preCICE

User's Guide

July 17, 2013

Contents

1 Introduction

preCICE stands for **pre**cise **c**ode **i**nteraction **c**oupling **e**nvironment. It has been written to facilitate partitioned coupled simulations, where partitioned is understood as a subdivisioning into separate systems of equations, surface coupled spatial discretization grids, and simulation softwares according to physical subproblems. preCICE has been written to deal with surface coupled problems, however, it is possible to perform volume coupling, too.

Another main task of preCICE is to support simulation programs using hexahedral grids. Surface representations of geometries read into preCICE can be queried to obtain positional (inside, outside, on geometry) and related information for points and hexahedrons (or rectangles in 2D). This functionality can be combined with the coupling functionality such that the surface representation is identical to the coupling surface.

This user's guide describes how to use preCICE to perform partitioned simulations, or as geometry interface.

Back to Contents.

2 Preparations

In order to use preCICE, first, a library has to be created from its source code. The solvers need to use the application programming interface (API) of preCICE in order to access coupling functionality, and the solver executables (or their interfaces for user programming) need to be linked to the preCICE library. This section describes how to accomplish this.

Back to Contents.

2.1 Build Prerequesits under Linux

- Obtain a current version of preCICE (http://www5.in.tum.de/wiki/index.php/PreCICE_Download).
- The Boost C++ libraries (http://www.boost.org/) need to be available (no build or installation is necessary). To choose a suitable version of Boost, please check out the preCICE release notes. To make the location of Boost available to preCICE, set the environment variable `BOOST_ROOT` to name the path to the Boost libraries root directory.
- In order to build preCICE, Python (version smaller than 3.0, http://www.python.org/) and SCons (http://www.scons.org/) need to be installed. If the Python extensions are intended to be used, NumPy (http://numpy.scipy.org/) needs to be installed in addition.
- If communication via MPI is required, a suitable implementation of the MPI2.0 standard is required to be installed (http://www.mcs.anl.gov/research/projects/mpich2/, e.g.).
- The GNU g++/gcc compiler (http://www.mingw.org/) needs to be available.

Back to Contents.

2.2 Build Prerequesits under Windows

- Obtain a current version of preCICE (http://www5.in.tum.de/wiki/index.php/PreCICE_Download).
- The Boost C++ libraries (http://www.boost.org/) need to be available, and the system and thread libraries need to be built when using socket communication. To choose a suitable version of Boost, please check out the preCICE release notes. To make the location of Boost available to preCICE, set the environment variable `BOOST_ROOT` to name the path to the Boost libraries root directory.
- In order to build preCICE, Python (version smaller than 3.0, http://www.python.org/) and SCons (http://www.scons.org/) need to be installed. If the Python extensions are intended to be used, NumPy (http://numpy.scipy.org/) needs to be installed in addition.
- If communication via MPI is required, a suitable implementation of the MPI2.0 standard is required to be installed (http://www.mcs.anl.gov/research/projects/mpich2/, e.g.).

- The MinGW g++/gcc compiler (http://www.mingw.org/) needs to be available.

Back to Contents.

2.3 Building preCICE

Switch to the root directory of preCICE, containing the `SConstruct` files. On a Linux system type

```
scons
```

On a Windows system type

```
scons -f SConstruct-windows
```

This will start building a debug version of preCICE, and may take several minutes. To speed up the build, add the option `-jn`, where `n` is the number of cores involved in the build and has to be larger than 1 in order to see speedup.

The following options can be enabled additionally, the first value is selected as default, when the option is not mentioned.

- `build=debug/release` For a production scenario, release mode should be used.
- `mpi=on/off` MPI is used as communication between solvers.
- `python=on/off` Python is used to enable configurable Python actions scripts.
- `sockets=on/off` Sockets are used as communication between solvers and recommended between solver and server.
- `spirit2=on/off` Boost.Spirit is used for writing checkpoints and reading geometries from VRML 1.0 files.

Back to Contents.

3 Running a Simulation with preCICE

3.1 Parallel Solvers

When running a solver in parallel, currently a preCICE server executable has to be run in addition. The server has all coupling data and communicates with the coupled solver (or its server, when also running in parallel). The parallel solver processes have to communicate with the server in order to read/write data from/to the coupling interface. To configure this use-case, the XML-tag `<server/>` has to be used in the configuration of the parallel participant (see XML reference). There, a communication method between solver processes and server has to be specified. The server executable is created in the build directory of preCICE (also containing the library), and is named `binprecice`. The possible console options for `binprecice` are obtained by running it without parameters. To run it as server type

```
./binprecice server SolverName ConfigName
```

The `SolverName` is the name of the configured participant, which is used to identify the solver in preCICE. The `ConfigName` is the preCICE XML configuration file used for the simulation. Server and solver processes can be started in any order. The server executable can currently only be run by one process.

3.2 Coupling Multiple Solvers

One preCICE adapter, i.e., implementational link of a solver's interface with the preCICE API, can couple more than just two solvers. The multi-solver coupling is specified by the XML configuration of preCICE by a sequence of one-to-one solver couplings. In a simple two-solver coupling one XML tag `communication`, one tag `coupling-scheme` as well as two tags `participant`, describing the solvers, are needed. In general, every solver needs to be described by an own tag `participant` and every coupling between two solver has to be described by a `communication` and `coupling-scheme` tag.

We first consider an example of three solvers $S1, S2, S3$, where $S1$ should be coupled to $S2$ by an explicit coupling scheme and $S2$ should be coupled to $S3$ by an implicit coupling scheme. This implies that solver $S1$ is coupled only indirectly via $S2$ to solver $S3$. In the XML configuration, we describe the solvers by tags

```
<participant name="S1"> ... </participant>
<participant name="S2"> ... </participant>
<participant name="S3"> ... </participant>
```

and we need to define communications between them

```
<communication:... from="S1" to="S2">
<communication:... from="S2" to="S3">
```

The association of the solvers in the same communication tag to attributes `from` and `to` does not matter here. Then, the coupling schemes need to be configured as

```
<coupling-scheme:implicit>
  <participants first="S2" second="S3"/>
  ...
</coupling-scheme:implicit>
<coupling-scheme:explicit>
  <participants first="S1" second="S2"/>
  ...
</coupling-scheme:explicit>
```

where the association of solvers to the attributes `first` and `second` of tag `participants` determines the execution sequence relative to one coupling. This configuration will create an explicit coupling scheme for solver $S1$, an implicit coupling scheme for $S3$ and an explicit as well as an implicit coupling scheme for solver $S2$. While the sequence of first giving the implicit and then the explicit scheme has no importance for $S1$ and $S3$, it is crucial for the coupling sequence in $S2$. In the described setup, in $S2$ the implicit

coupling scheme will first be iterated until convergence, before the explixit coupling scheme is executed (in every timestep). There will be no additional iteration necessary to execute the explicit scheme in $S2$, since it is triggered in the same iteration that leads to the convergence of the implicit scheme.

While the execution of the different coupling schemes of one solver is sequentialized, the overall execution is not. Thus, looking at the overall computation sequence of the coupled solvers, we would first observe one computation of $S1$ and $S2$, followed by repeated computations of $S3$ and $S2$ until the convergence of the implicit scheme. If in the expicit coupling scheme the first and second participant would be exchanged, $S1$ would have to wait until the convergence of the implicit coupling until it computes its timestep. This sequence ensures that a converged coupling state of $S2$ and $S3$ is achieved before the explicit coupling is executed in $S1$ which might be crucial to obtain stability also in the explicit coupling.

If the sequence of coupling schemes would be inversed, first, the explicit coupling scheme would be executed in $S2$ and, in the following, implicit iterations would follow taking the results from the computation of $S1$ as fixed.

For a general setup of coupling schemes $C1, C2, C3, ...$, the following rules apply for one time step:

- The sequence of schemes is executed from left to right.
- Explicit schemes are executed once.
- Implicit schemes are iterated until convergence.
- When several explicit schemes are in a sequence not broken by implicit schemes, they are executed together.
- When several implicit schemes are in a sequence not broken by explicit schemes, they are iterated together.
- When one or several implicit schemes follow one or several expicit schemes, the first iteration of the implicit schemes is done in the same call to `advance()`.
- When one or several explicit schemes follow one or several implicit schemes, the execution of the explicit schemes is done in the same call to `advance()` that lead to the convergence of all preceding implicit schemes.
- When during the iteration of several implicit schemes an implicit scheme is converged but the others not, the converged scheme is hold back (in converged status) until all other implicit schemes iterating together are also converged.