Science and Technology Facilities Council

Technical Report
DL-TR-2022-001

Comparison of code coupling libraries for high performance multi-physics simulation

P Rubin

April 2022

Enquiries concerning this report should be addressed to:

Chadwick Library
STFC Daresbury Laboratory
Sci-Tech Daresbury
Keckwick Lane
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at:
https://epubs.stfc.ac.uk

DOI: 10.5286/dltr.2022001

ISSN 1362-0207

Comparison of Code Coupling Libraries for High Performance Multi-Physics Simulation

Philippa Rubin*

Science and Technology Facilities Council, Hartree Centre, Daresbury Laboratory, Daresbury, Warrington, WA4 4AD, United Kingdom

March 2022

Abstract

The usability and performance of code coupling libraries are compared. In this work, MUI, MOOSE, preCICE, OpenPALM and PLE are considered. To compare performance, a 3D field exchange example was provided by the Scientific Computing Department at STFC. It is intended that this is a sufficiently general example to ensure this work applied to a wider range of the UK's Computational Collaborative Projects and High-End Computing Consortia.

1 Introduction

As available computational power increases, there is a growing interest in the use of large-scale multi-physics solvers in a range of life science, physical science and engineering projects. In most cases, a single solver will not offer all of the required capabilities and, hence, a coupling framework is required to translate data between solvers and co-ordinate their separate calculations. A number of code coupling libraries have been developed and, over the course of this project's work, many of them ran their first ever training workshops and have gained more and more support. Many have reached a level of maturity for us to compare their attributes and performance when used on large-scale problems. This document walks through some of the available code coupling libraries, followed by a performance comparison with a 3D field exchange example.

2 Code coupling perspectives

What is required of a code coupling library depends on the multi-physics problem a developer is trying to put together. Over the course of this project, literature and documentation has universally agreed that a code-coupling library should be a helpful tool to make two large-scale codebases talk to each other without having to write a third codebase yourself to do that. A developer can safely use a library for mass data-send between coupled components knowing this is the appropriate tool for this task, organising

*philippa.rubin@stfc.ac.uk

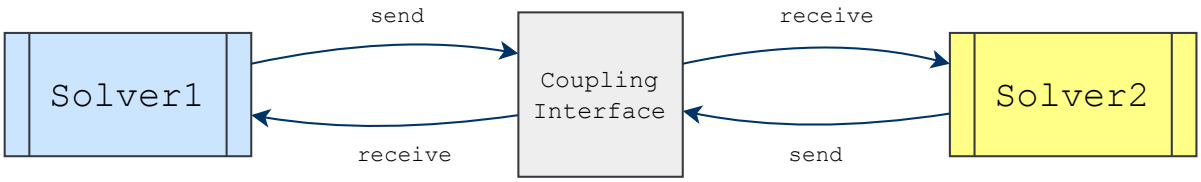

Figure 1: Coupling through a communication interface

computing allocation between the sections is straight-forward, and the work required to put this together is certainly less than writing this from scratch yourself. Using a library has the advantage of letting different scientific codebases stand alone in their own right as separate codebases: many agree that this is a better solution than creating monolithic 'do-everything' programs, which can take a long time and can lead to situations where a team of developers have a codebase where certain individuals know how parts of it work but nobody really knows how the full codebase works as the original solvers are no longer discernible from each other.

However, how far this sentiment has been taken varies across the libraries. Some libraries have gained popularity through focusing on being lightweight. That is, they only handle sending data points between the separate solvers. Some developers say fundamentally for code-coupling to work, all that needs to occur from a computational perspective is data transfer from one solver to another [1]. Two of the libraries in this report follow this approach of creating a lightweight communication framework, namely MUI in Section 3.1 and PLE in Section 3.5, see Figures 1 and 2. This approach certainly has its advantages, mainly that it allows for extremely general coupling: the only requirement is that the coupled variables' data is associated with a single point in space. This strategy is the most flexible in terms of its capability of combining codebases.

Other libraries take a more traditional approach: defining a mathematical schema for coupling. A multi-physics system consists of more than one component, each governed by its own principles for evolution or equilibrium, typically conservation or constitutive laws [2]. Some libraries encapsulate this idea when building applications using their software and they rely on your coupled problem's ability to be stated in an algebraic form such as a coupled equilibrium problem,

$$F_1(u_1, u_2) = 0 \tag{1a}$$
$$F_2(u_1, u_2) = 0 \tag{1b}$$

or a coupled evolution problem

$$\partial_t u_1 = f_1(u_1, u_2) \tag{2a}$$
$$\partial u_2 = f_2(u_1, u_2) \tag{2b}$$

and then formulate your problem in terms of a single residual that includes all of your components

$$F(u) = \begin{pmatrix} F_1(u_1, u_2) \\ F_2(u_1, u_2) \end{pmatrix} = 0 \tag{3}$$

If this is possible, then Equation 3 may be solved with an algorithm such as Jacobian-free Newton-Krylov (JFNK) [3], implemented with a non-linear implicit solver such as PETSc [4], SUNDIALS [5] or Trilinos [6]. This approach is possible for many code-coupling fields, such as in fluid structure interaction, nuclear fission, conjugate heat transfer, climate modelling, crack propagation: examples and strategies of which are all given in great detail in [2]. If this residual-based approach seems well suited to your work, then MOOSE in Section 3.2 should be of interest to you. While these kinds of solvers may save some computational scientists a lot of time, more general examples of coupling may not easily be expressed in an algebraic form such as in Equation 1 or Equation 2, nor would this be necessary. For more general examples, such as a mechanics program and an analysis program wanting to run in parallel rather than writing results of each solver to a file, the lighter communication-framework approach to coupling would be more appropriate.

3 Overview of chosen libraries

3.1 MUI

The Multiscale Universal Interface (MUI) is an open source code-coupling library originally developed by Brown University [7]. The primary developers and maintainers of MUI today are UKRI-STFC at Daresbury Laboratory [8]. There are practical training materials on MUI available which were very helpful over the course of this project: there is a GitHub repository of demo examples available [9], and the developers hold training events, for example, in February 2020 there was a training event at Daresbury Laboratory on high-performance code-coupling as part of the Hartree Centre IROR Training Series. The MUI project aims to create " ... a light weight plugin library that can glue together essentially all numerical methods including, but not limited to, Finite Difference, Finite Volume, Finite Element, Spectral Method, Spectral Element Method, Lattice Boltzmann Method, Molecular Dynamics, Dissipative Particle Dynamics and Smoothed Particle Hydrodynamics" [7]. MUI achieves this portability as it is very lightweight and straight-forward to use plus existing solvers do not have to be refactored before using it. You can integrate MUI into a code in around 10 new lines, as is shown below in Section 3.1.1. Due to this, MUI has been the quickest library to learn to use over the course of this project.

MUI provides a small set of programming interfaces to conduct send and receive messages between domains through an interface. In this way it does not put restrictions on the multi-physics solvers themselves: "MUI also follows PLE's philosophy of not prescribing how the physics of a problem should be coupled, only that coupling should be achieved by passing data through an interface. Assuming the data in question is associated with a point structure then the only requirement is that the data can be associated with a single point in space" [1]. The entire library is header-based and the only external library is the Message Passing Interface (MPI). It can be used in the same way any other C++ standard library would be used, without the need for pre-compilation. However, it has the bonus advantage of not interfering with pre-existing MPI communications within the pre-existing solvers, as Figure 2 and the example in 3.1.1 demonstrate.

3.1.1 Demonstration

Here is a short example of how to implement a MUI interface with two coupling components. This example sends just one variable value from one domain to another through the MUI interface. What follows is

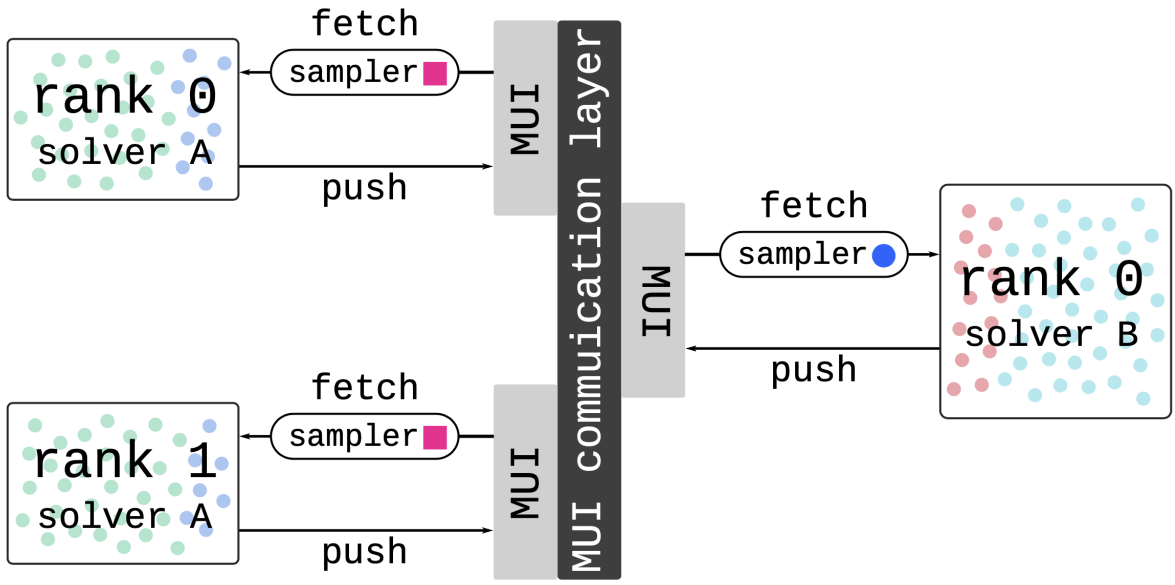

Figure 2: Solvers push and fetch data points through the MUI interface [7]

very similar to the first example from the MUI Demo's GitHub repository [9].

First, clone the latest release of MUI from their GitHub repository [10] as shown in Listing 1.

```
$ git clone https://github.com/MxUI/MUI.git
```

Listing 1: Clone MUI

Nothing further needs to be done here for MUI itself, just take note of where the `mui.h` header file is kept.

Create `hello.cpp` with the include shown in Listing 2. This is the only reference to the MUI download necessary to allow us to create a MUI interface.

```
#include "MUI/mui.h"

int main(int argc, char ** argv)
{
    return 0;
}
```

Listing 2: Include MUI header

We need to provide a name for each domain and a name for the coupling interface. Together, this will create the Uniform Resource Identifier, URI, a string which takes the form `mpi:// domain name / interface name`. The URI is submitted to the MUI interface as shown in Listing 3.

```
#include "MUI/mui.h"

int main(int argc, char ** argv)
{
    std::string domain_name ( argv[1] );
    std::string interface_name ( argv[2] );
    std::string uri( "mpi://" + domain_name + "/" + interface_name );

```

```
    mui::uniface1d interface(uri);

    std::cout << "MUI_Interface : " << uri << std::endl;

    return 0;
}
```

Listing 3: Submit URI

Alternatively, the URI could have been declared using a template header file as shown in the other demos [9]. When declaring with a template of your own design, you can specify what data types you would like to be used with your interface and its dimensionality. However, in this short example this information has been declared in main as shown in Listing 4 but for your own projects you would most likely use a template for this configuration.

```
    mui::sampler_exact1d<double> spatial_sampler;
    mui::chrono_sampler_exact1d chrono_sampler;
    mui::point1d push_point;
    mui::point1d fetch_point;
```

Listing 4: Configure MUI

We create a piece of data to be sent from one domain to the other domain. This data is then pushed to the interface using the `interface.push` command. Here it is given the simple name of `data`. This is followed by the `interface.commit` command. The second domain receives this information `data` with `interface.fetch`, with its reference `data` as shown in Listing 5

These crucial commands are the MUI wrappers for the MPI send and receive messages. For further detail on this, definitions for the push, commit and fetch functions can be found in the `uniface.h` header file in your clone of MUI. These commands are also overloaded such that you can provide different arguments depending on your URI set up.

```
    // Push value to interface
    double push_value ( std::stod(argv[3]) );
    push_point[0] = 0;
    interface.push( "data", push_point, push_value );
    std::cout << "Domain " << domain_name << " has pushed " << push_value << std::endl;

    // Commit
    interface.commit( 0 );

    // Fetch value
    int time = 0;
    fetch_point[0] = 0;
    double fetch_value = interface.fetch( "data", fetch_point, time, spatial_sampler,
    chrono_sampler );
    std::cout << "Domain " << domain_name << " has received " << fetch_value << std::endl
    ;

    return 0;
}
```

Listing 5: MUI push, commit and fetch commands

Finally, compile using your normal `mpic++` as shown in Listing 6

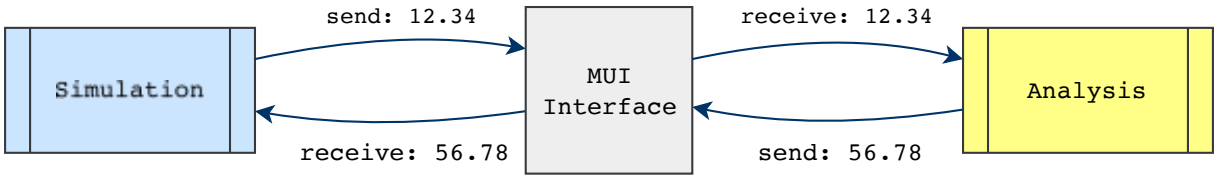

Figure 3: One point send and receive in MUI

```
$ mpic++ -std=c++11 -O3 hello.cpp -o hello
```

Listing 6: Compile MUI application

Run with `mpirun` as shown in Listing 7, specifying the name of each domain, the name of the interface, and the value you intend to send from the respective domains. In this example we have a `simulation` domain and an `analysis` domain sending `float` values to each other through the `global_interface`. The result is also demonstrated in Figure 3

```
$ mpirun -np 1 ./hello simulation global_interface 12.34 : -np 1 ./hello analysis
    global_interface 56.78
rank 0  identifier mpi://simulation/global_interface  domain size 1 peer number 1
rank 1  identifier mpi://analysis/global_interface  domain size 1 peer number 1
MUI Interface : mpi://simulation/global_interface
MUI Interface : mpi://analysis/global_interface
Domain simulation has pushed 12.34
Domain analysis has pushed 56.78
Domain simulation has received 56.78
Domain analysis has received 12.34
```

Listing 7: Run MUI application

In the previous example we say `./hello` in the `mpirun` command twice. However, to make each domain do something different, you would have two different executables, one for each domain, which would be knitted together with MUI with the correct send and receives in each program. For example, you could have one executable `simulation_work`, and another `analysis_work`, which would be put together such as in Listing 8,

```
$ mpirun -np 1 ./simulation_work simulation global_interface 12.34 : -np 1 ./
    analysis_work analysis global_interface 56.78
```

Listing 8: Run multiple MUI enabled applications

allowing the user to implement a coupled problem of two different programs, with data exchange between them. If the two executables `simulation_work` and `analysis_work` had their own MPI communication prior to adding MUI, this would not be interfered with. The `np` flag values above could be changed to whatever you liked for each piece of work. In this way, MUI allows easy co-ordination of solvers and computing allocation.

3.2 MOOSE

The Multiphysics Object-Oriented Simulation Environment (MOOSE) is a finite-element framework developed by Idaho National Laboratory [11]. Development of MOOSE started in 2008 and was made

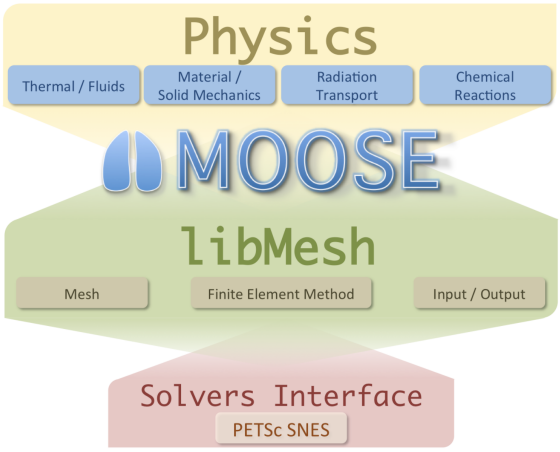

Figure 4: The MOOSE framework. MOOSE acts as a bridge between your physics and other software; libMesh and PETSc. Source: MOOSE training slides [13].

open-source in 2014. In Summer 2020, MOOSE updated their website and training materials: there is an excellent training video from their two-day virtual workshop in June 2020, which is freely available on YouTube [12] with slides [13]. This was the most helpful material in learning how to use MOOSE because the MOOSE team explain how your mathematical equations can be translated into MOOSE `C++` objects in a very accessible way. MOOSE has been the most transparent and resource-rich coupler over the course of this project and on their website you can find help on anything related to MOOSE, including help with general Object Orientated Programming in `C++` and material on Finite-Element Modelling.

MOOSE was also one of the most straight-forward couplers to access and install. MOOSE is available on GitHub [14]. In this work, their provided Conda environment was used to manage dependencies, which worked very well. Building applications and tests worked exactly as described in the documentation and could not be faulted. Software quality is stated as a strong priority for MOOSE: they follow a Nuclear Quality Assurance Level 1 (NQA-1) development process. MOOSE includes a test suite and documentation system to allow for agile development while maintaining a NQA-1 process. More detail about their software quality standards can be found on their website [15].

As with the other libraries, MOOSE helps you make two pieces of code talk to each other, without having to write a third piece of code to do that. However, how exactly it does this is quite different to the other couplers that are more communication-framework based. Rather than explicitly stating data points to share between coupling components, far less work is required from the user with MOOSE. They state in their training material "If you understand your physics, you should be able to use MOOSE" [12], the idea being that if you know the equations that represent your model and you can write them down in a list, this should be quite easy to translate into a MOOSE input file. MOOSE is a "fully-coupled, fully-implicit multiphysics solver", that is if you have 15 equations, you can put them all together in one MOOSE input file and solve them all at the same time. They say they have made their input file system "The simplest input that we can get from you that explains your physics", and then from this, MOOSE then can do all of the more difficult multi-physics solving for you, and is automatically parallel. This is demonstrated in Section 3.2.1. Under the hood, MOOSE uses PETSc [4] for its non-linear solvers, and directly relies on libMesh [16] for its finite-element framework, see Figure 4.

3.2.1 Demonstration

Since MOOSE works in such a different way to the other couplers featured in this report, the single variable exchange example shown in 3.1.1 and 3.4.1 doesn't put forward the key ideas of how MOOSE is intended to be used, or its advantages. That is, you write a set of mathematical equations down, put them into a MOOSE input file, and let MOOSE solve it. And so, instead, the following example shows how you would do that. This example demonstrates how to solve two equations where two variables are coupled together, through writing the appropriate input file and then building a MOOSE application. This example is taken from Example 3 in their examples set, which is available online [13].

We access MOOSE via GitHub [14] and we use their provided Conda environment to manage dependencies such as MPI, PETSc and libMesh. Their 'Getting Started' instructions on their website [11] explain how to do this and were completely faultless in this project.

We use MOOSE to solve for u and v with the following problem statement

$$-\nabla \cdot \nabla u + \nabla v \cdot \nabla u = 0 \tag{4a}$$

$$-\nabla \cdot \nabla v = 0 \tag{4b}$$

over a 3D domain Ω, which in this instance is a cup shaped object provided by an Exodus file. We have $u = v = 0$ on the top of the cup, $u = 2$ and $v = 1$ on the bottom, and natural boundary conditions $\nabla u \cdot \hat{n} = 0$, $\nabla v \cdot \hat{n} = 0$.

In their virtual workshop, the MOOSE team recommend you approach turning your strong-form PDEs into MOOSE code by transforming them into weak-form. They recommend that, in general, if you're not too familiar with finite-element methods, the following should get you through using MOOSE successfully without having to learn the underlying theory behind why this works.

The MOOSE team recommend that if you don't already know how to turn the PDE into weak-form, then the following process should be followed:

1. Write down the strong-form PDE;
2. Rearrange so that the right-hand side is equal to zero in all equations;
3. Multiply by ϕ_i, which is a test function;
4. Integrate over your domain Ω, applying integration-by-parts and the divergence theorem to arrive at the desired derivative order and generate boundary integrals.

Applying this process to Equation 4, the weak-form is given by

$$\underbrace{(\nabla u_h, \nabla \phi_i)}_{\text{Diffusion}} + \underbrace{(\nabla v_h \cdot \nabla u_h, \phi_i)}_{\text{Convection}} = 0 \quad \forall \phi_i \tag{5a}$$

$$\underbrace{(\nabla v_h, \nabla \phi_i)}_{\text{Diffusion}} = 0 \quad \forall \phi_i \tag{5b}$$

with test functions ϕ_i, and finite-element solutions u_h and v_h. For more information on this, the virtual workshop has plenty of details about shape functions, numerical integration and quadrature.

We must now represent the three terms in Equation 5 in terms of MOOSE `Kernel` objects. MOOSE C++ objects always start with a capital letter. Usually in MOOSE it is required that the user creates their own custom `Kernel` objects. We need two diffusion `Kernels`, one to act on each variable u and v, and a convection `Kernel`. In this instance, a `Diffusion` object is already defined in MOOSE but we must define a convection object inherited from the `Kernel` object in MOOSE.

In defining the convection `Kernel`, first we must register the object with MOOSE, see Listing 9. Then we must provide its contribution to the residual and its Jacobian. That is, we override the `computeQpResidual()` and `computeQpJacobian()`, using the already provided attributes `_u`, `_grad_u`, `_test`, `_grad_test`, `_phi`, `_grad_phi`, `_q_point`, `_i`, `_j` and `_qp`. See Listing 10.

```
registerMooseObject("ExampleApp", ExampleConvection);
```

Listing 9: Registering a MOOSE object

```
Real
ExampleConvection::computeQpResidual()
{
  return _test[_i][_qp] * (_grad_some_variable[_qp] * _grad_u[_qp]);
}

Real
ExampleConvection::computeQpJacobian()
{
  return _test[_i][_qp] * (_grad_some_variable[_qp] * _grad_phi[_j][_qp]);
}
```

Listing 10: Defining a MOOSE `Kernel` object

Now we can create the all-important MOOSE input file. The input file is written with six parts - Mesh, Variables, Kernels, Boundary Conditions, Executioner and Outputs.

This file will include everything about our multi-physics problem and will be passed as an input to MOOSE when we run the MOOSE app.

First, we define the domain of the problem, the `Mesh`, which can be from an input file or you could define a more simple Cartesian domain with boundaries. In Listing 11 we are providing an Exodus file.

```
[Mesh]
    file = 'mug.e'
[]
```

Listing 11: Provide a `Mesh`

Next, we give MOOSE all of the variables in our problem, 'diffused' and 'convected', with linear Lagrange shape functions. See Listing 12.

```
[Variables]
  [./convected]
    order = FIRST
    family = LAGRANGE
  [../]

  [./diffused]
    order = FIRST
    family = LAGRANGE
  [../]
```

```
[]
```

Listing 12: Provide `Variables`

Now we define the three `Kernels` discussed earlier. We use the pre-defined `Diffusion` `Kernel` object twice, and then our user-defined `ExampleConvection` object, always providing too the variable the `Kernel` is supposed to be acting on. See Listing 13.

```
[Kernels]
  [./diff_convected]
    type = Diffusion
    variable = convected
  [../]

  [./conv]
    type = ExampleConvection
    variable = convected

    # Couple a variable into the convection kernel using local_name = simulationg_name
    syntax
    some_variable = diffused
  [../]

  [./diff_diffused]
    type = Diffusion
    variable = diffused
  [../]
[]
```

Listing 13: Provide `Kernels`

Next, we define boundary conditions for the problem, as discussed earlier the cup as $u = v = 0$ on the top and $u = 2$, $v = 1$ on the bottom. See Listing 14

```
[BCs]
  [./bottom_convected]
    type = DirichletBC
    variable = convected
    boundary = 'bottom'
    value = 1
  [../]

  [./top_convected]
    type = DirichletBC
    variable = convected
    boundary = 'top'
    value = 0
  [../]

  [./bottom_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'bottom'
    value = 2
  [../]

  [./top_diffused]
```

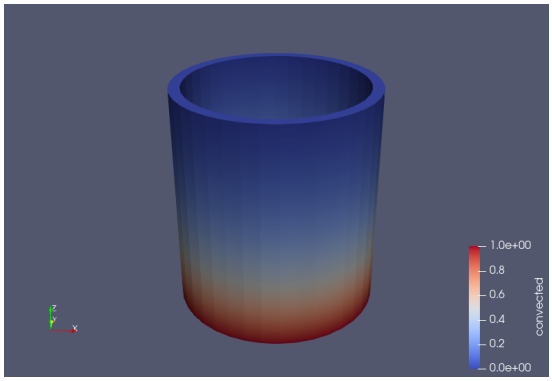

Figure 5: Convected Variable

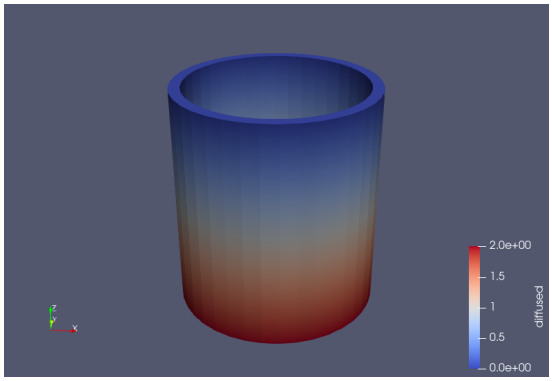

Figure 6: Diffused Variable

```
    type = DirichletBC
    variable = diffused
    boundary = 'top'
    value = 0
  [../]
[]
```

Listing 14: Provide `BCs`

We select the solver for our problem in the `Executioner` block. Here we use PJFNK [3], which is the default. Alternative solver types are JFNK, Newton and FD for debugging. The virtual training video [12] goes into these solvers in more detail. See Listing 15.

```
[Executioner]
  type = Steady
  solve_type = 'PJFNK'
[]
```

Listing 15: Provide the `Executioner`

Finally, we say what kind of output we want. We can output to the console, and to an Exodus file. See Listing 16.

```
[Outputs]
  execute_on = 'timestep_end'
  exodus = true
[]
```

Listing 16: Provide `Outputs`

All we need to do now is, in the MOOSE Conda environment, run `make`, and run the generated app with the input file passed as an input argument. You should see the two variables converge on your console. It will also give you an `out.e` file of your coupled variables, `diffused` and `convected`, which you can view in Paraview, or even in MOOSE's own graphical front end, called Peacock. See Figures 5 and 6.

3.3 preCICE

The Precise Code Interaction Coupling Environment (preCICE) [17] is an open source coupling library which aims to couple existing solvers together, creating what they call 'partitioned' simulations. In this

way, they are motivated similarly to MUI in Section 3.1, wanting the highest flexibility possible in reusing existing components. Their team have particular interests in coupling codes relating to fluid-structure interaction and conjugate heat transfer simulations, but stress that they are not limited to such fields. preCICE has been developed by doctoral candidates from the Technical University of Munich and the University of Stuttgart.

Similarly to MOOSE, an input file has to be written to couple your code with preCICE. preCICE is configured with XML file, for which they do provide a reference guide [18]. One particular quirk of preCICE is that in this XML file, you can establish an `m2n` communication channel (i.e. from `m` processes of `Solver1` to `n` processes of the `Solver2`) based on TCP/IP sockets. This makes preCICE particularly different to other libraries here that rely on MPI for communication between solvers. You can change this to use MPI ports if you wish but the preCICE team recommend using `sockets`. Once this XML configuration is complete, the user can then insert calls to the preCICE API in original solvers, and at this stage preCICE is very similar to MUI.

The preCICE Team ran their first workshop in February 2020, material from which is available on their website and their YouTube channel. They provide quite elaborate examples in their official tutorials and, hence, a new user may prefer to learn how to use preCICE using the `solverdummy` code shown below, which is also available under their examples on GitHub [19]. During this work, their step-by-step XML configuration guide [20] and their step-by-step 'Couple your code' guide [21] were found to be the most helpful resources when learning how to use preCICE. If you need help using preCICE, you can also post questions on their Discourse forum [22]. The preCICE Team list their main reference guide [23] as well as dissertations of main preCICE developers as 'Starting Points' in their own literature guide. However, we would recommend using the resources mentioned previously over these if you are a completely new user.

preCICE is available on GitHub [19]; it is also possible to build it using Spack. preCICE did successfully build on ARCHER2 but this was not an easy process and this has been documented in Appendix B. If you intend to use preCICE on a HPC system, this appendix item, or possibly one of their other cluster examples [24], may be of use to you. We also encountered difficulty building preCICE on our local machines. There is a demo virtual-machine to run preCICE [25], which we did find very useful as we could try out some examples before having to go through the difficult process of building the library ourselves.

3.3.1 Demonstration

All of the following can be run on the preCICE demo virtual machine [25]. We would highly recommend using this if this is your first time looking into preCICE, as trying to build preCICE both with Spack and from source was quite difficult in this project.

As with the other couplers, this demonstration will show how to exchange variables between two solvers, `SolverOne` and `SolverTwo`. preCICE uses meshes and so, rather than exchanging one variable as shown in 3.1.1 and 3.4.1, a size 3×3 field will be exchanged between the two solvers. The following example is available on preCICE's GitHub [19] called `solverdummies`. It is used in their build testing and equivalent `C`, `C++` and `Fortran` examples are available.

First, we must write a `.xml` file providing our desired configuration for the coupled program. As shown in Listing 17, this consists of five parts: `data`, `mesh`, `participant`, `m2n`, and `coupling-scheme`.

```
<precice-configuration>
  <solver-interface dimensions="3">
    <data .../>
```

```
    <mesh .../>
    <participant .../>
    <m2n .../>
    <coupling-scheme .../>
  </solver-interface>
</precice-configuration>
```

Listing 17: preCICE Configuration Steps

The first component to declare in this file is the data we wish to exchange between solvers. In this case, we will have one `data` entry for each solver.

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

Listing 18: Coupling Data

After this, in your code you can now access these coupled variables using the preCICE API, using `getDataID` as shown in Listing 19.

```
int dataOneID  = interface.getDataID("dataOne", meshID);
int dataTwoID = interface.getDataID("dataTwo", meshID);
```

Listing 19: Coupling data with preCICE API

The next component to declare is the meshes we wish to use. Similarly to before, we will have one `mesh` entry for each solver. These will have access to both data variables to read and write on each side.

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

Listing 20: Coupling meshes

And now in your code you can access the mesh vairables using the preCICE API as shown in Listing 21.

```
int meshOneID = interface.getMeshID("MeshOne");
int meshTwoID = interface.getMeshID("MeshTwo");
```

Listing 21: Coupling meshes with preCICE API

The coupled simulation has a solver on each side, of which we need at least two. These solvers are defined through providing a `participant` for each solver. This definition is one of the most important as it brings many of the other components together. The `participant` provides the `mesh` and if a `participant` uses more than one `mesh`, you can define a `mapping` between those. There are three mapping types: `nearest-neighbor`, `nearest-projection` and some radial-basis function mappings. You can also give these mappings directions and constraints. The reader should refer to preCICE's documentation on mappings for further information [26].

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

```
  <read-data name="dataTwo" mesh="MeshOne" />
</participant>
<participant name="SolverTwo">
  <use-mesh name="MeshOne" from="SolverOne" />
  <use-mesh name="MeshTwo" provide="yes" />
  <mapping:nearest-neighbor
    direction="write"
    from="MeshTwo"
    to="MeshOne"
    constraint="conservative" />
  <mapping:nearest-neighbor
    direction="read"
    from="MeshOne"
    to="MeshTwo"
    constraint="consistent" />
  <write-data name="dataTwo" mesh="MeshTwo" />
  <read-data name="dataOne" mesh="MeshTwo" />
</participant>
```

Listing 22: Coupling participants

Near the top of your code, you should create a preCICE `interface` object with the same name as you gave the participant in the configuration file.

```
SolverInterface interface("SolverOne", configFileName, commRank, commSize);
```

Listing 23: Coupling interface with preCICE API

You should also define the coordinates of your mesh using this interface since the `participant` provides the `mesh`. In `solverdummies`, this is done in the following way:

```
interface.setMeshVertices(meshID, numberOfVertices, vertices.data(), vertexIDs.data());
```

Listing 24: Mesh coordinates with the preCICE API

For your participants to exchange data, they need a defined communication channel. The default communication channel is based on TCP/IP sockets. preCICE express that you can use MPI ports as an alternative to TCP/IP sockets but state in their documentation that they recommend you use TCP/IP sockets instead. Programs you write using MPI for the communication channel will also generate warnings suggesting you use sockets as the MPI ports program can cause problems. You establish a parallel connection of `m` processes of `SolverOne` to `n` processes of `SolverTwo` in the following way.

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

Listing 25: Communication channel

And so if you wanted to try using MPI ports, you can swap the `sockets` argument in Listing 25 for `mpi`.

The last part to configure is the `coupling-scheme`. You must define which participants are coupled together and the data which is exchanged in this coupling. You can choose an implicit or explicit coupling scheme and also decide if participants should be executed either in serial or parallel. With implicit schemes, participants are executed multiple times until convergence, whereas for explicit schemes you define this behaviour. A very simple coupling scheme is shown in Listing 26.

```
<coupling-scheme:serial-implicit>
```

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

Listing 26: Coupling scheme

With this the configuration file is now finished. Many of the preCICE API calls have been shown above but the full file can be shown in their `solverdummy` example on Github [19]. On the preCICE virtual machine, this code exists at

```
~/precice/examples/solverdummies/cpp/solverdummy.cpp
```

Listing 27: preCICE VM solverdummies location

At this location, you can run `cmake .` and `make` to compile your coupled program. To run, enter the command given in Lsting 28 and your coupled program should print the variable exchanges to the console.

```
./solverdummy ../precice-config.xml SolverOne MeshOne & ./solverdummy ../precice-config.
    xml SolverTwo MeshTwo
```

Listing 28: Run preCICE solverdummies on their Virtual Machine

3.4 OpenPALM

OpenPALM (Projet d'Assimilation par Logiciel Multimethodes) is another coupling library enabling a user to execute components of code concurrently with communication between. OpenPALM comes in two parts; PrePALM and PALM. PrePALM is the graphical user interface and PALM is the driver of coupling framework itself. PALM handles the elementary components of the coupling algorithm with MPI communication to exchange data between.

On the OpenPALM website, it states they have had simulations of 130,000 cores on Titan, 130,000 cores on Turing and 12,000 cores on Curie [27]. It is also provided in [2] as one of the successes in multi-physics software. It has a lengthy user guide [28] and does not appear to be restricted to any particular scientific field. For these reasons, OpenPALM was of great interest in this investigation into high-performance code-coupling.

The OpenPALM team is joint between CERFACS and ONERA. CERFACS have also created another coupling library called OASIS [29], dedicated to geophysical applications. In contrast to OASIS, OpenPALM provides a more generic interpolation framework based on an unstructured mesh formalism [30], provided by the CWIPI library developed by ONERA [31].

OpenPALM applications are implemented via a user interface called PrePALM. The user designs their coupling algorithm into sequential and parallel sections, loops, conditional executions, and communication between components. The designed algorithm is presented to the user clearly in a window as seen in Figure 7. OpenPALM stress you can create parallel code "...without anything to know about MPI, only by drawing! It is one of the PALM features: one can make parallel computing without any further specific knowledge" [28].

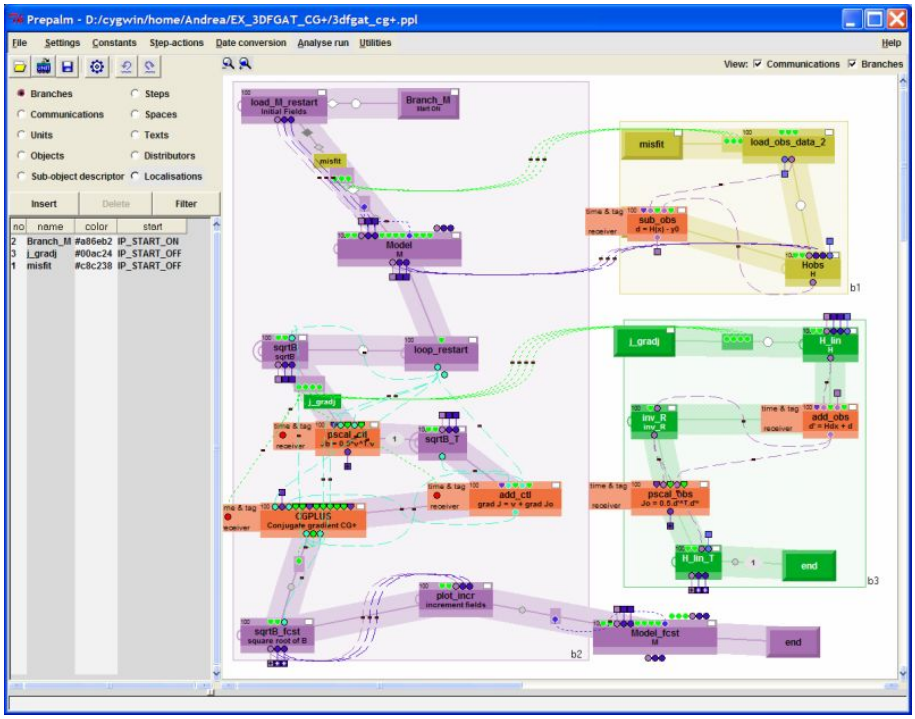

Figure 7: The PrePALM Interface. This example was sourced from OpenPALM's website [27].

You cannot create a coupled application in OpenPALM without creating a scheme in the GUI such as in Figure 7. This does not mean you cannot use OpenPALM on a remote machine but you would not attempt to use PrePALM on a supercomputer. It is expected that you 'build' your application on a local machine with the PrePALM interface, that is you generate all of the 'palm' files through this interface. These files are not machine specific, you can copy this automatically generated source code to the remote machine. Given the remote machine has the PALM library installed, you can amend the `Make.include` file to provide paths for the PALM installation and the MPI compilers on the remote machine, and the coupled application can compile and run remotely.

3.4.1 Demonstration

Similar to in Section 3.1.1 with MUI, here we demonstrate how to send a single variable value from one coupling branch to another with OpenPALM.

After installing PALM and PrePALM with the user guide, launch PrePALM from the terminal as shown in Listing 29

```
$ prepalm
```

Listing 29: Launch the PrePALM interface

The PrePALM window should launch with a blank canvas for your coupling program, as shown in Figure 8.

While the menu on the left has `Branches` selected, click `Insert`. A window should appear to give your branch a name and even a colour of your choice. This will result in a new branch appearing on the canvas. Double click on this to edit the branch code, the window shown in Figure 9 will appear. You could attach

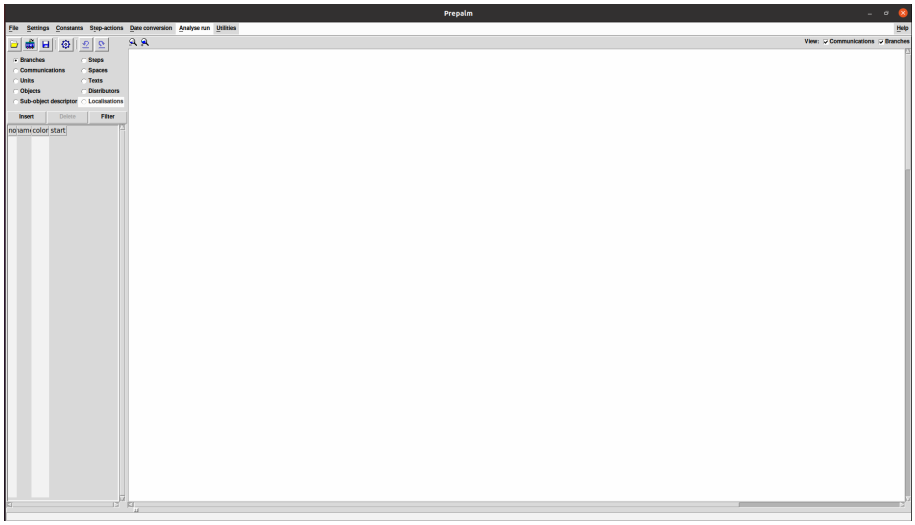

Figure 8: Launching PrePALM

'PALM units' in the branch code window, i.e. your own source files, which can be either a `.c`, `.cpp`, `.f` or `.f90` file, and you can do a mixture too. In this simple example, we will just write the `send` and `receive` calls in PrePALM itself.

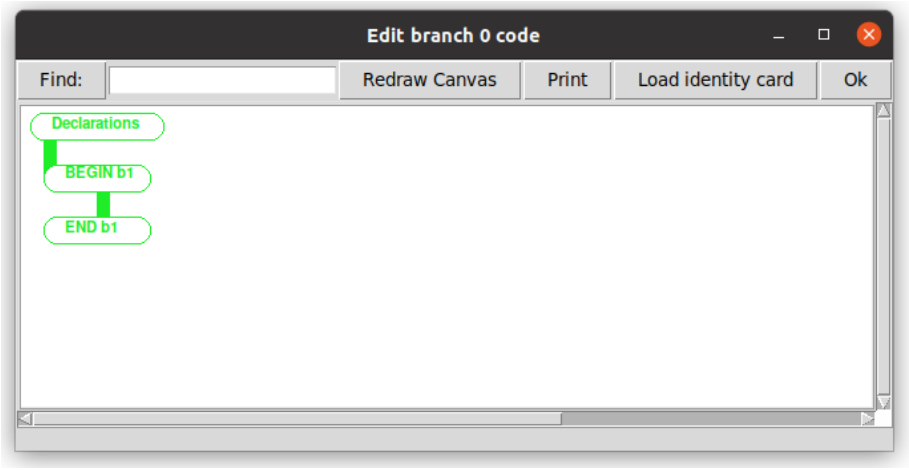

Figure 9: Creating branches in PrePALM

Under `Declarations`, we create an integer to send from the branch `b1` and give a value. Under `BEGIN b1` we can click and call `PALM_Put` and provide the variable to send as shown in Figure 10. This is equivalent to MPI Send.

Next, create another branch `b2` in the same way as `b1` was created. It will appear on the canvas alongside `b1`. You can click and drag these around as you please. This time `b2` will have a `PALM_GET` call shown in Figure 11, the equivalent of MPI Receive.

The two branches will be presented side by side on the canvas, with a little dot on each representing `int_send` and `int_recv`. These can be linked in the interface by clicking on them both. This is shown in Figure 12.

From the top left go to `File`, then save your design by clicking `Save Prepalm file (.ppl)`. Then

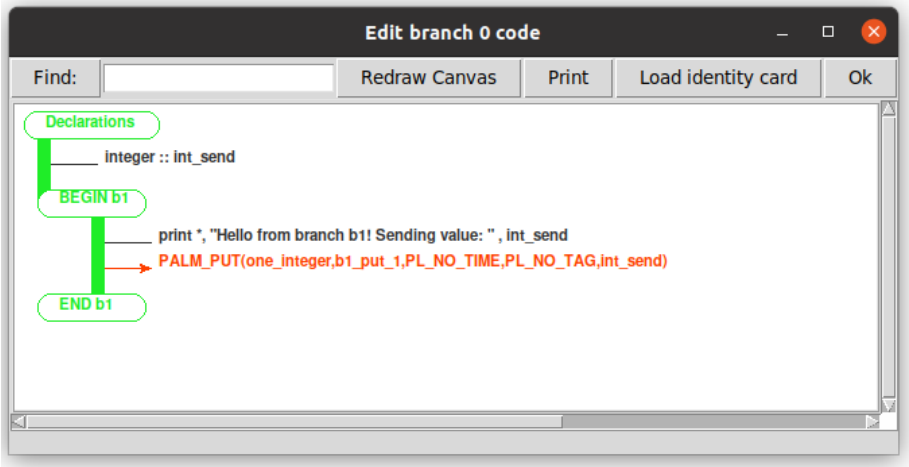

Figure 10: Using `PALM_PUT` in PrePALM

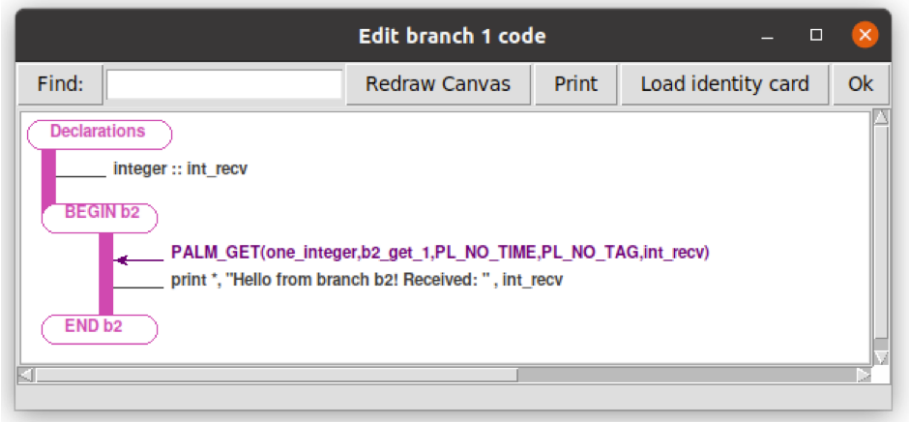

Figure 11: Using `PALM_GET` in PrePALM

generate the Palm service files by clicking `Make Palm files`. You will have to choose if you are using `mpi1mode` or `mpi2mode` in the window that comes up here. Prepalm will then generate many palm service files from this, in this example with `mpi1mode` this will create 15 various palm files. More files would be created as you added more of your own source code to the design. These files are not machine specific.

Create a `Make.include` file, providing the path to `$PALMHOME` where your installation of PALM exists, as well as the `F90`, `F77`, `CC` and `C++` compiler paths. Now in the terminal, you should be able to run `make` which will create the `palm_main` executable. Run this with `mpiexec` as shown in Listing 30.

```
$ mpiexec -np 1 ./palm_main
Hello from branch b1! Sending value:  1234
Hello from branch b2! Received:   1234
```

Listing 30: Run OpenPALM application

This example has achieved a very similar result to the earlier demonstration with MUI in Section 3.1.1.

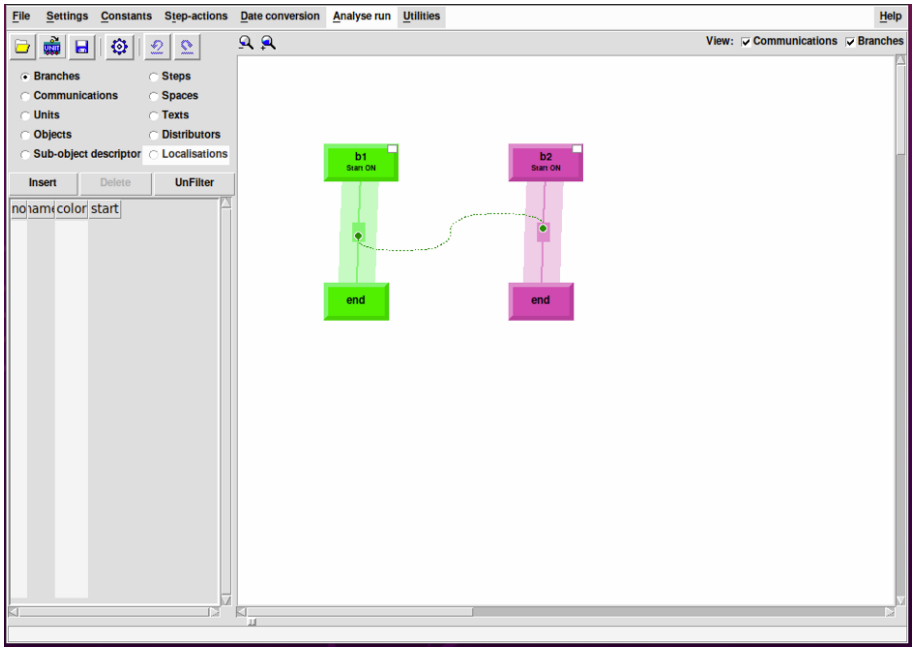

Figure 12: Completed PrePALM implementation

3.4.2 Requirements and Building

Over the course of writing this report, there was great difficulty with building PALM. Appendix C documents some steps taken when we investigated porting OpenPALM onto a HPC system.

3.5 PLE

The Point Location Exchange library, PLE, is part of Code_Saturne [32], a CFD software released by EDF, which is open-source. PLE was specifically designed to simplify parallel couplings using a minimalist API and dependencies. The library allows location of points on meshes and compare themselves to MUI and preCICE [33]. It is used to couple Code_Saturne to other EDF codebases, such as EDF's thermal software SYRTHES [34]. Code_Saturne is available on GitHub [35] with PLE included in this repository [36]. PLE has Doxygen documentation [37]. It also featured in CIUK in 2019 [38] in a presentation about Exascale CFD code.

PLE provides a framework through which data can be passed between different solvers. PLE creates an interface between two solvers, an MPI communicator is created for each interface, thus allowing pre-existing MPI functionality within the original solvers. Data is transferred between solvers at specfic spatial points [1]. In this way, it works in a very similar way to MUI in Section 3.1.

However, apart from the Doxygen documentation, unfortunately, PLE does not have anything further that we could find such as a user guide to help new users learn how to use PLE. Regretfully, because of the lack of support in this way, it is difficult to justify recommending PLE as the coupling library that a team should use if they have never used it before. While it does appear to be a current and popular choice, we would recommend using MUI instead as the two libraries work in a very similar way and MUI has more support for new users.

4 Performance Comparison

The same code-coupling example has been implemented in each library to compare their at-scale performance. Prior to this work, this example was originally implemented in MUI and it was part of the testing suite used by the MUI contributors in the Scientific Computing Department at STFC. In this project, this example was translated into the most equivalent problem statement in the other code coupling libraries featured in this report.

The example in this section is a simple 3D field exchange problem. A field u is created of the form $u(\vec{x}) = \vec{x}$. This is then exchanged with a field $v(\vec{x})$ by setting $v(\vec{x}) = u(\vec{x})$, see Figure 13. How exactly $v(\vec{x})$ is set to be $u(\vec{x})$ is different in every library in this report.

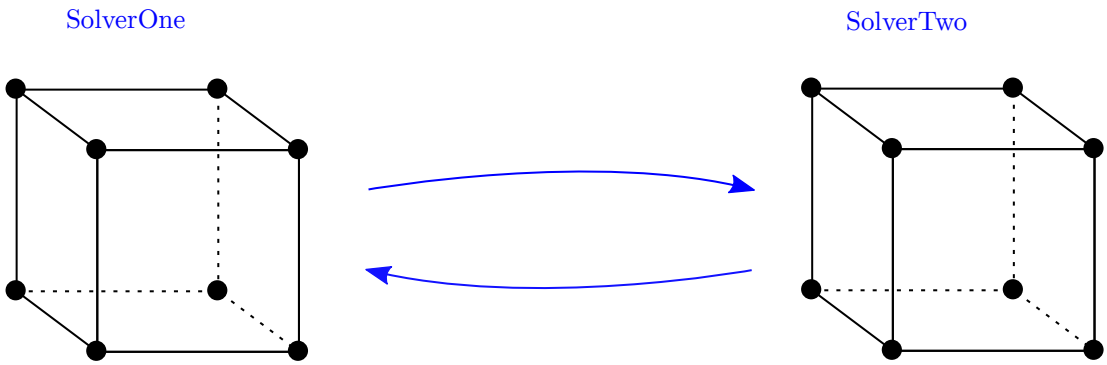

Figure 13: 3D Field Exchange Benchmark Case

This problem statement was chosen because it is the most general code-coupling problem of two variables that could be thought of and has easy scalability. The primary intention behind this being that it would ensure this project's work applied to the widest audience of CCPs and HECs interested in high-performance coupling, as it would zone-in on the mechanism of coupling variables together as much as possible.

4.1 Set-up

This comparison was carried out on ARCHER2 [39], the UK National Supercomputing Service. At the time of performing the benchmark runs, the full ARCHER2 system was not yet available to users. As such the four cabinet system of ARCHER2 was used in all cases. The four cabinet system consisted of 1,024 compute nodes, each with dual AMD EPYC Zen2 (Rome) 64 core processors.

At the time of writing, all of the libraries listed in this report have been fairly recently released. Almost all of them ran their first public workshops over the course of this project. It is thought that this may have contributed to there being considerable difficulty in building these libraries on ARCHER2 and even just on our local machines. To combat this, where package manager solutions existed, this was taken advantage of. For MUI, being the most lightweight, we only needed to use the ARCHER2 `GNU` environment with `gcc/10.1.0`, using heterogeneous job submissions, see Appendix A. For MOOSE, we used the provided

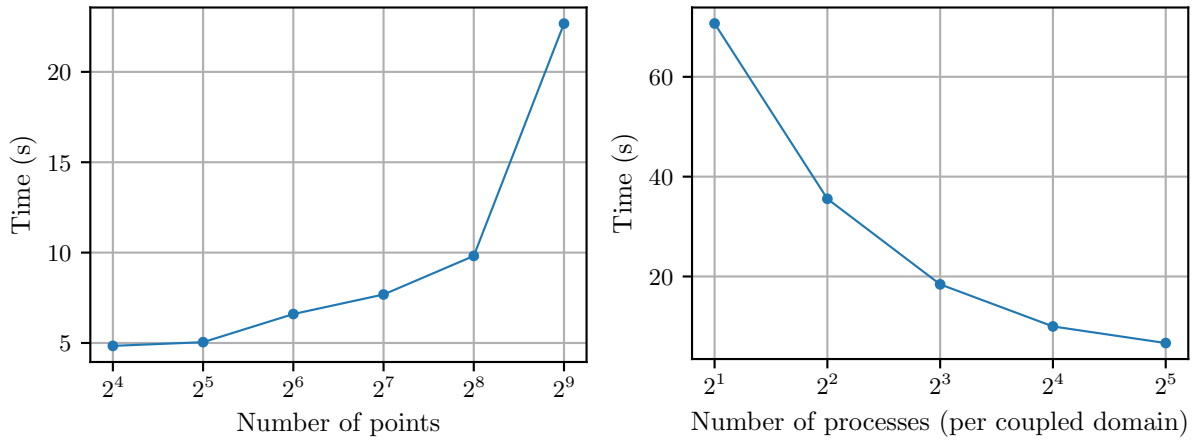

Figure 14: 3-D field exchange example with MUI. Run times are shown plotted against number of points on the side of a cube, using 64 processes per coupled domain (left), and plotted against number of MPI processes per coupled domain, using 256 points on the side of a cube (right).

Conda environment. preCICE was built with Spack, see Appendix B.

4.2 Results

4.2.1 MUI

The applicability of MUI to large-scale simulations can be seen most transparently of all the libraries featured in this report. We can see its scalability both by increasing the number of points and by increasing the number of processes in Figure 14. We believe this is because it is a lightweight communication framework only depending on MPI, and so its scaling reflects this. MUI jobs were submitted as two heterogeneous jobs. For more information regarding submitting heterogeneous jobs for this library on ARCHER2, see Appendix A.

4.2.2 MOOSE

We consider the time it takes to run a MOOSE app implementing the objects shown in Figure 13 with a corresponding `Kernel`. The time taken can be seen as a function of the number of points or the number of processes as shown in Figure 15.

It should be noted that MOOSE does a lot of work when it implements a MOOSE app, i.e. creating all of the objects described in Section 3.2.1. This may be superfluous to your own coupling problem statement or may be very useful to you if you have a problem complementary to what is described in Keyes et al. [2] mentioned earlier. You should take this into consideration when you choose which library is most appropriate for your work. It was decided that the field exchange problem shown in Figure 13 was the minimum initialisation required for a coupled problem, i.e. a domain for the simulation to take place and, therefore, this data preparation would be in common with nearly all coupling problems.

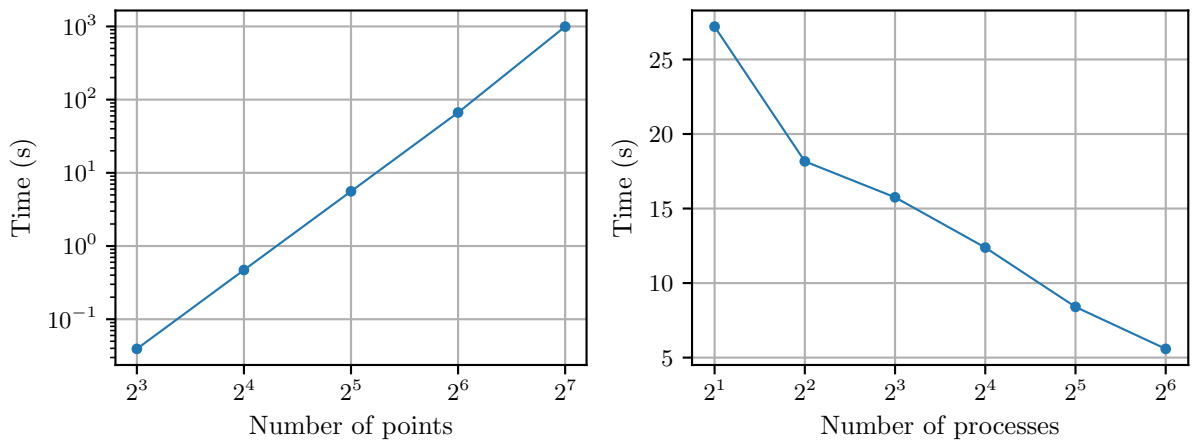

Figure 15: 3-D field exchange example with MOOSE. Run times are shown plotted against number of points on the side of a cube, using 64 processes (left), and against number of MPI processes, using 32 points on the side of a cube (right).

4.2.3 preCICE

preCICE was the most difficult library to port onto ARCHER2, both by trying to build from source and with Spack. Eventually, we successfully built preCICE with Spack and OpenMPI but this needed specific configuration, see Appendix B.

We were able to run the field exchange example using the TCP/IP socket option for the communication protocol. The implementation with sockets is shown in Figure 16 with time taken shown as a function of the number of points.

The option of configuring the coupling communication messages specifically with MPI on ARCHER2 unfortunately was not successful. This is advertised as an option on their website, however, it is not described as their preferred method in the documentation. It was not clear to us why they stated this preference for the `sockets` method when we were initially researching coupling options. However on running our example, we were met with the following message when configured to use MPI:

```
preCICE:WARNING:  preCICE was compiled with OpenMPI and configured to use <m2n:mpi />,
    which can cause issues in connection build-up. Consider switching to sockets if you
    encounter problems.
```

Listing 31: preCICE using MPI for coupling communication messages

which did not happen when using `sockets`.

While this is unfortunate, it is still worth considering preCICE with using their preferred TCP/IP sockets method for communication, as with these options, it does work well, Figure 16. Indeed, you may actually prefer that it does not use MPI for the coupling mechanism if you are concerned that adding more MPI messages may be problematic when you already use MPI in your applications. While MUI does appear to do this well, you may still personally lack confidence that this would transfer to your own example and you may prefer to avoid trying this entirely, and preCICE gives you the option of doing that using the `sockets` method.

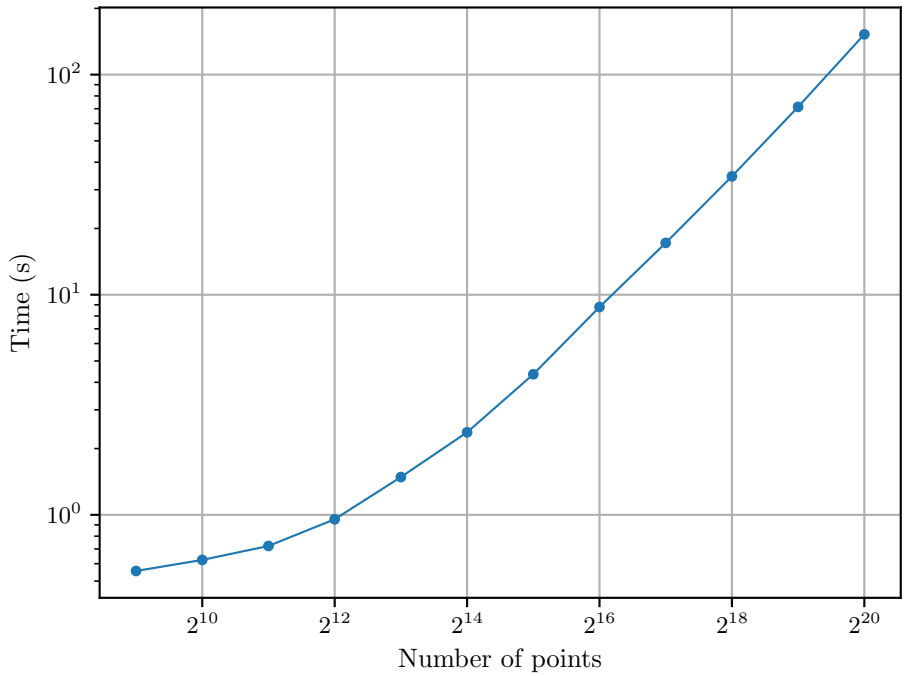

Figure 16: 3-D field exchange example with preCICE using the TCP/IP sockets communication protocol. Run times are shown plotted against number of points on the side of a cube.

4.2.4 Direct comparison of libraries

The communication mechanism responsible for sending and receiving each point in Figure 13 is the difference of interest when comparing these libraries. How developers have chosen to implement this varies considerably. As such, the most universal way to compare the libraries' applicability to large-scale simulations is to consider their run-time as a function of the number of points in the field-exchange.

5 Conclusion

In this work, we have investigated what libraries are currently available to couple scientific codebases together. In particular, we aimed to find software that could be run as part of large scale, high performance simulations.

Initially, we investigated using MUI, MOOSE, preCICE, OpenPALM and PLE. Nearly all of these libraries were running their first tutorials and workshops over the duration of this project: some time was spent going through this material and understanding which problems each library would be appropriate for.

In this initial stage, we ruled out some of those libraries, namely OpenPALM and PLE. While OpenPALM did have large amounts of documentation available, we still had difficulty building OpenPALM applications on our machines, meaning we could not replicate their claimed applicability to high-performance simulations. In this initial stage, PLE was shown to have limited documentation. This was a problem when we tried to create basic coupling applications using PLE. It was decided this was not an appropriate avenue to consider. Not only would it have been time consuming to try and learn how to use PLE using their limited documentation but we were aware that this would also be true of anyone we

recommended PLE to and, so, for this reason we did not take investigating PLE further than this.

We were able to run tutorial examples of MUI, MOOSE and preCICE on the machines available to us and, hence, more time was spent focusing on these when coming up with universal case-studies for comparison. The chosen case study to implement in each of these libraries one by one was the field-exchange example shown in Figure 13. This example was chosen because it was the most general code-coupling problem that we could think of. Our primary intention in making this choice was ensuring this project's work applied to the widest audience of CCPs and HECs interested in high-performance coupling. We wanted to zone-in on the mechanism of coupling variables together as much as possible and limit the compute time for anything that wasn't the coupling aspect of the example.

The result of this comparison was positive. We were able to provide evidence that MUI, MOOSE and preCICE with the `sockets` method are valid options for high performance code coupling. This is a fortunate result, as these three library options are all quite different and this provides the reader with options to consider for their own specific coupling problems. Primarily, one should consider if their problem fits under the category of a 'multi-physics' simulation as described by Equation 1 and 2, or a more general coupling problem that can be simplified down to communication messages. For the former, we recommend using MOOSE because it has been written precisely for people who have a list of coupled mathematical equations that they want to put into code. For the latter, we recommend either using MUI or preCICE. MUI uses MPI for its communication protocol, which you might like, but you might also prefer to keep your existing MPI implementation as it is without complicating your application by adding more messages. If this is the case, you could consider preCICE instead using TCP/IP sockets method. If you do chose to do this, however, we hope that you bear in mind that you may have a lot more difficulty building applications with preCICE compared to MUI, as the list of dependencies is far greater. In any case, we were pleased with this outcome as we are able to provide two options for the so-called 'general' type of coupling applications. A summary of our findings is provided in Table 1.

Library	Advertised as high performance	General coupling or multi-physics	Documentation available	Proven performance in this work	Recommended
MUI	✓	General Coupling	✓	✓	✓
MOOSE	✓	Multi-physics	✓	✓	✓
preCICE	✓	General Coupling	✓	✓	✓
OpenPALM	✓	General Coupling	✓	✗	✗
PLE	✓	General Coupling	✗	✗	✗

Table 1: Overall comparison chart

Acknowledgements

This work made use of computational support by CoSeC, the Computational Science Centre for Research Communities, through its Software Outlook activity.

I would also like to thank Stephen Longshaw and Judicael Grasset at STFC's Scientific Computing Department for their advice on useful coupling problems, and for providing the 3D field exchange example mentioned in this report.

A Running MUI with Heterogeneous Jobs on ARCHER2

Jobs using MUI should be submitted to the slurm cluster as heterogeneous jobs. We followed the ARCHER2 documentation to do this [40]. In short, if you would like to try out running some of the demo examples above, the following command should enable you to do this the same way as if you were on your own local machine.

```
salloc --nodes=1 --tasks-per-node=2 --cpus-per-task=1 --time=0:10:0 --partition=standard
    --qos=standard --account=<account-name> : --nodes=1 --tasks-per-node=2 --cpus-per-
    task=1 --time=0:10:0 --partition=standard --qos=standard --account=<account-name>
```

Listing 32: Heterogeneous jobs on ARCHER2 with `salloc`

B Building preCICE on ARCHER2

The most straightforward way to successfully build preCICE on ARCHER2 was to use Spack with some specific configurations. The following was found to be considerably easier than compiling from source or specifying pre-installed dependencies.

In `/work`, clone Spack.

```
$ git clone -b develop https://github.com/spack/spack.git
```

Listing 33: Clone Spack

Source Spack, allowing you to use Spack commands.

```
$ source spack/share/spack/setup-env.sh
```

Listing 34: Source Spack

You need to specify the location of Slurm on your system for Spack if you intend to submit jobs using the scheduler. Specifically, you need to find an `include/` directory containing `.h` files such as `pmi.h` and `slurm.h`, and a `lib64/` directory containing libraries such as `libpmi.so` and `libpmi2.so`. The following commands may help you locate these on your system:

```
$ whereis slurm
$ whereis libpmi
$ whereis libpmi2
$ whereis srun
$ whereis sinfo
```

Listing 35: Find system's Slurm installation

At the time of writing, the two directories were found at the locations below. As the two were not located in the same parent directory, a new directory had to be created with symlinks to the correct `include/` and `lib64/`.

```
$ mkdir slurm_hub
$ cd slurm_hub
$ ln -s /usr/inlcude/slurm include
$ ln -s /usr/lib64 lib64
```

Listing 36: Create `slurm_hub/ directory`

Next, in the `$HOME/.spack` directory, you will have to amend or create a `packages.yaml` file, providing the above slurm location

```
packages:
  slurm:
    buildable: false
    externals:
    - spec: slurm
      prefix: /path/to/slurm_hub
```

Listing 37: Specify pre-installed dependency for Spack

Now we can use Spack to build preCICE. You must first use the `GNU` environment.

```
$ module restore PrgEnv-gnu
```

Listing 38: Use GNU Environment

Export compiler name variables for ARCHER2

```
$ export CC=cc export CXX=CC export FC=ftn export F77=ftn export F90=ftn
```

Listing 39: Use GNU Environment

Then you can use the following command to configure MPI using the `--with-slurm` flag, which is not the default. It may take a long time to complete, but this is normal. At the time of writing this completes with no warnings or errors.

```
$ spack install precice ^boost@1.74.0 ^openmpi +pmi schedulers=slurm
```

Listing 40: Build preCICE

To run preCICE jobs you must use the interactive nodes on ARCHER2. This is because Spack needs to use your `$HOME/.spack` directory to do basic Spack commands. Unfortunately, Spack always looks for `/home/<project_name>` and during this work no method was found to redirect this. You can run the `solverdummy` example code in the following way as an interactive job. Note that you must provide arguments for `<account_name>` and `<spec>`.

```
$ salloc --nodes=1 --tasks-per-node=2 --cpus-per-task=1 --time=0:5:0 --partition=standard
    --qos=standard --account=<account_name>
$ source  spack/share/spack/setup-env.sh
$ spack load precice
$ cd $SPACK_ROOT/opt/spack/cray-sles15-zen2/gcc-10.2.0/precice-2.2.1-<spec>/share/precice
   /examples/solverdummies/cpp/
$ cmake .
$ make
$ ./solverdummy ../precice-config.xml SolverOne MeshOne & ./solverdummy ../precice-config
    .xml SolverTwo MeshTwo
```

Listing 41: Interactive job on ARCHER2

C OpenPALM Building and Requirements

The OpenPALM GUI, PrePALM, is written in Tcl/Tk and C, and also requires the STEPLANG interpreter. Configuration of the GUI on the local machine was very straightforward.

The OpenPALM's coupling library itself, PALM, requires MPI, as well as a Fortran 90 and C compiler. The PALM library allows configuration with MPI-1 mode and MPI-2 mode. It is stated in the OpenPALM documentation that users are not encouraged to use the MPI-1 mode, unless the MPI-2 mode is unavailable.

Over the course of writing this report, there was great difficulty with building PALM. On ARCHER 1 and ARCHER 2 building the library was only possible with very old versions of OpenMPI. The user guide specifies using OpenMPI 1.2.7, dated August 25th 2008 [41], despite the user guide itself being dated April 2019. Indeed, it was necessary to use the 2008 release of OpenMPI for successful building. It was not at all possible to use the default Cray MPICH libraries on ARCHER 1 or ARCHER 2. It was also difficult to find a compatible GNU suite. On ARCHER 1, PALM would only compile successfully using `gcc/4.8.1`, and compilers newer than this were not recognised. However, in this instance it is not clear that this is due to age, as on ARCHER 2 `gcc/9.3.0` was recognised by PALM and built.

After taking this caution above with compiler choices, both MPI-1 mode and MPI-2 mode were built. However, on a local linux machine, on ARCHER 1 and on ARCHER 2, the MPI-2 mode was found to be extremely error prone when attempting to build very simple applications such as those found in tutorials from the user guide, even with using OpenPALM's provided solutions. Therefore, over the course of this work, only MPI-1 mode was used. This is unfortunate as it states in their documentation that the original PALM development was based on the MPI-2 standard. MPI-1 mode is described as being the "restrained" and "degraded" version of PALM in their documentation. There are features which are only possible with the MPI-2 standard, for example you cannot relaunch blocks multiple times with MPI-1 mode and so the source code has to be amended to cope with this limited functionality. Running applications with MPI-1 mode was still difficult and inconsistent, with still some tutorial examples still not running but this was completely impossible with MPI-2 mode.

PALM could only be built on Linux machines successfully, although their website does have user guides for installing on a Mac this was not possible at all in my experience.

Attempts were made to create the 3D field exchange implementation in OpenPALM in MPI-1 mode, however on ARCHER 1 and ARCHER 2 this did not run reliably and there was indication that the communication between the two sides were not working as expected and, for this reason, this does not feature in the performance comparison. This could be down to not using the MPI-2 mode as advertised, however, given the other difficulties described in this report, this is not clear.

It is unfortunate that there was such difficulty to use OpenPALM over the course of this report, as OpenPALM appeared to be a prime candidate for code coupling at scale, given the quoted successes on their website as stated earlier. As well as this, it also initially appeared to be one of the more user-friendly libraries, having a lengthy user guide with plenty of very thorough examples. However, having made this assessment in Software Outlook that such problems exist, the difficulties with OpenPALM are presented in this report in the hope that it will save time for the CCPs and HECs we support. Research groups looking to choose a code coupling library for their multi-physics work would hopefully take the recommendation from this report to use one of the other libraries featured in this document, where there was more success in building applications.

References

[1] Stephen Longshaw, Alex Skillen, et al. "Code Coupling At Scale: Towards The Digital Product". In: May 30, 2017, p. 28. ISBN: 978-1-874672-72-2. DOI: 10.4203/csets.40.

[2] David E Keyes, Lois C McInnes, et al. "Multiphysics simulations: Challenges and opportunities". In: *The International Journal of High Performance Computing Applications* 27.1 (Feb. 1, 2013), pp. 4–83. DOI: 10.1177/1094342012468181.

[3] D. A. Knoll and D. E. Keyes. "Jacobian-free Newton–Krylov methods: a survey of approaches and applications". In: *Journal of Computational Physics* 193.2 (Jan. 20, 2004), pp. 357–397. DOI: 10.1016/j.jcp.2003.08.010.

[4] *PETSc Website*. URL: https://www.mcs.anl.gov/petsc/.

[5] *SUNDIALS Website*. URL: https://computing.llnl.gov/projects/sundials.

[6] *Trilinos Website*. URL: https://trilinos.github.io/.

[7] Yu-Hang Tang, Shuhei Kudo, et al. "Multiscale Universal Interface: A concurrent framework for coupling heterogeneous solvers". In: *Journal of Computational Physics* 297 (Sept. 15, 2015), pp. 13–31. DOI: 10.1016/j.jcp.2015.05.004.

[8] *Hartree Centre - Multiphysics and Multiphase Code Coupler — Scalable and flexible code coupling*. URL: https://www.hartree.stfc.ac.uk/Pages/scalable-and-flexible-code-coupling-.aspx.

[9] *MUI Demos GitHub Repository*. URL: https://github.com/MxUI/MUI-demo.

[10] *MUI GitHub Repository*. URL: https://github.com/MxUI/MUI.

[11] *MOOSE Website*. URL: https://mooseframework.inl.gov/.

[12] *MOOSE Virtual Workshop (Summer 2020)*. June 15, 2020. URL: https://www.youtube.com/watch?v=2tJwBsYaLaI.

[13] *MOOSE Examples and Tutorials*. URL: https://mooseframework.inl.gov/getting_started/examples_and_tutorials/index.html.

[14] *MOOSE GitHub Repository*. URL: https://github.com/idaholab/moose.

[15] *MOOSE Software Quality*. URL: https://mooseframework.inl.gov/sqa/index.html.

[16] *libMesh Website*. URL: https://libmesh.github.io/.

[17] *preCICE Website*. URL: https://precice.org/index.html.

[18] *preCICE XML reference*. URL: https://precice.org/configuration-xml-reference.html.

[19] *preCICE GitHub Repository*. URL: https://github.com/precice/precice.

[20] *preCICE Configuration Guide*. URL: https://precice.org/configuration-introduction.html.

[21] *preCICE 'Couple your code' Guide*. URL: https://precice.org/couple-your-code-preparing-your-solver.html.

[22] *preCICE Discourse Forum*. URL: https://precice.discourse.group/.

[23] Hans-Joachim Bungartz, Florian Lindner, et al. "preCICE – A fully parallel library for multi-physics surface coupling". In: *Computers & Fluids*. Advances in Fluid-Structure Interaction 141 (Dec. 15, 2016), pp. 250–258. DOI: 10.1016/j.compfluid.2016.04.003. URL: https://www.sciencedirect.com/science/article/pii/S0045793016300974.

[24] *Building preCICE on clusters and supercomputers*. URL: https://precice.org/installation-special-systems.html.

[25] *preCICE Demo Virtual Machine*. URL: https://precice.org/installation-vm.html.

[26] *preCICE Mapping Configuration*. URL: https://precice.org/configuration-mapping.html.

[27] *OpenPALM Website*. URL: https://www.cerfacs.fr/globc/PALM_WEB/index.html.

[28] Thierry Morel, Florent Duchaine, et al. *OpenPALM coupler version 4.3.0 User guide and training manual*. Apr. 2019. URL: https://www.cerfacs.fr/globc/PALM_WEB/pdfs/user_guide.pdf.

[29] *OASIS Website*. URL: https://portal.enes.org/oasis (visited on 02/27/2021).

[30] *CERFACS Research Areas: Code Coupling and User Interface*. URL: https://cerfacs.fr/en/code-coupling-and-user-interface/.

[31] *The CWIPI coupling library*. URL: https://w3.onera.fr/cwipi/bibliotheque-couplage-cwipi.

[32] *Code_Saturne Website*. URL: https://www.code-saturne.org/cms/.

[33] Yvan Fournier. "Massively Parallel Location and Exchange Tools for Unstructured Meshes". In: *International Journal of Computational Fluid Dynamics* 34.7 (Sept. 13, 2020), pp. 549–568. DOI: 10.1080/10618562.2020.1810676.

[34] *EDF R&D Simulation Software Webpage*. EDF France. URL: https://www.edf.fr/en/the-edf-group/world-s-largest-power-company/activities/research-and-development/scientific-communities/simulation-softwares.

[35] *Code Saturne GitHub Repository*. URL: https://github.com/code-saturne/code_saturne.

[36] *PLE GitHub Repository*. URL: https://github.com/code-saturne/code_saturne/tree/master/libple.

[37] *PLE (Parallel Location and Exchange) documentation*. URL: https://www.code-saturne.org/cms/sites/default/files/docs/ple-2.0/html/index.html.

[38] *SCD CIUK 2019 Presentations*. URL: https://www.scd.stfc.ac.uk/Pages/CIUK-2019-Presentations.aspx.

[39] *ARCHER2 Website*. URL: https://www.archer2.ac.uk/.

[40] *ARCHER2 User Documentation: Running heterogeneous jobs*. URL: https://docs.archer2.ac.uk/user-guide/scheduler/#heterogeneous-jobs.

[41] *Open MPI Releases*. URL: https://www.open-mpi.org/software/ompi/v1.2/.