value="-1" valid-digits="10" method="fixed"/>
```

Attribute	Type	Description	Default	Options
value	float	The maximum time window size.	`-1`	none
valid-digits	integer	Precision to use when checking for end of time windows used this many digits. $\phi = 10^{-validDigits}$	`10`	none
method	string	The method used to determine the time window size. Use `fixed` to fix the time window size for the participants.	`fixed`	`fixed`

participants

Defines the participants of the coupling scheme.

Example:

```
<participants first="{string}" second="{string}"/>
```

Attribute	Type	Description	Default	Options
first	string	First participant to run the solver.	*none*	none
second	string	Second participant to run the solver.	*none*	none

exchange

Defines the flow of data between meshes of participants.

Example:

```
<exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initialize="0"/>
```

Attribute	Type	Description	Default	Options
data	string	The data to exchange.	*none*	none
from	string	The participant sending the data.	*none*	none
mesh	string	The mesh which uses the data.	*none*	none
to	string	The participant receiving the data.	*none*	none
initialize	boolean	Should this data be initialized during initializeData?	`0`	none

acceleration:constant

Accelerates coupling data with constant underrelaxation.

Example:

```
<acceleration:constant> <relaxation value="{float}"/> </acceleration:constant>
```

Valid Subtags:

- relaxation (page 136) `1`

relaxation

Example:

```
<relaxation value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	Constant relaxation factor.	*none*	none

acceleration:aitken

Accelerates coupling data with dynamic Aitken under-relaxation.

Example:

```
<acceleration:aitken> <initial-relaxation value="{float}"/> <data mesh="{string}" name="{string}"/>
</acceleration:aitken>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- data (page 125) `1..*`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

acceleration:IQN-ILS

Accelerates coupling data with the interface quasi-Newton inverse least-squares method.

Example:

```
<acceleration:IQN-ILS> <initial-relaxation value="{float}" enforce="0"/> <max-used-iterations valu
e="{integer}"/> <time-windows-reused value="{integer}"/> <data scaling="1" mesh="{string}" name="{s
tring}"/> <filter limit="1e-16" type="{string}"/> <preconditioner freeze-after="-1" type="{strin
g}"/> </acceleration:IQN-ILS>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- max-used-iterations (page 128) `1`
- time-windows-reused (page 128) `1`
- data (page 125) `1..*`
- filter (page 128) `0..1`
- preconditioner (page 129) `0..1`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none
enforce	boolean	Enforce initial relaxation in every time window.	`0`	none

max-used-iterations

Maximum number of columns used in low-rank approximation of Jacobian.

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

time-windows-reused

Number of past time windows from which columns are used to approximate Jacobian.

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of time windows.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	`1`	none
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

filter

Type of filtering technique that is used to maintain good conditioning in the least-squares system. Possible filters:

- `QR1-filter` : updateQR-dec with (relative) test $R(i,i) < \epsilon * \|R\|_F$
- `QR1_absolute-filter` : updateQR-dec with (absolute) test $R(i,i) < \epsilon$
- `QR2-filter` : en-block QR-dec with test $\|v_{\text{orth}}\|_2 < \epsilon * \|v\|_2$

Please note that a QR1 is based on Given's rotations whereas QR2 uses modified Gram-Schmidt. This can give different results even when no columns are filtered out.

Example:

```
<filter limit="1e-16" type="{string}"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit eps of the filter.	`1e-16`	none
type	string	Type of the filter.	*none*	`QR1` , `QR1-absolute` , `QR2`

preconditioner

To improve the performance of a parallel or a multi coupling schemes a preconditioner can be applied. A constant preconditioner scales every acceleration data by a constant value, which you can define as an attribute of data. A value preconditioner scales every acceleration data by the norm of the data in the previous time window. A residual preconditioner scales every acceleration data by the current residual. A residual-sum preconditioner scales every acceleration data by the sum of the residuals from the current time window.

Example:

```
<preconditioner freeze-after="-1" type="{string}"/>
```

Attribute	Type	Description	Default	Options
freeze-after	integer	After the given number of time steps, the preconditioner weights are frozen and the preconditioner acts like a constant preconditioner.	`-1`	none
type	string	The type of the preconditioner.	*none*	`constant` , `value` , `residual` , `residual-sum`

acceleration:IQN-IMVJ

Accelerates coupling data with the interface quasi-Newton inverse multi-vector Jacobian method.

Example:

```
<acceleration:IQN-IMVJ always-build-jacobian="0"> <initial-relaxation value="{float}" enforce="0"/>
<imvj-restart-mode truncation-threshold="0.0001" chunk-size="8" reused-time-windows-at-restart="8"
type="RS-SVD"/> <max-used-iterations value="{integer}"/> <time-windows-reused value="{integer}"/>
<data scaling="1" mesh="{string}" name="{string}"/> <filter limit="1e-16" type="{string}"/> <precon
ditioner freeze-after="-1" type="{string}"/> </acceleration:IQN-IMVJ>
```

Attribute	Type	Description	Default	Options
always-build-jacobian	boolean	If set to true, the IMVJ will set up the Jacobian matrix in each coupling iteration, which is inefficient. If set to false (or not set) the Jacobian is only build in the last iteration and the updates are computed using (relatively) cheap MATVEC products.	`0`	none

Valid Subtags:

- initial-relaxation (page 124) `1`
- imvj-restart-mode (page 140) `0..1`
- max-used-iterations (page 128) `1`
- time-windows-reused (page 128) `1`
- data (page 125) `1..*`
- filter (page 128) `0..1`
- preconditioner (page 129) `0..1`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none
enforce	boolean	Enforce initial relaxation in every time window.	`0`	none

imvj-restart-mode

Type of IMVJ restart mode that is used:

- `no-restart` : IMVJ runs in normal mode with explicit representation of Jacobian
- `RS-ZERO` : IMVJ runs in restart mode. After M time steps all Jacobain information is dropped, restart with no information
- `RS-LS` : IMVJ runs in restart mode. After M time steps a IQN-LS like approximation for the initial guess of the Jacobian is computed.
- `RS-SVD` : IMVJ runs in restart mode. After M time steps a truncated SVD of the Jacobian is updated.

- `RS-SLIDE` : IMVJ runs in sliding window restart mode.

Example:

```
<imvj-restart-mode truncation-threshold="0.0001" chunk-size="8" reused-time-windows-at-restart="8"
type="RS-SVD"/>
```

Attribute	Type	Description	Default	Options
truncation-threshold	float	If IMVJ restart-mode=RS-SVD, the truncation threshold for the updated SVD can be set.	`0.0001`	none
chunk-size	integer	Specifies the number of time steps M after which the IMVJ restarts, if run in restart-mode. Defaul value is M=8.	`8`	none
reused-time-windows-at-restart	integer	If IMVJ restart-mode=RS-LS, the number of reused time steps at restart can be specified.	`8`	none
type	string	Type of the restart mode.	`RS-SVD`	`no-restart` , `RS-0` , `RS-LS` , `RS-SVD` , `RS-SLIDE`

max-used-iterations

Maximum number of columns used in low-rank approximation of Jacobian.

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

time-windows-reused

Number of past time windows from which columns are used to approximate Jacobian.

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	`1`	none
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

filter

Type of filtering technique that is used to maintain good conditioning in the least-squares system. Possible filters:

- `QR1-filter`: updateQR-dec with (relative) test $R(i,i) < \epsilon * \|R\|_F$
- `QR1_absolute-filter`: updateQR-dec with (absolute) test $R(i,i) < \epsilon$
- `QR2-filter`: en-block QR-dec with test $\|v_{\text{orth}}\|_2 < \epsilon * \|v\|_2$

Please note that a QR1 is based on Given's rotations whereas QR2 uses modified Gram-Schmidt. This can give different results even when no columns are filtered out.

Example:

```
<filter limit="1e-16" type="{string}"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit eps of the filter.	`1e-16`	none
type	string	Type of the filter.	*none*	`QR1`, `QR1-absolute`, `QR2`

preconditioner

To improve the performance of a parallel or a multi coupling schemes a preconditioner can be applied. A constant preconditioner scales every acceleration data by a constant value, which you can define as an attribute of data.

- A value preconditioner scales every acceleration data by the norm of the data in the previous time window.
- A residual preconditioner scales every acceleration data by the current residual.
- A residual-sum preconditioner scales every acceleration data by the sum of the residuals from the current time window.

Example:

```
<preconditioner freeze-after="-1" type="{string}"/>
```

Attribute	Type	Description	Default	Options
freeze-after	integer	After the given number of time steps, the preconditioner weights are frozen and the preconditioner acts like a constant preconditioner.	`-1`	none
type	string	Type of the preconditioner.	*none*	`constant` , `value` , `residual` , `residual-sum`

acceleration:broyden

Accelerates coupling data with the (single-vector) Broyden method.

Example:

```
<acceleration:broyden> <initial-relaxation value="{float}" enforce="0"/> <max-used-iterations value="{integer}"/> <time-windows-reused value="{integer}"/> <data scaling="1" mesh="{string}" name="{string}"/> </acceleration:broyden>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- max-used-iterations (page 128) `1`
- time-windows-reused (page 128) `1`
- data (page 125) `1..*`

initial-relaxation

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float		*none*	none
enforce	boolean		`0`	none

max-used-iterations

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer		*none*	none

time-windows-reused

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer		*none*	none

data

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	1	none
mesh	string		*none*	none
name	string		*none*	none

absolute-convergence-measure

Absolute convergence criterion based on the two-norm difference of data values between iterations.

$$\left\| H(x^k) - x^k \right\|_2 < \text{limit}$$

Example:

```
<absolute-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffice
s="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in ((0, 1]).	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	0	none

Attribute	Type	Description	Default	Options
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	0	none

relative-convergence-measure

Relative convergence criterion based on the relative two-norm difference of data values between iterations.

$$\frac{\left\|H(x^k) - x^k\right\|_2}{\left\|H(x^k)\right\|_2} < \text{limit}$$

Example:

```
<relative-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffice
s="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in $(0, 1]$.	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	0	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	0	none

residual-relative-convergence-measure

Residual relative convergence criterion based on the relative two-norm differences of data values between iterations.

$$\frac{\left\|H(x^k) - x^k\right\|_2}{\left\|H(x^{k-1}) - x^{k-1}\right\|_2} < \text{limit}$$

Example:

```
<residual-relative-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" s
uffices="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in ((0, 1]).	*none*	none
data	string	Data to be measured.	*none*	none

Attribute	Type	Description	Default	Options
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	0	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	0	none

min-iteration-convergence-measure

Convergence criterion used to ensure a miminimal amount of iterations. Specifying a mesh and data is required for technical reasons and does not influence the measure.

Example:

```
<min-iteration-convergence-measure min-iterations="{integer}" data="{string}" mesh="{string}" strict="0" suffices="0"/>
```

Attribute	Type	Description	Default	Options
min-iterations	integer	The minimal amount of iterations.	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	0	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	0	none

max-iterations

Allows to specify a maximum amount of iterations per time window.

Example:

```
<max-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum value of iterations.	*none*	none

extrapolation-order

Sets order of predictor of interface values for first participant.

Example:

```
<extrapolation-order value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The extrapolation order to use.	*none*	none

coupling-scheme:multi

Multi coupling scheme according to block Jacobi iterations. Improved implicit iterations are achieved by using a acceleration (recommended!).

Example:

```
<coupling-scheme:multi> <max-time value="{float}"/> <max-time-windows value="{integer}"/> <time-win
dow-size value="-1" valid-digits="10" method="fixed"/> <participant name="{string}" control="0"/>
<exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initialize="0"/> <accelerat
ion:constant> ... </acceleration:constant> <absolute-convergence-measure limit="{float}" data="{str
ing}" mesh="{string}" strict="0" suffices="0"/> <relative-convergence-measure limit="{float}" dat
a="{string}" mesh="{string}" strict="0" suffices="0"/> <residual-relative-convergence-measure limi
t="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <min-iteration-convergence-me
asure min-iterations="{integer}" data="{string}" mesh="{string}" strict="0" suffices="0"/> <max-ite
rations value="{integer}"/> <extrapolation-order value="{integer}"/> </coupling-scheme:multi>
```

Valid Subtags:

- max-time (page 118) `0..1`
- max-time-windows (page 119) `0..1`
- time-window-size (page 119) `1`
- participant (page 148) `1..*`
- exchange (page 120) `1..*`
- absolute-convergence-measure (page 144) `0..*`
- relative-convergence-measure (page 145) `0..*`
- residual-relative-convergence-measure (page 145) `0..*`
- min-iteration-convergence-measure (page 146) `0..*`
- max-iterations (page 146) `1`
- extrapolation-order (page 146) `0..1`
- acceleration
    - constant (page 136) `0..1`
    - aitken (page 136) `0..1`
    - IQN-ILS (page 137) `0..1`
    - IQN-IMVJ (page 139) `0..1`
    - broyden (page 143) `0..1`

max-time

Defined the end of the simulation as total time.

Example:

```
<max-time value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	The value of the maximum simulation time.	*none*	none

max-time-windows

Defined the end of the simulation as a total count of time windows.

Example:

```
<max-time-windows value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum count of time windows.	*none*	none

time-window-size

Defines the size of the time window.

Example:

```
<time-window-size value="-1" valid-digits="10" method="fixed"/>
```

Attribute	Type	Description	Default	Options
value	float	The maximum time window size.	`-1`	none
valid-digits	integer	Precision to use when checking for end of time windows used this many digits. $\phi = 10^{-validDigits}$	`10`	none
method	string	The method used to determine the time window size. Use `fixed` to fix the time window size for the participants.	`fixed`	`fixed`

participant

Example:

```
<participant name="{string}" control="0"/>
```

Attribute	Type	Description	Default	Options
name	string	Name of the participant.	*none*	none
control	boolean	Does this participant control the coupling?	`0`	none

exchange

Defines the flow of data between meshes of participants.

Example:

```
<exchange data="{string}" from="{string}" mesh="{string}" to="{string}" initialize="0"/>
```

Attribute	Type	Description	Default	Options
data	string	The data to exchange.	*none*	none
from	string	The participant sending the data.	*none*	none
mesh	string	The mesh which uses the data.	*none*	none
to	string	The participant receiving the data.	*none*	none
initialize	boolean	Should this data be initialized during initializeData?	`0`	none

acceleration:constant

Accelerates coupling data with constant underrelaxation.

Example:

```
<acceleration:constant> <relaxation value="{float}"/> </acceleration:constant>
```

Valid Subtags:

- relaxation (page 136) `1`

relaxation

Example:

```
<relaxation value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	Constant relaxation factor.	*none*	none

acceleration:aitken

Accelerates coupling data with dynamic Aitken under-relaxation.

Example:

```
<acceleration:aitken> <initial-relaxation value="{float}"/> <data mesh="{string}" name="{string}"/>
</acceleration:aitken>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- data (page 125) `1..*`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

acceleration:IQN-ILS

Accelerates coupling data with the interface quasi-Newton inverse least-squares method.

Example:

```
<acceleration:IQN-ILS> <initial-relaxation value="{float}" enforce="0"/> <max-used-iterations valu
e="{integer}"/> <time-windows-reused value="{integer}"/> <data scaling="1" mesh="{string}" name="{s
tring}"/> <filter limit="1e-16" type="{string}"/> <preconditioner freeze-after="-1" type="{strin
g}"/> </acceleration:IQN-ILS>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- max-used-iterations (page 128) `1`
- time-windows-reused (page 128) `1`
- data (page 125) `1..*`

- filter (page 128) `0..1`
- preconditioner (page 129) `0..1`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none
enforce	boolean	Enforce initial relaxation in every time window.	`0`	none

max-used-iterations

Maximum number of columns used in low-rank approximation of Jacobian.

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

time-windows-reused

Number of past time windows from which columns are used to approximate Jacobian.

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of time windows.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	`1`	none
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

filter

Type of filtering technique that is used to maintain good conditioning in the least-squares system. Possible filters:

- `QR1-filter` : updateQR-dec with (relative) test $R(i,i) < \epsilon * \|R\|_F$
- `QR1_absolute-filter` : updateQR-dec with (absolute) test $R(i,i) < \epsilon$
- `QR2-filter` : en-block QR-dec with test $\|v_{\text{orth}}\|_2 < \epsilon * \|v\|_2$

Please note that a QR1 is based on Given's rotations whereas QR2 uses modified Gram-Schmidt. This can give different results even when no columns are filtered out.

Example:

```
<filter limit="1e-16" type="{string}"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit eps of the filter.	`1e-16`	none
type	string	Type of the filter.	*none*	`QR1` , `QR1-absolute` , `QR2`

preconditioner

To improve the performance of a parallel or a multi coupling schemes a preconditioner can be applied. A constant preconditioner scales every acceleration data by a constant value, which you can define as an attribute of data. A value preconditioner scales every acceleration data by the norm of the data in the previous time window. A residual preconditioner scales every acceleration data by the current residual. A residual-sum preconditioner scales every acceleration data by the sum of the residuals from the current time window.

Example:

```
<preconditioner freeze-after="-1" type="{string}"/>
```

Attribute	Type	Description	Default	Options
freeze-after	integer	After the given number of time steps, the preconditioner weights are frozen and the preconditioner acts like a constant preconditioner.	`-1`	none

Attribute	Type	Description	Default	Options
type	string	The type of the preconditioner.	*none*	`constant` , `value` , `residual` , `residual-sum`

acceleration:IQN-IMVJ

Accelerates coupling data with the interface quasi-Newton inverse multi-vector Jacobian method.

Example:

```
<acceleration:IQN-IMVJ always-build-jacobian="0"> <initial-relaxation value="{float}" enforce="0"/>
<imvj-restart-mode truncation-threshold="0.0001" chunk-size="8" reused-time-windows-at-restart="8"
type="RS-SVD"/> <max-used-iterations value="{integer}"/> <time-windows-reused value="{integer}"/>
<data scaling="1" mesh="{string}" name="{string}"/> <filter limit="1e-16" type="{string}"/> <precon
ditioner freeze-after="-1" type="{string}"/> </acceleration:IQN-IMVJ>
```

Attribute	Type	Description	Default	Options
always-build-jacobian	boolean	If set to true, the IMVJ will set up the Jacobian matrix in each coupling iteration, which is inefficient. If set to false (or not set) the Jacobian is only build in the last iteration and the updates are computed using (relatively) cheap MATVEC products.	`0`	none

Valid Subtags:

- initial-relaxation (page 124) `1`
- imvj-restart-mode (page 140) `0..1`
- max-used-iterations (page 128) `1`
- time-windows-reused (page 128) `1`
- data (page 125) `1..*`
- filter (page 128) `0..1`
- preconditioner (page 129) `0..1`

initial-relaxation

Initial relaxation factor.

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float	Initial relaxation factor.	*none*	none
enforce	boolean	Enforce initial relaxation in every time window.	`0`	none

imvj-restart-mode

Type of IMVJ restart mode that is used:

- `no-restart` : IMVJ runs in normal mode with explicit representation of Jacobian
- `RS-ZERO` : IMVJ runs in restart mode. After M time steps all Jacobain information is dropped, restart with no information
- `RS-LS` : IMVJ runs in restart mode. After M time steps a IQN-LS like approximation for the initial guess of the Jacobian is computed.
- `RS-SVD` : IMVJ runs in restart mode. After M time steps a truncated SVD of the Jacobian is updated.
- `RS-SLIDE` : IMVJ runs in sliding window restart mode.

Example:

```
<imvj-restart-mode truncation-threshold="0.0001" chunk-size="8" reused-time-windows-at-restart="8"
type="RS-SVD"/>
```

Attribute	Type	Description	Default	Options
truncation-threshold	float	If IMVJ restart-mode=RS-SVD, the truncation threshold for the updated SVD can be set.	`0.0001`	none
chunk-size	integer	Specifies the number of time steps M after which the IMVJ restarts, if run in restart-mode. Defaul value is M=8.	`8`	none
reused-time-windows-at-restart	integer	If IMVJ restart-mode=RS-LS, the number of reused time steps at restart can be specified.	`8`	none
type	string	Type of the restart mode.	`RS-SVD`	`no-restart` , `RS-0` , `RS-LS` , `RS-SVD` , `RS-SLIDE`

max-used-iterations

Maximum number of columns used in low-rank approximation of Jacobian.

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

time-windows-reused

Number of past time windows from which columns are used to approximate Jacobian.

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The number of columns.	*none*	none

data

The data used to compute the acceleration.

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	`1`	none
mesh	string	The name of the mesh which holds the data.	*none*	none
name	string	The name of the data.	*none*	none

filter

Type of filtering technique that is used to maintain good conditioning in the least-squares system. Possible filters:

- `QR1-filter`: updateQR-dec with (relative) test $R(i,i) < \epsilon * \|R\|_F$
- `QR1_absolute-filter`: updateQR-dec with (absolute) test $R(i,i) < \epsilon$
- `QR2-filter`: en-block QR-dec with test $\|v_{\text{orth}}\|_2 < \epsilon * \|v\|_2$

Please note that a QR1 is based on Given's rotations whereas QR2 uses modified Gram-Schmidt. This can give different results even when no columns are filtered out.

Example:

```
<filter limit="1e-16" type="{string}"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit eps of the filter.	`1e-16`	none
type	string	Type of the filter.	*none*	`QR1`, `QR1-absolute`, `QR2`

preconditioner

To improve the performance of a parallel or a multi coupling schemes a preconditioner can be applied. A constant preconditioner scales every acceleration data by a constant value, which you can define as an attribute of data.

- A value preconditioner scales every acceleration data by the norm of the data in the previous time window.
- A residual preconditioner scales every acceleration data by the current residual.
- A residual-sum preconditioner scales every acceleration data by the sum of the residuals from the current time window.

Example:

```
<preconditioner freeze-after="-1" type="{string}"/>
```

Attribute	Type	Description	Default	Options
freeze-after	integer	After the given number of time steps, the preconditioner weights are frozen and the preconditioner acts like a constant preconditioner.	`-1`	none
type	string	Type of the preconditioner.	*none*	`constant` , `value` , `residual` , `residual-sum`

acceleration:broyden

Accelerates coupling data with the (single-vector) Broyden method.

Example:

```
<acceleration:broyden> <initial-relaxation value="{float}" enforce="0"/> <max-used-iterations value="{integer}"/> <time-windows-reused value="{integer}"/> <data scaling="1" mesh="{string}" name="{string}"/> </acceleration:broyden>
```

Valid Subtags:

- initial-relaxation (page 124) `1`
- max-used-iterations (page 128) `1`
- time-windows-reused (page 128) `1`
- data (page 125) `1..*`

initial-relaxation

Example:

```
<initial-relaxation value="{float}" enforce="0"/>
```

Attribute	Type	Description	Default	Options
value	float		*none*	none
enforce	boolean		`0`	none

max-used-iterations

Example:

```
<max-used-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer		*none*	none

time-windows-reused

Example:

```
<time-windows-reused value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer		*none*	none

data

Example:

```
<data scaling="1" mesh="{string}" name="{string}"/>
```

Attribute	Type	Description	Default	Options
scaling	float	To improve the performance of a parallel or a multi coupling schemes, data values can be manually scaled. We recommend, however, to use an automatic scaling via a preconditioner.	1	none
mesh	string		*none*	none
name	string		*none*	none

absolute-convergence-measure

Absolute convergence criterion based on the two-norm difference of data values between iterations.

$$\left\| H(x^k) - x^k \right\|_2 < \text{limit}$$

Example:

```
<absolute-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffice
s="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in ((0, 1]).	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

relative-convergence-measure

Relative convergence criterion based on the relative two-norm difference of data values between iterations.

$$\frac{\left\|H(x^k) - x^k\right\|_2}{\left\|H(x^k)\right\|_2} < \text{limit}$$

Example:

```
<relative-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffice
s="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in $(0, 1]$.	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

residual-relative-convergence-measure

Residual relative convergence criterion based on the relative two-norm differences of data values between iterations.

$$\frac{\left\|H(x^k) - x^k\right\|_2}{\left\|H(x^{k-1}) - x^{k-1}\right\|_2} < \text{limit}$$

Example:

```
<residual-relative-convergence-measure limit="{float}" data="{string}" mesh="{string}" strict="0" suffices="0"/>
```

Attribute	Type	Description	Default	Options
limit	float	Limit under which the measure is considered to have converged. Must be in ((0, 1]).	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

min-iteration-convergence-measure

Convergence criterion used to ensure a miminimal amount of iterations. Specifying a mesh and data is required for technical reasons and does not influence the measure.

Example:

```
<min-iteration-convergence-measure min-iterations="{integer}" data="{string}" mesh="{string}" strict="0" suffices="0"/>
```

Attribute	Type	Description	Default	Options
min-iterations	integer	The minimal amount of iterations.	*none*	none
data	string	Data to be measured.	*none*	none
mesh	string	Mesh holding the data.	*none*	none
strict	boolean	If true, non-convergence of this measure ends the simulation. “strict” overrules “suffices”.	`0`	none
suffices	boolean	If true, convergence of this measure is sufficient for overall convergence.	`0`	none

max-iterations

Allows to specify a maximum amount of iterations per time window.

Example:

```
<max-iterations value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The maximum value of iterations.	*none*	none

extrapolation-order

Sets order of predictor of interface values for first participant.

Example:

```
<extrapolation-order value="{integer}"/>
```

Attribute	Type	Description	Default	Options
value	integer	The extrapolation order to use.	*none*	none

Tools for preCICE

Creating your own simulation or doing a rigorous performance study of a method? There are probably a few common tasks that could use some automation for.

Here you will find a few tools to:

- Visualize the preCICE configuration file (page 162) to understand if you are really asking preCICE to do what you meant to.
- Analyze the performance of the coupled simulation (page 165) to understand where the runtime comes from.
- Compute parameters for the RBF mapping configuration (page 167) to optimize the accuracy and performance of your RBF mapping.

Config visualization

Summary: Understanding, handling and debugging preCICE configuration files can be difficult and tedious. This tool simplifies this process by visualizing the configuration as a dot graph.

Motivation

Understanding, handling and debugging preCICE configuration files can be difficult and tedious. As so many problems, also this problem grows superlinear with the size of the input. Especially the configuration of the data-flow can be tricky to get right for beginners and sometimes even seasoned preCICE users.

This tool is supposed to tackle this issue.

It naively interprets the given configuration file and visualizes it as a graph. This has a few important benefits:

- Configuration mistakes can be difficult to spot in XML, but are often trivial to spot in a graph.
- Students and co-workers have less trouble understanding relations of components in a graph format.
- A graph is a good format to present the simulation scenario in presentations

Installation

Please first install the dependencies:

- `python3` and `pip`
- `graphviz` for rendering the result.

We recommend installing the `config-visualizer` straight from GitHub:

```
pip3 install --user https://github.com/precice/config-visualizer/archive/master.zip
```

In case you want to tinker with the software, you can clone the repository and install the package locally.

```
git clone https://github.com/precice/config-visualizer.git pip3 install --user -e config-visualizer
```

Note: You maybe need to add your user pip installations to your path to make the config visualizer findable, i.e.

```
export PATH=$PATH:$HOME/.local/bin
```

Usage

1. Use `precice-config-visualizer -o config.dot precice-config.xml` to generate the graph in the `.dot` format.
2. Use `dot -Tpdf -ofile config.pdf config.dot` to layout the result and output a given format such as pdf. This program is part of graphviz.

These commands support piping, so you can also execute:

```
cat precice-config.xml | precice-config-visualizer | dot -Tpdf > config.pdf
```

Controlling the output

For big cases, the generated output can be visually too busy. This is why the tool allows you to control the verbosity of some elements. For some properties, the following options are available:

- **full** shows the available information in full detail. This is the default.
- **merged** shows available relations between components without full detail. Multiple edges between components will be merged into a single one.
- **hide** hided all relations.

These options are currently available for:

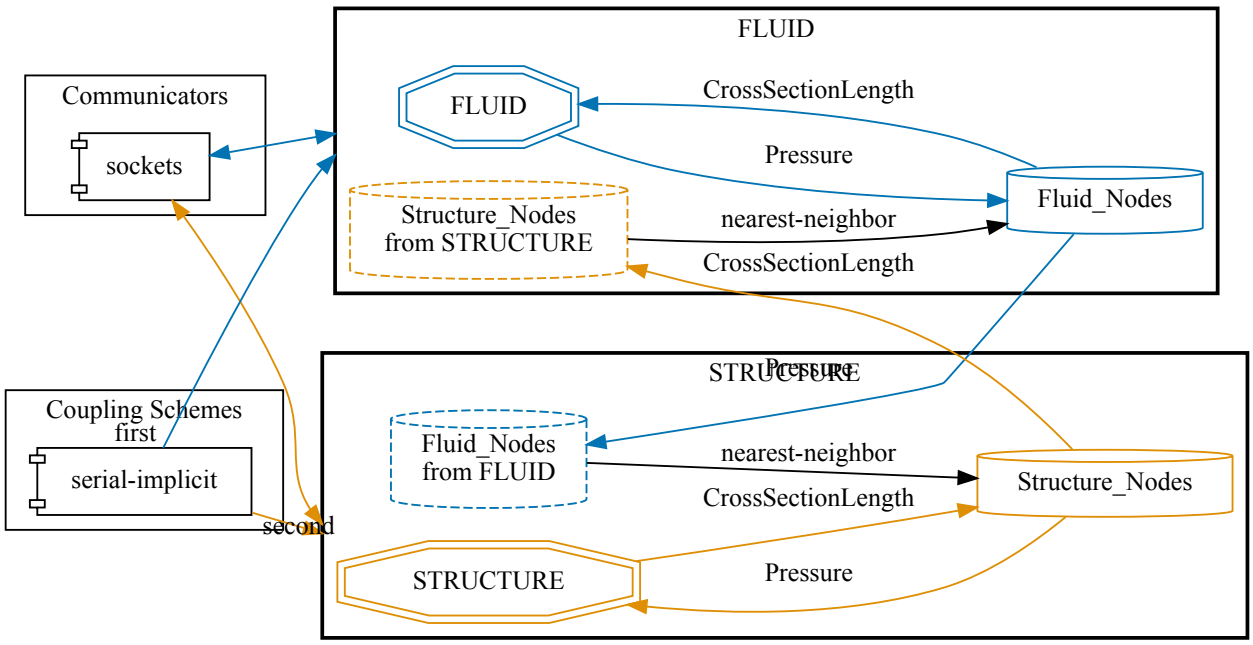
- **data access** participants using `read-data` and `write-data` to access data on meshes.
- **data exchange** participants `exchange` ing data between meshes.
- **communicators** configured `m2n` connections between participants.
- **coupling schemes** configured `cplscheme` s between participants.

Examples

These examples are based on the elastictube1d example.

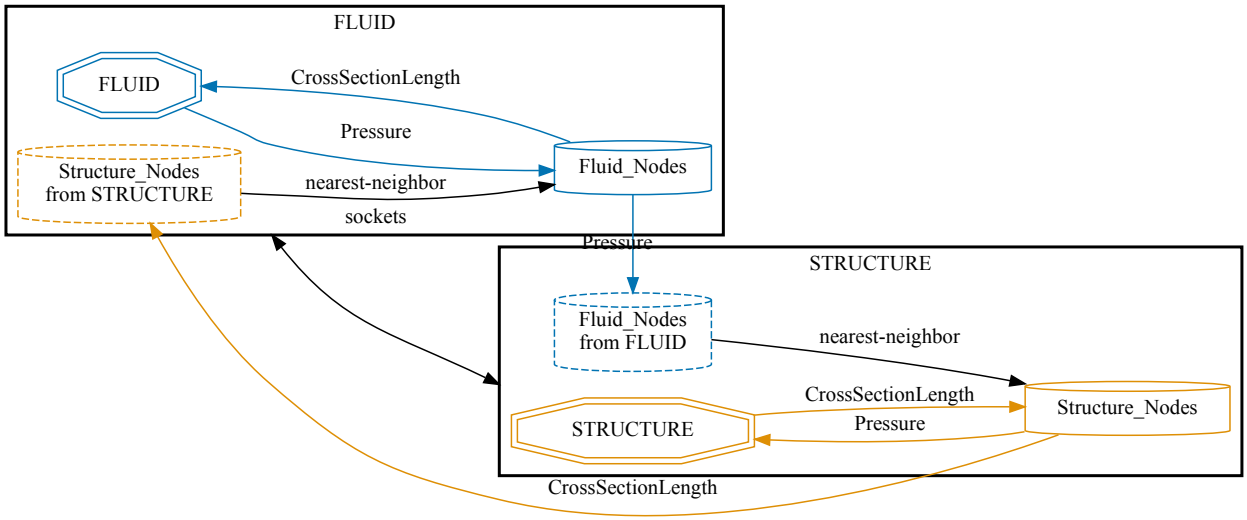
The full picture

```
precice-config-visualizer --communicators=merged --cplschemes=merged precice-config.xml | dot -Tpdf
> graph.pdf
```



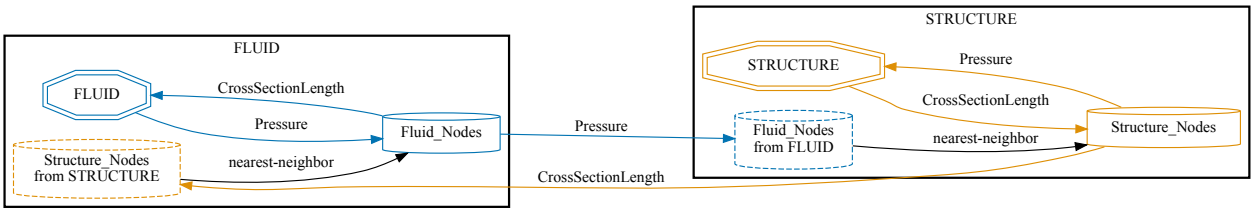
Reduced information of coupling schemes and communicators

```
precice-config-visualizer --communicators=merged --cplschemes=merged precice-config.xml | dot -Tpdf
> graph.pdf
```



Data flow visualization

```
precice-config-visualizer --communicators=hide --cplschemes=hide precice-config.xml | dot -Tpdf > g
raph.pdf
```



Performance analysis

Summary: A guide to the main reference literature for each component and feature of preCICE

Working with events

preCICE uses the EventTimings library to profile major logical blocks of work. The library generates files during the finalization step of each participant and writes them to their current working directories.

For a participant called `MySolver`, the files are called as follows:

- `precice-MySolver-events.json`
- `precice-MySolver-events-summary.log`

The events summary file

The events summary file contains a table of events, their occurences and some statistics on their runtime. This can be helpful to quickly identify where the preCICE library spends most of its time.

It is especially helpful to focus on noteworthy events (page 166).

This is an example output:

```
Run finished at Wed Aug 1 09:41:10 2018 Global runtime = 12859ms / 12s Number of processors = 4 # R
ank: 0 Event | Count | Total[ms] | Max[ms] | Min[ms] | Avg[ms] | T[%] |
------------------------------------------------------------------------------------------ _GLOBAL | 1
| 12859 | 12859 | 12859 | 12859 | 99 | advance | 1 | 84 | 84 | 84 | 84 | 0 | Name | Max | MaxOnRank
| Min | MinOnRank | Min/Max |
------------------------------------------------------------------------- _GLOBAL | 12859 | 0 | 12
859 | 0 | 1 | advance | 119 | 2 | 83 | 1 | 0 |
```

`T[%]` prints the relative runtime. Note that this can be more than 100% summed up, since events can be nested, like in the example above.

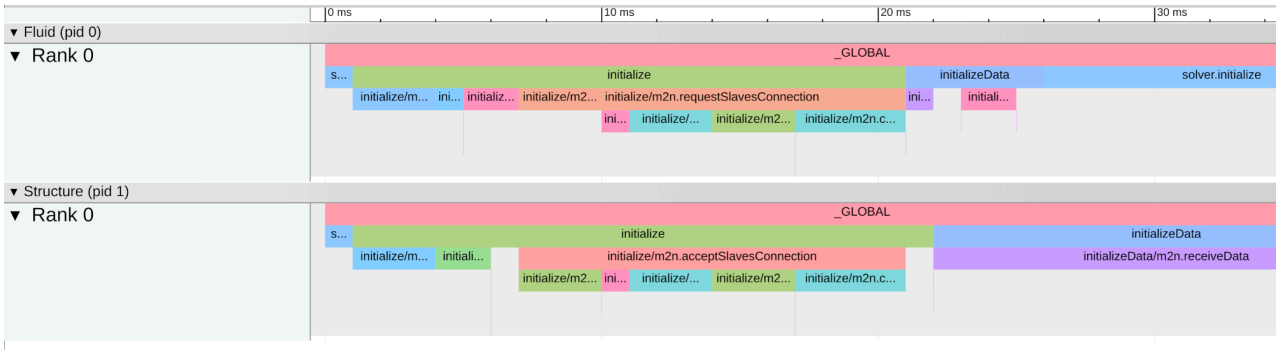
The events JSON file

The events JSON file contains the full picture of events and attached data.

You can use the events2trace tool to convert events to the trace format. The tool allows to merge the events output of multiple participants into a single output in the trace format. This trace format can then be visualized using the following tools:

- speedscope.app
- ui.perfetto.dev
- `chrome://tracing/` (page 0) in Chromium browsers *(see full list)*

An example trace visualization using `chrome://tracing/` of the elastictube1d example looks as following:



You can also evaluate the data in the events files yourself. Please pay special attention to the timestamps as they are based on the system clock. Larger clusters are often subject to clock-drift, so you need to shift and normalize the time-scales yourself.

Noteworthy events

Noteworthy events are

1. The communication buildup, which can become very expensive on clusters. This is generally bound to the filesystem. This is not a problem for The main latency here is the distributed file-system.

RBF shape calculator

Selecting an appropriate shape parameter for radial basis function mappings can be a bit tricky.

To simplify this task, you can find the script rbgShape.py in the preCICE repository.

Given the width of the mesh and the amount of vertices to cover by the support radius, this script calculates an appropriate shape parameter for Gaussian basis-functions. The script also allows to specify a custom decay.

This solves the following equation:

$$\mathbf{shape} = \frac{\sqrt{-log(\mathbf{decay})}}{\mathbf{vertices} \cdot \mathbf{meshwidth}}$$

Overview of adapters

There are various codes - free and proprietary ones - currently coupled with preCICE. If you want to add your code here, please let us know.

Official adapters

We host adapters for the following codes in the preCICE GitHub organization and we maintain them to work with the latest release of preCICE (unless stated otherwise).

Adapter for	Resources	Typical applications	Comments
OpenFOAM	code, docs (page 170)	Fluid part in CHT, FSI, FF	
deal.II	code, docs (page 183)	Structure part in FSI, any FEM	
FEniCS	code, docs (page 213)	Structure part in CHT, FSI, any FEM	
Nutils	docs (page 221)	Structure part in CHT, any FEM	
CalculiX	code, docs (page 197)	Structure part in CHT, FSI	
SU2	code, docs (page 209)	Fluid part in FSI	Maintainer needed
code_aster	code, docs (page 215)	Structure part in CHT	
Ansys Fluent	code, docs	Fluid part in FSI	Experimental
COMSOL Multiphysics	code	Structure part in FSI	Currently not maintained

Third-party adapters

The preCICE community has successfully coupled the following codes with preCICE for community projects (page 0). Wherever meaningful (license, maturity of the project, no other home), we host the code repository.

Adapter for	Contact	Resources	Typical applications
LS-DYNA	LKR	code example	Continuous metal casting process
MBDyn	TU Delft Wind Energy	code	Structure part in FSI
MBDyn	Politecnico di Milano DAER	documentation, code	Structure part in FSI

Alya	TUM SCCS	Fluid and structure part in FSI	Not actively maintained (but not abandoned)
Ateles (APES)	Univ. Siegen STS	code	Fluid-Acousting, Fluid-Fluid coupling
FASTEST	TU Darmstadt FNB	None	Fluid-Structure-Acoustics interaction
FEAP	TU Darmstadt FNB	None	Structure part in FSI
Palabos	University of Stuttgart	Code	Fluid-Structure interaction (Experimental)
DUNE	Max Firmbach, UniBW M	Thesis, Code tbc.	Structure part in FSI

Legacy adapters

These adapters and/or the respective solvers are not maintained and might not work anymore, but are listed here as an example of which other projects have used preCICE in the past.

Adapter for	Contact	Resources	Typical applications
Carat++	TUM Statik	None	Structure part of FSI
EFD	TUM SCCS	code	Fluid part of FSI
foam-extend	TU Delft Aerodynamics	code	Fluid and structure part of FSI, Fluid-Fluid coupling
Peano	Durham University	None	Fluid part of FSI

The OpenFOAM adapter

Summary: An OpenFOAM function object for CHT, FSI, and fluid-fluid coupled simulations using preCICE.

What is this?

This preCICE adapter is a plug-in (function object) for OpenFOAM, which can work with any recent version of OpenFOAM (.com / .org, see supported OpenFOAM versions). It supports fluid-structure interaction (fluid part), conjugate heat transfer (fluid and solid parts), and fluid-fluid simulations, while it is also easily extensible.

What can it do?

This adapter can read/write the following fields:

- Temperature (read + write)
- Heat flux (read + write)
- Sink temperature (read + write)
- Heat transfer coefficient (read + write)
- Force (write)
- Stress (write)
- Displacement (read)
- Displacement delta (read)
- Pressure (read + write)
- Pressure gradient (read + write)
- Velocity (read + write)
- Velocity gradient (read + write)

All features of preCICE are supported, including implicit coupling and nearest-projection mapping. Even though OpenFOAM is 3D, this adapter can also work in the 2D mode of preCICE, defining only one layer of interface nodes (automatically).

Try

Here you will find how to get the adapter, how to configure a case, how to extend the adapter to cover additional features, as well as a few notes on supported OpenFOAM versions.

Learn

Apart from following the documentation here, you will also often find us in OpenFOAM-related conferences. Before diving into preCICE and the OpenFOAM adapter for the first time, you may want to watch the recording of our training session from the 15th OpenFOAM Workshop:

Cite

We are currently working on an up-to-date reference paper. Until then, please cite this adapter using [1]:

```
Gerasimos Chourdakis. A general OpenFOAM adapter for the coupling library preCICE. Master's thesis,
Department of Informatics, Technical University of Munich, 2017.
```

For CHT-specific topics, you may want to additionally look into [2] and for FSI into [3].

Related literature

[1] Gerasimos Chourdakis. A general OpenFOAM adapter for the coupling library preCICE. Master's thesis, Department of Informatics, Technical University of Munich, 2017.

[2] Lucia Cheung Yau. Conjugate heat transfer with the multiphysics coupling library preCICE. Master's thesis, Department of Informatics, Technical University of Munich, 2016.

[3] Derek Risseeuw. Fluid Structure Interaction Modelling of Flapping Wings. Master's thesis, Faculty of Aerospace Engineering, Delft University of Technology, 2019.

Get the OpenFOAM adapter

Summary: Get the code from GitHub and run ./Allwmake. If this fails, look into wmake.log and ldd.log.

To build the adapter, you need to install a few dependencies and then execute the `Allwmake` script.

1. Install a compatible OpenFOAM distribution.
2. Install preCICE.
3. Download the latest release for your OpenFOAM version.
4. Execute the build script: `./Allwmake`.
    - See and adjust the configuration in the beginning of the script first, if needed.
    - Check for any error messages and suggestions at the end.
    - Modify the `adapter_build_command` to e.g. build using more threads, e.g. `wmake -j 4 libso`.

Adding `-DADAPTER_DEBUG_MODE` flag to the `ADAPTER_PREP_FLAGS` activates additional debug messages. You may also change the target directory or specify the number of threads to use for the compilation. See the comments in `Allwmake` for more.

Next: configure and load the adapter or run a tutorial.

Troubleshooting

The following are common problems that may appear during building the OpenFOAM adapter if something went wrong in the described steps. Make sure to always check for error messages at every step before continuing to the next.

The `Allwmake` script prints the environment variables it uses in the beginning (as well as in `Allwmake.log`) and it writes the building commands in the file `wmake.log`. Afterwards, it checks (using `ldd`) if the library was linked correctly and writes the output to `ldd.log`. **Please check these files and include them in your report if you have need help.**

Unknown function type `preciceAdapterFunctionObject`

- Did building & linking the adapter succeed? Any errors in wmake.log or ldd.log? Details: (click)

If in the beginning of the simulation you get the following warning:

```
Starting time loop --> FOAM Warning : From function void* Foam::dlOpen(const Foam::fileName&, bool)
in file POSIX.C at line 1604 dlopen error : libprecice.so: cannot open shared object file: No such
file or directory --> FOAM Warning : From function bool Foam::dlLibraryTable::open(const Foam::file
Name&, bool) in file db/dynamicLibrary/dlLibraryTable/dlLibraryTable.C at line 105 **could not load
"libpreciceAdapterFunctionObject.so"** --> FOAM Warning : From function bool Foam::dlLibraryTabl
e::open(const Foam::dictionary&, const Foam::word&, const TablePtr&) [with TablePtr = Foam::HashTab
le<Foam::autoPtr<Foam::functionObject> (*)(const Foam::word&, const Foam::Time&, const Foam::dictio
nary&), Foam::word, Foam: :string::hash>*] in file lnInclude/dlLibraryTableTemplates.C at line 62 C
ould not open library "libpreciceAdapterFunctionObject.so" --> FOAM Warning : Unknown function type
preciceAdapterFunctionObject
```

then this probably means that something went wrong while building the OpenFOAM adapter. Check the files `wmake.log` (for building errors) and `ldd.log` (for runtime linking errors). Make sure that, when you run the simulation, you have the same OpenFOAM and any other required environment variables as when you built the adapter.

If everything during building has gone well, the adapter must be installed into your `$FOAM_USER_LIBBIN` directory. Check that it exists (`ls $FOAM_USER_LIBBIN`) and that `ldd $FOAM_USER_LIBBIN/libpreciceAdapterFunctionObject.so` does not return any errors.

Note that the simulation will continue without loading the adapter and there will be no coupling.

wmkdep: could not open file X

This is an info/warning message that is printed when WMake tries to distinguish between the object files it already has (and can save time by not recompiling them) and the files it needs to compile. You can safely ignore this message.

A header file cannot be found (during compilation)

This is a common problem e.g. when installing dependencies in non-system directories. Have a look in the page linking to preCICE.

Rellocation-related errors

Make sure to build both preCICE as a shared library (i.e. `.so` , not `.a`).

Configure the OpenFOAM adapter

Summary: Write a system/preciceDict, set compatible boundary conditions, and activate the adapter in your system/controlDict.

In order to run a coupled simulation, you need to:

1. prepare a preCICE configuration file (described in the preCICE configuration),
2. prepare an adapter's configuration file,
3. set the coupling boundaries in the OpenFOAM case,
4. load the adapter, and
5. start all the solvers normally, from the same directory, e.g. in two different terminals.

If you prefer, you may find an already prepared case in our Tutorial for CHT: Flow over a heated plate.

You may skip the section *"Advanced configuration"* in the beginning, as it only concerns special cases. You may also find more details in the Pull Request #105, especially for changes regarding the previous, yaml-based configuration format.

The adapter's configuration file

The adapter is configured via the file `system/preciceDict`. This file is an OpenFOAM dictionary with the following form:

```
FoamFile { version 2.0; format ascii; class dictionary; location "system"; object preciceDict; } pr
eciceConfig "precice-config.xml"; participant Fluid; modules (CHT); interfaces { Interface1 { mesh
Fluid-Mesh; patches (interface); locations faceCenters; readData ( Heat-Flux ); writeData ( Tempera
ture ); }; };
```

The `participant` needs to be the same as the one specified in the `preciceConfig`, which is the main preCICE configuration file. The `preciceConfig` can be a path and needs to be wrapped with quotation marks.

The list `modules` can contain `CHT` or/and `FSI` (separated by space).

In the `interfaces`, we specify the coupling interfaces (here only one). The `mesh` needs to be the same as the one specified in the `preciceConfig`. The `patches` specifies a list of the names of the OpenFOAM boundary patches that are participating in the coupled simulation. These need to be defined in the files included in the `0/` directory. The names of the interfaces (e.g. `Interface1`) are arbitrary and are not used.

The `locations` field is optional and its default value is `faceCenters` (with `faceCentres` also accepted), signifying that the interface mesh is defined on the cell face centers. The alternative option is `faceNodes`, which defines the mesh on the face nodes and is needed e.g. for reading displacements in an FSI scenario.

The values for `readData` and `writeData` for conjugate heat transfer can be `Temperature`, `Heat-Flux`, `Sink-Temperature`, or `Heat-Transfer-Coefficient`. Values like `Sink-Temperature-Domain1` are also allowed. For a Dirichlet-Neumann coupling, the `writeData` and `readData` can be either:

```
readData ( Heat-Flux ); writeData ( Temperature );
```

or:

```
readData ( Temperature ); writeData ( Heat-Flux );
```

For a Robin-Robin coupling, we need to write and read both of `Sink-Temperature` and `Heat-Transfer-Coefficient`:

```
readData ( Sink-Temperature // e.g. Sink-Temperature-Solid Heat-Transfer-Coefficient // e.g. Heat-Transfer-Coefficient-Solid ); writeData ( Sink-Temperature // e.g. Sink-Temperature-Fluid Heat-Transfer-Coefficient // e.g. Heat-Transfer-Coefficient-Fluid );
```

For fluid-structure interaction, `writeData` can be `Force` or `Stress`, where `Stress` is essentially a force vector scaled by the cell face in spatial coordinates (with any postfix), thus, a conservative quantity as well. `readData` can be `Displacement` and `DisplacementDelta` (with any postfix). `DisplacementDelta` refers to the last coupling time step, which needs to considered in the case of subcycling.

Warning: You will run into problems when you use `Displacement(Delta)` as write data set and execute RBF mappings in parallel. This would affect users who use OpenFOAM and the adapter as the Solid participant in order to compute solid mechanics with OpenFOAM (currently not officially supported at all). Have a look at this issue on GitHub for details.

Configuration of the OpenFOAM case

A few changes are required in the configuration of an OpenFOAM case, in order to specify the interfaces and load the adapter. For some solvers, additional parameters may be needed (see “advanced configuration”).

Boundary conditions

The type of the `readData` needs to be compatible with the respective boundary conditions set for each field in the `0/` directory of the case.

Read the OpenFOAM User Guide for more on boundary conditions.

CHT

- For `readData(Temperature)`, use `type fixedValue` for the `interface` in `0/T`. OpenFOAM requires that you also give a (redundant) `value`, but the adapter will overwrite it. ParaView uses this value for the initial time. As a placeholder, you can e.g. use the value from the `internalField`.

```
interface { type fixedValue; value $internalField; }
```

- For `readData(Heat-Flux)`, use `type fixedGradient` for the `interface` in `0/T`. OpenFOAM requires that you also give a (redundant) `gradient`, but the adapter will overwrite it.

```
interface { type fixedGradient; gradient 0; }
```

- For `readData(Sink-Temperature)` or `Heat-Transfer-Coefficient`, use `type mixed` for the `interface` in `0/T`. OpenFOAM requires that you also give (redundant) values for `refValue`, `refGradient`, and `valueFraction`, but the adapter will overwrite them.

```
interface { type mixed; refValue uniform 293; valueFraction uniform 0.5; refGradient uniform 0; }
```

FSI

- For `readData(Displacement)` or `DisplacementDelta`, you need the following:
  - `type movingWallVelocity` for the interface (e.g. `flap`) in `0/U`,
  - `type fixedValue` for the interface (e.g. `flap`) in the `0/pointDisplacement`, and

- `solver displacementLaplacian` in the `constant/dynamicMeshDict`.

```
// File 0/U interface { type movingWallVelocity; value uniform (0 0 0); } // File 0/pointDisplaceme
nt interface { type fixedValue; value $internalField; } // File constant/dynamicMeshDict dynamicFvM
esh dynamicMotionSolverFvMesh; motionSolverLibs ("libfvMotionSolvers.so"); solver displacementLapla
cian;
```

Load the adapter

To load this adapter, you must include the following in the `system/controlDict` configuration file of the case:

```
functions { preCICE_Adapter { type preciceAdapterFunctionObject; libs ("libpreciceAdapterFunctionOb
ject.so"); } }
```

This directs the solver to use the `preciceAdapterFunctionObject` function object, which is part of the `libpreciceAdapterFunctionObject.so` shared library. The name `preCICE_Adapter` can be arbitrary.

Advanced configuration

These additional parameters may only concern some users is special cases. Keep reading if you want to use nearest-projection mapping, an incompressible or basic (e.g. laplacianFoam) solver, if you are using a solver with different variable names (e.g. a multiphase solver) or if you are trying to debug a simulation.

Nearest-projection mapping

An example for for nearest-projection mapping is provided in the nearest-projection tutorial case. The preCICE documentation contains a detailed description of nearest-projection mappings in preCICE. In summary, we need to explicitly enable the `connectivity` option to create edges between the interface mesh points and give them to preCICE:

```
interfaces { Interface1 { mesh Fluid-Mesh-Centers; locations faceCenters; connectivity false; patch
es (interface); // ... writeData, readData ... }; Interface2 { mesh Fluid-Mesh-Nodes; locations fac
eNodes; connectivity true; patches (interface); // ... writeData, readData ... }; };
```

This `connectivity` boolean is optional and defaults to `false`. Note that `connectivity true` can only be used with `locations faceNodes`.

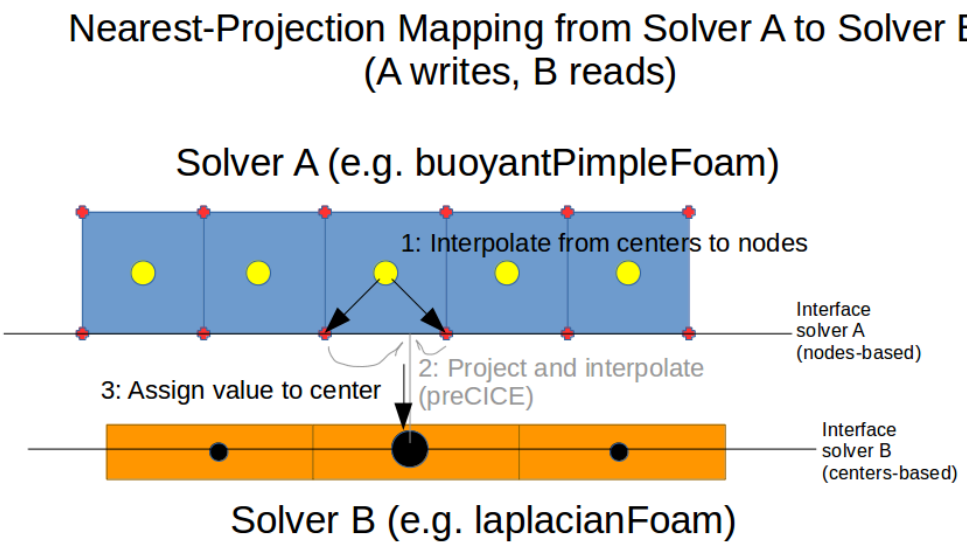
Even if the coupling data is associated to `faceCenters` in the solver, we can select `faceNodes` as locations type: the respective data will be interpolated from faces to nodes. Also, connectivity is only needed and supported for `writeData`. Therefore, we need to split the interface in a “read” and a “write” part, as shown above.

More details about the rationale are given in the following section.

Adapter Implementation

Since OpenFOAM is a finite-volume based solver, data is located in the middle of the cell, or on the cell face centers for a coupling interface. Mesh connectivity can be given to preCICE using the methods `setMeshTriangle` and `setMeshEdge`. Using the face centers as arguments for these methods is cumbersome. The main reason is that, although OpenFOAM decomposes the mesh for parallel simulations and distributes the subdomains to different processes, mesh connectivity needs to be defined over the partitioned mesh boundaries. This problem vanishes if we define mesh connectivity based on the face nodes, since boundary nodes can be shared among processors. Therefore, mesh connectivity can only be provided on the face nodes (not on the face centers).

As described already, the data is not stored on the face nodes, but on the face centers. Therefore, we use OpenFOAM functions to interpolate from face centers to face nodes. The following image illustrates the workflow:



Data is obtained at the face centers, then interpolated to face nodes. Here, we have provided mesh connectivity and finally, preCICE performs the nearest-projection mapping. It is important to notice that the target data location is again the face center mesh of the coupling partner. In the standard CHT case, where both data sets are exchanged by a nearest-projection mapping, this leads to two interface meshes (centers and nodes) per participant. Having both the centers and nodes defined, we can skip one interpolation step and read data directly to the centers (cf. picture solver B).

Note: As already mentioned, the `Fluid` participant does not need to provide the mesh connectivity in case of a standard FSI. Therefore, the `Solid` participant needs to provide it and nothing special needs to be considered compared to other mapping methods. This implementation supports all CHT-related fields, which are mapped with a `consistent` constraint.

Additional properties for some solvers

Some solvers may not read all the material properties that are required for a coupled simulation. These parameters need to be added in the `preciceDict`.

Conjugate heat transfer

For conjugate heat transfer, the adapter assumes that a solver belongs to one of the following categories: *compressible*, *incompressible*, or *basic*. Most of the solvers belong in the *compressible* category and do not need any additional information. The other two need one or two extra parameters, in order to compute the heat flux.

For **incompressible solvers** (like the buoyantBoussinesqPimpleFoam), you need to add the density and the specific heat in a `CHT` subdictionary of `preciceDict`. For example:

```
CHT { rho [ 1 -3 0 0 0 0 0 ] 50; Cp [ 0 2 -2 -1 0 0 0 ] 5; };
```

For **basic solvers** (like the laplacianFoam), you need to add a constant conductivity:

```
CHT { k [ 1 1 -3 -1 0 0 0 ] 100; };
```

The value of `k` is connected to the one of `DT` (set in `constant/transportProperties`) and depends on the density (`rho [ 1 -3 0 0 0 0 0 ]`) and heat capacity (`Cp [ 0 2 -2 -1 0 0 0 ]`). The relation between them is `DT = k / rho / Cp`.

Fluid-structure interaction

The adapter's FSI functionality supports both compressible and incompressible solvers.

For incompressible solvers, it tries to read uniform values for the density and kinematic viscosity (if it is not already available) from the `FSI` subdictionary of `preciceDict`:

```
nu nu [ 0 2 -1 0 0 0 0 ] 1e-03; rho rho [1 -3 0 0 0 0 0] 1;
```

Notice that here, in contrast to the `CHT` subdict, we need to provide both the keyword (first `nu`) and the word name (second `nu`). We are working on bringing consistency on this.

Additional parameters in the adapter's configuration file

Some optional parameters can allow the adapter to work with more solvers, whose type is not determined automatically, their fields have different names, or they do not work well with some features of the adapter.

User-defined solver type

The adapter tries to automatically determine the solver type, based on the dictionaries that the solver uses. However, you may manually specify the solver type to be `basic`, `incompressible` or `compressible` for a CHT or FSI simulation:

```
CHT { solverType incompressible; };
```

This will force the adapter use the boundary condition implementations for the respective type.

Parameters and fields with different names

The names of the parameters and fields that the adapter looks for can be changed, in order to support a wider variety of solvers. You may specify the following parameters in the adapter's configuration file (the values correspond to the default values):

```
CHT { # Temperature field nameT T1; # Thermal conductivity nameKappa k1; # Density nameRho rho1; #
Heat capacity for constant pressure nameCp Cp1; # Prandtl number namePr Pr1; # Turbulent thermal di
ffusivity nameAlphat alphat1; };
```

Debugging

The adapter also recognizes a few more parameters, which are mainly used in debugging or development. These are optional and expect a `true` or a `false` value. Some or all of these options may be removed in the future.

The user can toggle debug messages at build time.

Coupling OpenFOAM with 2D solvers

The adapter asks preCICE for the dimensions of the coupling data defined in the `precice-config.xml` (2D or 3D). It then automatically operates in either 3D (normal) or 2D (reduced) mode, with z-axis being the out-of-plane dimension. Read more.

Porting your older cases to the current configuration format

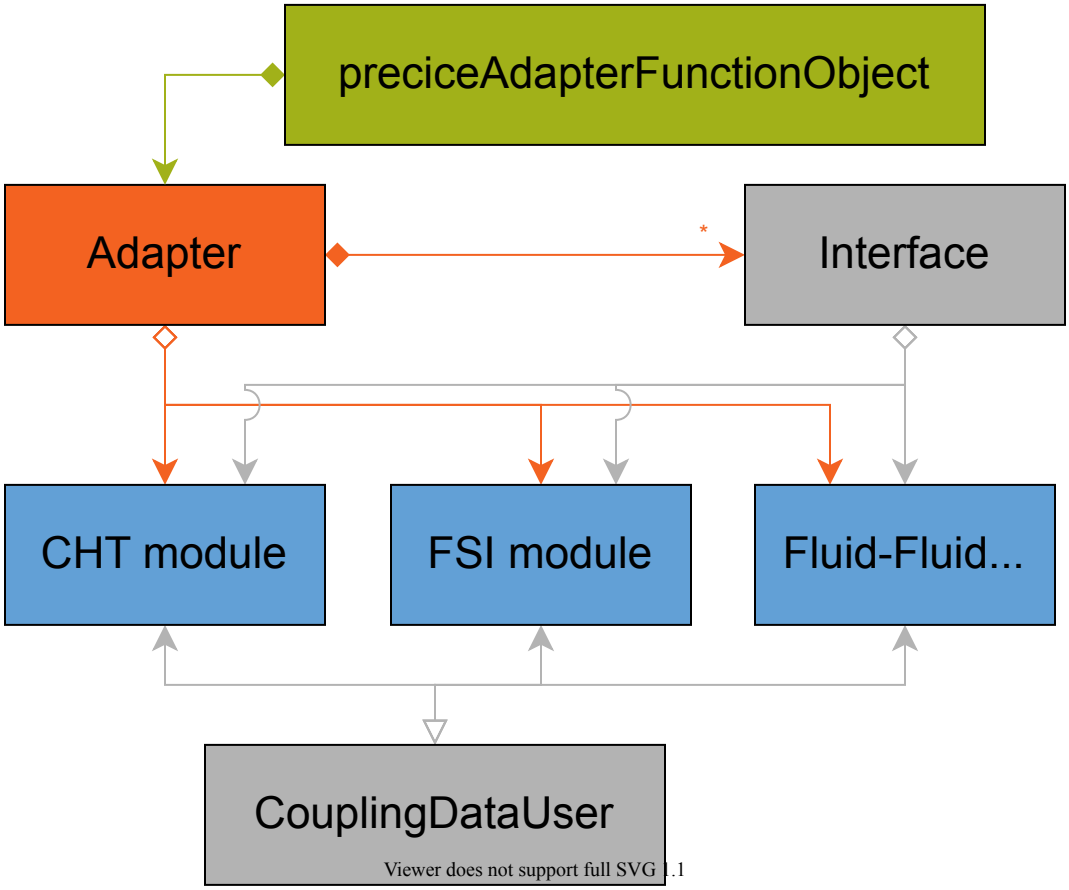
In earlier versions of the adapter, we were using a yaml-based configuration format, with the adapter configuration file usually named as `precice-adapter-config.yml`. We moved to a OpenFOAM dictionary format in #105, to reduce the dependencies. You may also find the tutorials #69 to be a useful reference (file changes).

Extend the OpenFOAM adapter

Summary: An overview of the OpenFOAM adapter's architecture and which parts to modify if you want to add functionality.

Architecture

The OpenFOAM adapter separates the core functionality (e.g. calling preCICE methods) from the problem-specific methods (e.g. accessing fields and converting quantities). The latter is encapsulated into "modules", which add only a few lines of code in the core. The following, simplified UML diagram gives an overview:



While in the begining the adapter only included a module for conjugate heat transfer, a module for fluid-structure interaction and a module for fluid-fluid coupling have been added since then.

Starting points

In case you just want to couple a different variable, you need to create a new coupling data user class in the `preciceAdapter::CHT` namespace or in a new one. Then you need to add an option for it in the configuration part to add objects of it into the `couplingDataWriters` and `couplingDataReaders` whenever requested.

There are some `NOTE`s in the files Adapter.H, Adapter.C, CHT/CHT.C, and CHT/Temperature.H to guide you through the process.

Note: make sure to include any additional required libraries in the `LIB_LIBS` section of the `Make/options`. Since the adapter is a shared library, another missing library will trigger an "undefined symbol" runtime error.

See also the notes and discussion in issue #7: Create a module for fluid-structure interaction.

OpenFOAM support

Summary: Recent OpenFOAM.com versions work out-of-the-box. Recent OpenFOAM.org versions are also supported, but you will need a version-specific branch.

How to get OpenFOAM

The easiest way to start is to get binary packages for your Linux distribution. For example, to get OpenFOAM v2012 on Ubuntu:

```
# Add the signing key, add the repository, update: wget -q -O - https://dl.openfoam.com/add-debian-repo.sh | sudo bash # Install OpenFOAM v2012: sudo apt-get install openfoam2012-dev
```

As these steps change your `.profile`, you need to log out and in again to make OpenFOAM fully discoverable.

Supported OpenFOAM versions

OpenFOAM is a project with long history and many forks, of which we try to support as many as possible.

We provide version-specific branches and archives for:

- OpenCFD / ESI (openfoam.com):
  - OpenFOAM v1812-v2012 or newer (main target)
  - OpenFOAM v1806 or older
- OpenFOAM Foundation (openfoam.org):
  - OpenFOAM 8 (experimental).
  - OpenFOAM 7 (experimental).
  - OpenFOAM 6 (experimental).
  - OpenFOAM 5.x.
  - OpenFOAM 4.0/4.1.

Known not supported versions: OpenFOAM v1606+ or older, OpenFOAM 3 or older, foam-extend (any version).

Supported OpenFOAM solvers

We support mainstream OpenFOAM solvers such as pimpleFoam (for FSI) or buoyantPimpleFoam, buoyantSimpleFoam, laplacianFoam (for CHT). Our community has tried the adapter with multiple different solvers that support function objects.

Notes on OpenFOAM features

End of the simulation

The adapter (by default) ignores the `endTime` set in the `controlDict` and stops the simulation when preCICE says so.

Let's see this with more details. During the simulation, both the solver and preCICE try to control when the simulation should end. While in an explicit coupling scenario this is clearly defined, in an implicit coupling scenario the solver may schedule its exit (and therefore the last call to the adapter) before the coupling is complete. See how function objects are called for more details on this.

In order to prevent early exits from the solver, the solver's `endTime` is set to infinity and it is later set to the current time when the simulation needs to end. This has the side effect of not calling any function object's `end()` method normally, so these are triggered explicitly at the end of the simulation.

Function Objects

In principle, using other function objects alongside the preCICE adapter is possible. They should be defined *before* the adapter in the `system/controlDict`, as (by default and opt-out) the adapter controls when the simulation should end and explicitly triggers (only) the `end()` methods of any other function objects at the end of the simulation. If the `end()` of a function object depends on its `execute()`, then the latter should have been called before the preCICE adapter's `execute()`.

If you want to test this behavior, you may also include e.g. the `systemCall` function object in your `system/controlDict`:

```
functions { systemCall1 { type systemCall; libs ("libutilityFunctionObjects.so"); executeCalls ( "echo \*\*\* systemCall execute \*\*\*" ); writeCalls ( "echo \*\*\* systemCall write \*\*\*" ); endCalls ( "echo \*\*\* systemCall end \*\*\*" ); } preCICE_Adapter { type preciceAdapterFunctionObject; libs ("libpreciceAdapterFunctionObject.so"); } }
```

Writing results

As soon as OpenFOAM writes the results, it will not try to write again if the time takes the same value again. Therefore, during an implicit coupling, we write again when the coupling timestep is complete. See also a relevant issue.

Adjustable timestep and modifiable runTime

In the `system/controlDict`, you may optionally specify the following:

```
adjustTimeStep yes; maxCo 0.5; runTimeModifiable yes;
```

The adapter works both with fixed and adjustable timestep and it supports the `runTimeModifiable` feature. However, if you set a *fixed timestep* and *runTimeModifiable*, changing the configured timestep *during the simulation* will not affect the timestep used. A warning will be shown in this case.

The deal.II adapter

Summary: Coupled structural solver written with the C++ finite element library deal.II

What is deal.II?

From their documentation: *deal.II is a C++ program library targeted at the computational solution of partial differential equations using adaptive finite elements. It uses state-of-the-art programming techniques to offer you a modern interface to the complex data structures and algorithms required.* A more extensive answer can be found on the deal.II webpage.

Aim of this adapter

This adapter has two use cases: On the one hand, it provides coupled structural solvers, which could be used for FSI simulations steered by preCICE. On the other hand, it serves as an example of how to couple your own deal.II project with other solvers using preCICE. Have a look in the build your own adapter (page 190) section for more details.

Tip: In addition to our coupled solid mechanics related codes of the dealii-adapter repository, we contributed a minimal deal.II-preCICE example to the deal.II project. If you want to couple your own deal.II-code, or want to gain insight in the preCICE coupling with deal.II this tutorial is probably the best place to start.

How to install the adapter?

The adapter requires deal.II version 9.2 or greater and preCICE version 2.0 or greater. The building can be done using CMake, as usual. A detailed installation guide can be found on the deal.II adapter building page (page 184).

How to use the coupled codes?

The coupled codes cover the solid part of partitioned FSI simulations. If you want to use them for your own partitioned case, you can read up in the configuration section (page 187) how to change parameters and use different meshes.

How can I use my own solver with the adapter?

The provided deal.II adapter is (as opposed to other adapter) not applicable for any arbitrary solver or project. Nevertheless, the required infrastructure and code to couple a solver different than the given solid solver is similar. You can find a detailed description of the relevant functionality and how to use it for your own solver in the own project section (page 190).

How general are the already coupled codes?

preCICE provides a large variety of various functionalities. The coupled codes do not yet cover everything. You can find further information about recent limitations of the codes in the limitation section (page 192).

What is the theory behind the coupled solver?

deal.II is a finite element library where user can implement whatever they want. If you are interested in theoretic details of the coupled solid solver, you can find the relevant information in the solver details section (page 193).

Get the deal.II adapter

Summary: Use CMake to install deal.II and build the individual programs.

This adapter is a collection of examples of a deal.II solver adapted for preCICE. To build the adapter, we first need to get the deal.II and preCICE header files and libraries. Afterwards, we can build the adapter using CMake and we can run a tutorial.

Get deal.II

Building the adapter requires deal.II version 9.2 or greater. You can find all available download options on the deal.II website.

Binary packages

deal.II is available in several Linux distributions. For example, if you are using Ubuntu, you can get the `libdeal.ii-dev` package (see also the backports ppa):

```
sudo apt install libdeal.ii-dev libdeal.ii-doc cmake make g++
```

Note: The package libdeal.ii-doc installs the deal.II own tutorials ('steps'), which are not necessarily required for the dealii-adapter. However, they can be helpful in order to test the correct installation of the deal.II library. The following steps copy and test the `step-1` tutorial of deal.II:

```
cp -r /usr/share/doc/libdeal.ii-doc/examples/step-1 . cd step-1 cmake . make run
```

Building from source

Get the latest release from the deal.II repository and build using CMake:

```
git clone https://github.com/dealii/dealii.git mkdir build cd build/ cmake \ -D DEAL_II_WITH_UMFPAC
K="ON" \ -D DEAL_II_WITH_THREADS="ON" \ -D DEAL_II_COMPONENT_EXAMPLES="OFF" \ ../dealii make -j 4
```

The direct solvers in this examples require `UMFPACK`. The nonlinear-solver utilizes a shared-memory parallelization. We disable building the examples only to significantly reduce the building time and storage needs.

Advanced: Building in production

If you want to use deal.II in production, there may be several options you may want to tune. In this case, use ccmake or check the deal.II CMake documentation. For example:

```
cmake \ -D CMAKE_BUILD_TYPE="DebugRelease" \ -D CMAKE_CXX_FLAGS="-march=native \ -D DEAL_II_CXX_FLA
GS_RELEASE="-O3" \ -D DEAL_II_WITH_UMFPACK="ON" \ -D DEAL_II_WITH_THREADS="ON" \ -D DEAL_II_COMPONE
NT_EXAMPLES="OFF" \ -D CMAKE_INSTALL_PREFIX=/path/install/dir \ ../dealii make -j 4
```

Detailed installation instructions are given in the installation section of the deal.II webpage.

Get preCICE

Have a look at our preCICE installation guide (page 14).

Build the adapter

If you have deal.II and preCICE globally installed in your system and want to run a tutorial, building the adapter is as simple as `cmake . && make`:

1. Clone the repository and naviagte to the top-level directory

   ```
   git clone https://github.com/precice/dealii-adapter.git && cd dealii-adapter
   ```

2. The solvers are compiled into a single executable. Configuration is carried out using `cmake`:

   - If you have deal.II and preCICE installed globally on your system:

   ```
   cmake .
   ```

   - If you have deal.II and preCICE installed in a local directory:

   ```
   cmake -DDEAL_II_DIR=/path/to/deal.II -DpreCICE_DIR=/path/to/precice .
   ```

   where `*_DIR` points to your installation (not source) directory. This should be the same as the `CMAKE_INSTALL_PREFIX` you used when installing the respective libraay. If you have set either of these variables globally, you could skip it in the command above.

3. Run `make` to build the adapter. This will generate the `elasticity` executable.

4. Ensure that the executable is run-time discoverable by adding it to your `PATH` variable, e.g. for bash

   ```
   export PATH="/path/to/dealii/adapter/elasticity:${PATH}"
   ```

☑ **Tip:** Our tutorials (page 0) include scripts (`run.sh`) in order to start individual cases. The deal.II adapter scripts accept an option `-e=<executable_to_run>` to locate the executable, in case it is not globally discoverable.

2D vs 3D simulations

By default, the adapter is built as a 2D example in release mode. If you want to run a 3D example (quasi 2D, meaning that the out-of-plane direction is clamped but we use real cells for the calculation), you can define this when configuring:

```
cmake -DDIM=3 .
```

Note that you need to run `make distclean` if you switch from one to another dimension in order to overwrite the dimension value.

Debug vs Release mode

You can switch between debug and release mode using `make debug` or `make release`. By default, programs are built in release mode.

Next steps

To run the deal.II codes, copy the parameter file (`parameters.prm`) into your target directory, e.g. `solid-dealii/`. Afterwards, run the executable as

```
./elasticity path/to/parameters.prm
```

Example cases can be found in our FSI tutorial cases (page 0).

Note: The deal.II related examples have already a preconfigured parameter file, so that the parameter file doesn't need to be copied.

Configure the deal.II codes

Summary: Define your geometry in the individual source code file and case specific parameters (e.g. coupling parameters) in the respective parameter file (*.prm)

If you like to setup your own FSI simulation using the provided dealii-adapter, this section should help you to configure the source code and the parameter file.

In order to change your geometry and set appropriate boundary conditions, you need to modify the source file. The parameter file (e.g. `parameters.prm`) is used to set certain properties: material properties, numerical properties or preCICE-related properties.

☑ **Tip:** The linear elastic solver is designed for single core and single threaded computations. The non-linear solver supports shared memory parallelism. If that is still not enough for your case, there is also an unofficial non-linear elastic solid solver for massively parallel systems.

☑ **Tip:** The number of allocated threads in case of shared-memory parallel computations can be specified via the environment variable `DEAL_II_NUM_THREADS`. By default, all available cores on the respective machine are utilized.

Parameter file

This section gives additional information about the parameter files. Here is an example:

```
subsection Time # End time set End time = 10 # Time step size set Time step size = 0.05 # Write res
ults every x timesteps set Output interval = 10 # Output folder set Output folder = dealii-output e
nd
```

☑ **Tip:** A reference parameter file including all important options can be found in the adapter repository.

The first subsection deals with specifications for time-related settings. The output interval specifies when simulation results are written to an output file. In this example, the program will store the results every 10 time steps. Using a time step size of 0.05 seconds, a result file is written every 0.5 seconds.

```
subsection Discretization # Polynomial degree of the FE system set Polynomial degree = 3 # Time int
egration scheme # 0 = forward, 1 = backward set theta = 0.5 # Newmark beta beta = 0.25 # Newmark ga
mma gamma = 0.5 end
```

This subsection configures the numerical discretization: The polynomial degree is associated to the degree of the applied shape functions. Theta is related to the time integration scheme of the linear solver, which is a one-step-theta method. Accordingly, its value can be chosen between 0 and 1, where 0 denotes an explicit forward Euler method and 1 denotes an implicit backward Euler method with each having first order accuracy. It is recommended to use theta to 0.5, which results in a second order accurate and energy-conserving Crank-Nicolson scheme. If you prefer dissipative behavior, you need to choose theta greater than 0.5. Have a look in the Solver details (page 193) for more information. The non-linear solver uses, however, an implicit Newmark scheme, which allows a configuration using the paramters beta and gamma.

```
subsection System properties # Poisson's ratio set Poisson's ratio = 0.4 # Shear modulus set mu =
0.5e6 # density set density = 1000 # Body forces x,y,z set body forces = 0.0,0.0,0.0 end
```

This section defines the material properties and allows the definition of body forces. Possion's ratio and lambda define the material properties. For an overview of all available parameters and conversion formulas have a look at the conversion table at the bottom of the elastic moduli wikipedia article: Body forces are usually gravitational forces and defined direction-wise (x,y,z).

```
subsection Solver # Structural model to be used: linear or neo-Hookean set Model = linear # Linear solver: CG or Direct set Solver type = Direct # Max CG solver iterations (multiples of the system matrix size) # In 2D, this value is best set at 2. In 3D, a value of 1 works fine. set Max iteration multiplier = 1 # Absolute CG solver residual (multiplied by residual norm, ignored if Model == linear) set Residual = 1e-6 # Number of Newton-Raphson iterations allowed (ignored if Model == linear) set Max iterations Newton-Raphson = 10 # Relative displacement error tolerance for non-linear iteration (ignored if Model == linear) set Tolerance displacement = 1.0e-6 # Relative force residual tolerance for non-linear iteration (ignored if Model == linear) set Tolerance force = 1.0e-9 end
```

This subsection defines parameters for the applied solver. First of all, the underlying model needs to specified: you can either choose a linear elastic model or employ a hyper-elastic non-linear neo-Hookean solid. The non-linear solvers applies an iterative Newton-Raphson scheme to solve the system iteratively. The following selections determine the properties of the linear and non-linear solver. Depending on your configuration, some parameters might not be relevant. The residual of the linear solver is only relevant for the non-linear model, since the residual is adjusted between individual Newton iterations. For the linear model, this value is hard-coded.

Note: You need to build deal.II with `UMFPACK` in order to use the direct solver, which is enabled by default.

```
subsection precice configuration # Cases: FSI3 or PF for perpendicular flap set Scenario = FSI3 # PF x-location set Flap location = 0.0 # Name of the precice configuration file set precice config-file = precice-config.xml # Name of the participant in the precice-config.xml file set Participant name = Solid # Name of the coupling mesh in the precice-config.xml file set Mesh name = Solid-Mesh # Name of the read data in the precice-config.xml file set Read data name = Stress # Name of the write data in the precice-config.xml file set Write data name = Displacement end
```

This section defines preCICE-related settings. The scenario and flap-location parameters can be deleted for your own project since they are just needed for the configuration of our tutorial cases. The other parameters are related to the `precice-config.xml` file. Have a look at the respective entry in the preCICE configuration section (page 62) for details. Make sure the names are the same as in the `precice-config.xml`.

Source code file

Grid generation

Similar to the deal.II tutorial cases, the grid is generated in a function called `make_grid()`, which is called in the beginning of the `run()` function. There are a bunch of options to construct the mesh inside this function, which are extensively described in the deal.II documentation: If your geometry is rather simple (e.g. a shell or a sphere), have a look at the GridGenerator class in the documentation. If you have complex geometries, you might want to create your mesh with external software and load the geometry file in the source code file. In this case, have a look at the GridIn class. The documentation also provides a list of supported mesh file formats.

In our case, we configured the source code file for two tutorial cases (page 0). Hence, there is additionally an `if` condition in the `make_grid()` function, which asks for the chosen tutorial case. Since both cases have a rectangular grid, we generate our mesh by using the `subdivided_hyper_rectangle()` function. Moreover, the deal.II tutorial programs provide various examples for the grid generation.

Boundary conditions

Boundary conditions are applied to specific mesh regions via boundary IDs. We need to distinguish three mesh regions:

1. Dirichlet boundaries, where a constant zero displacement is prescribed
2. Neumann boundaries, where a prescribed traction acts on the surface. This is solely the coupling interface in our case
3. Boundaries without a specified condition (strictly speaking zero traction)

Hence, the first task is the assignment of mesh IDs to the desired mesh region. This is done in the `make_grid()` function. In our case, we used the `colorized = true` option during the grid generation, which automatically assigns each side of our rectangle an individual boundary ID. But you could also iterate over all cells and ask for your own condition (e.g. geometric conditions) and set the boundary ID accordingly. Make sure you only have one boundary ID for the interface mesh in the end. If you have more than one, sum them up in a single ID in a second step as done at the end of the `make_grid()` function. The interface mesh ID is a global variable and needed by the `Adapter` constructor.

The interface mesh is assumed to be the only Neumann boundary. If you have other loads acting on the surface, you need to add it manually during assembly. Constant volume loads (gravity) can be used and are switched off by default, since the tutorial cases don't need them.

Similar to the collection of the interface mesh ID in a single boundary ID, you could sum up your Dirichlet boundaries in one ID. In this case, you could simply use the `clamped_mesh_id` as in the tutorial cases. If you want to set more specific Dirichlet boundaries e.g. in a specific direction, have a look at the bottom of the `assemble_rhs()` / the `make_constraints()` function. You need to modify the `interpolate_boundary_values()` function by e.g. choosing a different direction in the `fe.component_mask()`. The tutorials give an example for doing this in the out-of-plane direction. A detailed documentation is given in the deal.II documentation.

Use the adapter for your own project

Summary: This section will help you couple your own deal.II-code based on the provided deal.II solid codes.

The deal.II adapter provides examples of deal.II codes, which have been coupled using preCICE. This section explains the preCICE-related code changes and introduces the `Adapter` class, which is located in the `include/adapter` directory. A step-by-step tutorial is also available on the preCICE wiki (page 222).

Tip: In addition to our coupled solid mechanics related codes of the dealii-adapter repository, we contributed a minimal deal.II-preCICE example to the deal.II project. If you want to couple your own deal.II-code, this tutorial is probably the best place to start.

Note: Contact us (page 0) if you have any questions. Even if you don't have any questions, please let us know about your experience when your adapter is ready!

Which information is needed by preCICE?

preCICE uses a black-box coupling approach, which means the solver only needs to provide a minimal set of information. In the simplest case, this includes configuration information, e.g. the name of the participant and the coordinates of the data you want to exchange (the interface mesh vertices). If you want to use a nearest-projection mapping, you need to additionally specify mesh connectivity between the vertices (page 239), which is currently not included in this adapter example.

About fluid-structure coupling

For every multi-physics coupling, proper coupling data needs to be exchanged between all participants. In our example case, the `Fluid` participant calculates stresses, which are passed to the `Solid` participant. Using the stress for the structural calculations, the `Solid` participant calculates displacements, which are then passed back to the `Fluid` participant. As outlined above, preCICE needs coordinates of the data points you want to exchange.

The `Adapter` class

This section introduces the `Adapter` class. This class is located in the `include/adapter` directory and contains various files: the files `time_handler.h` and `adapter.h` are the most important files for the adapter. The `time_handler.h` file contains a class to keep track of the current time step and absolute time values. There are other ways to handle this task, but we directly use its functionality in the adapter class to restore the time and therefore included it in this class. The main functionalities are, however, provided in the `adapter.h` file. An exhaustive documentation for all functions can be found directly in the `adapter.h` source code.

In order to use the adapter, we first create an `adapter` object:

```
Adapter::Adapter<dim, Vector<double>, Parameters::AllParameters> adapter(parameters, interface_boundary_id);
```

The first template argument specifies the coupling dimension, the second argument specifies the vector type of your simulation. The third template argument specifies the `Parameter` class type. The `parameter` object is directly passed to the constructor and all preCICE related settings are read by the adapter. In this case, required information is grouped in the parameter file in the subsection `precice configuration`. You can copy and insert it directly in your own parameter class or copy the class from the provided parameter class. Apart from the parameter object, the constructor needs to know your `interface_boundary_id`, which is the `boundary_id` of your deal.II triangulation. Make sure it is unique and doesn't change during simulation.

With the `adapter` object and the `time` object, you can simply modify your time loop in the following way: This code has in large parts been copied from the `linear_elasticity` `run()` function:

```
// In the beginning, we create the mesh and set up the data structures make_grid(); setup_system();
output_results(); // Output initial state assemble_system(); // Then, we initialize preCICE i.e. we
pass our mesh and coupling // information to preCICE // displacement and stress are the coupling da
ta sets for FSI adapter.initialize(dof_handler, displacement, stress); // Then, we start the time l
oop. The loop itself is steered by preCICE. This // line replaces the usual 'while( time < end_tim
e)' while (adapter.precice.isCouplingOngoing()) { // In case of an implicit coupling, we need to st
ore time-dependent // data, in order to reload it later. The decision, whether it is // necessary t
o store the data is handled by preCICE as well adapter.save_current_state_if_required(state_variabl
es, time); // Afterwards, we start the actual time step computation time.increment(); // Assemble t
he time dependent contribution obtained from the Fluid // participant assemble_rhs(); // ...and sol
ve the system solve(); // Update time-dependent data according to the theta-scheme update_displacem
ent(); // Then, we exchange data with other participants. Most of the work is // done in the adapte
r: We just need to pass both data vectors with // coupling data to the adapter. In case of FSI, 'di
splacement' is the // data we calculate and pass to preCICE and 'stress' is the (global) // vector
filled by preCICE/the Fluid participant. adapter.advance(displacement, stress, time.get_delta_t());
// Next, we reload the data we have previously stored in the beginning // of the time loop. This is
only relevant for implicit couplings and // preCICE steers the reloading depending on the specific
// configuration. adapter.reload_old_state_if_required(state_variables, time); // At last, we ask p
reCICE whether this coupling time step (= time // window in preCICE terms) is finished and write th
e result files if (adapter.precice.isTimeWindowComplete() && time.get_timestep() % parameters.outpu
t_interval == 0) output_results(); }
```

Since now the coupling data is time-dependent, you need to assemble parts of your matrices/vectors in every time step, depending on your coupling problem/implementation. In most of the cases and in our example, this will be the rhs of the equations. The function `assemble_rhs()` uses the coupling data, namely stresses, to rebuild the rhs vector. How to apply coupling data is strongly problem-dependent. In our case, we use global data vectors to exchange data with preCICE. Afterwards, we use `get_function_values()` to extract the relevant data locally from the global vector. This way, we also enable to run the assembly with a shared-memory parallelization as shown in the nonlinear elastic case. More details can be found in the `assemble_rhs()` function in the linear elastic solver or the `assemble_neumann_contribution_one_cell()` function in the nonlinear elastic solver.

Your adapter is now ready for most of the preCICE features. For nearest-projection mapping, mesh connectivity needs to be provided as well, which is not yet supported.

Limitations and assumptions

In case you want to use the adapter for your own work, consider the following notes.

Number of interface meshes

You can only define one `read` and one `write` mesh per deal.II executable. On the one hand, extending this is not in particular difficult and could be done in the `Adapter` class itself, but since we have only FSI (solid part) ready-to-run tutorials and programs, it should be sufficient to summarize different boundary parts into a single mesh (in deal.II terminology `boundary_id()`) for writing and reading, respectively. The applied boundary conditions are on each `write` and on each `read` mesh the same. If this is not sufficient, you may want to consider more than one deal.II participant.

Boundary conditions

If you want to couple your own deal.II code, note that we apply here Neumann boundary conditions. In case you would like to couple e.g. a `Fluid` participant, the application of Dirichlet boundary conditions is much more common. The treatment of boundary conditions is very specific to the considered application case and cannot be included in the adapter as presented here. Even if you couple your own structure solver, the boundary conditions resulting from the coupling might be different, e.g. depending on your configuration (Lagrangian vs Eulerian). However, in this code we show everything based on coupling the application to other participant by Neumann boundary conditions. The treatment of Dirichlet boundary conditions is dedicated to future work of this adapter / adapter examples. You might find useful information in the deal.II own tutorials. If you have done or are currently doing something similar, let us know.

Coupling data

As opposed to preCICE itself, there is currently no option to switch between the coupling data type (i.e. vector data and scalar data). This is triggered by the fact that we use purely vector valued data for (FSI) coupling. Have a look in the extensive source code documentation of the `Adapter` class if you would like to change this in your own project.

Mapping

In order to benefit from the high-order property of finite element methods as applied here, we use the support points of high-order polynomials in the coupling mesh. These points are in general different from the quadrature points, where data is evaluated. Therefore, we define our read mappings (and data) to be `consistent`, since we need to interpolate from the support points to the quadrature points. If we use a `conservative` mapping, we might violate the conservation property due to the interpolation step. In one of the previous implementations, we used a conservative read data, but selected the face-centers of the triangulation as read mesh leading to a loss of the natural subcell resolution property of the finite element methods. A rather simple remedy would be the application of collocation techniques, where the interpolation step vanishes. Another option would be to ensure the conservation property in the interpolation step itself. However, this is not considered in the current implementations.

Theoretic details of coupled deal.II codes

Since deal.II is a library and you are free to implement your own stuff, this section provides details about the implemented solver analogous to the commented tutorial programs in deal.II. In case you want to modify this solver or use it for your own project. The linear elastic solver was built on the step-8 tutorial program of the deal.II library, which deals with linear elastostatics. The nonlinear elastic solver was built on the Quasi static Finite strain Compressible Elasticity code gallery example. A lot of aspects are already explained there and in the source code files. However, these programs deal with elastodynamics. Therefore, we need to consider a time discretization.

As a quick overview:

- the linear-elastic solver uses a one-step theta method as described below
- the nonlinear-elastic solver uses an implicit Newmark method. The implemented theory can directly be found in Solution Methods for Time Dependent Problems pp 212 ff

Linear elastic solver

Mathematical aspects

Our starting equation, which is basically the Navier-Cauchy equation, reads as follows:

$$\begin{cases} \rho\ddot{\mathbf{u}} & = \nabla \cdot \sigma + \mathbf{b} \\ \sigma & = \mathbf{C} : \varepsilon \\ \varepsilon & = \frac{1}{2}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right) \end{cases} \tag{1.1}$$

Where u is the displacement field, rho the material density, b the body forces and sigma the stress tensor, which is related to the linear strain measure epsilon via the fourth order elasticity tensor C. Equation 1.1 needs to be satisfied in the whole domain Omega and we apply the following boundary conditions:

$$\begin{aligned} \mathbf{u} &= \mathbf{0} \quad \text{on } \Gamma_u \\ \sigma \cdot \mathbf{n} &= \hat{\mathbf{t}} \quad \text{on } \Gamma_\sigma \end{aligned} \tag{1.2}$$

Here, the first boundary condition is applied to the Dirichlet boundary Gamma_u and we prescribe a zero displacement. The second boundary condition is applied to the Neumann boundary Gamma_sigma and describes basically our coupling interface, since the traction vector is obtained from our flow solver. As last point, the initial conditions are given in equation 1.3:

$$\begin{aligned} \mathbf{u}(\mathbf{x}, t_0) &= \mathbf{0} \quad \text{in } \Omega \\ \dot{\mathbf{u}}(\mathbf{x}, t_0) &= \mathbf{0} \quad \text{in } \Omega \end{aligned} \tag{1.3}$$

Both initial values are chosen to be zero, but you are free to choose them differently according to your problem. The material is assumed as isotropic and thus fully described by the Lamé coefficients:

$$\mathbf{C} = 2\mu\mathbf{I} + \lambda\mathbf{1} \otimes \mathbf{1} \tag{1.4}$$

Where 1 and I are the second and fourth order unit tensors respectively. Finally, the weak formulation of equation 1.1 is given as

$$\int_\Omega \delta\mathbf{u}\cdot\rho\ddot{\mathbf{u}}\,\mathrm{d}\Omega = -\int_\Omega \delta\nabla\mathbf{u}:\mathbf{C}:\frac{1}{2}\left(\nabla\mathbf{u}+(\nabla\mathbf{u})^T\right)\,\mathrm{d}\Omega + \int_\Omega \delta\mathbf{u}\cdot\mathbf{b}\,\mathrm{d}\Omega + \int_{\Gamma_\sigma}\delta\mathbf{u}\cdot\hat{\mathbf{t}}\,\mathrm{d}\Gamma \tag{1.5}$$

Discretization

Discretization in space is done using Finite Elements. By default, linear shape functions are applied, but you are free to specify the polynomial degree in the `parameters.prm` file. More details about the Finite Element discretization are available in the step-8 tutorial description (see link above). The following section focuses on the time discretization. Therefore, the governing second order differential equation is transformed, similar to a state space model, in two first order equations:

$$\begin{cases} \dot{\mathbf{D}}(t) = & \mathbf{V}(t) \\ \mathbf{M}\dot{\mathbf{V}}(t) = & -\mathbf{K}\mathbf{D}(t)+\mathbf{F}(t) \end{cases} \tag{2.1}$$

Here, a block notation for the global vectors and matrices is used, where M denotes the mass matrix, K the stiffness matrix, D the displacement vector, V the velocity vector, and F the load vector, which includes body loads and the prescribed traction. Note that the load vetor F is due to the coupling time dependent. Time derivatives are approximated using a one-step theta method

$$\begin{cases} \frac{\mathbf{D}_{n+1}-\mathbf{D}_n}{\Delta t} = & \theta\mathbf{V}_{n+1}+(1-\theta)\mathbf{V}_n \\ \mathbf{M}\frac{\mathbf{V}_{n+1}-\mathbf{V}_n}{\Delta t} = & \theta\left(-\mathbf{K}\mathbf{D}_{n+1}+\mathbf{F}_{n+1}\right)+(1-\theta)\left(-\mathbf{K}\mathbf{D}_n+\mathbf{F}_n\right) \end{cases} \tag{2.2}$$

where theta is a parameter $[0,1]$, which allows to modify the time stepping properties. $\theta=0$ results in a forward Euler method and $\theta=1$ results in a backward Euler method, which are both first order accurate. Solely $\theta=0.5$ results in a second order accurate Crank-Nicolson scheme, which additionally provides energy conservation in the system.

Performing some equation massaging finally leads to the following system, which is actually implemented:

$$\begin{cases} \mathbf{D}_{n+1} = & \mathbf{D}_n+\theta\Delta t\mathbf{V}_{n+1}+(1-\theta)\mathbf{V}_n \\ \left(\mathbf{M}+\theta^2\Delta t^2\mathbf{K}\right)\mathbf{V}_{n+1} = & \left(\mathbf{M}-\theta(1-\theta)\,\Delta t^2\mathbf{K}\right)\mathbf{V}_n-\Delta t\mathbf{K}\mathbf{D}_n \\ & +(1-\theta)\Delta t\mathbf{F}_n+\theta\Delta t\mathbf{F}_{n+1} \end{cases} \tag{2.3}$$

Hence, we solve in each time step for the unknown velocity and update later on for the unknown displacement. Before the time loop is entered, time invariant global matrices are assembled in the `assemble_system()` function, namely the stiffness matrix K, the mass matrix M, and the constant body loads (gravity). Since the composition of the linear system on the left-hand side (which consists of M and K) is also constant, we store it in a stepping matrix, in order to save the rebuilding in each time step. Note that in the tutorial cases, no gravity is needed and therefore, the term is zero in the example program.

The time-dependent terms are assembled in the `assemble_rhs()` function in each time step. This is straightforward and comments have been added in the source code for more information. Part two of equation 2.3 is finally solved in the `solve()` function, before the displacement vector is updated in the `update_displacement()` function (part one of eq. 2.3).

Capability

This `linear` solver is designed for single-core and single-thread computations. If you like to change the source code for parallel computations, have a look at the step-17 tutorial program, which shows how this can be done using PETSc.

Furthermore, this section should point out that the underlying physical description of the linear structural mechanics is not suitable for large deformations and large rotations. The reason is simply the linear measurement of strains, which are only valid for small deviations. Rigid body rotations lead already to an indicated artificial strain. Due to this, the structure usually gets bigger and a small rotation assumption is violated. Hence, a linear strain measure is typically used for rotations smaller than 6°.

Coupling meshes in deal.II

Summary: The polynomial support points are used to define the coupling mesh.

Defining a coupling mesh in finite element programs is not trivial and there are multiple solution strategies. We rely here on the support points of the high-order polynomials. There are several reasons to do so: First, we would like to keep the high resolution property for higher-order shape functions. An alternative would be to choose the mesh vertices, but it would lead to a loss of information between grid points. Another reason is the consistency to the deal.II infrastructure during the right-hand side assembly, where we treat the coupling data as global vectors. When reading global vectors, data location is assumed to be at the support points i.e. the actual solution points. The third reason is, that we use (by default) a direct solver and therefore, want to keep the overall number of unknowns rather small, while providing a high accuracy. From a solver perspective, high-order polynomial degrees are in most of the applications superior considering the classical solid mechanics as in the tutorials.

However, from a coupling perspective, fewer unknowns are not the best choice, since our mapping methods are restricted to (at most) second order in space. Therefore, having more interface nodes is preferable. Also, using the support points is difficult in case of `conservative` mappings, because data is interpolated to quadrature points and conservation property is not guaranteed. A simple remedy would be to apply collocation techniques, where the interpolation is essentially omitted.

Where are support points located?

The concept of support points is explained in this glossary entry of the deal.II documentation. In particular (by default), we use here the standard `FE_Q` finite element, where the support points are located according to Gauss-Lobatto quadrature points. Note that these points are not equidistant. A detailed description of the `FE_Q` finite element can be found in the `FE_Q` deal.II documentation.

The CalculiX adapter

Summary: The CalculiX adapter can be used to couple CalculiX to CFD solvers for FSI or CHT application or even to couple CalculiX to itself.

Start here

1. Get CalculiX and the dependencies (page 198)
2. Build the Adapter (page 201)
3. Configure and run simulations (page 203)
4. Follow a tutorial:
    - Tutorial for CHT with OpenFOAM and CalculiX: Flow in a shell-and-tube heat exchanger
    - Tutorial for FSI with OpenFOAM and CalculiX: Flow in a channel with an elastic flap, either perpendicular, or parallel to the flow and attached to a cylinder.
    - Tutorial on structure-structure coupling: Elastic beam artificially cut into two halves.

Are you encountering an unexpected error? Have a look at our Troubleshooting (page 206) page.

Do you Want to build on a cluster? Look at our instructions for SuperMUC (page 207) (outdated).

Versions

Please check the Calculix adapter README for the newest compatible CalculiX version.

Adapters for older versions of CalculiX and preCICE are available in various branches. Branches compatible with **preCICE v2.x:**

- master
- v2.15_preCICE2.x

All other branches are compatible with **preCICE v1.x**.

History

The adapter was initially developed for conjugate heat transfer (CHT) simulations via preCICE by Lucia Cheung in the scope of her master's thesis [1] in cooperation with SimScale. For running the adapter for CHT simulations refer to this thesis. The adapter was extended to fluid-structure interaction by Alexander Rusch [2].

References

[1] Lucia Cheung Yau. Conjugate heat transfer with the multiphysics coupling library precice. Master's thesis, Department of Informatics, Technical University of Munich, 2016.

[2] Benjamin Uekermann, Hans-Joachim Bungartz, Lucia Cheung Yau, Gerasimos Chourdakis and Alexander Rusch. Official preCICE Adapters for Standard Open-Source Solvers. In Proceedings of the *7th GACM Colloquium on Computational Mechanics for Young Scientists from Academia*, 2017.

Get CalculiX

Summary: Building CalculiX itself can already be quite a challenges. That's why we collected here some recipe.

The CalculiX adapter for preCICE directly modifies the source code of CalculiX and produces an alternative executable `ccx_preCICE`. Therefore, we first need to get and (optionally) build CalculiX from source.

CalculiX consists of the solver, called "CCX" and a pre- and postprocessing software with graphical user interface "CGX".

- The installation procedure of CCX is described in its `src/README.INSTALL` files, but we also give a summary here.
- We don't modify CGX, so you can simply get a binary package (if needed, e.g. as a preprocessor in our FSI tutorials)

You don't need to build the "vanilla" CalculiX before building the adapter. But you do need to get all the dependencies and the source code of CCX.

Dependencies

CalculiX itself depends on SPOOLES2.2 and ARPACK.

Additionally, our adapter also depends on yaml-cpp.

These can be found in many distributions as binary packages. For example, in Ubuntu, do:

```
sudo apt install libarpack2-dev libspooles-dev libyaml-cpp-dev
```

Building Spooles from source

•If you cannot get a binary for Spooles, try these instructions.

Download SPOOLES, e.g:

```
wget http://www.netlib.org/linalg/spooles/spooles.2.2.tgz
```

Extract it in a separate directory

```
mkdir SPOOLES.2.2 tar zxvf spooles.2.2.tgz -C SPOOLES.2.2 cd SPOOLES.2.2
```

Edit by hand configuration file `Make.inc` to change the compiler version in line 14-15

```
CC = gcc #CC = /usr/lang-4.0/bin/cc
```

Now build the library:

```
make lib
```

Building ARPACK from source

•If you cannot get a binary for ARPACK, try these instructions.

Download Arpack and patch:

```
wget https://www.caam.rice.edu/software/ARPACK/SRC/arpack96.tar.gz wget https://www.caam.rice.edu/software/ARPACK/SRC/patch.tar.gz
```

Unpack them (they will be unpacked in the newly created directory `ARPACK`)

```
tar xzfv arpack96.tar.gz tar xzfv patch.tar.gz cd ARPACK
```

Edit by hand `ARmake.inc` to specify build instructions. The following changes will depend on the directory structure of your system:

- **Line 28**: Change `home = $(HOME)/ARPACK` to directory where ARPACK in extracted
- **Line 115**: Change `MAKE = /bin/make` to e.g. `MAKE = make` (if needed)
- **Line 120**: Change `SHELL = /bin/sh` to e.g. `SHELL = sh` (if needed)
- **Lines 104 - 105**: Specify your fortran compiler and compiler flags, e.g. for the gnu systems:

```
FC = gfortran #FFLAGS = -O -cg89
```

- **Line 35**: Modify the platform suffix for the library and remember it, since Calculix adapter makefile will depend on it (by default it will use suffix INTEL for Linux and MAC for mac systems). For example change `PLAT = SUN4` to `PLAT = INTEL`
- You will probably get linking errors related to ETIME, which you can bypass: In the file `UTIL/second.f` append `*` to the beginning of line 24 (that comments it out)

  ```
  * EXTERNAL ETIME
  ```

Now we are ready to build the library with `make lib`

Building yaml-cpp from source

- If you cannot get a binary for yaml-cpp, try these instructions.

Get the latest version of yaml-cpp and build it as a shared library. For example:

```
wget https://github.com/jbeder/yaml-cpp/archive/yaml-cpp-0.6.2.zip unzip yaml-cpp-0.6.2.zip cd yaml-cpp-yaml-cpp-0.6.2 mkdir build cd build cmake -DBUILD_SHARED_LIBS=ON .. make
```

After building, make sure that you make yaml-cpp discoverable by setting e.g. your `LD_LIBRARY_PATH`. You don't need this for the CalculiX adapter, but you would need it e.g. for the OpenFOAM adapter.

Note: If you use Boost 1.67 or newer, then you also need to install yaml-cpp 0.6 or newer. Similarly, for an older Boost version, you also need an older yaml-cpp. Unfortunately, this is not related to the adapter's code.

Building CalculiX with the preCICE adapter

Get the source

Once the libraries are installed, you can finally install Calculix with preCICE adapter (adapt the `VERSION` in the link, see the beginning of the adapter's README.md to find out which one you need).

```
cd ~ wget http://www.dhondt.de/ccx_VERSION.src.tar.bz2 tar xvjf ccx_VERSION.src.tar.bz2
```

The source code is now in the `~/CalculiX/ccx_VERSION/src` directory. The adapter's `Makefile` is looking for CCX in this directory by default, so modify it if needed.

Building the "vanilla" CalculiX (optional)

If you want to build the "vanilla" (i.e. without preCICE) CalculiX, you can now run `make` inside the `src/` directory. Depending on how you installed the dependencies above (using `apt` or from source), you might get compilation errors, such as `spooles.h:26:10: fatal error: misc.h: No such file or directory`. Often these errors can be easily fixed by modifying CalculiX `Makefile`. Please refer to our adapter's makefile options (page 201) for a list of library and include flag you might have to set depending on your installation procedure.

Building the modified CalculiX

Continue to the page get the adapter (page 201).

Get the CalculiX adapter

Summary: How to build the adapted CalculiX `ccx_preCICE`

After installing preCICE (page 14) and getting the CalculiX source and the required dependencies (page 198), you can now build the adapter, i.e. a modified CCX executable.

Building the adapted CalculiX

1. Download and unzip the adapter (e.g. in the `CaluliX` folder):

   ```
   wget https://github.com/precice/calculix-adapter/archive/master.zip unzip master.zip cd ca
   lculix-adapter-master
   ```

2. Edit the `Makefile` to set the paths to dependencies.
   - If you have the CalculiX source in `~/CalculiX/` and the dependencies in your global paths, you don't need to change anything.
   - Otherwise, set `CCX` and, if built from source, the include and lib flags for SPOOLES, ARPACK, and yaml-cpp.
3. Clean any previous builds with `make clean`.
4. Build with `make` (e.g. `make -j 4` for parallel).
5. You should now have a new executable `ccx_preCICE` in the `bin/` folder of the adapter. You may move this file to a path known by your system, or add this to you `PATH` (careful when doing this!).

Makefile options

The adapter is built using GNU Make. The `Makefile` contains a few variables on top, which need to be adapted to your system:

1. `CCX`: Location of the original CalculiX solver (CCX) source code ("src" directory)
   - Example: `$(HOME)/CalculiX/ccx_2.16/src`
2. `SPOOLES_INCLUDE`: Include flags for SPOOLES
   - Example 1: `SPOOLES_INCLUDE = -I/usr/include/spooles/` (installed)
   - Example 2: `SPOOLES_INCLUDE = -I$(HOME)/SPOOLES.2.2/` (source)
3. `SPOOLES_LIBS`: Library flags for SPOOLES
   - Example 1: `SPOOLES_LIBS = -lspooles` (installed)
   - Example 2: `SPOOLES_LIBS = $(HOME)/SPOOLES.2.2/spooles.a` (source)
4. `ARPACK_INCLUDE`: Include flags for ARPACK
   - Example 1: `ARPACK_INCLUDE =` (installed, nothing needed)
   - Example 2: `ARPACK_INCLUDE = -I$(HOME)/ARPACK` (source)
5. `ARPACK_LIBS`: Library flags for ARPACK
   - Example 1: `ARPACK_LIBS = -larpack -llapack -lblas` (installed)
   - Example 2: `ARPACK_LIBS = $(HOME)/ARPACK/libarpack_INTEL.a` (source)

6. `YAML_INCLUDE`: Include flags for yaml-cpp
    - Example 1: `YAML_INCLUDE = -I/usr/include/` (installed)
    - Example 2: `YAML_INCLUDE = -I$(HOME)/yaml-cpp/include` (source)
7. `YAML_LIBS`: Library flags for yaml-cpp
    - Example 1: `YAML_LIBS = -lyaml-cpp` (installed)
    - Example 2: `YAML_LIBS = -L$(HOME)/yaml-cpp/build -lyaml-cpp` (source)

You may also want to adjust the compiler `FC` from `mpifort` to `mpif90` or to any other compiler your system uses.

Compiling with GCC 10 or newer

If you compile with GCC 10 or newer, you will get the following error, originating from CalculiX:

```
Error: Rank mismatch between actual argument at (1) and actual argument at (2) (rank-1 and scalar)
```

To work around this, you need to add `-fallow-argument-mismatch` to the `FFLAGS` inside `Makefile`:

```
- FFLAGS = -Wall -O3 -fopenmp $(INCLUDES) + FFLAGS = -Wall -O3 -fopenmp -fallow-argument-mismatch
$(INCLUDES)
```

Notes on preCICE versions

- In case you are using some very old preCICE version, please upgrade. Our community is happy to help you. Click here and keep reading if you loved preCICE v1.x and (optionally) wish The Beatles were still around.

1. This adapter expects the preCICE C bindings in `[prefix]/include/precice/SolverInterfaceC.h` and gets this path from pkg-config. In other words, this assumes that preCICE (at least v1.4.0) has been built & installed with CMake (e.g. using a Debian package). In case you want to keep using preCICE built with SCons, see the changes invoked by Pull Request #14.
2. Starting from preCICE v1.2.0, the name (and the respective paths) of the language “adapters” have changed to language “bindings”. This affects the line `#include "precice/bindings/c/SolverInterfaceC.h"` in `calculix-adapter/adapter/PreciceInterface.c`. To compile with older preCICE versions, change `bindings` to `adapters`.

Configure the CalculiX adapter

Summary: Write a config.yml, write a CalculiX case input file, and run an adapted CalculiX executable.

Layout of the YAML configuration file

The layout of the YAML configuration file, which should be named `config.yml` (default name), is explained by means of an example for an FSI simulation:

```
participants: Calculix: interfaces: - nodes-mesh: Calculix_Mesh patch: interface read-data: [Force
s] write-data: [DisplacementDeltas] precice-config-file: ../precice-config.xml
```

The adapter allows to use several participants in one simulation (e.g. several instances of Calculix if several solid objects are taken into account). The name of the participant “Calculix” must match the specification of the participant on the command line when running the executable of “CCX” with the adapter being used (this is described later). Also, the name must be the same as the one used in the preCICE configuration file *precice-config.xml*.
One participant may have several FSI interfaces. Note that each interface specification starts with a dash.
For FSI simulations the mesh type of an interface is always “nodes-mesh”, i.e. the mesh is defined node-wise, not element-wise. The name of this mesh, “Calculix_Mesh”, must match the mesh name given in the preCICE configuration file.
For defining which nodes of the CalculiX domain belong to the FSI interface, a node set needs to be defined in the CalculiX input files. The name of this node set must match the name of the patch (here: “interface”).
In the current FSI example, the adapter reads forces from preCICE and feeds displacement deltas (not absolute displacements, but the change of the displacements relative to the last time step) to preCICE. This is defined with the keywords “read-data” and “write-data”, respectively. The names (here: “Forces” and “DisplacementDeltas”) again need to match the specifications in the preCICE configuration file. In the current example, the coupled fluid solver expects displacement deltas instead of displacements. However, the adapter is capable of writing either type. Just use “write-data: [Displacements]” for absolute displacements rather than relative changes being transferred in each time step. Valid `readData` keywords in CalculiX are:

```
* Forces * Displacements * Temperature * Heat-Flux * Sink-Temperature * Heat-Transfer-Coefficient
```

Valid `writeData` keywords are:

```
* Forces * Displacements * DisplacementDeltas * Temperature * Heat-Flux * Sink-Temperature * Heat-T
ransfer-Coefficient
```

From CalculiX version 2.15, additional `writeData` keywords are available:

```
* Positions * Velocities
```

Note that the square brackets imply that several read- and write-data types can be used on a single interface. This is not needed for FSI simulations (but for CHT simulations). Lastly, the “precice-config-file” needs to be identified including its location. In this example, the file is called *precice-config.xml* and is located one directory above the folder, in which the YAML configuration file lies.

CalculiX case input file

An exemplary CalculiX case input file may look like the following:

```
*INCLUDE, INPUT=all.msh *INCLUDE, INPUT=fix1.nam *INCLUDE, INPUT=fix2.nam *INCLUDE, INPUT=fix3.nam
*INCLUDE, INPUT=interface.nam *MATERIAL, Name=EL *ELASTIC 100000000, 0.3 *DENSITY 10000.0 *SOLID SE
CTION, Elset=Eall, Material=EL *STEP, NLGEOM, INC=1000000 *DYNAMIC 0.01, 5.0 *BOUNDARY Nfix1, 3, 3,
0 Nfix2, 1, 1, 0 Nfix2, 3, 3, 0 Nfix3, 1, 3, 0 *CLOAD Ninterface, 1, 0.0 Ninterface, 2, 0.0 Ninterf
ace, 3, 0.0 *NODE FILE U *EL FILE S, E *END STEP
```

The adapter internally uses the CalculiX data format for point forces to apply the FSI forces at the coupling interface. This data structure is only initialized for those nodes, which are loaded at the beginning of a CalculiX analysis step via the input file. Thus, it is necessary to load all nodes of the node set, which defines the FSI interface in CalculiX (referring to the above example, the nodes of set “interface” (Note that in CalculiX a node set always begins with an “N” followed by the actual name of the set, which is here “interface”.) are loaded via the “CLOAD” keyword.), in each spatial direction. However, the values of these initial forces can (and should) be chosen to zero, such that the simulation result is not affected.

CalculiX CCX offers both a geometrically linear as well as a geometrically non-linear solver. Both are coupled via the adapter. The keyword “NLGEOM” (as shown in the example) needs to be included in the CalculiX case input file in order to select the geometrically non-linear solver. It is also automatically triggered if material non-linearities are included in the analysis. In case the keyword “NLGEOM” does not appear in the CalculiX case input file and the chosen materials are linear, the geometrically linear CalculiX solver is used. In any case, for FSI simulations via preCICE the keyword “DYNAMIC” (enabling a dynamic computation) must appear in the CalculiX input file.

More input files that you may find in the CalculiX tutorial cases:

- `<name>.inp`: The main case configuration file. Through this, several other files are included.
- `<name>.msh`: The mesh file.
- `<name>.flm`: Films
- `<name>.nam`: Names, e.g. indices of boundary nodes
- `<name>.sur`: Surfaces
- `<name>.dfl`: DFlux

Running the adapted calculiX executable

Running the adapted executable is pretty similar to running the original CalculiX CCX solver. The syntax is as follows:

```
ccx_preCICE -i [CalculiX input file] -precice-participant [participant name]
```

For example:

```
ccx_preCICE -i flap -precice-participant Calculix
```

The input file for this example would be *flap.inp*. Note that the suffix “.inp” needs to be omitted on the command line. The flag “-precice-participant” triggers the usage of the preCICE adapter. If the flag is not used, the original unmodified solver of CCX is executed. Therefore, the new executable “ccx_preCICE” can be used both for coupled preCICE simulations and CalculiX-only runs. Note that as mentioned above, the participant name used on the command line must match the name given in the YAML configuration file and the preCICE configuration file.

Supported elements

The preCICE CalculiX adapter supports solid and shell elements. It can been used with both linear and quadratic tetrahedral (C3D4 and C3D10) and hexahedral (C3D8 and C3D20) elements. For shell elements, currently S3 and S6 tetrahedral elements are supported. There is a restriction when using nearest-projection mapping that you have to use tetrahedral elements. If a quasi 2D-3D case is set up (single element in out-of-place direction) then only linear elements are supported.

Nearest-projection mapping

In order to use nearest-projection mapping, a few additional changes are required. The first is that the interface surface file (.sur) must be added to the Calculix input file. An example of the addition to the input file is given below

```
*INCLUDE, INPUT=all.msh *INCLUDE, INPUT=fix1.nam *INCLUDE, INPUT=fix2.nam *INCLUDE, INPUT=fix3.nam
*INCLUDE, INPUT=interface.nam *INCLUDE, INPUT=interface.sur *MATERIAL, Name=EL
```

This surface file is generated during the mesh generation process. The second addition is to the config.yml. In order for the adapter to know that the surface mesh must be read, the line

```
- nodes-mesh
```

must be changed to

```
- nodes-mesh-with-connectivity
```

Note that an error will only occur if nodes-mesh-with-connectivity is specified without a .sur file. The calculix-adapter with nearest-projection mapping only supports tetrahedral elements (C3D4 and C3D10) as preCICE only works with surface triangles for nearest-projection mapping.

Troubleshooting the CalculiX adapter

Summary: While working with the CalculiX adapter, you may run onto common issues. This is a collection of what we know could go wrong.

This list is definietly not complete. If after reading this, you still have issues, please ask in the preCICE forum.

Things to check

- Are you using the same version of CalculiX and of the CalculiX adapter? The adapter installation works by replacing files of the original CalculiX, so they should be compatible.
- Can you manually run the `ccx_preCICE` binary?
    - It should be in your `$PATH`
    - If autocompletion does not work (e.g. `ccx_` + TAB key), then it is probably not in your `$PATH`.
- Our tutorials also require CGX (pre- and post-processor of CalculiX).
    - Is CGX installed?
    - Is OpenGL (required by CGX) installed?

Building the CalculiX adapter on SuperMUC

Summary: This page explains how to build the CalculiX adapter on SuperMUC. Even though SuperMUC was shut down in 2019, this page may still be useful for other clusters.

Warning: This page needs updates for preCICE v2.

In order to install CalculiX and the adapter on superMUC, a number of depencies are first required. Initially, preCICE must be installed (page 14)

Additionally, SPOOLES, ARPACK and yaml-cpp are required.

To install SPOOLES, some changes are necessary.

1. `makefile` : `~/SPOOLES.2.2/Tree/src/makeGlobalLib` contains an error: file `drawTree.c` does not exist and should be replaced by `tree.c`.
2. Changes to the `Make.inc` file must be done according to the CalculiX install Manual, page 16 and 17.

In installing ARPACK, the HOME directory needs to be specified in the “ARmake.inc” file. No changes are necessary for the Makefile. To install ARPACK, run “make lib” in the ARPACk directory.

Any problems with the installation of SPOOLES and ARPACK can be searched in the installation instructions.

To install yaml-cpp, run in the source directory:

```
mkdir build cd build cmake .. make make install
```

yaml-cpp 0.5.3 is known to work. Newer version may also work. yaml-cpp can be downloaded from

```
wget https://github.com/jbeder/yaml-cpp/archive/release-0.5.3.tar.gz -O - | tar xz
```

Module List

The following modules available in superMUC are known to work for the CalculiX adapter installation.

1. python/3.5_anaconda_nompi
2. scons/3.0.1
3. valgrind/3.10
4. petsc/3.8
5. boost/1.65_gcc
6. gcc/6
7. mpi.intel/2017

Makefile Changes

The paths to the CalculiX CCX, SPOOLES, ARPACK and YAML must be specified. Line 61: “FC = mpifort” can be commented out and replaced with “FC = gfortran”.

The path to the pkgconfig file needs to be stated. The command “export PKG_CONFIG_PATH=/path/to/lib/pkgconfig” must be provided. It is easier to install preCICE with the “CMAKE_INSTALL_PREFIX” set to the path where preCICE is installed.

Adapter Installation

To install the adapter, the command with the following configurations is known to work:

```
cmake -DBUILD_SHARED_LIBS=ON -DCMAKE_INSTALL_PREFIX=/path -DCMAKE_BUILD_TYPE=RelWithDebInfo ..
```

The SU2 adapter

Summary: Modify native SU2 files to couple with other solvers or SU2 itself

References

[1] Alexander Rusch. Extending SU2 to fluid-structure interaction via preCICE. Bachelor's thesis, Munich School of Engineering, Technical University of Munich, 2016.

Build the adapter

Summary: Get SU2, get preCICE, execute adapter install script

SU2

Before installing the adapter SU2 itself must be downloaded from SU2 repository. If necessary unpack the code and move it to your preferred location. Please do not configure and build the package before installing the adapter. In case you have already used SU2 you will need to rebuild the suite after installing the adapter.

preCICE

It is assumed that preCICE has been installed successfully beforehand. Concerning installation instructions for preCICE, have a look at the preCICE installation documentation (page 14)

Adapter

In order to run SU2 with the preCICE adapter, some SU2-native solver routines need to be changed. The altered SU2 files are provided with this adapter in the directory “replacement_files”. Moreover, preCICE-specific files are contained in the directory “adapter_files”. These need to be added to the source code of SU2. A simple shell script called su2AdapterInstall comes with the adapter, which automates this process and replaces/copies the adapted and preCICE-specific files to the correct locations within the SU2 package including the appropriately adjusted Makefile of SU2. For the script to work correctly, the environment variable `SU2_HOME` needs to be set to the location of SU2 (top-level directory).

> **Note:** If you run `./configure --prefix=$SU2_HOME` and get the error `configure: error: cannot find python-config for /usr/bin/python`, check via `ls /usr/bin` whether there is a `python-config` and/or `python2.7-config`. If not, you can create a symbolic link via `ln /usr/bin/python3-config /usr/bin/python-config` such that `python-config` uses `python3-config`

It is recommended to set these variables permanently in your `~/.bashrc` (Linux) or `~/.bash_profile` (Mac). After setting these variables the script `su2AdapterInstall` can be run from the directory, in which it is contained: `./su2AdapterInstall` The script will not execute if the environment variables are unset or empty.

If you do not want to use this script, manually copy the files to the locations given in it. The two environment variables need to be defined as stated above, nevertheless.

After copying the adapter files to the correct locations within the SU2 package, SU2 can be configured and built just like the original version of the solver suite. Please refer to the installation instructions provided with the SU2 source code. SU2 should be built with MPI support in order to make use of parallel functionalities. The script su2AdapterInstall states recommended command sequences for both the configuration and the building process upon completion of the execution.

The SU2 executable is linked against the dynamic library of preCICE, so make sure you have built it like this.

Running simulations

Summary: Modify SU2 configuration file, specify interfaces by SU2 markers, run SU2 either serial or parallel

SU2 configuration file

The adapter is turned on or off via the native SU2 configuration file. If it is turned off, SU2 executes in its original version. Moreover, the adapter is configured in the SU2 configuration file. The following adapter-related options are currently available (default values given in brackets):

1. `PRECICE_USAGE` (NO): Determines whether a preCICE-coupled simulation is run or not.
2. `PRECICE_VERBOSITYLEVEL_HIGH` (NO): Produces more output, mainly for debugging purposes.
3. `PRECICE_LOADRAMPING` (NO): Allows to linearly ramp up the load on the structural component at the beginning of the simulation. This may resolve stability issues due to large loads at the beginning of simulations.
4. `PRECICE_CONFIG_FILENAME` (precice-config.xml): Location and name of the preCICE configuration file.
5. `PRECICE_PARTICIPANT_NAME` (SU2): Name of the participant in the preCICE configuration file.
6. `PRECICE_MESH_NAME` (SU2-Mesh): Name of the mesh in the preCICE configuration file.
7. `PRECICE_READ_DATA_NAME` (Displacements): Name of the read data in the preCICE configuration file. The SU2-adapter supports reading absolute displacements and relative displacements. The adapter picks up the respective data field according to the name: 'Displacement' for absolute displacement and 'DisplacementDelta' for relative displacement. ⚠ **Important:** Note that reading 'Displacement' data has been added for compatibility reasons with our tutorials and it cannot be used with `extrapolation` in your configuration. Only the relative `DisplacementDelta` data supports this feature.
8. `PRECICE_WRITE_DATA_NAME` (Forces): Name of the write data in the preCICE configuration file.
9. `PRECICE_WETSURFACE_MARKER_NAME` (wetSurface): Name of the marker, which identifies the FSI interface in the geometry file of SU2.
10. `PRECICE_LOADRAMPING_DURATION` (1): Number of time steps, in which the load ramping is active, counting from the beginning of the simulation. The ramped load increases linearly with each time step.
11. `PRECICE_NUMBER_OF_WETSURFACES` (1): In case multiple FSI-interfaces exist, their count needs to specified here.

If multiple interfaces exist, the names of all related entries (`PRECICE_WETSURFACE_MARKER_NAME` , `PRECICE_READ_DATA_NAME` , `PRECICE_WRITE_DATA_NAME` , `PRECICE_MESH_NAME`) must be appended by consecutive numbers. Hence, the names (also in the geometry file) need to be alike differing only by the appending number, which must be successively increasing from zero. E.g. for three interfaces, the marker name could be defined as `PRECICE_WETSURFACE_MARKER_NAME= wetSurface` in the SU2 configuration file, while the markers in the geometry file would need to be named *wetSurface*, *wetSurface1* and *wetSurface2*.

Moreover, in the SU2 configuration file grid movement must be allowed: `GRID_MOVEMENT= YES` and the type of grid movement must be set correctly for preCICE-coupled simulations: `GRID_MOVEMENT_KIND= PRECICE_MOVEMENT` . Also, the boundary, which is allowed to move needs to be specified. Here the name of the FSI-interface marker including its appending identifying number as stated above needs to be used, e.g., `MARKER_MOVING= ( wetSurface0 )` . If multiple FSI-interfaces exist (as in the example above), this may look like `MARKER_MOVING= ( wetSurface, wetSurface1, wetSurface2 )` .

Running the adapted SU2 executable

Since the adapter (as well as its options) is turned on or off via the SU2 configuration file, the execution procedure is just as for the original version of SU2. For execution with one process working on the fluid domain from the directory, in which both the SU2_CFD executable and the SU2 configuration file are located:

```
./SU2_CFD su2ConfigurationFile.cfg
```

The adapter is designed such that it can be executed in an intra-parallel manner meaning that the flow domain is decomposed into several parts. The execution is then as follows (again assuming that executable and configuration file are within the current directory; exemplifying a decomposition of the fluid domain with eight processes):

```
mpirun -n 8 ./SU2_CFD su2ConfigurationFile.cfg
```

The FEniCS adapter

Summary: A general adapter for the open source computing platform FEniCS

What is FEniCS ?

From the FEniCS website: *FEniCS is a popular open-source (LGPLv3) computing platform for solving partial differential equations (PDEs). FEniCS enables users to quickly translate scientific models into efficient finite element code. With the high-level Python and C++ interfaces to FEniCS, it is easy to get started, but FEniCS offers also powerful capabilities for more experienced programmers. FEniCS runs on a multitude of platforms ranging from laptops to high-performance clusters.* More details can be found at fenicsproject.org.

Aim of this adapter

This adapter supports the Python interface of FEniCS and offers an API that allows the user to use FEniCS-style data structures for solving coupled problems. We provide usage example for the adapter for heat transport, conjugate heat transfer and fluid-structure interaction. However, the adapter is designed in a general fashion and can be used to couple any code using the FEniCS library.

How to install the adapter?

The adapter requires FEniCS and preCICE version 2.0 or greater and the preCICE language bindings for Python. The adapter is published on PyPI. After installing preCICE and the python language bindings one can simply run `pip3 install --user fenicsprecice` to install the adapter via your Python package manager.

Please refer to the installation instructions provided here for alternative installation procedures.

Examples for coupled codes

The following tutorials can be used as a usage example for the FEniCS adapter:

- Solving the heat equation in a partitioned fashion (heat equation solved via FEniCS for both participants), see [1]
- Flow over plate (heat equation solved via FEniCS for solid participant), see [1]
- Perpendicular flap (structure problem solved via FEniCS), see [1, 2]
- Cylinder with flap (structure problem solved via FEniCS), see [2]

How can I use my own solver with the adapter ?

The FEniCS adapter does not couple your code out-of-the-box, but you have to call the adapter API from within your code. You can use the tutorials from above as an example. The API of the adapter and the design is explained and usage examples are given in [1].

You need more information?

Please don't hesitate to ask questions about the FEniCS adapter on discourse or in gitter.

How to cite

If you are using our adapter, please consider citing our paper "FEniCS-preCICE: Coupling FEniCS to other Simulation Software" [1].

Related literature

[1] Benjamin Rodenberg, Ishaan Desai, Richard Hertrich, Alexander Jaust, Benjamin Uekermann. FEniCS-preCICE: Coupling FEniCS to other Simulation Software. preprint on arXiv, arxiv.org/abs/2103.11191, 2021
[2] Richard Hertrich. Partitioned fluid structure interaction: Coupling FEniCS and OpenFOAM via preCICE. Bachelor's thesis, Munich School of Engineering, Technical University of Munich, 2019.

The code_aster adapter

Summary: On this page, we give a step-by-step guide how to get and install code_aster and the code_aster adapter. The adapter currently supports usage of code_aster as solid solver for conjugate heat transfer problems. We use the Python command files of code_aster and call the preCICE Python bindings from there.

Requirements

The adapter requires at least preCICE v2.0. It was developed and tested again code_aster v14.4 and v14.6.

Since code_aster works with 'command files' that include integrated python code, you will need to the python bindings of preCICE to use this adapter:

- Get preCICE (page 14)
- Get Python bindings of preCICE (page 43)

Get code_aster

This part is meant as a brief overview for those who are not yet familiar with code_aster. Please consult the official documentation of code_aster for any issues.

There are two possible ways to install the code_aster solver on your system, but we only support the second one:

1. The easiest and most intuitive way, is to install Salome-Meca. This is a user-friendly code_aster implementation, that also provides pre- and post-processing software. It can be used to create the mesh and model, and it also provides the post-processing software ParaVis.
2. The second method to install code_aster on your system is to download a package containing the code_aster source code. This grants the possibility to run the code_aster solver from a script, but brings about some additional complexities during installation. This implementation of code_aster is supported for coupling with preCICE.

To install code_aster, download the full package from code-aster.org, under download. It is recommended to install a stable version of code_aster (here 14.6).

Dependencies

Make sure to have the code_aster dependencies installed before building code_aster.

For Ubuntu, you may install the following packages:

```
bison cmake make flex g++ gcc gfortran \ grace liblapack-dev libblas-dev \ libboost-numpy-dev libbo
ost-python-dev \ python3 python3-dev python3-numpy \ tk zlib1g-dev
```

Building and installation

To initiate the installation of code_aster, redirect the terminal to the location of the `setup.py` file, and run the following command.

```
python3 setup.py install --prefix=/your/target/path
```

The installation will ask you to confirm the automatically-set environment soon after it starts. Make sure that none of the dependencies listed are missing, and that there are no unexpected messages. It can happen that some optional dependencies (such as nedit, geany or gvim) are not found, this is not a problem. Once confirmed that everything is correct, you can go ahead and tell the terminal to continue the installation.

```
Checking for GNU compilers... yes
Checking for global values...

Compiler variables (set as environment variables):

export               CC='/usr/bin/gcc'
export           CFLAGS='-O2 -fno-stack-protector -fPIC'
export       CFLAGS_DBG='-g  -fno-stack-protector -fPIC'
export    CFLAGS_OPENMP='-fopenmp'
export              CXX='/usr/bin/g++'
export           CXXLIB='-L/usr/lib/gcc/x86_64-linux-gnu/7 -lstdc++'
export          DEFINED='LINUX64  _USE_OPENMP'
export              F90='/usr/bin/gfortran'
export         F90FLAGS='-O2 -fPIC'
export     F90FLAGS_DBG='-g  -fPIC'
export      F90FLAGS_I8=' -fdefault-double-8 -fdefault-integer-8 -fdefault-real-8'
export  F90FLAGS_OPENMP=' -fopenmp'
export        FFLAGS_I8=' -fdefault-double-8 -fdefault-integer-8 -fdefault-real-8'
export               LD='/usr/bin/gfortran'
export   LDFLAGS_OPENMP=' -fopenmp'
export          MATHLIB='/usr/lib/x86_64-linux-gnu/liblapack.a /usr/lib/x86_64-linux-gnu/libblas.a'
export         OTHERLIB='-L/usr/lib/x86_64-linux-gnu -lpthread -L/usr/lib/x86_64-linux-gnu -lz'

# Environment settings :

--------------------------------------------------------------------------------
Checking for ps... /bin/ps
Checking for xterm... /usr/bin/xterm
Checking for nedit... no
Checking for geany... no
Checking for gvim... no
Checking for gedit... /usr/bin/gedit
Checking for gdb... /usr/bin/gdb
Checking for ddd... no
Checking for flex... /usr/bin/flex
Checking for ranlib... /usr/bin/ranlib
Checking for bison... /usr/bin/bison
Checking for cmake... /usr/bin/cmake

--------------------------------------------------------------------------------
Checking for host name...   michel-HP-ZBook-Studio-G4
Checking for network domain name...    (empty)
Checking for full qualified network name...    michel-HP-ZBook-Studio-G4

--------------------------------------------------------------------------------
Checking for dependencies and required variables for '__main__'...     [  OK  ]

--------------------------------------------------------------------------------
Checking for dependencies and required variables for '__cfg__'...      [  OK  ]
Filling cache...                                                       [  OK  ]


--------------------------------------------------------------------------------
Check if found values seem correct. If not you can change them using 'setup.cfg'.
Do you want to continue (y/n, default n) ?
```

code_aster and the bundled dependencies will now be built. This can take a while.

After the installation is done, check that all dependencies have been installed correctly. If a dependency was not installed correctly, go through the log file, and try to run the installation again. Alternatively, install the dependency manually and specify its path in `setup.cfg`. In this case, make sure that the required version of the tool is installed.

```
--------------------------------------------------------------------------------
Installation of aster 14.4.0 successfully completed
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
     SUMMARY OF INSTALLATION
--------------------------------------------------------------------------------

 Installation of   : hdf5 1.10.3
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/hdf5-1.10.3
Elapsed time       : 106.36 s
                                                                      [  OK  ]
 Installation of   : med 4.0.0
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/med-4.0.0
Elapsed time       : 84.08 s
                                                                      [  OK  ]
 Installation of   : gmsh 3.0.6
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/gmsh-3.0.6-Linux
Elapsed time       : 0.58 s
                                                                      [  OK  ]
 Installation of   : scotch 6.0.4
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/scotch-6.0.4
Elapsed time       : 19.43 s
                                                                      [  OK  ]
 Installation of   : astk 2019.0
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/lib/python3.6/site-packages
Elapsed time       : 0.49 s
                                                                      [  OK  ]
 Installation of   : metis 5.1.0
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/metis-5.1.0
Elapsed time       : 20.44 s
                                                                      [  OK  ]
 Installation of   : tfel 3.2.1
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/tfel-3.2.1
Elapsed time       : 221.39 s
                                                                      [  OK  ]
 Installation of   : mumps 5.1.2
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/mumps-5.1.2
Elapsed time       : 84.18 s
                                                                      [  OK  ]
 Installation of   : homard 11.12
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch/public/homard-11.12
Elapsed time       : 1.26 s
                                                                      [  OK  ]
 Installation of   : aster 14.4.0
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch
Elapsed time       : 279.64 s
                                                                      [  OK  ]
 Installation of   : Code_Aster + 10 of its prerequisites
Destination        : /home/michel/Desktop/aster-full-src-14.4.0-1.noarch
Elapsed time       : 832.01 s
                                                                      [  OK  ]
```

Once the solver has been installed successfully, add the following line to the bashrc (run `gedit ~/.bashrc`) and start a new session:

```
source $ASTER_ROOT/etc/codeaster/profile.sh
```

where `$ASTER_ROOT` you should replace with the actual path where you build code_aster. This line will make sure that once the user runs the just built code_aster version with the `as_run` command. If your package manager suggests to install a binary package of code_aster (do not install it), you may have not set the `$ASTER_ROOT` path correctly.

Testing

We can test the installation of code_aster with the following command:

```
as_run --vers=14.6 --test forma01a
```

If everything is as expected, the output should be `--- DIAGNOSTIC JOB : OK`.

```
<INFO> Code_Aster run ended, diagnostic : OK

-------------------------------------------------------------------------------
 Copying results


OK                 Code_Aster run ended



-------------------------------------------------------------------------------------------
                                          cpu      system     cpu+sys     elapsed
-------------------------------------------------------------------------------------------
  Preparation of environment             0.00       0.00        0.00        0.00
  Copying datas                          0.01       0.01        0.02        0.03
  Code_Aster run                         1.05       0.13        1.18        1.17
  Copying results                        0.00       0.00        0.00        0.00
-------------------------------------------------------------------------------------------
  Total                                  1.13       0.17        1.30        1.30
-------------------------------------------------------------------------------------------

as_run 2019.0

------------------------------------------------------------
--- DIAGNOSTIC JOB : OK
------------------------------------------------------------
```

Get the code_aster adapter

1. Download the adapter code or, even better, clone the repository `https://github.com/precice/code_aster-adapter.git`
2. Place the file `cht/adapter.py` in the code_aster directory, under `$ASTER_ROOT/14.6/lib/aster/Execution`.

Test cases

There is a tutorial (page 0) available to help you get started with coupling code_aster through preCICE. In this tutorial, we couple code_aster as a solid solver, and OpenFOAM as a fluid solver for a flow-over-plate conjugate-heat-transfer scenario.

Required files for a coupled Simulation

You can find the tutorial files in the tutorial repository. The Solid directory contains, among others, the following files:

- `solid.astk`: In a Code_Aster case, there is always an export file that links all the separate case files, specifying their functionality and their location. The export file also sets additional, system-dependent variables. The export file is to be generated from the `solid.astk` file, which can be done in ASTK as described below.
- example.export: This is a template export file to run this tutorial. Apart from being a reference example for the `solid.export` file that is to be generated, it can be used to run this tutorial without using ASTK for generating `solid.export`. See Alternative: Skipping ASTK configuration below for more information.

- `solid.mmed` : This is the file that contains the mesh of the solid domain for Code_Aster. It can be opened and adapted with Salome Meca.

code_aster works with command files, which are the basis of every simulation case. The command files define the problem, the boundary conditions, the mesh that is used, and more parameters. When we couple Code_Aster with preCICE, we mainly use three command files:

- `adapter.comm` : This is the main command file of a Code_Aster coupling. Code_Aster starts at this command file, which wraps the solver call in a loop and triggers the coupling operations. Through the INCLUDE command, invoked at the beginning, the other command files are included. This file is part of the Code_Aster adapter.
- `def.comm` : The test-case is defined in this command file. It is in charge of setting the mesh, model, materials, initial and boundary conditions. This file is case specific and is found in the tutorial repository.
- `config.comm` : This file is used to configure the coupling. This file is part of the Code_Aster adapter

Additionally, the following files are created when the coupling is run:

- `solid.mess` : An output (message) file, which contains the Code_Aster log of a run.
- `solid.rmed` : This file is the 'result mesh' file, and has the same format as the mesh file `solid.mmed` . It contains the result of the solid domain after a run, and can be opened with Salome Meca. In this tutorial, multiple rmed files will be generated and saved in the REPE_OUT folder.
- `solid.resu` : This file is also a 'result mesh' file, but it saves the results in ASCII format. It is not relevant for this tutorial.

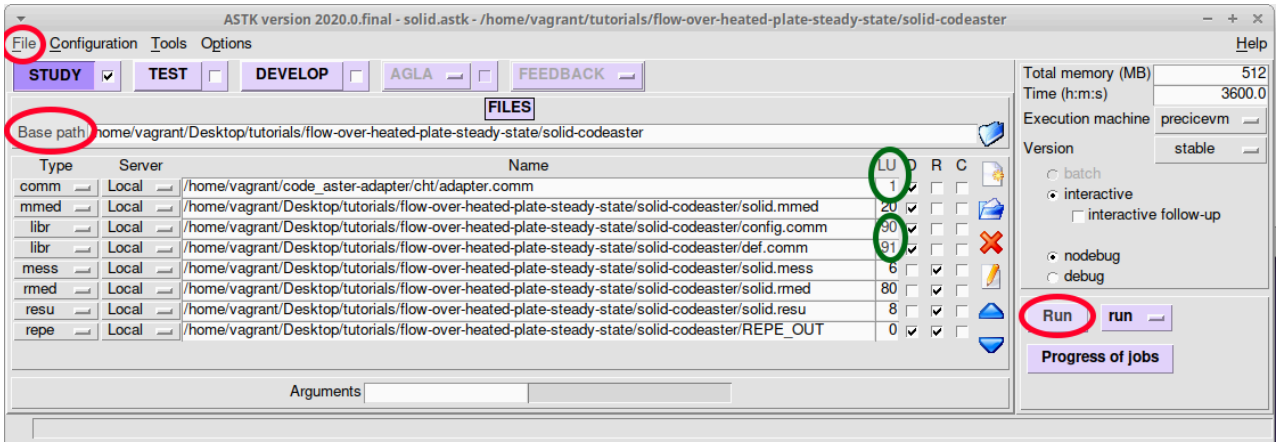
Configuring the adapter

Generating the Export file

The `solid.export` file that is included in the tutorial needs to be configured for your local system. Alternatively, you can skip the generation of `solid.export` , and use the template export file provided. Please keep in mind that generating `solid.export` through ASTK, will tune Code_Aster to your local system and run more efficiently. To generate this file using ASTK, follow these steps:

1. Start astk from your terminal.
2. Click "File > Open..." and select the file `solid.astk` .
3. In the Base path field, set the path to your `solid-codeaster` directory. Note that clicking on a text field and then clicking on the Base path, astk will auto-fill the selected field with the base path.
4. The following should already be set by default:
    a. Select under `D` (input) the files `adapter.comm` , `solid.mmed` , `config.comm` , `def.comm` . Select under `R` (output) the files `solid.mess` , `solid.rmed` , `solid.resu` .
    b. For the `.comm` files, look at the `LU` values and make sure that `adapter.comm` is assigned to `UNIT=1` , `def.comm` is assigned to `UNIT=91` , and `config.comm` has `UNIT=90` . The adapter.comm is the command file that comes with the Code_Aster adapter. For the rest of the files, ASTK will give the default `UNIT` values. Make sure that these correspond to the values in the image below.
    c. Make sure that in ASTK, the `nodebug` mode is selected.
    d. Lastly, add a new field of type `repe` , by pressing the `Add Entry` button on the right. In this field, point `REPE_OUT` to be located in the `solid-codeaster` directory, as shown in the image below. Make sure to also create this directory on your system. This `REPE_OUT` folder will hold the `rmed` output files of Code_Aster.

5. Now that you have updated the `solid.astk` file, save and export it from the `File` menu. You need to give a name for your file, e.g. `solid.export`.
6. Click "Run" to generate the rest of the files and exit ASTK.
7. Run the case from a terminal as `as_run --run solid.export`.



Setting the preCICE exchange directory

We currently need to manually set the `exchange-directory` in the `m2n` :sockets node:

```
<m2n:sockets from="Fluid" to="Solid" exchange-directory="tutorials/flow-over-heated-plate-steady-state"/>
```

See the respective issue for more details on why this is needed.

Post-processing

There are two methods to visualize the results for Code_Aster:

1. Salome-Meca is a intergated graphical interface, which also offers a post-processing unit called ParaViS (based on ParaView). The nice thing about ParaViS, is that it can open both the results of OpenFOAM and Code-Aster at the same time. Please make sure to have salome-meca 2018 or newer, as the med files are not compatible with older versions. Before installing Salome-Meca, please make sure that the environment on your system uses Python 2.7 (see the respective Salome-Meca issue).
2. GMSH is a stand-alone visualization tool that can open files of med format. Please make sure to get a version that is compatible with med 4.0 (GMSH 4.5 is known to work).

History

The adapter was implemented as part of the master thesis of Lucia Cheung in cooperation with SimScale. For quick access: an excerpt of Lucia's thesis focusing on the adapter

The Nutils adapter

Summary: There is currently not really such a thing as a Nutils adapter. Coupling Nutils is so simple that directly calling the preCICE Python API from the application scripts is the way to go.

The best way to learn how to couple a Nutils application script is to look at some examples:

- Two heat conduction scripts coupled to one another
- A heat conduction script coupled to CFD for conjugate heat transfer
- An ALE incompressible Navier-Stokes solver coupled to solid mechanics for fluid-structure interaction
- A fracture mechanics solver volume-coupled to a dummy electro-chemistry corrosion model
- A 1D compressible fluid solver coupled to a 3D compressible fluid solver (uses deprecated version of Python bindings)

Couple your code

You want to couple your own code? In this section, you learn how to do that.

Do I need to be a preCICE expert to couple my own code?

No, not at all. Much more important is that you know the code you want to couple very well. Everything related to preCICE will come easy then.

Application programming interface

Coupling your own code means basically to work with the preCICE API (page 223). Go to this page for an overview of the available languages and some helpful links.

Are you just getting started to couple your code?

There is a step-by-step guide (page 225), which takes you through all necessary steps to couple your own code. We recommend that you first have a brief look at all steps before you start. Afterwards, you can really couple your code step by step: read what the next step is about and then implement it in your code.

Advanced topics

There is a list of advanced topics. These topics are not all relevant to every user. They deal with specific problems for certain types of codes: how to handle FEM meshes, how to handle moving meshes, etc.

You want to port your adapter to a newer preCICE version?

There is a specific page on porting adapters from preCICE 1.x to 2.x (page 250).

Application programming interface

Summary: This page gives an overview on available preCICE APIs and minimal reference implementations.

preCICE is written in C++. Thus, the native API language of preCICE is C++ as well. Its definite documentation is available on the preCICE doxygen. If you are new to the preCICE API, however, we recommended that you first follow the step-by-step guide (page 225).

Bindings

Besides the C++ API, there are also bindings to other languages available:

Language	Location	Installation
C	`precice/precice/tree/master/extras/bindings/c`	native bindings (page 35)
Fortran	`precice/precice/tree/master/extras/bindings/fortran`	native bindings (page 35)
Fortran Module	`precice/fortran-module`	`make` (page 42)
Python	`precice/python-bindings`	`pip3 install pyprecice` (page 43)
Matlab	`precice/matlab-bindings`	installation script (page 44)

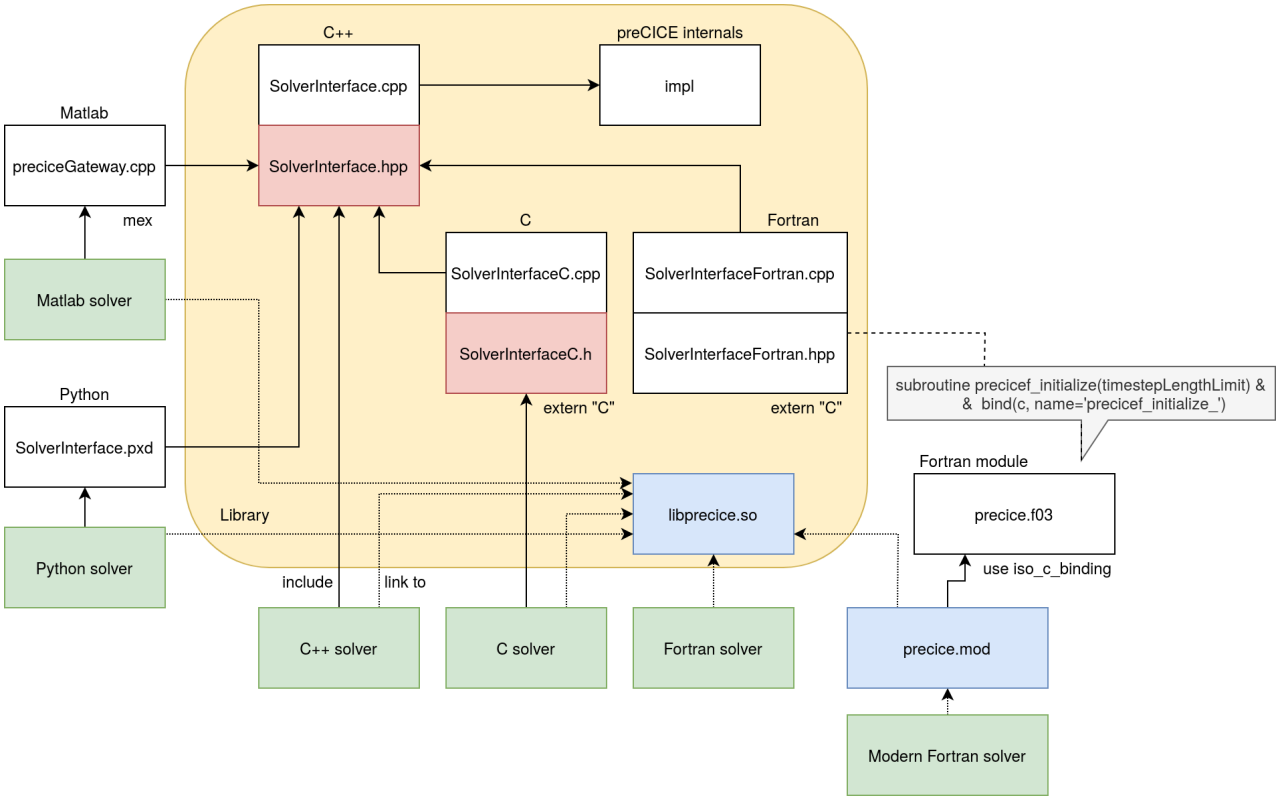
Minimal reference implementations

For all languages, we provide minimal reference implementations, so called *solver dummies*. They can be a great source to copy from.

Language	Location
C++	`precice/precice/examples/solverdummies/cpp`
C	`precice/precice/examples/solverdummies/c`
Fortran	`precice/precice/examples/solverdummies/fortran`
Fortran Module	`precice/fortran-module/examples/solverdummy`
Python	`precice/python-bindings/solverdummy`
Matlab	`precice/matlab-bindings/solverdummy`

Architectural overview of bindings

All the language bindings are calling the C++ API of preCICE and some of them are interdependent. Here is an overview of what uses what:



Step 1 – Preparation

> **Summary:** If you want to couple your own code you need to properly understand it. That is why, in this first step, we have a look at your own code. We discuss what you need to do to prepare the code for coupling.

Let's say you want to prepare a fluid solver for fluid-structure interaction and that your code looks like this:

```
turnOnSolver(); //e.g. setup and partition mesh double dt; // solver timestep size while (not simul
ationDone()){ // time loop dt = beginTimeStep(); // e.g. compute adaptive dt solveTimeStep(dt); end
TimeStep(); // e.g. update variables, increment time } turnOffSolver();
```

Probably most solvers have such a structures: something in the beginning (reading input, domain decomposition), a big time loop, and something in the end. Each timestep also falls into three parts: some pre-computations (e.g. computing an adaptive timestep size), the actual computation (solving a linear or non-linear equation system), and something in the end (updating variables, incrementing time). Try to identify these parts in the code you want to couple.

In the following steps, we will slowly add more and more calls to the preCICE API in this code snippet. Some part of the preCICE API is briefly described in each step. More precisely (no pun intended :grin:), we use the native `C++` API of preCICE. The API is, however, also available in other scientific programming languages: plain `C`, `Fortran`, `Python`, and `Matlab` (see Application Programming Interface (page 0)).

Tip: Also have a look at the definite C++ API documentation.

Note: This example refers to preCICE v2.x: see the differences from preCICE v1.x (page 250).

Step 2 – Steering methods

> **Summary:** In this step, you get to know the most important API functions of preCICE: initialize, advance, and finalize.

As a first preparation step, you need to include the preCICE library headers. In C++, you need to include the file SolverInterface.hpp. The handle to the preCICE API is the class `precice::SolverInterface`. Its constructor requires the participant's name, the preCICE configuration file's name and the `rank` and `size` of the current thread. Once the basic preCICE interface is set up, we can *steer* the behaviour of preCICE. For that we need the following functions:

```
SolverInterface( String participantName, String configurationFileName, int rank, int size ); double initialize(); double advance ( double computedTimestepLength ); void finalize();
```

What do they do?

- `initialize` establishes communication channels, sets up data structures of preCICE, and returns the maximum timestep size the solver should use next. But let's ignore timestep sizes for the moment. This will be the topic of Step 5 (page 232).
- `advance` needs to be called after the computation of every timestep to *advance* the coupling. As an argument, you have to pass the solver's last timestep size. Again, the function returns the next maximum timestep size you can use. More importantly, it maps coupling data between the coupling meshes, it communicates coupling data between the coupled participants, and it accelerates coupling data. One could say the complete coupling happens within this single function.
- `finalize` frees the preCICE data structures and closes communication channels.

So, let's extend the code of our fluid solver:

```
#include "precice/SolverInterface.hpp" turnOnSolver(); //e.g. setup and partition mesh precice::SolverInterface precice("FluidSolver","precice-config.xml",rank,size); // constructor double dt; // solver timestep size double precice_dt; // maximum precice timestep size precice_dt = precice.initialize(); while (not simulationDone()){ // time loop dt = beginTimeStep(); // e.g. compute adaptive dt dt = min(precice_dt, dt); // more about this in Step 5 solveTimeStep(dt); precice_dt = precice.advance(dt); endTimeStep(); // e.g. update variables, increment time } precice.finalize(); // frees data structures and closes communication channels turnOffSolver();
```

Step 3 – Mesh and data access

> **Summary:** In this step, we see how to define coupling meshes and access coupling data.

For coupling, we need coupling meshes. Let's see how we can tell preCICE about our coupling mesh. For the moment, we define coupling meshes only as clouds of vertices. In Step 8 (page 239), we will learn how to define mesh connectivity, so edges, triangles, and quads.

Coupling meshes and associated data fields are defined in the preCICE configuration file, which you probably already know from the tutorials. The concrete values, however, you can access with the API:

```
int getMeshID (const std::string& meshName); int setMeshVertex (int meshID, const double* positio
n); void setMeshVertices (int meshID, int size, double* positions, int* ids);
```

- `getMeshID` returns the ID of the coupling mesh. You need the ID of the mesh whenever you want to something with the mesh.
- `setMeshVertex` defines the coordinates of a single mesh vertex and returns a vertex ID, which you can use to refer to this vertex.
- `setMeshVertices` defines multiple vertices at once. So, you can use this function instead of calling `setMeshVertex` multiple times. This is also good practice for performance reasons.

To access coupling data, the following API functions are needed:

```
int getDataID (const std::string& dataName, int meshID); void writeVectorData (int dataID, int vert
exID, const double* value); void writeBlockVectorData (int dataID, int size, int* vertexIDs, doubl
e* values);
```

- `getDataID` returns the data ID for a coupling data field (e.g. "Displacements", "Forces", etc).
- `writeVectorData` writes vector-valued data to the coupling data structure.
- `writeBlockVectorData` writes multiple vector data at once, again for performance reasons.

Similarly, there are methods for reading coupling data: `readVectorData` and `readBlockVectorData`. Furthermore, preCICE distinguishes between scalar-valued and vector-valued data. For scalar data, similar methods exist, for example `writeScalarData`.

> **Note:** The IDs that preCICE uses (for data fields, meshes, or vertices) have arbitrary integer values. Actually, you should never need to look at the values. The only purpose of the IDs is to talk to preCICE. You also do not look at the value of a C pointer, it is just a non-readable address. In particular, you should not assume that vertex IDs are ordered in any certain way (say from 0 to N-1) or, for example, that 'Forces' always have the same ID '2' on all meshes.

Let's define coupling meshes and access coupling data in our example code:

```
turnOnSolver(); //e.g. setup and partition mesh precice::SolverInterface precice("FluidSolver","precice-config.xml",rank,size); // constructor int dim = precice.getDimensions(); int meshID = precice.getMeshID("FluidMesh"); int vertexSize; // number of vertices at wet surface // determine vertexSize double* coords = new double[vertexSize*dim]; // coords of coupling vertices // determine coordinates int* vertexIDs = new int[vertexSize]; precice.setMeshVertices(meshID, vertexSize, coords, vertexIDs); delete[] coords; int displID = precice.getDataID("Displacements", meshID); int forceID = precice.getDataID("Forces", meshID); double* forces = new double[vertexSize*dim]; double* displacements = new double[vertexSize*dim]; double dt; // solver timestep size double precice_dt; // maximum precice timestep size precice_dt = precice.initialize(); while (not simulationDone()){ // time loop precice.readBlockVectorData(displID, vertexSize, vertexIDs, displacements); setDisplacements(displacements); dt = beginTimeStep(); // e.g. compute adaptive dt dt = min(precice_dt, dt); solveTimeStep(dt); computeForces(forces); precice.writeBlockVectorData(forceID, vertexSize, vertexIDs, forces); precice_dt = precice.advance(dt); endTimeStep(); // e.g. update variables, increment time } precice.finalize(); // frees data structures and closes communication channels delete[] vertexIDs, forces, displacements; turnOffSolver();
```

Did you see that your fluid solver now also needs to provide the functions `computeForces` and `setDisplacements`? As you are an expert in your fluid code, these functions should be easy to implement. Most probably, you already have such functionality anyway. If you are not an expert in your code try to find an expert :smirk:.

Once your adapter reaches this point, it is a good idea to test your adapter against one of the solverdummies (page 0), which then plays the role of the `SolidSolver`.

You can use the following `precice-config.xml`:

```
<?xml version="1.0"?> <precice-configuration> <solver-interface dimensions="3"> <data:vector name="Forces"/> <data:vector name="Displacements"/> <mesh name="FluidMesh"> <use-data name="Forces"/> <use-data name="Displacements"/> </mesh> <mesh name="StructureMesh"> <use-data name="Forces"/> <use-data name="Displacements"/> </mesh> <participant name="FluidSolver"> <use-mesh name="FluidMesh" provide="yes"/> <use-mesh name="StructureMesh" from="SolidSolver"/> <write-data name="Forces" mesh="FluidMesh"/> <read-data name="Displacements" mesh="FluidMesh"/> <mapping:nearest-neighbor direction="write" from="FluidMesh" to="StructureMesh" constraint="conservative"/> <mapping:nearest-neighbor direction="read" from="StructureMesh" to="FluidMesh" constraint="consistent"/> </participant> <participant name="SolidSolver"> <use-mesh name="StructureMesh" provide="yes"/> <write-data name="Displacements" mesh="StructureMesh"/> <read-data name="Forces" mesh="StructureMesh"/> </participant> <m2n:sockets from="FluidSolver" to="SolidSolver"/> <coupling-scheme:serial-explicit> <participants first="FluidSolver" second="SolidSolver"/> <max-time-windows value="10" /> <time-window-size value="1.0" /> <exchange data="Forces" mesh="StructureMesh" from="FluidSolver" to="SolidSolver"/> <exchange data="Displacements" mesh="StructureMesh" from="SolidSolver" to="FluidSolver"/> </coupling-scheme:serial-explicit> </solver-interface> </precice-configuration>
```

Coupling flow

> **Summary:** Do you wonder why there is no `sendData` and `receiveData` in preCICE? Instead, there is simply `advance`. We call this a high-level API. On this page, you learn which advantages a high-level API has and how communication and control flow in preCICE works.

preCICE distinguishes between serial and parallel coupling schemes:

- **serial**: the participants run after one another,
- **parallel**: the participants run simultaneously.

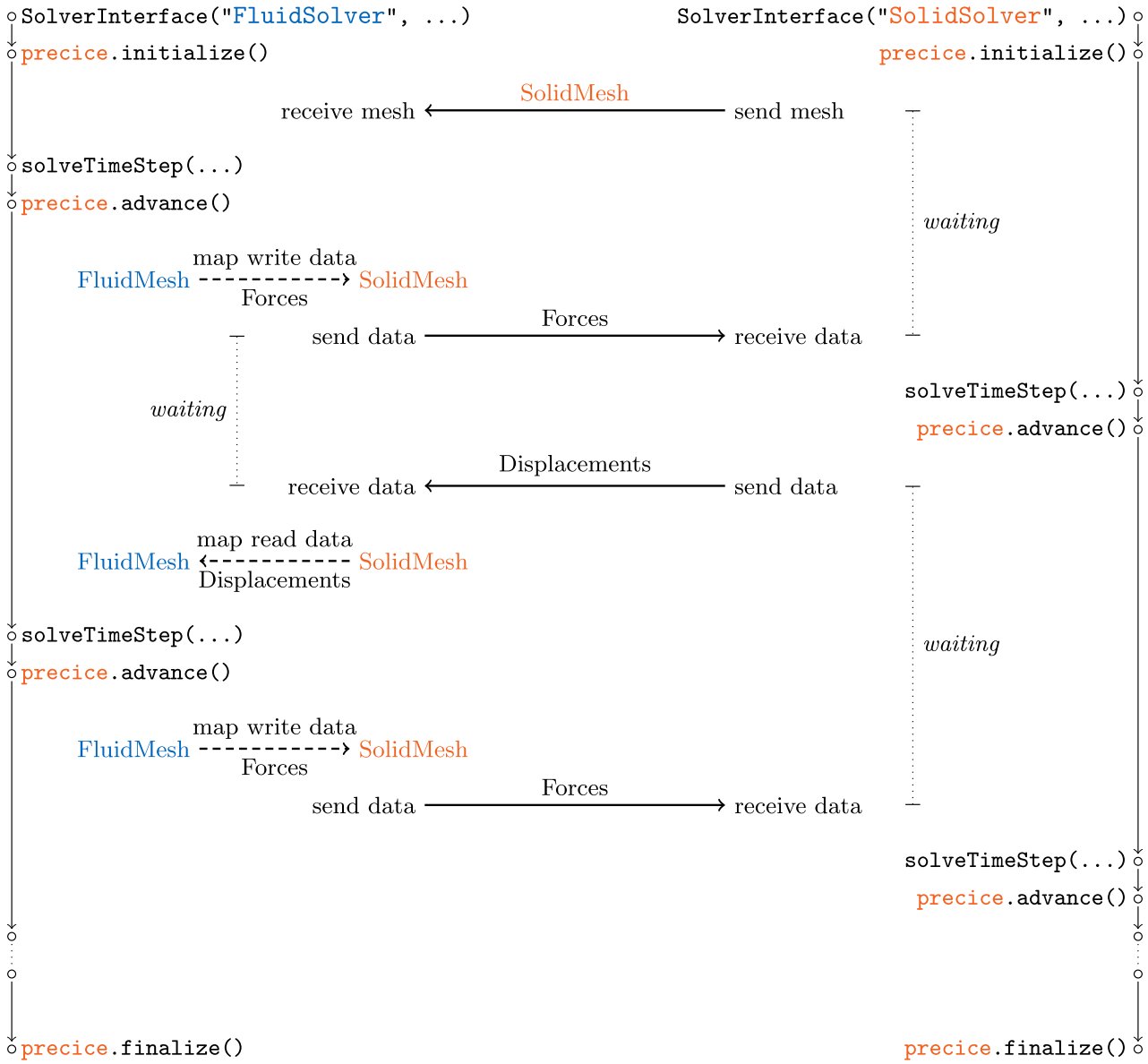
Serial coupling schemes

In our example, we currently use a serial coupling scheme:

```
<coupling-scheme:serial-explicit> <participants first="FluidSolver" second="SolidSolver"/> ... </co
upling-scheme:serial-explicit>
```

`FluidSolver` is first and `SolidSolver` second. This means that `FluidSolver` starts the simulation and computes the first timestep, while `SolidSolver` still waits. Where does it wait? Well, communciation in preCICE only happens within `initialize` and `advance` (and `initializeData`, but more about this in Step 7 (page 238)):

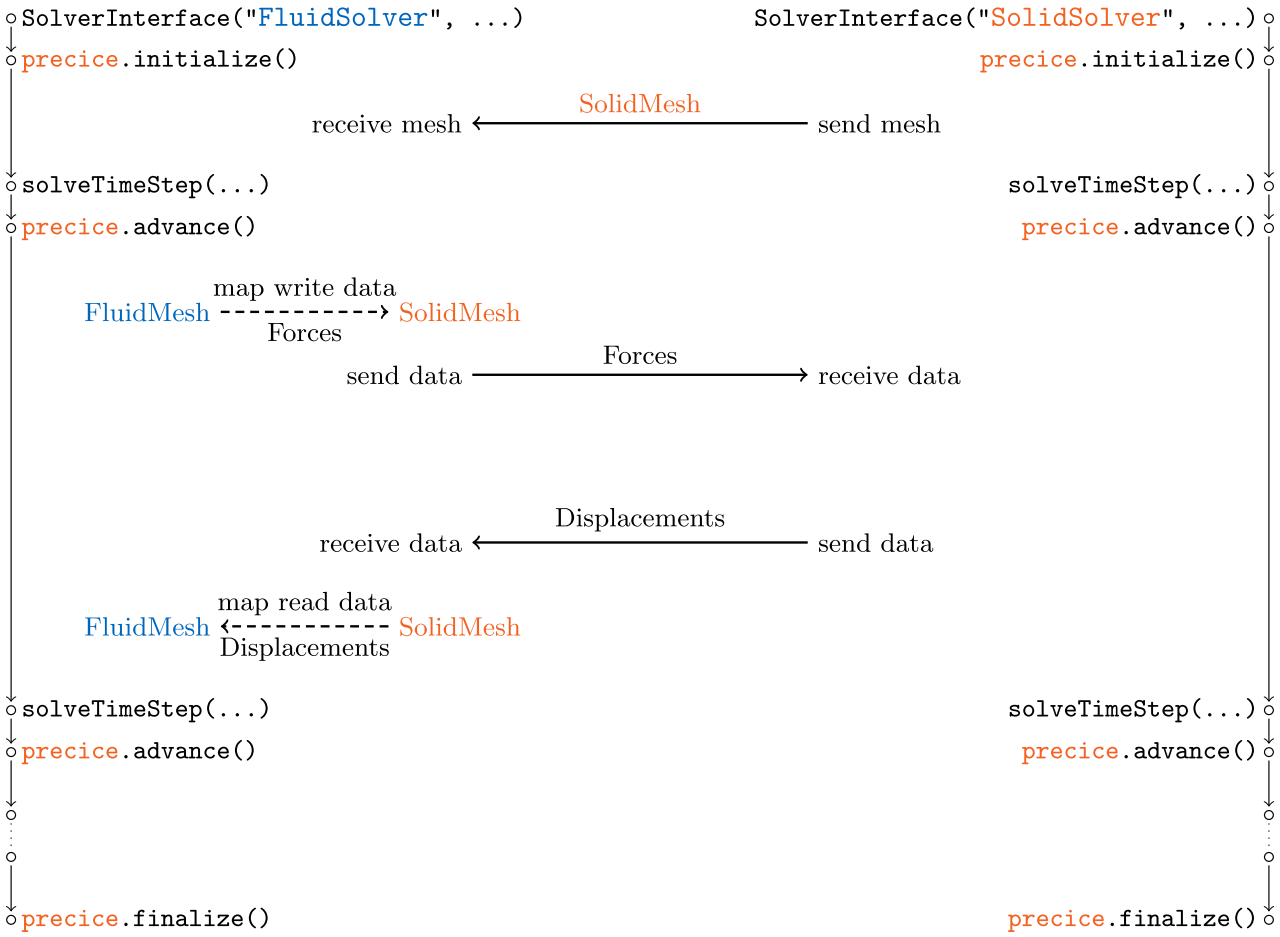
- `FluidSolver` computes the first timestep and then sends and receives data in `advance`. The receive call blocks.
- `SolidSolver` waits in `initialize` for the first data. When it receives the data it computes its first timestep and then calls `advance`.
- Now, `FluidSolver` receives data and `SolidSolver` blocks again.
- ...



Try to swap the roles of `first` and `second` in your example. Do you see the difference? If everything is just too fast, add some `sleep` calls.

Parallel coupling schemes

In a way, parallel coupling schemes are much easier here (numerically, they are not, but that's a different story). Everything is symmetric:



⚠ **Important:** The neat thing about the high-level API of preCICE is that you don't need to change anything in your code to switch between a serial and a parallel coupling scheme. This becomes even more important if you want to couple not only two participants, but three or more. The coupling logic, meaning who sends data to whom can be fully configured at runtime.

Step 5 – Non-matching timestep sizes

> **Summary:** In this step, you learn how preCICE handles non-matching timestep sizes and a few more things about simulation time.

In previous steps, you have already seen that there are quite some things going on with timestep sizes. Let us now have a look at what is actually happening.

```
... double dt; // solver timestep size double precice_dt; // maximum precice timestep size precic
e_dt = precice.initialize(); while (not simulationDone()){ // time loop dt = beginTimeStep(); //
e.g. compute adaptive dt dt = min(precice_dt, dt); solveTimeStep(dt); precice_dt = precice.advanc
e(dt); endTimeStep(); // e.g. update variables, increment time }
```

Good thing to know: in this step, you do really not have to alter your code. Everything needed is already there :relieved:. We now simply want to learn on what actually happens.

There are basically two options to choose from in the configuration.

```
<time-window-size value="..." method="..."/>
```

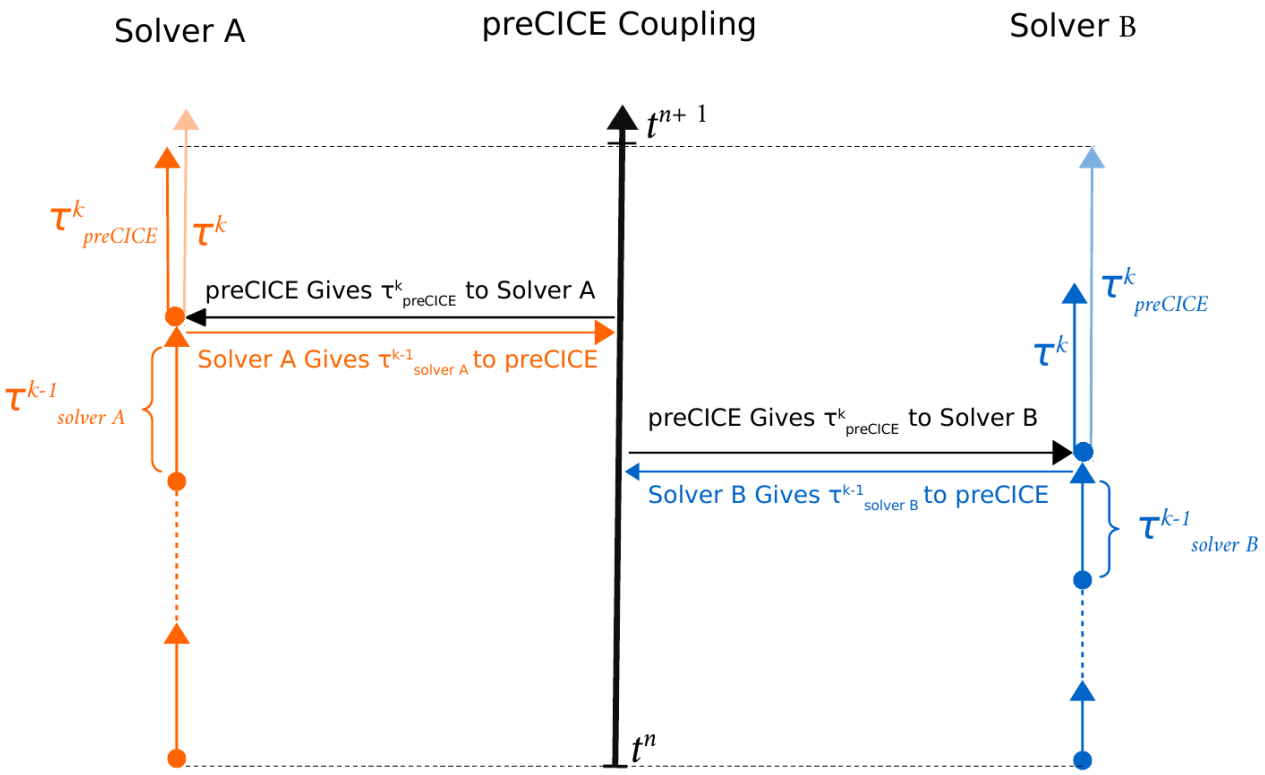
`method` can be:

- `fixed`: A fixed time window with size `value` is prescribed, during which both participants can use whatever timestep sizes they want. Communication of coupling data happens only after each time window.
- `first-participant`: The first participant prescribes the timestep size for the second one. Communication of coupling data happens after each timestep. This option is only available for serial coupling schemes (more about configuration of coupling schemes (page 71)). The attribute `value` is not applicable.

Let us have a closer look at both options.

Fixed time window

The preCICE configuration defines a fixed time window. Both participants can use smaller timestep sizes, but then they *subcycle*, i.e. coupling data is only communicated at the end of each time window. The figure below illustrates this procedure (k is the subcycling index, the dashed lines mark the time window):



- After each timestep, both participants tell preCICE which timestep size `dt` they just used. This way, preCICE can keep track of the total time. preCICE returns the remainder time to the next window.

```
precice_dt = precice.advance(dt);
```

- Both participants compute their next (adaptive) timestep size. It can be larger or smaller than the remainder.

```
dt = beginTimeStep();
```

If it is larger, the remainder `dt_precice` is used instead (orange participant, dark orange is used). If it is smaller, the participant's timestep size `dt` can be used (blue participant, dark blue is used). These two cases are reflected in:

```
dt = min(precice_dt, dt)
```

- Once both participants reach the end of the time window, coupling data is exchanged.

Note: This procedure is independent of whether a serial or a parallel coupling scheme is used. For parallel coupling, both solvers run together and everything happens simultaneously in both participants, while for serial coupling, the first participant needs reach the end of the window before the second one can start.

If a participant subcycles it is actually not necessary to write data to or read data from preCICE. To avoid unnecessary calls, preCICE offers two optional helper functions:

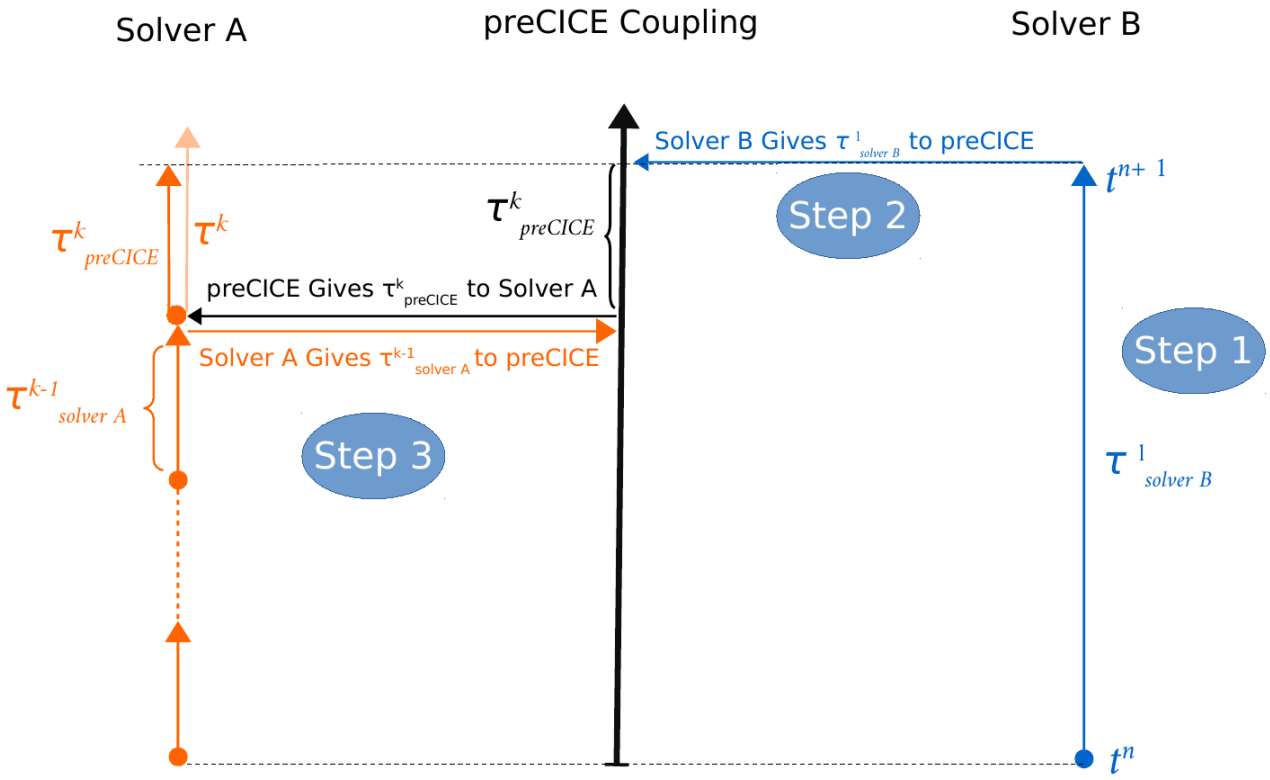
```
bool isReadDataAvailable () const; bool isWriteDataRequired (double computedTimestepLength) const;
```

You can use them as follows:

```
while (not simulationDone()){ // time loop if (precice.isReadDataAvailable()){ precice.readBlockVec
torData(displID, vertexSize, vertexIDs, displacements); setDisplacements(displacements); } dt = beg
inTimeStep(); // e.g. compute adaptive dt dt = min(precice_dt, dt); solveTimeStep(dt); if (precic
e.isWriteDataRequired(dt)){ computeForces(forces); precice.writeBlockVectorData(forceID, vertexSiz
e, vertexIDs, forces); } precice_dt = precice.advance(dt); endTimeStep(); // e.g. update variables,
increment time }
```

First participant prescribes timestep size

The `first` participant sets the timestep size. This requires that the `second` participant runs after the `first` one. Thus, as stated above, this option is only applicable for serial coupling.



- The blue participant B is `first` and computes a step with its timestep size (Step 1).
- In `advance`, this timestep size is given to preCICE (Step 2).

```
precice_dt = precice.advance(dt);
```

- preCICE has tracked the time level of the orange participant A and returns the remainder to reach B's timestep size.
- A computes its next (adaptive) timestep size. It can now be larger or smaller than the remainder.

```
dt = beginTimeStep();
```

If it is larger, the remainder `dt_precice` is used instead (the case below in Step 3, dark orange is used). If it is smaller, the participant's timestep size `dt` can be used (not visualized). These two cases are again reflected in the formula:

```
dt = min(precice_dt, dt)
```

- The procedure starts over with the blue participant B.

Note: `precice_dt` on the blue side is always infinity such that `min(dt,precice_dt)==dt`.

Important: You never need to alter your code if you want to switch the first and second participant, if you want to switch between a serial and a parallel coupling scheme, or if you want to switch between `fixed` and `first-participant` timestepping. Everything can be configured. Even if implicit coupling is used.

Steering the end of the simulation

One last thing about time. There is also a helper function in preCICE that allows you to steer the end of a coupled simulation:

```
bool isCouplingOngoing();
```

This functions looks at `max-time-windows` and `max-time` as defined in the preCICE configuration and knows when it is time to go. Then, you should call `finalize`. It replaces your `simulationDone()`.

```
while (not precice.isCouplingOngoing()){ // time loop ... } precice.finalize();
```

Step 6 – Implicit coupling

> **Summary:** In previous steps, we only considered explicit coupling. We now move onto implicit coupling, so sub-iterating each timestep multiple times until a convergence threshold is reached. This stabilzes strongly-coupled problems.

The main ingredient needed for implicit coupling is move backwards in time. For that, we need a flux capacitor. Just kidding :wink:. What we really need is that your solver can write and read iteration checkpoints. An iteration checkpoint should contain all the information necessary to reload a previous state of your solver. What exactly is needed depends solely on your solver. preCICE tells you when you need to write and read checkpoints. To this end, preCICE uses the following action interface:

```
bool isActionRequired(const std::string& action) void markActionFulfilled(const std::string& actio
n) const std::string& constants::actionReadIterationCheckpoint() const std::string& constants::acti
onWriteIterationCheckpoint()
```

- `isActionRequired` inquires the necessity of a certain action. It takes a string argument to reference the action.
- `markActionFulfilled` tells preCICE that the action is fulfilled. This is a simple safeguard. If a certain action is required and you did not mark it as fulfilled preCICE will complain.
- The Methods in the `precice::constants` namespace return strings to reference specific actions. For implicit coupling, we need `actionReadIterationCheckpoint` and `actionWriteIterationCheckpoint`.

Let's extend our example code to also handle implicit coupling.

```
turnOnSolver(); //e.g. setup and partition mesh precice::SolverInterface precice("FluidSolver","pre
cice-config.xml",rank,size); // constructor const std::string& coric = precice::constants::actionRe
adIterationCheckpoint(); const std::string& cowic = precice::constants::actionWriteIterationCheckpo
int(); int dim = precice.getDimension(); int meshID = precice.getMeshID("FluidMesh"); int vertexSiz
e; // number of vertices at wet surface // determine vertexSize double* coords = new double[vertexS
ize*dim]; // coords of vertices at wet surface // determine coordinates int* vertexIDs = new int[ve
rtexSize]; precice.setMeshVertices(meshID, vertexSize, coords, vertexIDs); delete[] coords; int dis
plID = precice.getDataID("Displacements", meshID); int forceID = precice.getDataID("Forces", meshI
D); double* forces = new double[vertexSize*dim]; double* displacements = new double[vertexSize*di
m]; double dt; // solver timestep size double precice_dt; // maximum precice timestep size
```

```
precice_dt = precice.initialize(); while (precice.isCouplingOngoing()){ if(precice.isActionRequire
d(cowic)){ saveOldState(); // save checkpoint precice.markActionFulfilled(cowic); } precice.readBlo
ckVectorData(displID, vertexSize, vertexIDs, displacements); setDisplacements(displacements); dt =
beginTimeStep(); // e.g. compute adaptive dt dt = min(precice_dt, dt); solveTimeStep(dt); computeFo
rces(forces); precice.writeBlockVectorData(forceID, vertexSize, vertexIDs, forces); precice_dt = pr
ecice.advance(dt); if(precice.isActionRequired(coric)){ // timestep not converged reloadOldState();
// set variables back to checkpoint precice.markActionFulfilled(coric); } else{ // timestep converg
ed endTimeStep(); // e.g. update variables, increment time } } precice.finalize(); // frees data st
ructures and closes communication channels delete[] vertexIDs, forces, displacements; turnOffSolve
r();
```

The methods `saveOldState` and `reloadOldState` need to be provided by your solver. You wonder when writing and reading checkpoints is required? Well, that's no black magic. In the first coupling iteration of each time window, preCICE tells you to write a checkpoint. In every iteration in which the coupling does not converge, preCICE tells you to read a checkpoint. This gets a bit more complicated if your solver subcycles (we learned this in Step 5 (page 0)), but preCICE still does the right thing. By the way, the actual convergence measure is computed in `advance` in case you wondered about that as well.

⚠ **Important:** Did you see that we moved the function `endTimeStep()` into the `else` block? This is to only move forward in time if the coupling converged. With this neat trick, we do not need two loops (a time loop and a coupling loop), but both are combined into one.

Of course, with the adapted code above, explicit coupling still works. You do not need to alter your code for that. In case of explicit coupling, both actions reading and writing iteration checkpoints simply always return `false`.

At this state, you can again test your adapted solver against a solver dummy (page 0). Make sure to adjust the config file for implicit coupling scheme:

```
[...] <coupling-scheme:serial-implicit> <participants first="FluidSolver" second="SolidSolver" />
<max-time-windows value="10" /> <time-window-size value="1.0" /> <max-iterations value="15" /> <rel
ative-convergence-measure limit="1e-3" data="Displacements" mesh="SolidSolver"/> <exchange data="Fo
rces" mesh="StructureMesh" from="FluidSolver" to="SolidSolver" /> <exchange data="Displacements" me
sh="StructureMesh" from="SolidSolver" to="FluidSolver"/> </coupling-scheme:serial-implicit> [...]
```

c ✅ **Tip:** For stability and faster convergence also use an acceleration method (page 0).

⚠ **Important:** You need to implement `saveOldState` and `reloadOldState` in such a way that a single coupling iteration becomes a proper function. Meaning, for two times the same input (the values you read from preCICE), the solver also needs to return two times the same output (the values you write to preCICE). Only then can the quasi-Newton acceleration methods work properly. This means, you need to include as much information in the checkpoint as necessary to really be able to go back in time. Storing complete volume data of all variables is the brute-force option. Depending on your solver, there might also be more elegant solutions. Be careful: this also needs to work if you jump back in time more than one timestep.

Step 7 - Data initialization

> **Summary:** As default values, preCICE assumes that all coupling variables are zero initially. For fluid-structure interaction, for example, this means that the structure is in its reference state. Sometimes, you want to change this behavior – for instance, you may want to restart your simulation.

For initializing coupling data, you can add the following **optional** method:

```
void initializeData();
```

Before jumping into the implementation, let's try to clarify how the usual the sequence of events in a serial and in a parallel coupling as studied in Step 4 (page 0) changes.

TODO: picture

In a serial coupling, only the second participant can send data inside `initializeData()`. In parallel coupling, both participants can initialize data.

The high-level API of preCICE makes it possible to enable this feature at runtime, irrelevant of serial or parallel coupling configuration. To support this feature, we extend our example as follows:

```
[...] const std::string& cowid = precice::constants::actionWriteInitialData(); [...] int displID = precice.getDataID("Displacements", meshID); int forceID = precice.getDataID("Forces", meshID); double* forces = new double[vertexSize*dim]; double* displacements = new double[vertexSize*dim]; [...] precice_dt = precice.initialize(); if(precice.isActionRequired(cowid)){ precice.writeBlockVectorData(forceID, vertexSize, vertexIDs, forces); precice.markActionFulfilled(cowid); } precice.initializeData(); while (precice.isCouplingOngoing()){ [...]
```

Now, you can specify at runtime if you want to initialize coupling data. For example to initialize displacements:

```
[...] <exchange data="Forces" mesh="StructureMesh" from="FluidSolver" to="SolidSolver" /> <exchange data="Displacements" mesh="StructureMesh" from="SolidSolver" to="FluidSolver" initialize="yes"/> [...]
```

Step 8 – Mesh connectivity

> **Summary:** So far, our coupling mesh is only a cloud of vertices. This is sufficient for most of the numerical methods that preCICE offers. For some features, however, preCICE also needs to know how vertices are connected to each other. In this step, you learn how to define this so-called mesh connectivity.

The most important example where mesh connectivity is needed is the `nearest-projection` mapping, where the mesh we project *into* needs mesh connectivity. For a consistent mapping, this is the mesh *from* which you map. For a conservative mapping, the mesh *to* which you map. More information is given on the mapping configuration page (page 0).

In 2D, mesh connectivity simply means to define edges between vertices. In 3D, you need to define triangles and / or quads. Both, we can either build up from edges or directly from vertices.

```
int setMeshEdge (int meshID, int firstVertexID, int secondVertexID); void setMeshTriangle (int mesh
ID, int firstEdgeID, int secondEdgeID, int thirdEdgeID); void setMeshTriangleWithEdges (int meshID,
int firstVertexID, int secondVertexID, int thirdVertexID); void setMeshQuad(int meshID, int firstEd
geID, int secondEdgeID, int thirdEdgeID, int fourthEdgeID); void setMeshQuadWithEdges(int meshID, i
nt firstVertexID, int secondVertexID, int thirdVertexID, int fourthVertexID);
```

- `setMeshEdge` defines a mesh edge between two vertices and returns an edge ID.
- `setMeshTriangle` defines a mesh triangle by three edges.
- `setMeshTriangleWithEdges` defines a mesh triangle by three vertices and also creates the edges in preCICE on the fly. Of course, preCICE takes care that no edge is defined twice. Please note that this function is computationally more expensive than `setMeshTriangle`.
- `setMeshQuad` defines a mesh quad by four edges.
- `setMeshQuadWithEdges` defines a mesh quad by four vertices and also creates the edges in preCICE on the fly. Again, preCICE takes care that no edge is defined twice. This function is computationally more expensive than `setMeshQuad`.

If you do not configure any features in the preCICE configuration that require mesh connectivity, all these API functions are no-ops. Thus, don't worry about performance.

Maybe interesting to know: preCICE actually does internally not compute with quads, but creates two triangles. Read more.

> **Warning:** Quads are only supported since v2.1. For older version, the methods only exist as empty stubs.

The following code shows how mesh connectivity can be defined in our example. For sake of simplification, let's only define one triangle and let's assume that it consists of the first three vertices.

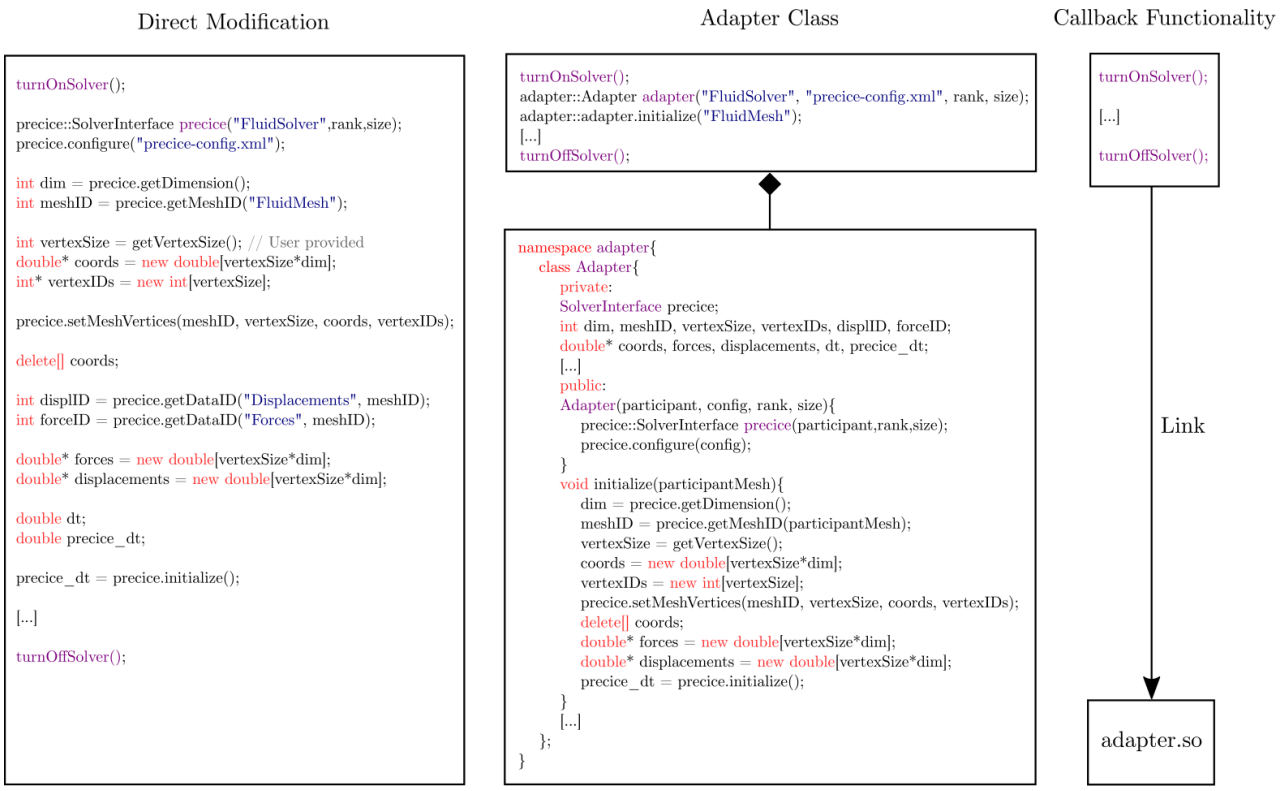
```
[...] int* vertexIDs = new int[vertexSize]; precice.setMeshVertices(meshID, vertexSize, coords, ver
texIDs); delete[] coords; int edgeIDs[3]; edgeIDs[0] = precice.setMeshEdge(meshID, vertexIDs[0], ve
rtexIDs[1]); edgeIDs[1] = precice.setMeshEdge(meshID, vertexIDs[1], vertexIDs[2]); edgeIDs[2] = pre
cice.setMeshEdge(meshID, vertexIDs[2], vertexIDs[0]); if(dim==3) precice.setMeshTriangle(meshID, ed
geIDs[0], edgeIDs[1], edgeIDs[2]); [...]
```

Adapter software engineering

Summary: The example developed in the step-by-step guide is a rather intrusive way of writing an adapter as we directly modify the main solver routines. This page discusses better software engineering approaches.

What we develop in the step-by-step guide (page 0) is best described as an *adapted code*, not an *adapter*. We directly modified the main solver routines. Better alternatives exist. Depending on the solver's functionalities, you could either consider creating a separate class for the preCICE adapter (as applied e.g. in the SU2 adapter) or using a callback functionality provided by the solver (as applied e.g. in the OpenFOAM adapter).

The diagram below summarizes these three different ways of using preCICE:



The direct modification approach is what the step-by-step guide (page 0) uses. It consists of directly modifying the solver code lines to couple with preCICE. This, however, is not an ideal approach since it requires changing of the solver source code, and you should try to avoid this to wherever possible to maintain a sustainable software development practice.

To minimize the lines of solver source code changes, you can create a separate class for the adapter. This adapter will be responsible for calling all the preCICE APIs, and from the solver source code you would only call the corresponding adapter class methods. As stated above, this approach is used e.g. for the SU2-adapter and explained in detail in Alexander Rusch's thesis.

If the solver you are using provides a callback functionality you can separate the adapter even better from the code. You simply call the preCICE API from the callback. As stated above this is realized in the OpenFOAM adapter and is explained in detail in Gerasimos Chourdakis' thesis.

Initialization in existing MPI environment

Summary: preCICE uses MPI for communication between different participants (and also for communication between ranks of the same participant). So are there any problems if the solver that you intend to couple also already uses MPI (e.g. for parallelization)? Who should initialize MPI? Who should finalize MPI? This is what we discuss here.

It is not complicated. There are just three rules that preCICE follows:

- preCICE only initializes MPI if it is not yet initialized (by e.g. the solver you want to couple).
- preCICE finalizes MPI if and only if it was also initialized by preCICE.
- preCICE only initializes MPI if it needs MPI.

So what does this mean for your adapter code:

- Initialize preCICE after you initialize MPI.
- Finalize preCICE before you finalize MPI.

```
[...] // start up your solver MPI_Init(NULL, NULL); int world_rank, world_size; MPI_Comm_rank(MPI_C
OMM_WORLD, &world_rank); MPI_Comm_size(MPI_COMM_WORLD, &world_size); [...] // maybe more initializa
tion precice::SolverInterface precice("SolverName", world_rank, world_size); precice.configure("pre
cice-config.xml"); [...] // declare meshes vertices etc. double precice_dt = precice.initialize();
[...] // solving and coupling precice.finalize(); [...] // more finalization MPI_Finalize();
```

Dealing with moving meshes

Summary: TODO

TODO

Dealing with FEM meshes

Summary: TODO

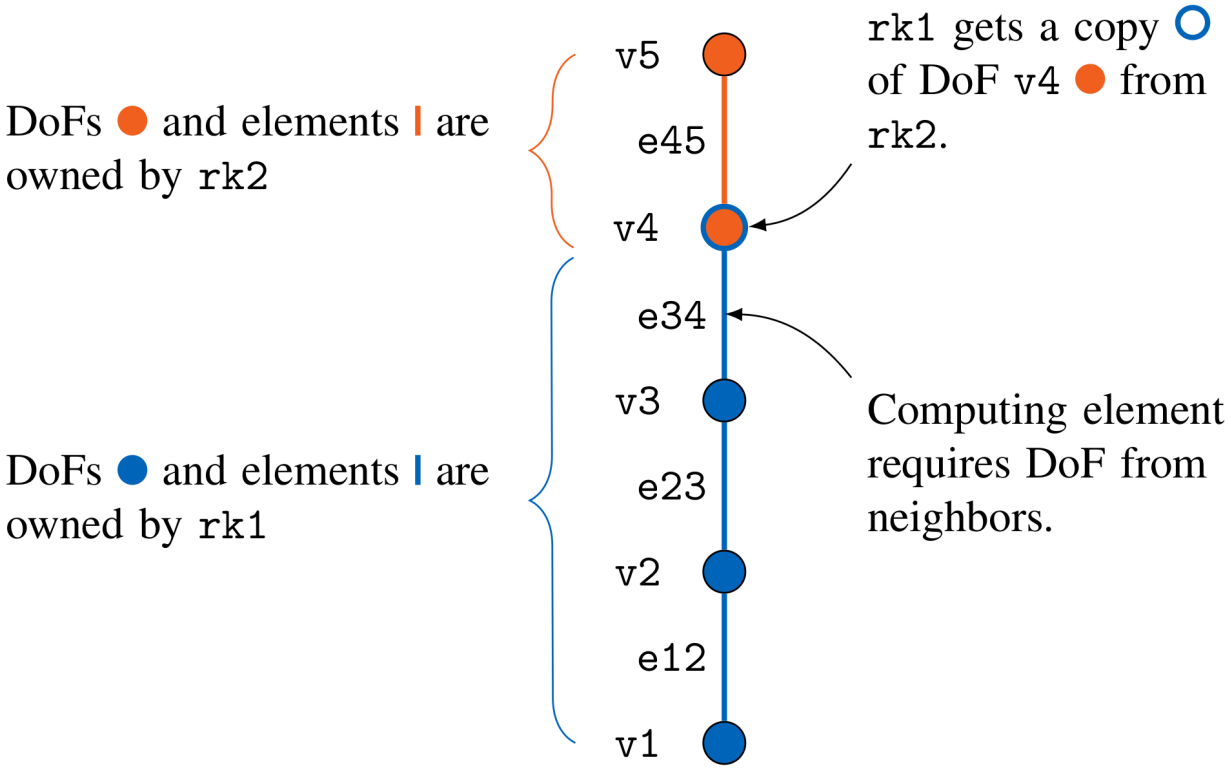
TODO

Dealing with distributed meshes

Summary: As preCICE is designed for HPC, adapter developers often have to deal with distributed meshes. There is no golden bullet how to best handle distributed meshes with preCICE. On this page, we compare different approaches.

General setup

We will focus on distributed meshes as they are often used in parallelized finite element codes, such as FEniCS or deal.II. In a distributed memory parallelization, the ranks usually only own a fraction of the mesh. At the interface between two partitions of the mesh vertices or elements owned by one rank usually have to be communicated to another one in order to make sure that all the information needed for the computations is available.

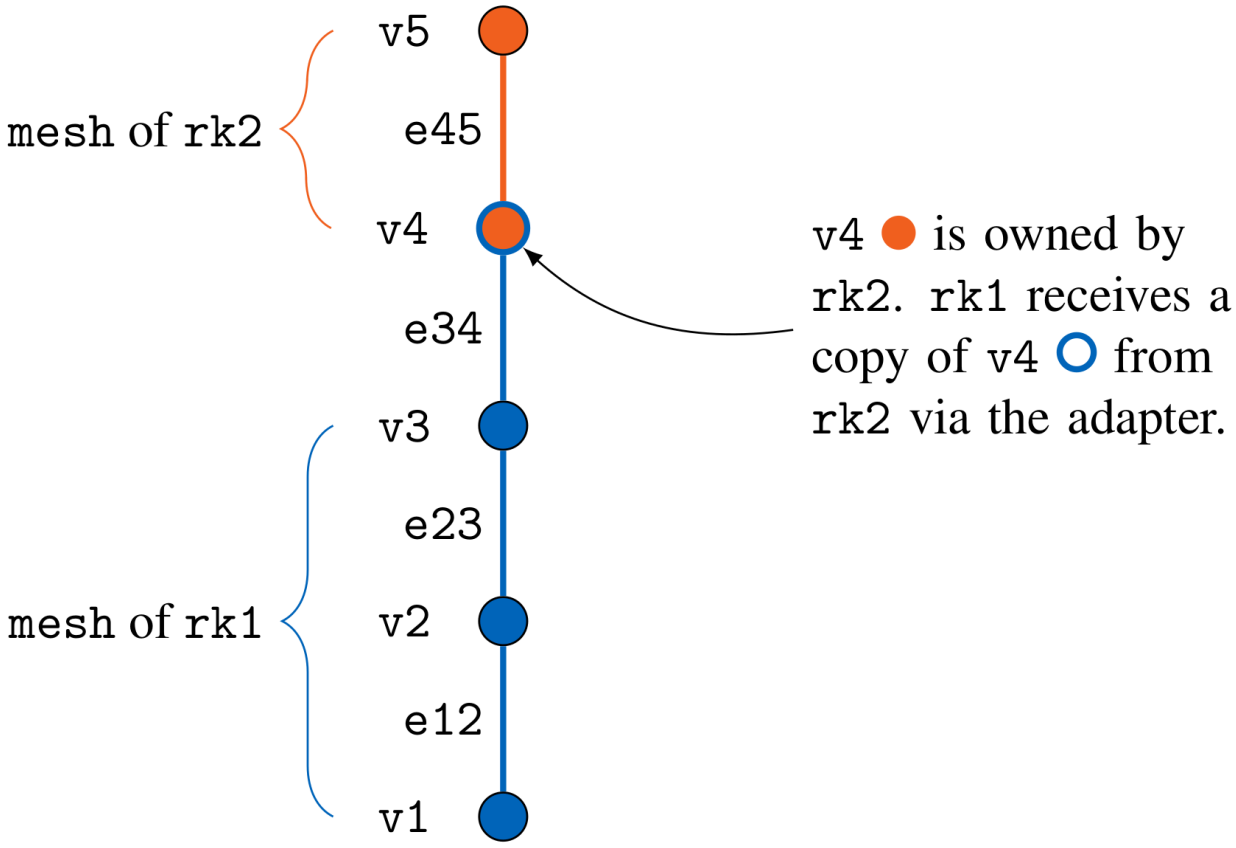


This distributed mesh setup is not only relevant for the internal degrees of freedom, but also for the coupling mesh of preCICE: The ranks have to call `precice::setMeshVertex(...)` or `precice::setMeshVertices(...)` to define the coupling mesh. In the following we want to discuss strategies how preCICE can be used in such a situation and how we can deal with the need for duplicate vertices.

Use a single `mesh` and communicate values for copied vertices inside adapter

In this approach we do not define any copied vertices in preCICE, but only the vertices owned by a rank. Therefore, each vertex is globally only defined once via `precice::setMeshVertex(...)`. The rank that owns the vertices uses the read and write functions of preCICE (`precice::readBlockScalarData(...)` and `precice::writeBlockScalarData(...)`) to update the coupling data on the mesh.

Note that it might be required to add another communication step inside the adapter or the solver to synchronize the data on the copied vertices among ranks.



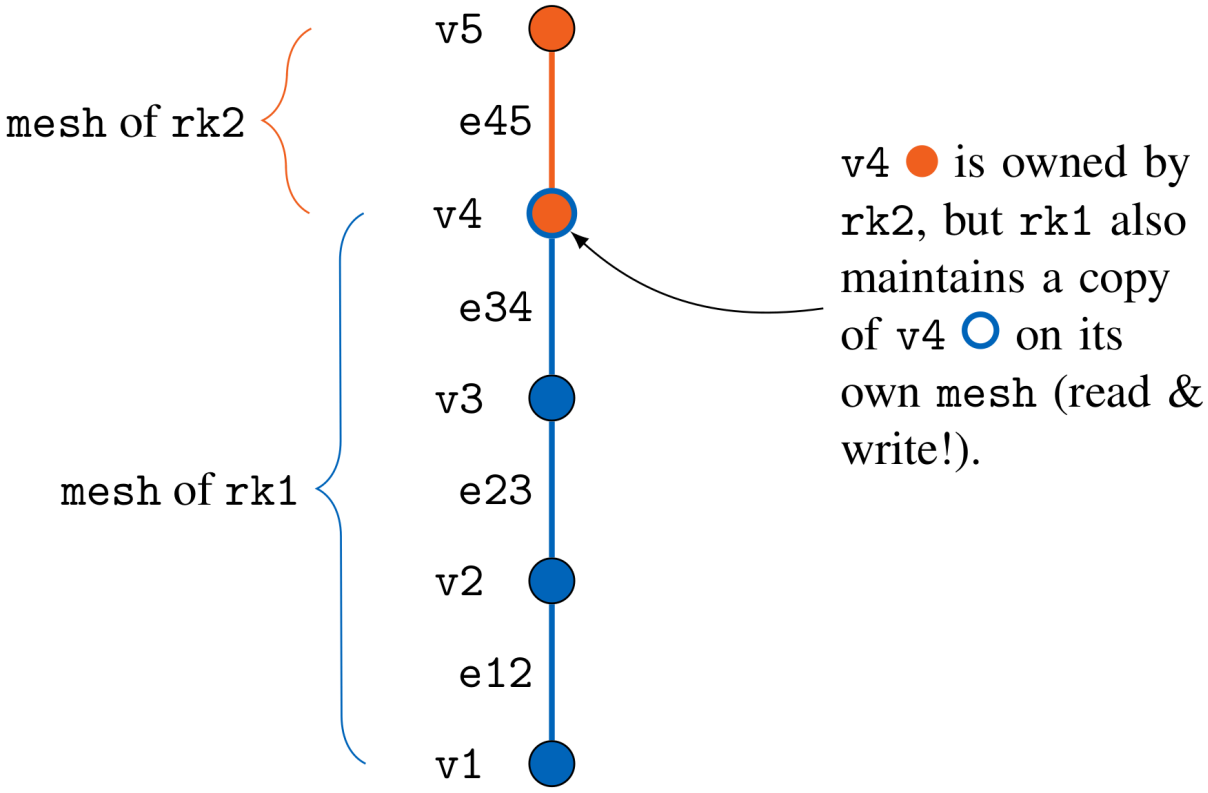
Discussion of this approach:

- Only one mesh is needed.
- Same `precice-config.xml` as for serial case.
- Additional communication step after preCICE communication is complete might be required.
- Mesh connectivity information is restricted to vertices owned by the rank. Therefore `precice::setMeshEdge(...)` cannot be called for edges that cross the border between two ranks.

Use a single `mesh` and duplicate copied vertices

Each rank can only access the vertices that it has previously defined. Therefore, in this approach we have to call `precice::setMeshVertex(...)` for all vertices owned by the rank **and** for vertices where the rank requires access to a copy, since we will have to read coupling data from these vertices, as well. Note that we will additionally have to write data to copied vertices, since they are equal to owned vertices from the perspective of preCICE.

Since we have to write duplicate vertices, it becomes especially important to make sure that the values written by the rank that owns the vertices and the rank(s) where the vertices are only a copy of the original vertex are written correctly: If a conservative mapping is used, only a single rank (usually the rank that owns the vertex) is allowed to write the updated values to preCICE, since otherwise the mapping of preCICE will cause the actual value to be a multiple of the “true” result. If a consistent mapping is used, all ranks that define the vertex also have to write the “true” value to it, since otherwise the result will be a combination of the “true” value and zeroes originating from the ranks owning copies of the vertex.



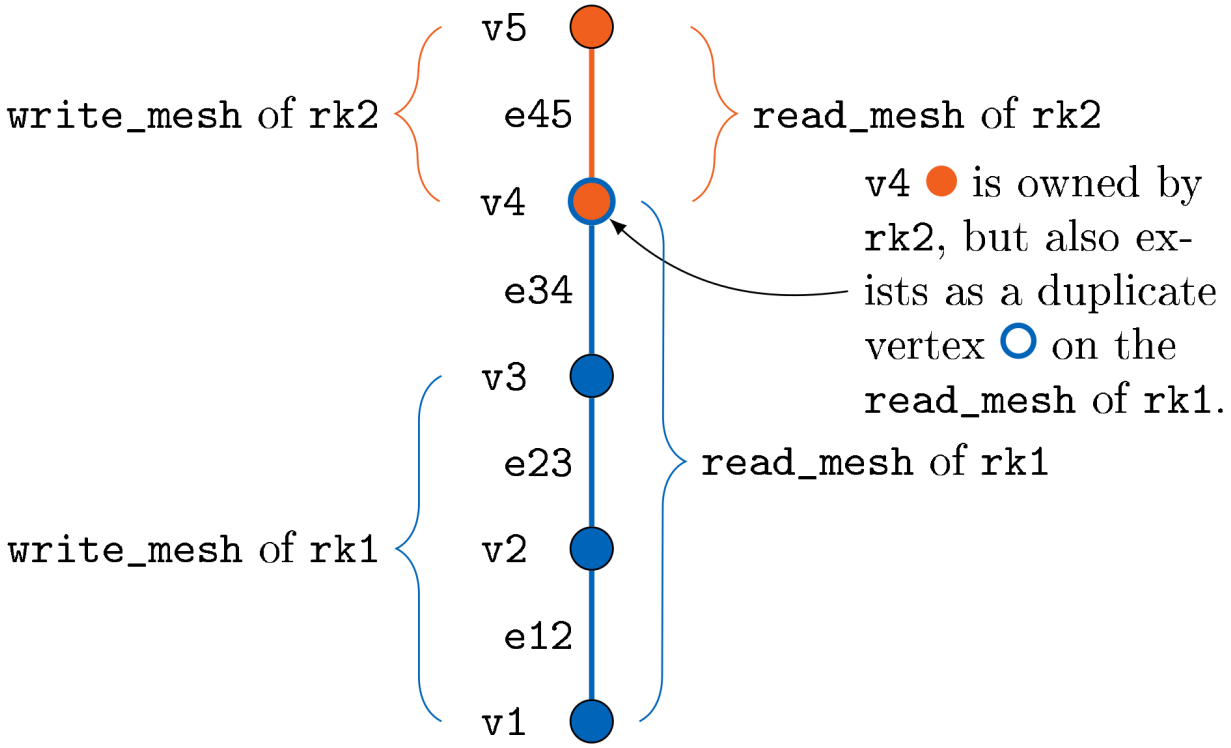
Discussion of this approach:

- Only one mesh is needed.
- Same `precice-config.xml` as for serial case can be used.
- User must call write function for owned **and** copied vertices correctly depending on the mapping technique used to avoid mistakes.
- Mesh connectivity information is available for vertices owned by the rank and the direct neighborhood. The adapter can `precice::setMeshEdge(...)` for edges that cross the border between two ranks.

Define two separate meshes as `read_mesh` and `write_mesh`

We create a `write_mesh` where we call `precice::setMeshVertex(...)` *only* for the vertices owned by the rank. We do no add any copies of vertices to the `write_mesh`, since they are owned by another rank and only the rank having ownership is allowed to write values (e.g. via `precice::writeBlockScalarData(...)`) to vertices on that mesh.

Additionally, we create a `read_mesh`, where we call `precice::setMeshVertex(...)` for vertices owned by the rank *and* vertices where a copy is required. This allows the rank to read the values for owned as well as for copied vertices (e.g. via `precice::readBlockScalarData(...)`).



Discussion of this approach:

- User has to deal with two meshes.
- `precice-config.xml` becomes more complex than for the serial case.
- The mapping of preCICE is taking care of providing information for copied vertices to ranks that would otherwise not be able to access these vertices.
- Mesh connectivity information is restricted to vertices owned by the rank. Therefore `precice::setMeshEdge(...)` cannot be called for edges that cross the border between two ranks.

Direct access to received meshes

Summary: You can access received meshes and their data directly by using specific optional API functions.

Warning: These API functions are work in progress, experimental, and are not yet released. The API might change during the ongoing development process. Use with care.

This concept is required if you want to access received meshes directly. It might be relevant in case you don't want to use the mapping schemes in preCICE, but rather want to use your own solver for data mapping. As opposed to the usual preCICE mapping, only a single mesh (from the other participant) is now involved in this situation since an 'own' mesh defined by the participant itself is not required any more. In order to re-partition the received mesh, the participant needs to define the mesh region it wants read data from and write data to. The complete concept on the receiving participant looks as follows:

```
// Allocate a bounding-box vector containing lower and upper bounds per // space dimension std::vec
tor<double> boundingBox(dim * 2); // fill the allocated 'boundingBox' according to the interested r
egion // with the desired bounds... // Get relevant IDs. Note that "ReceivedMeshname" is not a name
of a // provided mesh, but a mesh defined by another participant. Accessing // a received mesh dire
ctly is disabled in a usual preCICE configuration. const int otherMeshID = precice.getMeshID("Recei
vedMeshName"); const int writeDataID = precice.getDataID("WriteDataName", otherMeshID); // Define r
egion of interest, where we want to obtain the direct access. // See also the API documentation of
this function for further notes. precice.setMeshAccessRegion(otherMeshID, boundingBox.data()); // i
nitialize preCICE as usual double dt = precice.initialize(); // Get the size of the received mesh p
artition, which lies within the // defined bounding (provided by the coupling participant) const in
t otherMeshSize = precice.getMeshVertexSize(otherMeshID); // Now finally get the data. First alloca
te memory for the IDs and the // vertices std::vector<double> otherSolverVertices(otherMeshSize * d
im); std::vector<int> ids(otherMeshSize); // ... and afterwards ask preCICE to fill the vectors pre
cice.getMeshVerticesAndIDs(otherMeshID, otherMeshSize, ids.data(), otherSolverVertices.data()); //
continue with time loop and write data directly using writeDataID and // the received ids, which co
rrespond to the vertices
```

Concept and API

Defining a bounding box for serial runs of the solver (not to be confused with serial coupling mode) is valid. However, a warning is raised in case vertices are filtered out completely on the receiving side, since the associated data values of the filtered vertices are filled with zero data values in order to make the 'read' operation of the other participant valid.

In order to use the feature, it needs to be enabled explicitly in the configuration file. Using the same data and mesh names as in the code example above, a corresponding configuration would be

```
... <participant name="MyParticipant"> <use-mesh name="ReceivedMeshName" from="OtherParticipant" di
rect-access="true" /> <write-data name="WriteDataName" mesh="ReceivedMeshName" /> </participant>
...
```

Note that we write the data on a mesh we received and no mapping and no mesh need to be defined as opposed to the usual case. If you want to read data on a provided mesh additionally, a mesh can (and must) be provided, as usual. Note also that you probably need to reconfigure the mesh, which is used for the data exchange (`<exchange`

`data=..`), the data acceleration and convergence measure within the coupling scheme. Minimal configuration examples can also be found in the integration tests located in the preCICE repository `precice/src/precice/tests`. All relevant test files have '`direct-access`' in the file name, e.g. `explicit-direct-access.xml`.

Tip: A more application-oriented configuration, where both solver make use of this feature can be found in this deal.II example

Using the feature in parallel

Using the described concept in parallel computations requires some additional considerations. In particular, it is important to note that the geometric description of the domain via axis-aligned bounding-boxes is not exact. Thus, the resulting partitioning usually leads to overlapping regions between the individual rank partitions. If additional mappings on a directly accessed mesh are desired, overlapping partitions might even be necessary in order to compute the global mapping. preCICE does not know which rank finally writes data to or reads data from which vertices. Hence, preCICE performs a *sum over all data* it receives for a particular vertex. If you have overlapping regions, i.e., vertices are unique on a rank, but duplicated across all ranks, the contribution of each rank is summed up and finally passed to the other participant. It is the responsibility of the user to make sure that data is only written on a single rank or summing up data is actually desired since the data is conservative (e.g. summing up a force contribution across all ranks). An exemplary implementation of this feature, which works in parallel, is given in the deal.II example (a documentation of the implementation is given in the source code itself). There, a consensus algorithm, which selects the lowest rank for duplicated vertices across several ranks, is responsible for writing the data, all other ranks do not write data for the particular point.

Porting adapters from preCICE 1.x to 2.x

Summary: This guide helps you to upgrade from preCICE 1.x to preCICE 2.x.

We use semantic versioning for preCICE, which means that you can extract useful information from the version number. If the first digit (major version) does not change, this means that you don't need to update your adapter or (usually) your preCICE configuration file. However, when the major version number increases, this means that you need to update your code as well (we plan for a major version change once every 2-3 years). We recommend using the latest stable versions of preCICE and the corresponding bindings and adapters.

preCICE API

Single-step setup

This is described in detail in #614 and was done to simplify the setup. Change:

```
- SolverInterface interface(solverName, commRank, commSize); - interface.configure(configFileName);
+ SolverInterface interface(solverName, configFileName, commRank, commSize);
```

Typical error message that should lead you here:

```
error: no matching function for call to 'precice::SolverInterface::SolverInterface(std::__cxx11::st
ring&, int, Foam::label)' precice_ = new precice::SolverInterface(participantName_, Pstream::myProc
No(), Pstream::nProcs());
```

and

```
note: candidate: precice::SolverInterface::SolverInterface(const string&, const string&, int, int)
SolverInterface( ^ .../SolverInterface.hpp:52:3: note: candidate expects 4 arguments, 3 provided
```

Sorted out duplicate meaning of timestep

- Renamed API function `isTimestepComplete` to `isTimeWindowComplete` (#619)

Clarified `fulfilledAction`

- `fulfilledAction` was renamed to `markActionFulfilled` (#631)

Language bindings

C

- Moved to `extras/bindings/c`.
- Separated into `include` (header) and `src` (implementation).
- Renamed `precicec_isCouplingTimestepComplete` to `precicec_isTimeWindowComplete`.

Fortran

- Moved the intrinsic Fortran bindings to `extras/bindings/fortran`.
- Renamed the "Fortran 2003 bindings" to "Fortran module" and `Precice_solver_if_module` to `precice`.

Moved them to precice/fortran-module.

Python

- python bindings migration guide

preCICE configuration file

- Renamed `mapping:petrbf` to `mapping:rbf` (see #572).
- Remove `master:mpi-single` tags.
  preCICE defaults to `master:mpi-single` for parallel participants (see #572).
- Remove `distribution-type="..."` from `m2n` tags.
  It now defaults to `point-to-point`, use the attribute `enforce-gather-scatter=1` if this is not desired (see #572).
- Renamed `coupling-scheme` configuration option `timestep-length` to `time-window-size`
- Renamed `coupling-scheme` configuration option `max-timesteps` to `max-time-windows`
- Renamed `post-processing` to `acceleration`
- Renamed `acceleration` configuration option `timesteps-reused` to `time-windows-reused`
- Renamed `acceleration` configuration option `reused-timesteps-at-restart` to `reused-time-windows-at-restart`
- Renamed `export` configuration option `timestep-interval` to `every-n-time-windows`
- Renamed `action` configuration option `on-timestep-complete-post` to `on-time-window-complete-post`

Building

- Renamed CMake variables (#609)
  - `MPI` to `PRECICE_MPICommunication`
  - `PETSC` to `PRECICE_PETScMapping`
  - `PYTHON` to `PRECICE_PythonActions`
- CMake `CMAKE_BUILD_TYPE` is automatically set to `Debug`, if empty
- CMake variables for enabling C and Fortran
- Removed SCons completely. You can do everything and much more with CMake (page 30).

Side-changes

All the tutorials are adapted for preCICE v2. You can still find a version compatible with preCICE v1.6.1 here.

Most of the adapters (all apart from the “under initial development” ones) are only supporting the latest version of preCICE. Make sure also that you are using the correct branch: an adapter’s `develop` is supposed to work with preCICE `develop`, and similar for `master`.

At the same time as preCICE v2, we also changed the configuration format of the OpenFOAM adapter: Instead of a yaml file, it is now an OpenFOAM dictionary, making installation even easier. Please refer to the Configuration (page 174) page for more details.