Abstract

This paper presents the design of multiphysics integration platform called MuPIF. MuPIF mission lies in integrating existing computational codes in a multiphysics framework, taking care of code steering and data exchange under parallel or distributed mode. The platform requires that each application implements its own application interface (API), which allows access to solution domains and fields. MuPIF is written in Python scripting language and uses Pyro4 module for accessing remote objects over a network. A few examples demonstrate basic application of the platform to simple examples.

Keywords: multi-physics, simulations, software integration, distributed computing.

1 Introduction

Multiscale/multiphysics methods have contributed to a considerable progress in several fields such as electronics, photonics, nanotechnology, biotechnology, advanced manufacturing and processing of alloys, directed molecular evolution etc. The progress is traditionally based on computational science, formulating mathematical and computational models with verification and validation stages on several scales [1].

The reliable multiscale/multiphysics modeling requires including all relevant physical phenomena along the process chain and across multiple scales, combining the knowledge from multiple disciplines. Development of such a new tool would be generally difficult due to time and resource constraints. However, combination of single-physics tools presents a viable approach to build a customized multiphysics simulation chain for a particular problem. Such a modular approach allows reusing existing software tools to tackle multiscale/multiphysics problems and their coupling needs to

be resolved. A multiscale/multiphysics integration platform MuPIF presented in this paper opens this option for data exchange and steering individual applications.

A coupled problem usually requires global time steps for a proper data synchronization. The coupling basically occurs in two ways:

- Weakly coupled (staggered, one-way) formulation. There is no requirement on consistency of internal variables across applications. For example, thermo-mechanical coupling computes at first a temperature field which is passed to mechanical part, see Figure 1. If the thermal model does not depend on mechanical results, there is no requirement on consistency from both models.
- Strongly coupled formulation requires consistency based on global criteria. Fluid-structure interaction with moving boundaries presents such an example.

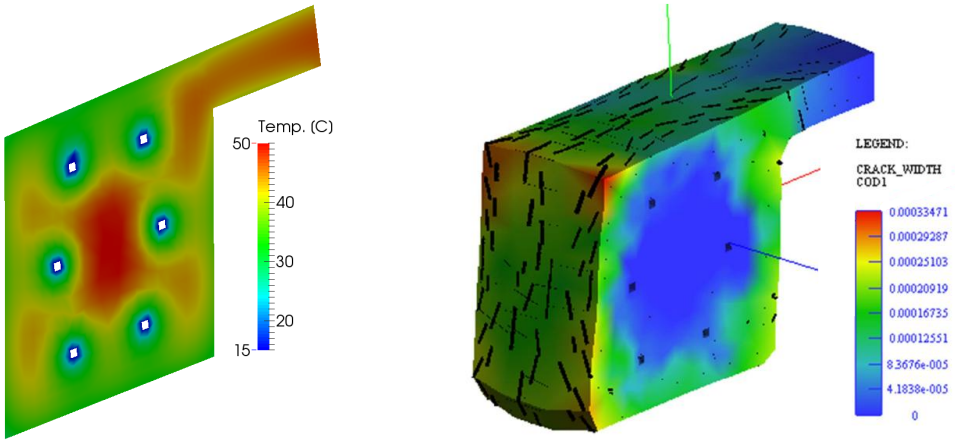

Figure 1: Example of thermo-mechanical simulation. Simulation of Oparno's bridge arch ending with the prediction of crack width [2].

Nowadays, several multiphysics simulation codes are available. To mention a few, commercial products include COMSOL Multiphysics [3], or multiphysics embodied within monolithic commercial codes (e.g. ADINA [4], MSC Nastran [5], ANSYS Multiphysics [6]), as well as open-source programs (e.g. Elmer [7], OpenFOAM [8]). However, the integration of external simulation tools is rather limited.

Multiscale/multiphysics computing are usually resource and time demanding. Tasks can run on different computers in distributed or parallel mode. This brings further requirements on multiscale modeling platform, defining communication mechanisms over a computer network, job management, data security etc.

Several frameworks have been developed to ease the implementation of large-scale parallel simulations, such as POOMA [9], Overture [10], SAMRAI [11], ALEGRA [12], ASV [13], OASIS [14], SIERRA [15], and PALM [16], adopted for a number of multiphysics simulations in computational fluid dynamics. These platform are typically focused on specific techniques and applications and none of them

has become widely adopted outside its field of application. A major difficulty of the above-mentioned platforms is the required conversion of software. This means rewriting, loss of control over many aspects of the software, and its dependence on the framework itself.

When designing a new platform, major challenge remains in flexibility which allows:

- Integration of external codes without reprogramming them.
- Supporting various data formats.
- Definition of simulation scenarios and coupling.

The lack of standardization for data formats or communication interfaces represents probably the main limiting factor in the widespread usage of integrated solutions by commercial as well as academic users. Today, there are ongoing standardization activities led by various groups; The Minerals, Metals, and Materials society (TMS) [17], Integrated Computational Materials Engineering expert group (ICMEg) [18], and European Materials Modelling Council (EMMC) [19] focusing on establishing the common language and standards.

This paper describes major features of a newly developed integration platform called MuPIF [20]. The platforms is object-oriented and focuses on services of different classes rather than on data itself. Data from various codes are exchanged through Application programming interfaces (APIs). MuPIF handles the distributed and parallel simulation scenarios, providing a transparent communication mechanism for the remote components.

2 Design of MuPIF platform

Design of MuPIF platform is based on an object-oriented approach. MuPIF is written in Python 2.7 for portability on different operating systems [20]. MuPIF defines base (abstract) classes which represent various entities such as field of unknowns, discretization etc. Each class contain services, e.g. *giveValue()* and *setValue()*, and attributes, e.g. units, origin, time step. The most relevant base classes are:

- **Application** - defines abstract services for data exchange and steering. Provides methods for e.g. getting and registering properties, fields, functions. Steering services allow executing solution for a specific solution step, update solution state, terminate the application, etc.
- **Field** - a scalar, vector, or tensor quantity defined on a spatial domain (represented by the Mesh class). The class provides interpolation services in space, but is assumed to be fixed in time (the application interface allows to request

field at specific time). The fields are usually created by the individual applications (sources) and are passed to target applications. Derived classes will typically implement fields defined on common discretizations, like fields defined on structured or unstructured FE meshes, finite difference grids, etc. Basic services provided by the fields include methods for evaluating the field at any spatial position and methods to support graphical export (creation of VTK dataset, for example).

- **Property** - characteristic value of a problem, which has no spatial variation, e.g. homogenized property. Property is identified by PropertyID, which is an enumeration determining its physical meaning. Properties keep their value, type, and units.
- **Function** - a class defined by mathematical expression and can be a function of spatial position, time, and other variables.
- **Mesh** - representation of a computational domain as a collection of Cells and Vertices. Derived classes represent structured, unstructured FE grids, FV grids, etc. Mesh is assumed to provide a suitable instance of cell and vertex localizers, allowing to perform efficient spatial requests. In general, the mesh services provide different ways how to access the underlying interpolation cells and vertices, based on their numbers, or spatial location.
- **Vertex** - represents a vertex (node) in discretized domain.
- **Cell** - represents a finite volume (finite element).
- **TimeStep** - holds solution time step.

The important role of the abstract classes is to define a common interface (in terms of provided services), that has to be implemented by any derived class. For example, a field from an application can be stored in Field instance of the class with associated Mesh instance. Calling a service *Field.evaluate(position)* calls a derived (inherited) service from the base class and interpolates a value from the field at a given position. Figure 2 shows such a mapping from a fine discretization to a coarse one which is commonly encountered when solving multiphysical/multiscale tasks.

2.1 Application interface - API

Application interface (API) presents a "glue" of different simulation tools which is defined by base **Application** class. The class declares services like *getMesh()*, *getField()*, *setField()*, *solveStep()* responsible for data exchange and steering of individual simulation tools. There are two API implementations for data exchange; direct or indirect.

Direct implementation requires direct communication with a particular simulation tool through memory access. This either requires a simulation tool written in Python,

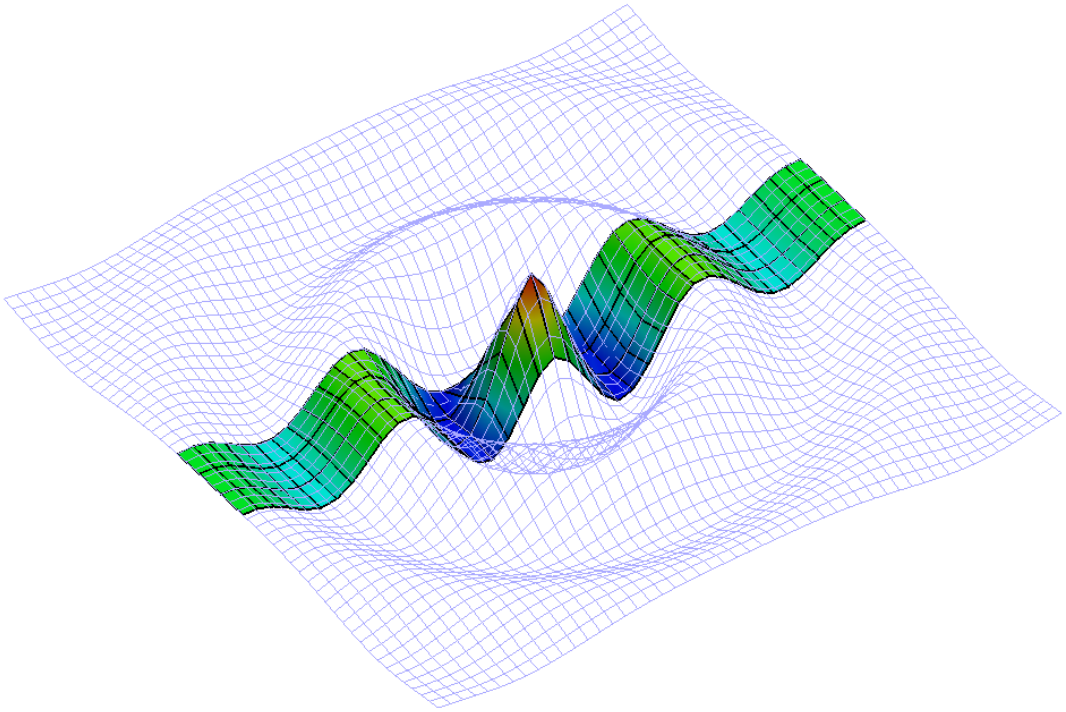

Figure 2: Mapping of a fine field to a coarser discretization in the MuPIF platform.

or with a Python interface. In general, each application (in the form of a dynamically linked library) can be loaded and called from Python, but care must be taken to convert Python data types into correct data types. More convenient is to use a wrapping tool (such as Boost) that can generate a Python interface to the function, generally taking care of data conversions for the basic types. The result of wrapping is a set of Python functions or classes, representing their application counterparts. The user calls an automatically generated Python function which performs data conversion and calls the corresponding native equivalent. The ability of the presented platform to support multiple data formats allows to directly support native application data format (by implementing underlying classes representing data fields) so finally data conversion is not necessary.

Indirect implementation is based on a wrapper class for API. A characteristic example is data exchange through input/output files which have some structure and an application proceeds with a single solution step. For a typical solution step, the wrapper class has to cache all input data internally (by overloading corresponding set methods), execute the application from previously stored state, passing input data, and parsing its output(s) to collect return data (requested using get methods). This is the case for the majority of commercial codes.

Comparing these two approaches, the direct approach is more efficient, as data can be exchanged directly in memory. However, it requires the access to source code of individual applications, or at least the access to application in the form of library

with necessary functions exposed. On the other hand, the indirect approach is less intrusive and applicable to a wide range of applications, requiring only capability to start solution step from the previously stored one. The disadvantages consist in rather inefficient input/output data passing.

The complex simulation pipeline is defined by a top-level script in Python language. This script invokes individual applications, synchronizes the data exchange, checks for convergence in each solution step. The listing in Table 1 illustrates the top-level script in the case of two weakly coupled linear applications. It illustrates the initialization of individual applications, construction of time-step loop, invocation of individual applications in a staggered way including the data exchange in terms of properties.

3 Parallel and distributed environments in MuPIF

MuPIF uses a module Pyro4 which is an abbreviation for PYthon Remote Objects. Pyro4 takes care of the network communication between the objects when they are distributed over different machines on the network, hiding all socket programming details. One just calls a method on a remote object as if it were a local object – the use of remote objects is (almost) transparent. This is achieved by the introduction of so-called proxies. A proxy is a special kind of object that acts as if it were the actual object. Proxies forward the calls to the remote objects, and pass the results back to the calling code. In this way, there is no difference between simulation script for local or distributed case, except for the initialization, where, instead of creating local application interfaces, one has to connect to their remote counterparts. A name server is devoted to localize URL of the remote counterparts.

Figure 3 demonstrates a classical example of multiphysics simulation, with results similar to Figure 1. Three computers could be involved; one runs the Top level script, the second runs heat transport and the third mechanical analysis. The application may have more instances on more CPU cores. App1 shows indirect implementation for data exchange through input/output files stored on a disk. App2 uses direct implementation with direct data access to the memory. TCP/IP protocols are exchanged over a network in encrypted or unencrypted way. Pyro4 name server registers URLs of involved remote objects to know where to send a request to.

The framework must ensure safe operation in distributed environments. Most of the concerns are naturally related to the use of communication layer built in a Pyro4 library. Pyro4 does not encrypt the data it sends over the network. This means that sending sensitive data over internet (especially user data, passwords) should be avoided as the data can be easily eavesdropped. When the data security is of concern, data can be passed through a secure ssh tunnel at an increased cost for the data transfer and coding.

When local objects are published (made available remotely), Pyro4 starts listening for incoming requests and processes them. This is handled by the Pyro4 daemon.

```
time  = 0
timestepnumber=0
targetTime = 10.0

app1 = application1(None)
app2 = application2(None)

while (abs(time -targetTime) > 1.e-6):
    #determine critical time step
    dt = min(app1.getCriticalTimeStep(), app2.getCriticalTimeStep())
    #update time
    time = time+dt
    if (time > targetTime):
        #make sure we reach targetTime at the end
        time = targetTime
    timestepnumber = timestepnumber+1
    # create a time step
    istep = TimeStep.TimeStep(time, dt, timestepnumber)

    try:
        #solve problem 1
        app1.solveStep(istep)
        #request Concentration property from app1
        c = app1.getProperty(PropertyID.PID_Concentration, istep)
        # register Concentration property in app2
        app2.setProperty (c)
        # solve second sub-problem
        app2.solveStep(istep)

    except APIError.APIError as e:
        print ("Following API error occurred:",e)
        break

prop = app2.getProperty(PropertyID.PID_CumulativeConcentration, istep)
print ("Result: ", prop.getValue())
# terminate
app1.terminate();
app2.terminate();
```

Table 1: Example of MuPIF script illustrating the data exchange between two applications.

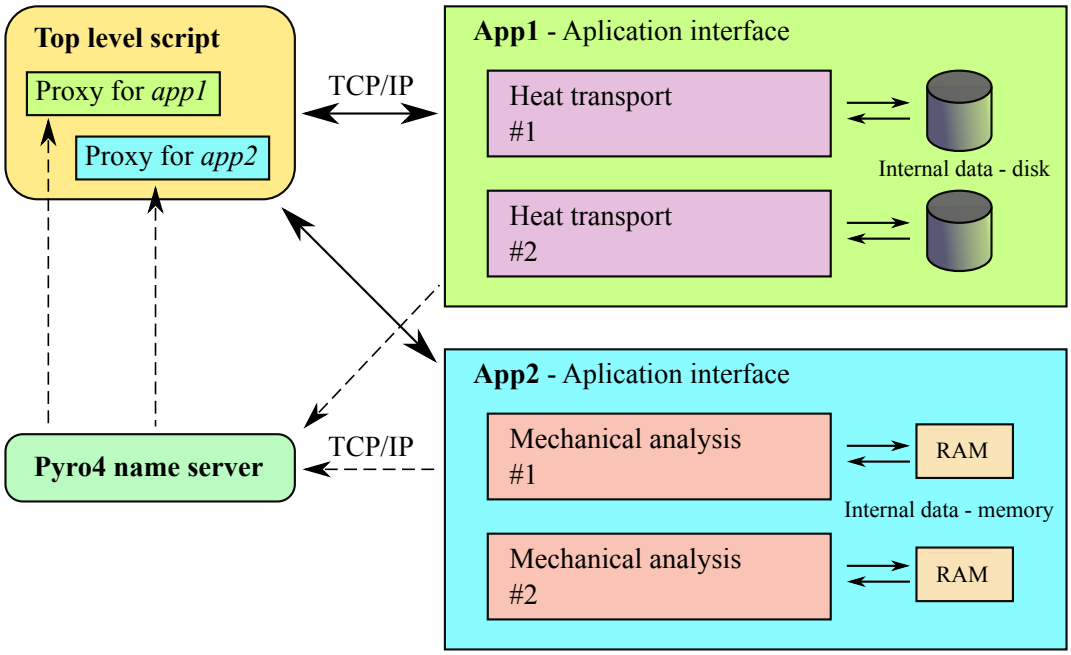

Figure 3: Operation of MuPIF platform in a thermo-mechanical analysis.

To prevent malicious requests to the daemon from the unauthorized source, Pyro4 provides a concept of HMAC signature that is attached to every network transfer to allow only legitimate connections. Additionally, the firewall rules can be set up on the participating computing nodes to allow remote connection to Pyro4 network ports only from trusted clients.

4 Conclusions

This paper presents basic description and features of an open-source, object-oriented, multiphysics integration platform called MuPIF [20]. The platform is designed to interconnect various commercial/open-source computing codes, to implement data exchange through a network, to provide various services for field mapping on different discretizations and to steer individual applications.

Acknowledgments

This work was carried out within the project Multiscale Modelling Platform: Smart design of nano-enabled products in green technologies (MMP, Grant agreement no: 604279) and the authors gratefully acknowledge the financial contribution of the European Community's Seventh Framework Programme.

References

[1] I. Babuška, F. Nobile, R. Tempone, "Reliability of computational science", *Numerical Methods for Partial Differential Equations*, 23(4): 753–784, 2007, ISSN 1098-2426.

[2] L. Jendele, V. Šmilauer, J. Červenka, "Multiscale Hydro-thermo-mechanical Model for Early-Age and Mature Concrete Structures", *Advances in Engineering Software*, 72: 134–146, 2014.

[3] Comsol, *COMSOL Multiphysics*, 2015, URL `http://www.comsol.com`.

[4] Adina R&D, *Adina*, 2015, URL `http://www.adina.com`.

[5] MSC Nastran, *Nastran*, 2015, URL `http://www.mscsoftware.com`.

[6] ANSYS, *Ansys*, 2015, URL `http://www.ansys.com`.

[7] CSC - IT Center for Science, *Elmer*, 2015, URL `http://www.csc.fi/english/pages/elmer`.

[8] OpenCFD, *OpenFOAM*, 2015, URL `http://www.openfoam.com`.

[9] Los Alamos National Laboratory, *POOMA*, 2015, URL `http://acts.nersc.gov/formertools/pooma/index.html`.

[10] Rensselear Polytechnic Institute and LLNL, *Overture*, 2015, URL `http://www.overtureframework.org/`.

[11] B.T. Gunney, A.M. Wissink, D.A. Hysom, "Parallel clustering algorithms for structured {AMR}", *Journal of Parallel and Distributed Computing*, 66(11): 1419 – 1430, 2006, ISSN 0743-7315, URL `http://www.sciencedirect.com/science/article/pii/S0743731506000803`.

[12] A. Robinson, W. Rider, et al., "ALEGRA: An Arbitrary Lagrangian-Eulerian Multimaterial, Multiphysics Code", in AIAA (Editor), *Proceedings of the 46th AIAA Aerospace Sciences Meeting*. Reno, NV, 2008.

[13] Advanced Visual Systems Inc., *AVS - Advanced Visual Systems*, 2015, URL `http://www.avs.com`.

[14] R. Redler, S. Valcke, H. Ritzdorf, "OASIS4 – a coupling software for next generation earth system modelling", *Geoscientific Model Development*, 3(1): 87 – 104, 2010.

[15] H.C.E. J. R. Stewart, "A Framework Approach for Developing Parallel Adaptive Multiphysics Applications", *Finite Elements in Analysis and Design*, 40(12): 1599–1617, 2004.

[16] D.D. S. Buis, A. Piacentini, "PALM: A Computational framework for assembling high performance computing applications", *Concurrency Computat.: Pract. Exper.*, 18(2): 247–262, 2006.

[17] The Minerals, Metals & Materials Society (TMS), 2015, URL `http://www.tms.org/`.

[18] Integrated Computational Materials Engineering expert group (ICMEg), 2015, URL `http://www.icmeg.eu`.

[19] The European Materials Modeling Council (EMMC), 2015, URL `http://www.emmc.info/`.

[20] Multi-Physics Integration Framework (MuPIF), 2015, URL `https://sourceforge.net/projects/mupif/`.