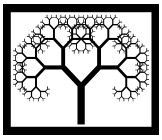
Paper 3



Proceedings of the Eighth International Conference
on Engineering Computational Technology,
B.H.V. Topping, (Editor),
Civil-Comp Press, Stirlingshire, Scotland

On the Design of a Parallel, Distributed Multi-Physics Integration Tool

B. Patzák, D. Rypl and J. Kruis
Department of Mechanics, Faculty of Civil Engineering
Czech Technical University, Prague, Czech Republic

Abstract

This paper presents the design of a distributed multi-physics integration tool, with the aim of providing high-level support for the integration of existing applications into full featured multi-physics simulation tools. The abstract and application of independent steering and data exchange are achieved by defining an application interface, that has to be implemented by individual applications. The design supports various coupling strategies and discretization techniques. The focus of the work presented is on the distributed aspects of the framework design.

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

1 Introduction

Numerical simulations are now routinely used in research and industry and are accepted as reliable analysis tools. The existing knowledge in various disciplines has been integrated into advanced simulation tools, that have been developed over many years. To enable further progress in many fields, including structural and material engineering, the integration of existing knowledge from different disciplines is necessary. Therefore, one of the challenges in numerical modeling is to enable an efficient use and inter-operation of a wide variety of problem-specific computational tools to simulate, analyze, and understand complex, multi-physics problems [1, 2, 3, 4].

Traditionally, single purpose programs or shell scripts are used to integrate existing applications. Such an approach may initially look promising, but later it becomes clear, that there are many restricting issues and limitations. Such an approach provides a very low level of interoperability - data are usually exchanged using files, allowing inter-operation at program level, rather than at function level, that allows to combine different codes. Usually, substantial coding and deep understanding of the code struc-

ture is required. The level of code reuse is typically very low and this makes extension and maintenance quite hard.

At present, there are many multi-physics simulation codes available, notably COMSOL Multiphysics [5]; advanced multi-physics capabilities can be found in many monolithic commercial codes (ADINA [6], MSC Nastran [7], ANSYS Multiphysics [8], etc), as well as open-source programs (Elmer [9], OpenFOAM [10], etc). Several frameworks have been developed to ease the implementation of large-scale, parallel simulations, such as POOMA [11], Overture [12], SAMRAI [13], ALEGRA [14], AVS [15], OASIS [16], SIERRA [17], and PALM [18], adopted for a number of multi-physics simulations in computational fluid dynamics. These frameworks have brought many innovations in computational and software technology, however, they are typically focused on specific techniques and applications and none of them has attracted a large user base or has been widely adopted outside their field of application. A major issue is the required conversion of software to make use of these frameworks. This means rewriting, loss of control over many aspects of the software, and its dependence on the framework itself.

The paper starts with introduction to the main objectives of the presented work. Next, the object-oriented design of developed multi-physics integration tool [19] is presented in Section 3, introducing the fundamental classes, discussing their role, interface and mutual relations. The support for distributed multi-physics applications is discussed in Section 4 and performance of distributed field mapping is illustrated on several examples in Section 5.

2 Motivation

The main goal of the presented framework is to provide a high-level support for mutual data exchange between codes, including support for different discretization techniques and specific field transfer operators, aware of underlying physical phenomena. The field representation and field exchange methods support various data types (scalar, vector, and tensorial values). The role of high-level services is to provide a simple and seamless data transfer between applications, which is independent of actual discretization. This naturally enables to design a general representation of solution fields, that will provide high-level mapping and interpolation services, allowing to evaluate solution fields at any point of problem domain and hiding all details of internal representation. Unified data access services provided by the framework will also facilitate other common tasks, such as post-processing. The implemented library is integrated into interactive scripting Python [20, 21] environment.

Individual applications typically require specific meshes and even different discretization techniques. The underlying physics may also impose additional constraints on mapping or interpolation operators, such as mass or energy conservation requirements. The object-oriented structure of the framework will naturally enable implementation of user-specific mapping and interpolation operators, as required by individ-

ual problems under consideration. Efficient steering and data exchange require some kind of control channel, implementing Application Interface. This channel would allow a framework to call the individual codes at appropriate times, handle exceptional situations and request/update application data. Such an approach is very flexible and allows communication with particular applications on an abstract level, permitting easy addition/replacement of components.

The available strategies for handling coupled problem composed of individual components (applications) can be divided into so-called strongly or weakly coupled schemes, depending whether consistency of internal values across applications is required at each global time step. In the case of strongly coupled scheme, the consistency based on a global convergence is required, while in the weakly coupled scheme not. In both approaches, a sequence of global time steps has to be established, based for example on global Courant Friedrichs Lewy (CFL) condition, time-scale of relevant physical phenomena, or accuracy constraints. Within the limits of the global time step, each application can choose its own local time stepping.

The parallel and distributed applications and associated aspects are addressed as well. The parallel applications typically come with distributed parallel data structure and several application instances running on different processors. The application interface design allows a unified application data access and steering, hiding the differences between individual applications and allowing to manage serial and parallel applications using the same interface. Moreover, the support for efficient parallel field transfer of distributed data is incorporated.

The object oriented design naturally allows to group several application together into one logical group, that can be manipulated using single interface. A task agent will steer the individual applications within a group (using their application interfaces) and will manage the inter-group data exchange. When task agent will itself implement application interface, it can be at higher level regarded as a single application. The hierarchical structure of cooperating agents and individual applications is characteristic design pattern of proposed framework (see Fig. 1). Such design will allow to efficiently utilize parallel and grid computing resources.

3 Framework design

The object-oriented data structure consists of several top level classes representing the fundamental blocks of the Multi-Physics Integration Framework, called MuPIF [19]. Its overall structure is shown in Fig. 2 using UML notation (see [22] for details). A first step is the abstraction of a computational domain, represented by a class derived from abstract *Domain* class. The purpose of domain is to provide unified description of the geometry of problem domain, represented by a set of interpolation cells, geometry of which is defined using vertices. The individual cells (represented by corresponding classes, derived from base *Cell* class) may be of different type, to represent finite-volume or finite-element meshes or finite difference grids, for example.

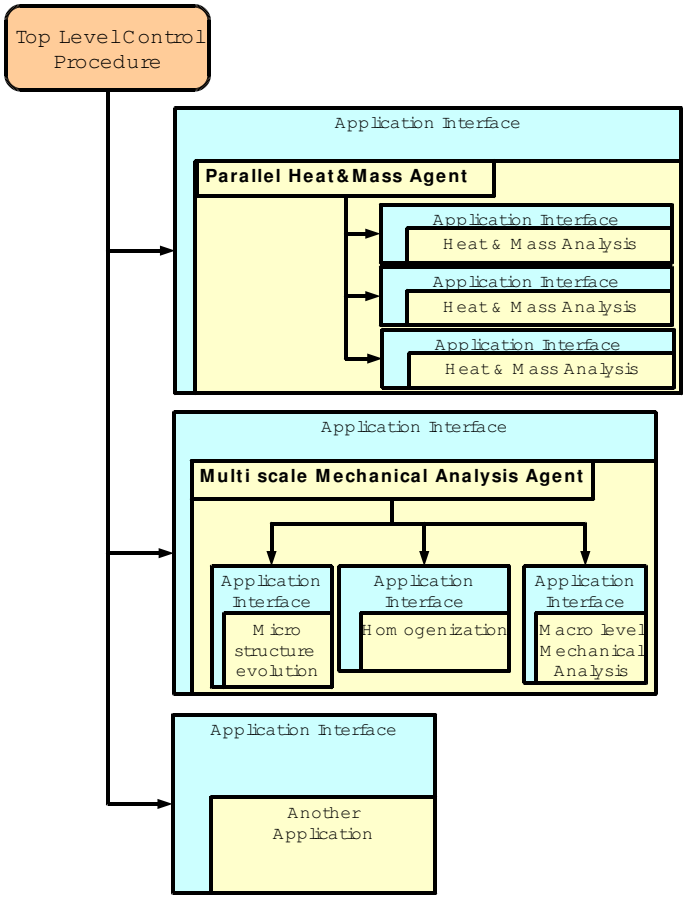


Figure 1: Example of hierarchy of cooperating agents.

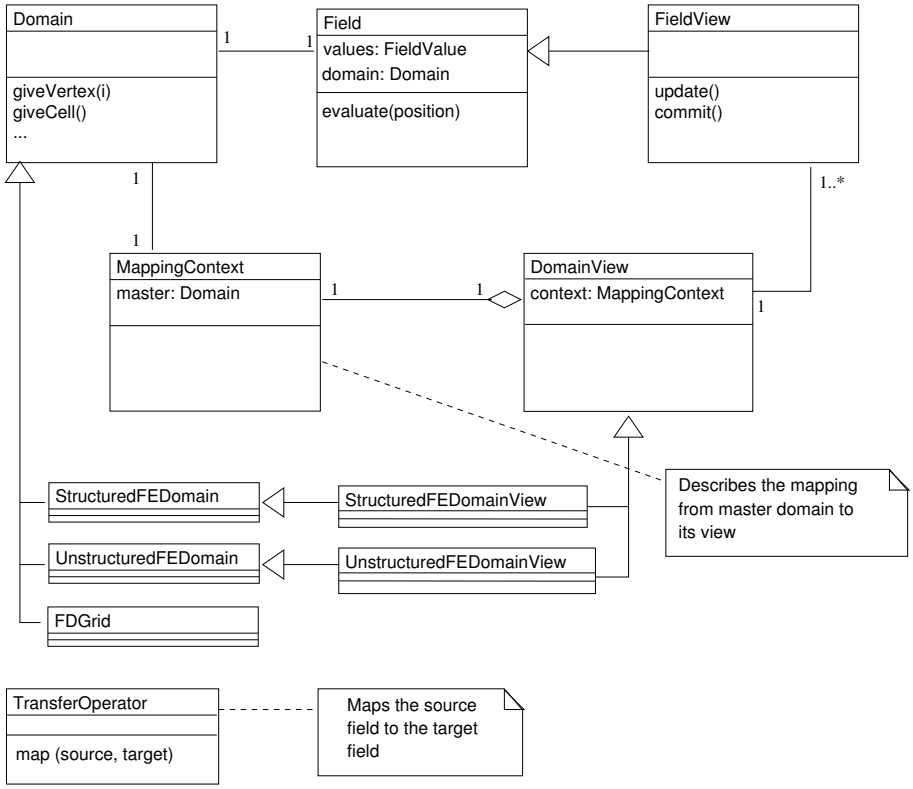


Figure 2: Design of MuPIF framework.

Domain class declares abstract services for spatial search (target cell location, for example) and interpolation (based on given field data). Inheritance allows to extend this approach and represent a subset of any domain using the same interface. Such subsets (thereafter called views) are naturally required in many applications, as they may represent material regions, boundaries of domain, or even decomposition of domain into partitions in case of parallel applications, for example. Usually there is one-to-one relation between cells in view and cells in corresponding master domain, however, in case of view representing boundary of a domain, the cell-types in the domain view are different from domain cell-types, as they represent a subset of lower dimension. To represent views, the framework introduces *DomainView* class, derived from *Domain* class. A direct consequence of being inherited from *Domain* class is the need to implement interface defined by its parent. Thus, the *DomainView* class can be used anywhere, where *Domain* class is expected. The mapping from master *Domain* to any of its *DomainView* is defined by corresponding *MappingContext* class, which is an attribute of *DomainView*. This class provides methods defining the mapping of view vertices and cells to their counterparts on the master domain. In a such way, any number of domain views can be created, depending on application needs. Typically, the instances of *Domain* are created by individual applications.

The individual fields may represent scalar, vector, or tensor quantities, defined on

different types of discretizations. One of the main objectives is to obtain an abstract representation of solution fields, allowing to evaluate individual fields at given position and implement field transfer operators independently of underlying discretization. In MuPIF, the solution fields are represented by hierarchy of classes derived from base *Field* class, declaring the common abstract interface. Each *Field* class manages the values associated to each vertex (or cell) of the corresponding domain. To support fields of different types, the abstract *FieldValue* class has been defined, where derived classes represent basic types of field values (scalar, vector, tensor, etc) and define common operations. The base *Field* class defines fundamental services for field data evaluation at given point. By relying on cell interpolation services, the implementation of *Field* is independent of particular type of mesh. Analogously to the concept of domain view, the *FieldView* class (derived form *Field*) enables transparent representation of field data associated to domain view. One may regard the *FieldView* as a proxy class, that maps field view data defined on associated domain view to global field data defined on master mesh. The *FieldView*, with associated *DomainView* (and its *MappingContext*), uniquely defines the subset of field data represented by the field view. Direct consequence of inheritance and domain abstraction allows to reuse many *Field* class methods by *FieldView* class, including *evaluate* service.

For some applications, it may be beneficial for the field view to maintain its local copy of master field data. This is particularly needed in distributed applications, where the master mesh is distributed data structure and the domain view and field view may represent data composed of contributions from different processors. In such a case, the data access to individual field view data entries may be very slow and inefficient due to non-locality of the source data. A logical solution is to maintain the local copy of the data, thus data is transferred only once. At the same time, when the field view data is updated, the changes can be recorded and later committed to master field. Both described approaches are supported by corresponding classes derived from base *FieldView* class.

The mutual data exchange between individual applications is assumed to happen through field transfer operations. In this model, one or more applications (providers) deliver source field(s), defined on corresponding domain(s), while another application (receiver) requires or accepts the source field. The source field is typically defined on discretization provided by provider, which is, in general, different from receiver's domain (the only requirement is that the domains should occupy the same space). The source field has to be mapped into receiver's discretization in order to obtain vertex or integration point values. The application to different problems requires the use of problem specific mapping algorithms, respecting physically or numerically based requirements. The generic mapping operator is represented by an abstract class *TransferOperator*. Its role is to hide all implementation details under common interface, consisting of *map* method, which takes the source and target fields as parameters and implements mapping from target to source field. Some operators may require to build some intermediate representation, like a least-square approximation of the target field and this is naturally encapsulated in a derived classes implementing particular algorithms. Also note, that since the *FieldView* is derived from *Field* class, the *FieldView*

can be passed as parameter to any method expecting *Field* class as parameter as well (polymorphism), extending the use of the same mapping operator for more general cases.

4 Distributed environments

In a case of parallel and distributed applications, an additional level of complexity has to be addressed. The individual applications can be physically distributed over the network. The important role of the framework is to provide a transparent communication mechanism between individual objects that will take care of the network communication between the objects if necessary. When solving large-scale problems, individual sub-problems are solved in parallel on shared or massively-parallel computing resources. The data retrieval and processing should be performed in parallel as well, without compromising the scalability. Particularly, the scalable implementation of field mapping is quite challenging. The key idea is to represent needed remote data on target computing node locally, so that the mapping can be performed in parallel. Moreover, when source field view of remote data is locally cached, the field values are transferred only once. This concept of parallel field transfer is illustrated in Fig. 3, where simple interpolation field projection is used. On the computing nodes containing target sub-domains, the field view of source data is set up in a such way that its underlying sub-domain spatially covers the target sub-domain. This mapping is represented by *MappingContext* class. Once the local representation of remote data matching the target sub-domain is available on all target computing nodes, the mapping itself can be done in parallel, without any additional communication.

The setup of mapping context on target application computing nodes requires global representation of remote data. This is needed because the target application should not be aware of source application deployment. Therefore, application agents have to be created by individual applications. They essentially hide the distributed character of underlying mesh or field and manage the proper message dispatching to individual computing nodes containing distributed data. The application agent implements the application interface and its role is to represent the overall global access point for application. Agent is aware of distributed application data structure, which allows to execute data request operations efficiently by splitting them based on application partitioning, routing the requests to processes owning the data, and assembling the results. Despite many advantages, the introduction of application agent has also some drawbacks. If all requests are passed through agent, it may become a bottleneck. However, due to the distributed nature, multiple data requests can be processed in parallel, creating thread for each request, for example. Also, as discussed in previous example of distributed mapping operation, the agent is needed only for setting up the mapping contexts which determine mapping of distributed source data. After the mapping contexts are set up, the data transfers from source to target computing nodes can be done in parallel, without the need of communication through agent - the mapping context contains all data necessary to communicate directly with source computing nodes, as

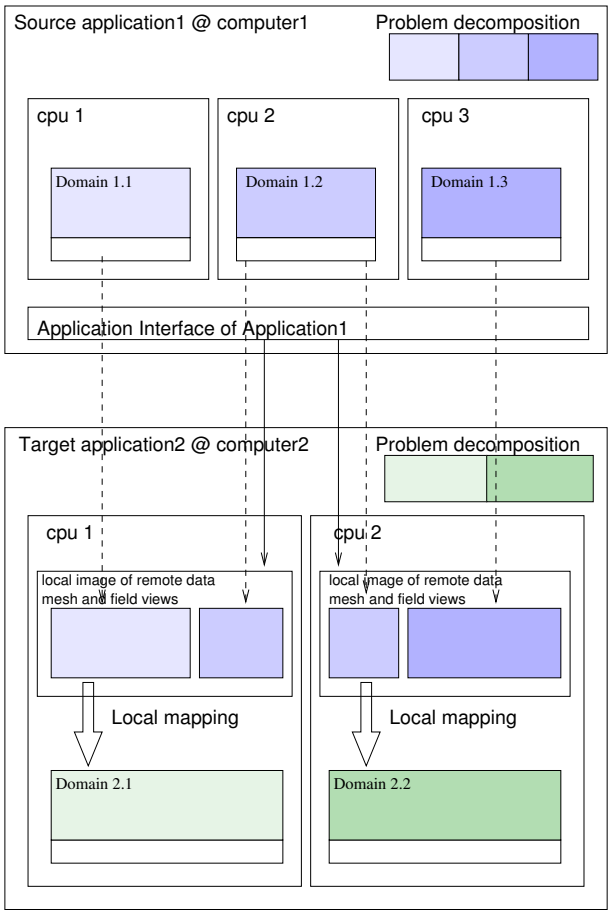


Figure 3: Concept of distributed mapping.

the data distribution is already known at this stage.

The abstract communication layer is built on Pyro [23] library, providing transparent distributed object system. It takes care of the network communication between the objects once 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. The proxy is a special kind of object that acts as if it were the actual -remote- object. Proxies forward method calls to the remote objects, and pass results back to the calling code. Pyro also provides Naming Service which keeps record of the location of objects. The utilization of Pyro allows to fully concentrate on application design, the distributed processing and data exchange is conveniently and transparently handled by Pyro. This is particularly convenient in initial phases of project, where the focus is put on design and prototype implementation of the framework.

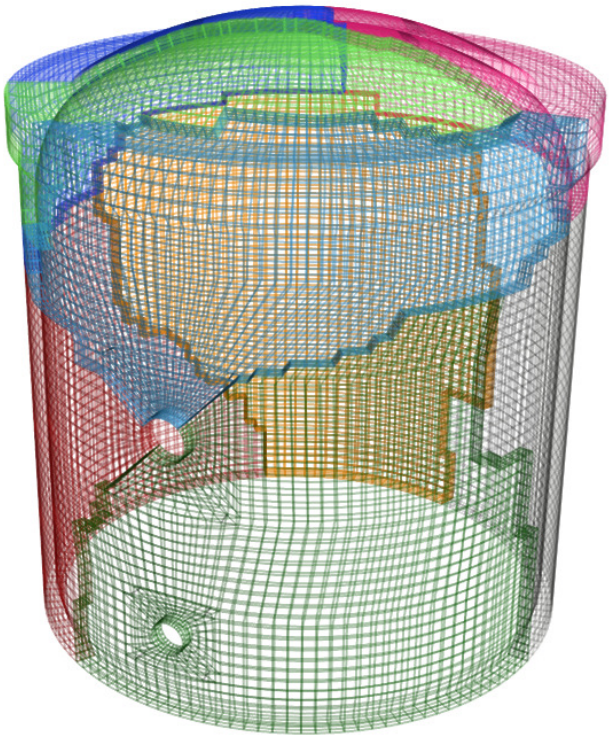


Figure 4: Specimen geometry, computational mesh, and its decomposition into 8 partitions.

5 Example

To demonstrate the use of the library and its performance, an example of parallel field mapping between two distributed finite element applications is presented. The scalar temperature field is transferred between two distributed FE applications, both using unstructured 3D finite element meshes with different decompositions. The performance of distributed field transfer algorithm is compared to performance of pure serial mapping. The source and target FE meshes consist in total of 52825 nodes and 41848 linear brick elements. Originally the target and source meshes were identical. In order to enforce element interpolation during the field mapping, the target mesh was perturbed. Different decompositions of source and target domains have been considered and have been obtained by Metis [24]. The geometry of the specimen, associated FE mesh, and its decomposition into 8 partitions are illustrated in Fig. 4.

The total wall clock time consumed as well as separate wall clock times consumed by mirroring the remote data and local mapping were recorded by each sub-process of target application. The mirroring time includes the set up of remote field views

```
#create a time step
tstep=timestep.TimeStep(1.,1.)
# querry source sub-problem APIs
for rank in range(4):
    api.append(Pyro4.Proxy("PYRONAME:Mupif.agent.example10."+str(rank)))
    remoteField.append(Pyro4.core.Proxy(api[rank].giveFieldUri(1,tstep)))
# get my rank
myrank = int(argv[len(argv)-1])
#read target sub-mesh from file
mymesh = util.readOfemMesh('jete4.np08.oofem.in.'+str(myrank))
# create empty target field
myfield = field.Field(mymesh, field.FieldID.FID_Temperature,
                      field.FieldType.vertex_based,
                      field.FieldValueType.scalar)

# determine mesh bbox
mybbox=bbox.BBox(list(mymesh.giveVertex(0).coords),
                 list(mymesh.giveVertex(0).coords))
for v in mymesh.vertices():
    mybbox.merge(v.coords)
print "Mymesh ", mybbox

#obtain local field view (lfv) of remote data
init = True
start = time.clock()
for rank in range(4):
    if init:
        lfv = api[rank].giveBBoxFieldView(mybbox, 1, tstep)
        lfv.update() #get remote field data
        init=False
    else:
        #merge with remaining
        _lfv = api[rank].giveBBoxFieldView(mybbox, 1, tstep)
        _lfv.update() #update from remote master values
        lfv.merge(_lfv)
print "Rank:", myrank, "Remote data miroring took ",
      (time.clock() - start), "[seconds]"

# map locally mirrored remote field to target mesh
start = time.clock()
transferoperator.CellInterpolationTO().map(lfv, myfield)
elapsed = (time.clock() - start)
print "Rank:", myrank, "Local Mapping took ", elapsed, "[seconds]"
```

Table 1: Listing of target process Mupif script illustrating distributed field mapping.

NSS	NTS	TS1[s]	TS2[s]	TS3[s]	TS4[s]	TT[s]	note
1	1	-/327				340	serial
1	1	5.7/328				348	distributed
2	2	8.6/163.4	6.7/189.4			206	distributed
2	4	7.8/105.6	5.9/87.8	5.2/97.9	4.6/106.8	126	distributed
4	2	9.4/164.4	7.1/189.8			209	distributed
4	4	6.8/89.3	7.0/100.9	4.9/100.2	4.8/107.4	123	distributed

Table 2: Wall clock times consumed by field mapping for different configurations (NSS is number of source sub-domains, NTS is number of target sub-domains, TS contains times for particular target sub-domain (mirroring/local mapping), and TT is the overall time spent by field transfer).

for individual source sub-domains, their mutual merging, and communication due to remote data transfer. The time for local mapping measures the field transfer between two local fields. The measurements were done on multicore workstation with 8 CPU cores running Linux operating system and the obtained results are summarized in Table 2.

From the presented results, several conclusions can be made. Firstly, by comparing the results in the first two lines, which compare times for serial mapping of the whole mesh (when source and target fields are managed by the single process) to the whole mesh mapping, when target and source fields are managed by different processes. The difference allows to assess the cost for inter-process communication, the cost of setting up remote data field views and the Pyro library overhead. In total, this overhead is approx. 6 seconds, which is small, compared to overall computational time. When source and target applications will be spatially distributed, the communication cost will be higher, as relatively slower network communication has to be used instead of inter-process communication on shared memory computer. Another interesting point is scalability which could be evaluated by comparing results with the same number of source sub-domains and different (increasing) number of target sub-domains (e.g., lines 2 and 3, or 4 and 5 in Table 2). It can be seen that the algorithm scales well, for example, the speedup in the case with four source sub-domains and two and four target ones is 1.7, while the ideal value is equal to 2. The difference is attributed to increased amount of communication needed and the additional overall overhead of distributed algorithm. On the other hand, the results can be improved by using smart request scheduling strategy. At present, all processes running target application start querying source application processes in the same sequential ordering. In this case, single source process has to respond to multiple requests in the same time. A better strategy would be to perturb the querying sequence on target processes in order to obtain better balanced query processing. Such an approach definitely could improve the overall scalability. The observed speedup is therefore mainly due to the local mapping on target sub-domains from mirrored source data which is done in parallel on each target process without further communication.

The presented timings also reveal relative inefficiency of plain Python for compu-

tationally intensive tasks. In this case the whole algorithm, including spatial octree localization, element interpolation, etc. has been implemented in Python. The time consumed by mapping is quite high, as it exceeds the time needed to solve the individual problems by compiled C++ solver. Fortunately, this can be improved easily. Once the prototype implementation in Python is developed, the individual subroutines and classes can be re-implemented in C or C++ and a Python interface can be generated, using the wrapping tools. In this way all advantages of scripting Python interface are retained, while the numerical efficiency is significantly improved to the level of compiled languages.

6 Conclusions

This paper presents the design and structure of multi-physics integration framework, with an object-oriented architecture. This framework facilitates the implementation of multi-physics simulations, built from independently developed components. The designed class hierarchy allows to represent solution fields, dicretizations and computational cells using an abstract interface that naturally supports different discretization techniques and allows the implementation of numerically- and physically-aware algorithms independent of underlying discretization. The interaction between framework and individual applications is relying on abstract application interface, that needs to be implemented for individual applications. Such an approach allows to perform transparent data exchange and steering of individual applications. Paper also addresses and illustrates important issues related to parallel and distributed computing platforms. The library is integrated into interactive scripting Python environment and is available as open source project, distributed under a GNU Public license.

Acknowledgments

This work has been supported by the Grant Agency of the Czech Republic, project no. P105/10/1402.

References

[1] P. van der Velde, G.D. Mallinson, The Design of a Component-Oriented Framework for Numerical Simulation Software, Advances in Engineering Software, Volume 38, Issue 3, 182-192, 2007.

[2] M. F. Sanner, Python: A Programming Language for Software Integration and Development. J. Mol. Graphics Mod., Vol 17, 57-61, 1999.

[3] E. de Sturler, J. Hoeflinger, L. Kale, and M. Bhandarkar, A New Approach to Software Integration Frameworks for Multi-Physics Simulation codes, in The

Architecture of Scientific Software, R.F. Boisvert, P.T.P. Tang, eds., Kluwer Academic Publishers, Boston, 2001, ISBN 0-7923-7339-1.

[4] P. Montarnal, A. Dimier, E. Deville, E. Adam, J. Gaombalet, A. Bengaouer, L. Loth, C. Chavant, Coupling methodology within the software platform Alliances, In E. Oate M. Papadrakakis and B. Schrefler, editors, Int. Conf. on Computational Methods for Coupled Problems in Science and Engineering COUPLED PROBLEMS 2005, Barcelona, 2005.

[5] COMSOL Multiphysics, http://www.comsol.com, 2011.

[6] ADINA, http://www.adina.com, 2011.

[7] MSC Nastran, http://www.mscsoftware.com, 2011.

[8] ANSYS, http://www.ansys.com, 2011.

[9] Elmer, http://www.csc.fi/english/pages/elmer, 2011.

[10] OpenFOAM, http://www.openfoam.com, 2011.

[11] S.Karmesin, S. Haney, B. Humphrey, J. Cummings, T. Williams, J. Crotinger, S. Smith and E. Gavrilov, POOMA - Parallel Object-Oriented Methods and Applications, http://acts.nersc.gov/pooma.

[12] D. Brown, K. Chand, W. Henshaw, B. Miller, J. Painter, R. Pember, B. Philip, D. Quinlan, T. Rutaganira, OVERTURE, http://acts.nersc.gov/overture.

[13] Gunney, B. T. N., A. M. Wissink, D. A. Hysom, Parallel Clustering Algorithms for Structured AMR, Journal of Parallel and Distributed Computing, 66(11), 1419-1430, 2006.

[14] ALEGRA, http://www.cs.sandia.gov/ALEGRA/Code_versions.html.

[15] AVS- Advanced Visual Systems, http://www.avs.com.

[16] OASIS - A Code Coupler for Climate Modelling, https://verc.enes.org/models/software-tools/oasis, 2011.

[17] J. R. Stewart, H. C. Edwards, A Framework Approach for Developing Parallel Adaptive Multiphysics Applications, Finite Elements in Analysis and Design, Volume 40, Issue 12, 1599-1617, 2004.

[18] PALM Software, http://pantar.cerfacs.fr/globc/PALM_WEB/index.html, 2011.

[19] B. Patzk, MuPIF - Multi-Physics Integration Framework, http://mech.fsv.cvut.cz/mupif, 2011.

[20] G. van Rossum, F. L. Drake, Jr. (Editor), An Introduction to Python, Network Theory Ltd, 164 pages, 2006.

[21] M. Lutz, Learning Python, O'Reilly & Associates, 701 pages, 2007.

[22] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed. ed.), Addison-Wesley, ISBN 0-321-19368-7, 2003.

[23] Pyro project home page, http://pyro.sourceforge.net/.

[24] G. Karypis, V. Kumar, A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs, SIAM Journal on Scientific Computing, Vol. 20, No. 1, pp. 359392, 1999.