AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF SCIENCE THESIS

MARCIN NOWAK

MULTISCALE APPLICATIONS COMPOSITION AND EXECUTION TOOLS BASED ON SIMULATION MODELS DESCRIPTION LANGUAGES AND COUPLING LIBRARIES

SUPERVISOR:
Katarzyna Rycerz Ph.D

Krakow 2011/2012

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

. . . .

PODPIS

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA INFORMATYKI

PRACA MAGISTERSKA

MARCIN NOWAK

NARZĘDZIA DO KONSTRUOWANIA I WYKONYWANIA WIELOSKALOWYCH APLIKACJI W OPARCIU O JĘZYKI OPISU MODELI SYMULACYJNYCH I BIBLIOTEKI ŁĄCZĄCE

OPIEKUN PRACY:
dr inż. Katarzyna Rycerz

Kraków 2011/2012

Acknowledgements

I wish to thank my supervisor—Dr. Katarzyna Rycerz—for her invaluable help and patient guidance she has provided during my work on this Thesis. Without her support and contribution this Thesis would not have been possible.

I would like to express my gratitude to Dr. Marian Bubak for his accurate advice.

I would also like to thank my colleague Paweł Pierzchała with whom I worked on the initial version of the MUST User Support Tool.

Next, the thanks for the help with integrating MUST User Support Tool with the GridSpace virtual laboratory go to Eryk Ciepiela and Daniel Harężlak from ACC CYFRONET.

Last, but not least, I would like to help Małgorzata Palej for her suggestions and proofreading this work.

This work is related with the Mapper project which receives funding from the EC's Seventh Framework Programme (FP7/2007-2013) under grant agreement no RI-261507.

Abstract

Multiscale applications are crucial in better understanding of the processes from various fields of science. Modeled processes frequently require substantial computing capabilities. This is the reason why the grid and the cloud computing environments are chosen as suitable platforms for executing multiscale applications.

However, execution of multiscale applications in remote distributed environments is relatively complicated. Therefore, we developed MUST User Support Tool which aids execution of multiscale applications in the grid and the cloud environments as a part of this Thesis. The proposed tool provides a simple interface which may be used to facilitate the work with multiscale applications in remote distributed environments.

Furthermore, we thoroughly compare the grid and the cloud computing environments. Various aspects ranging from the definitions of the grid and the cloud, through computing and programming models to performance results are taken into consideration.

This Thesis is organized as follows: Chapter 1 presents the problems discussed, the scope and main goals of this Thesis. Chapter 2 introduces multiscale applications, discusses their main requirements and describes examples from various fields of science. Chapter 3 describes model description languages commonly used to describe single and multiscale models frequently used in multiscale applications. Various libraries aiding development of multiscale applications are described in Chapter 4. Chapter 5 is a detailed comparison of the grid and the cloud computing environments. Chapter 6 presents various middleware tools used for accessing the remote environments. MUST User Support Tool is introduced and exhaustively described in Chapter 7. Chapter 8 presents MUST implementation details and the possibilities of expansion. Performance results of executing a scientific application using MUST User Support Tool with various setups are presented in Chapter 9. Finally, Chapter 10 summarizes this Thesis and presents its results and conclusions.

Appendix A is a glossary of terms and abbreviations used in this Thesis. Appendix B contains various MUST User Support Tool usage examples and installation details. Appendix C presents The Comparison of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations paper related to this Thesis.

Contents

Contents **9**

1 Introduction **13**
1.1 Problem outline . . . 13
1.2 Goals and scope . . . 14
1.3 Organization of this Thesis . . . 15
1.4 Contribution of other authors . . . 17

2 Multiscale applications **19**
2.1 Introduction . . . 19
2.2 Requirements . . . 19
2.3 Examples . . . 21
2.4 Summary . . . 25

3 Model description languages **27**
3.1 SBML and CellML . . . 27
3.2 MML and CxA . . . 29
3.3 Comparison . . . 32
3.4 Summary . . . 33

4 Coupling libraries **35**
4.1 AMUSE . . . 35
4.2 MCT . . . 36
4.3 MUSCLE . . . 37
4.4 Comparison . . . 39
4.5 Summary . . . 40

5 Infrastructures **41**
5.1 Grid . . . 41
5.2 Cloud . . . 43
5.3 Comparison . . . 44
5.4 Summary . . . 48

6 Accessing infrastructures **49**
6.1 Local resources—Portable Batch System . . . 49
6.2 Grid resources—GridSpace . . . 49
6.3 Cloud resources—Amazon Web Services . . . 51
6.3.1 Elastic Compute Cloud . . . 51
6.3.2 Simple Storage Service . . . 51

6.3.3 Elastic Block Storage 52
6.3.4 Simple Queue Service 52
6.4 Summary 52

7 MUST User Support Tool **55**
7.1 Concept of MUST 55
7.2 Requirements 56
7.3 Use cases 57
7.4 Architecture 58
7.4.1 MUST Architecture—Grid 59
7.4.2 MUST Architecture—Cloud 63
7.5 Summary 67

8 Implementation details **69**
8.1 MUST implementation 69
8.2 Expansion possibilities 72
8.3 Tools used 72
8.4 Summary 73

9 Case study **75**
9.1 ISR2D performance results 75
9.2 Performance results interpretation 78
9.3 Summary 79

10 Summary **81**
10.1 MUST User Support Tool 81
10.2 Grid and cloud comparison 82

List of Figures **85**

List of Tables **85**

References **87**

Appendices **91**

A Glossary **91**

B Examples **93**
B.1 Example GridSpace experiment 93
B.2 Example usage 98
B.3 Installation guide 99

B.3.1 Prerequisites 99
B.3.2 Access machine and nodes configuration 100
B.3.3 EC2 instance configuration 101
B.4 Summary 101

C Publication—Comparison of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations 103

1 Introduction

This Chapter introduces the user support tool that allows automatic execution of multiscale applications in distributed environments (MUST User Support Tool) proposed in this Thesis and its main requirements. The MUST tool facilitates the work with multiscale applications in the grid and the cloud environments. This Chapter also specifies the goals and the scope of this Thesis (Sections 1.1—1.2). Sections 1.3 and 1.4 present the organization of this Thesis and the contribution of other authors respectively.

1.1 Problem outline

The need for high-performance computing and storing vast amounts of data is constantly growing. Not everyone, however, is able to build and maintain their own supercomputers or mass storage devices. Therefore, this need is nowadays frequently fulfilled by remote systems. Details concerning the infrastructure or the physical localization of such systems are irrelevant to the end-users. Both grid and cloud computing provide applicable resources. Grid computing focuses on large-scale computations, commonly in the form of batch jobs. Grid systems process few large requests (e.g. jobs allocating several hundreds of nodes). On the other hand, cloud systems primarily allow users to deploy their services. Consequently, cloud systems process sizeable numbers of small requests.

Multiscale applications depend on such substantial computing capabilities. Simulation of three-dimensional models is a key task in numerous fields of science. Models at different spatial and temporal scales are widely used. Multiscale applications depend on numerous model description languages, coupling libraries and work-flow management systems. Integration and execution of different models is a crucial task. Unfortunately, it often has to be performed manually and, in consequence, is very time-consuming. In this paper, we propose a user support tool which assists in automatic execution and distribution of multiscale applications on various infrastructures.

Multiscale applications are generally used by non-programmers. Therefore, their execution on distributed architectures should be relatively uncomplicated.

The user support tool presented in this Thesis was built as a part of the GridSpace virtual laboratory which allows users to create *experiments*. Created experiments can be executed locally (on a computing cluster) or remotely (on the grid). The GridSpace virtual laboratory provides a user-friendly web interface. MUST can be used to facilitate work with multiscale applications in distributed environments. It allows users to automatically

distribute their GridSpace experiments and execute them on the grid and the cloud infrastructures.

Utilization of the resources provided by grid and cloud computing can be a great opportunity to develop multiscale applications.

1.2 Goals and scope

There were two main goals of this Thesis:

- Design and development of a user support tool which allows automatic distributed execution of multiscale applications in various infrastructures (hereinafter referred to as MUST—MUST User Support Tool). The main requirements:

  - **Support for distributed execution of multiscale applications.** The purpose of the tool is to allow distributed execution of multiscale applications i.e. applications which use coupling libraries to link multiple separate single-scale models. In this Thesis we focus on support for multiscale applications built using MUSCLE coupling library. The MUSCLE coupling library was chosen based on the conclusions of Chapter 4. The distributed execution of MUSCLE-based applications must not require substantial changes to the existing applications. The distribution should be based on the existing features of the supported library.
  - **Support for both grid and cloud distributed infrastructures.** The proposed tool should allow execution of multiscale applications on both grid and cloud infrastructures. Live transmission of the standard input and error streams from the remote environments should be possible as well as optional upload of the input sandbox files—allowing automatic deployment of simple multiscale applications (assuming that other prerequisites are available on the remote machines).
  - **Access from the GridSpace virtual laboratory level.** The proposed tool should be accessible from the GridSpace virtual laboratory web interface. Automated execution and distribution of the GridSpace *experiments* based on multiscale application should be possible using the proposed tool.

  A detailed description of the requirements is presented in Section 7.2. Chapters 7 and 8 describe the MUST tool in detail.

- Comparison of grid and cloud computing, including:

- **Theoretical comparison.** The aim of the throughout theoretical comparison of grid and cloud computing is to highlight differences and similarities between them. Aspects such as programming model, computing model, usability, standarization, etc. should be taken into consideration.
- **Performance results.** Performance results of the grid and the cloud infrastructures based on the execution of a scientific multiscale application using various setups should be compared. The comparison should specify, describe and separately confront all the steps necessary to execute an application in the distributed environment including the preparation step, the execution itself and downloading results from the remote machines.
- **Ease of access, difficulty of installation and amount of changes required in legacy applications.** Middleware tools used for accessing the grid and the cloud infrastructures should be described and compared as well as the APIs exposed and standards/proprietary solutions used. Installation and configuration steps required should also be discussed.

A comparison of the grid and the cloud infrastructures is presented in Chapter 5 while the performance results are discussed in Chapter 9.

To achieve these goals, we developed the MUST tool and then performed appropriate tests.

1.3 Organization of this Thesis

This Chapter presented the problem outline and briefly introduced MUST—a user support tool facilitating execution of multiscale applications in various distributed environments. The subsequent chapters present theoretical concepts related to multiscale applications and distributed environments (the Grid and the Cloud) as well as tools used for execution and description of multiscale applications on various levels (ranging from low level job scheduling systems through middleware coupling libraries to high-level description tools and execution environments).

The concept of multiscale applications is presented in Chapter 2 which discusses their requirements thoroughly and presents problems related to accurate model description and comprehensible inter-model communication. Eventually, a few examples of multiscale applications from various fields of science are described (with particular emphasis on different spatial and temporal scales at which each application operates).

The subsequent Chapter (3) focuses on an efficient and understandable description of multiscale applications. Therefore, we present various model description languages. Firstly, some recognized and widely used languages concentrating on single model description such as Cell Markup Language and Systems Biology Markup Language are presented. Then, we describe the concept of Multiscale Modeling Language (MML) enabling a description of multiple models and inter-model interactions. CxA—a format preceding MML—is also shortly discussed as a format closely related to MUltiScale Coupling Library and Environment (MUSCLE). Finally, a comparison of all the languages is presented.

Chapter 4 presents Astrophysical Multipurpose Software Environment (AMUSE), Model Coupling Toolkit (MCT) and MUltiScale Coupling Library and Environment (MUSCLE)—three different coupling libraries facilitating building and execution of multiscale applications. First of all, each library is separately described. Then, all the libraries are compared with emphasis on aiding execution of multiscale applications in distributed environments.

Chapter 5 leaves the subject of multiscale applications and focuses on generic distributed environments. The Grid and the Cloud computing environments are exhaustively described and compared. A theoretical comparison starting with quotations of various Grid and Cloud definitions is followed by a presentation of more pragmatic aspects, such as programming model, computing model and usability.

The middleware tools enabling access to the Grid and the Cloud environments are described in Chapter 6. It includes job scheduling systems used for local computing clusters as well as an API allowing access to various Cloud-based resources (on the example of Amazon Web Services API). In this Chapter, we also introduce and describe the GridSpace virtual laboratory. Utilization of the previously listed middleware tools in MUST is also mentioned in the summary.

The MUST tool is described in detail in Chapter 7, a general concept of the tool is specified, followed by a presentation of detailed requirements and a description of some use cases. Subsequently, the tool's layered architecture is described and depicted, divided into two sections—one for the Grid and one for the Cloud architecture. Detailed diagrams showing cooperation of consecutive layers are also included. We list as well the steps performed before, during and after execution of the particular multiscale application using the MUST tool, which are depicted on the suitable sequence diagrams.

Chapter 8 describes MUST's implementation details, with a class diagram included, showing the similarities and implementation differences on both Grid and Cloud computing infrastructures. The chapter ends with a discussion about possibilities of expansion, divided into supporting both new

execution environments and other coupling libraries.

Various performance results are included in Chapter 9. The tests performed include a division into various stages of execution (including submission, execution and results gathering stages). A scientific application (In-stent restenosis 2D, described in detail in Section 2.3) was used to achieve conclusive results. All tests were performed on both Grid and Cloud infrastructures. The chapter is summarized by a discussion of the measured performance results.

Appendix A is the short glossary of uncommon terms and abbreviations used in this Thesis.

Appendix B shows various examples of MUST usage. A sample GridSpace experiment which may be launched using MUST is included as well as its detailed description. Some MUST's command line usage are also demonstrated. The Chapter ends with a short description of the MUST installation.

Appendix C includes The Comparison of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations paper (by K. Rycerz and co-authors M. Nowak, P. Pierzchala, M. Bubak, E. Ciepiela and D. Harezlak).

1.4 Contribution of other authors

MUST was developed as a part of the GridSpace virtual laboratory (GS)[44]. GS enables development, sharing, execution and reusability of the so-called experiments—sets of high-level scripts created by scientists (more details in Section 6.2).

The initial version allowed running MUSCLE-based multiscale applications on the grid infrastructure. It was developed by Paweł Pierzchała and the author of this Thesis. Paweł Pierzchała focused on integration with GridSpace while the author of this Thesis concentrated on the usage of the grid resources. MUSCLE usage, live results streaming, etc. was a result of a joint work of Paweł Pierzchała and the author of this Thesis.

After finishing the initial version, Paweł Pierzchała continued building a graphical tool which allows mapping groups of single scale simulations to computing nodes. That JavaScript based tool was developed to cooperate with the GridSpace virtual laboratory as an external Web application[2].

Concurrently, the author of this Thesis developed a tool which allowed launching MUSCLE multiscale applications on the Amazon Web Services-based cloud infrastructure.

2 Multiscale applications

In this Chapter, we introduce the concept of multiscale applications (Section 2.1). The common requirements of multiscale applications are presented in Section 2.2 while Section 2.3 presents some exemplary multiscale problems and applications from various fields of science.

2.1 Introduction

Multiscale modeling is nowadays used in multiple fields of science. Examples include Physiology, Computational Biology, Engineering and Nano-material Science [1].

The multiscale model may be perceived either as a single model spanning many spatial and temporal scales or as a set of coupled single scale models (as presented on Figure 1). Models at various scales require simulation at different levels of detail (ranging from intermolecular to macroscale interactions).

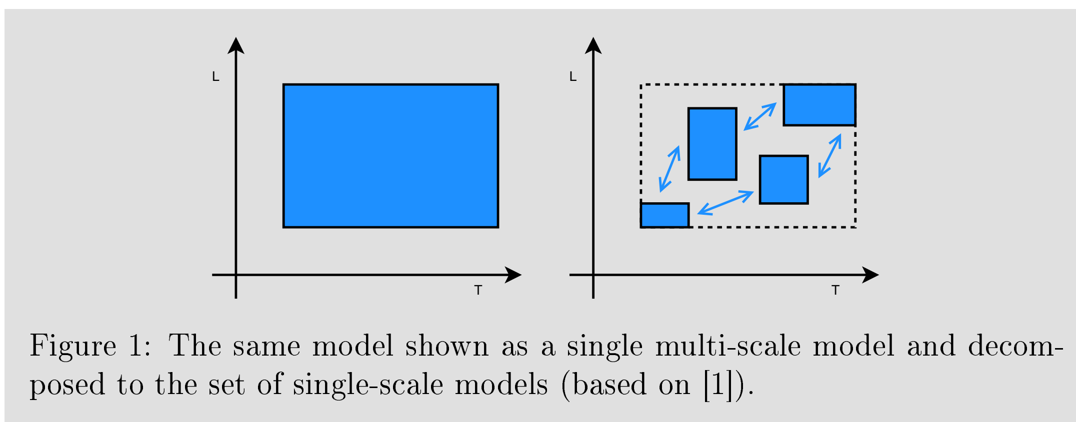


Figure 1: The same model shown as a single multi-scale model and decomposed to the set of single-scale models (based on [1]).

Multiscale applications cannot be easily decomposed by nature—they consist of many components which are often tightly coupled. Each component may be simulated separately using different techniques (by molecular dynamics, cellular automata or Monte-Carlo methods, etc.).

2.2 Requirements

Multiscale applications are widely used as tools helping to understand complex processes. Multiple single scale simulations forming one multiscale application are often computationally heavy. Receiving results in real time is crucial in some appliances (e.g. when aiding health-related decisions).

Computing, communication and storage requirements

Multiscale applications may require very **large computing capabilities** (sometimes exceeding PetaFlops [1]). Communication between computational processes (both at the intra- and inter-simulation level) may easily become the application's bottleneck.

This is the main reason why the concurrent execution and an effective communication between single scale models is a crucial task. There are several approaches to communication between single scale models (as described in [3], e.g. the timing of a particular simulation may be regulated by another one, there may be a rollback needed, etc.).

Significant amounts of data are often used and/or produced by multiscale applications. This is why many multiscale applications require the **utilization of large and easily accessible storage devices**.

Simulation dependencies and model description

Dependencies between single scale models are highly diversified. Models may have to be simulated concurrently or sequentially. Simulation of the same period in various spatial scales may require varied computing capabilities resulting in different simulation times (i.e. a simulation of 1 second of blood flow may take significantly longer when simulated on cell level than when simulated on vessel level—nevertheless simulation results on each level may affect another one).

The **description of models** themselves **and their cooperation** is an important requirement. Models should be described in a clear, structured way so that they could be easily understandable and reusable. XML-based model description languages (e.g. SBML, CellML) usually meet these requirements. Although there are many formats available a single standard has not been chosen yet (as discussed further in Chapter 3).

While there are a few model description languages describing single scale models, there are significantly less languages which would describe connections and dependencies between models (i.e. languages which would model a whole multiscale application). For instance, the MML language (described also in Chapter 3) attempts to achieve this goal.

There are multiple tools facilitating execution of multiscale applications. For example, the AMUSE library [18] is used for simulation of stellar systems. Other examples include MCT [19, 20] and MUSCLE [21], which are more general purpose tools. MCT operates on a slightly lower level than MUSCLE. MUSCLE uses the CxA model description format (described in Section 3.2. The tools mentioned above, allowing simulation of existing models, are described more widely in Chapter 4.

2.3 Examples

Mutliscale problems from various fields of Science are briefly described below. Example domains include Physiology, Flow Control and Fusion processes. All the presented applications span many spatial and temporal scales and require supercomputing capabilities to be modeled.

Physiology

The first example of a multiscale application is a multiscale model of in-stent restenosis created in the COAST project [4].

Coronary artery disease remains the most common cause of death in Europe. It refers to stenosis of coronary arteries caused by accumulation of atheromatous plaque[1, 5]. Possible treatment involves the use of a metal frame (stent) to maintain an open vessel lumen. Unfortunately, there is a common complication—in-stent restenosis (ISR)—which is the return of the vessel lumen to a size similar to that before intervention.

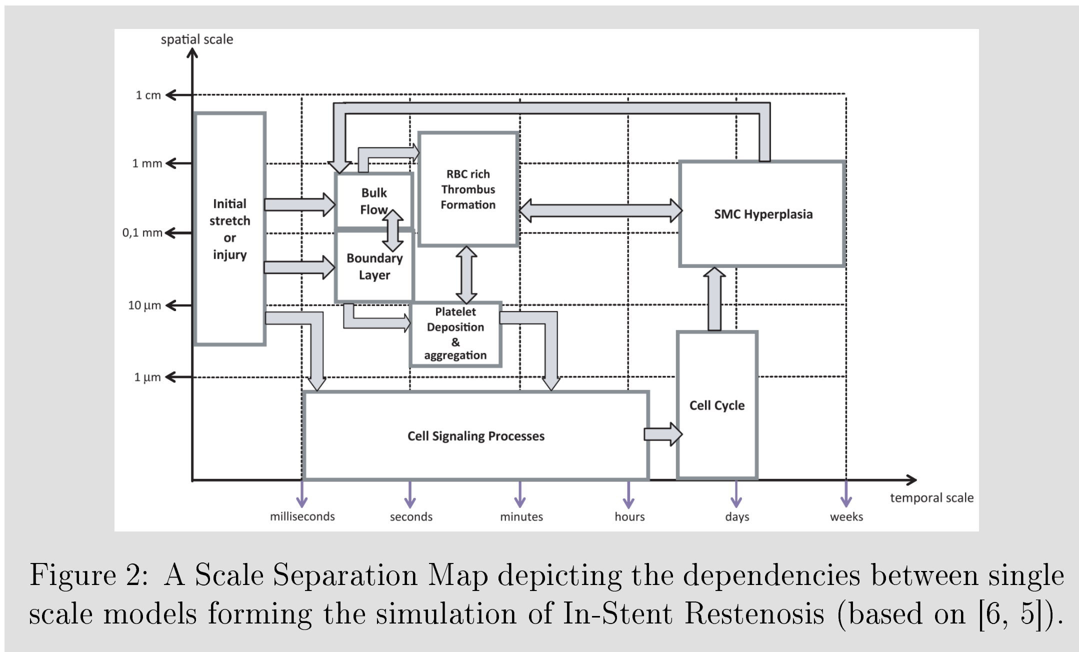


Figure 2: A Scale Separation Map depicting the dependencies between single scale models forming the simulation of In-Stent Restenosis (based on [6, 5]).

Modeling ISR may help to understand and prevent this ailment. Processes participating in ISR act on scales from microns up to centimeters. Temporal scales involved are also widely separated (from seconds to months). The factors such as administered drugs or changing blood pressure affect the patient on multiple scales, ranging from individual cell characteristics up to

global hemodynamics. Figure 2 shows a simplified Scale Separation Map (a diagram type described in Section 3.2) proposed by Evans et al. in [6], depicting dependencies between single scale models forming a simulation of ISR.

A real time simulation of 3D ISR models is crucial in many areas (e.g. designing patient-specific chemotherapy and radiotherapy applications). This simulation however requires up to thousands of processors.

Flow Control

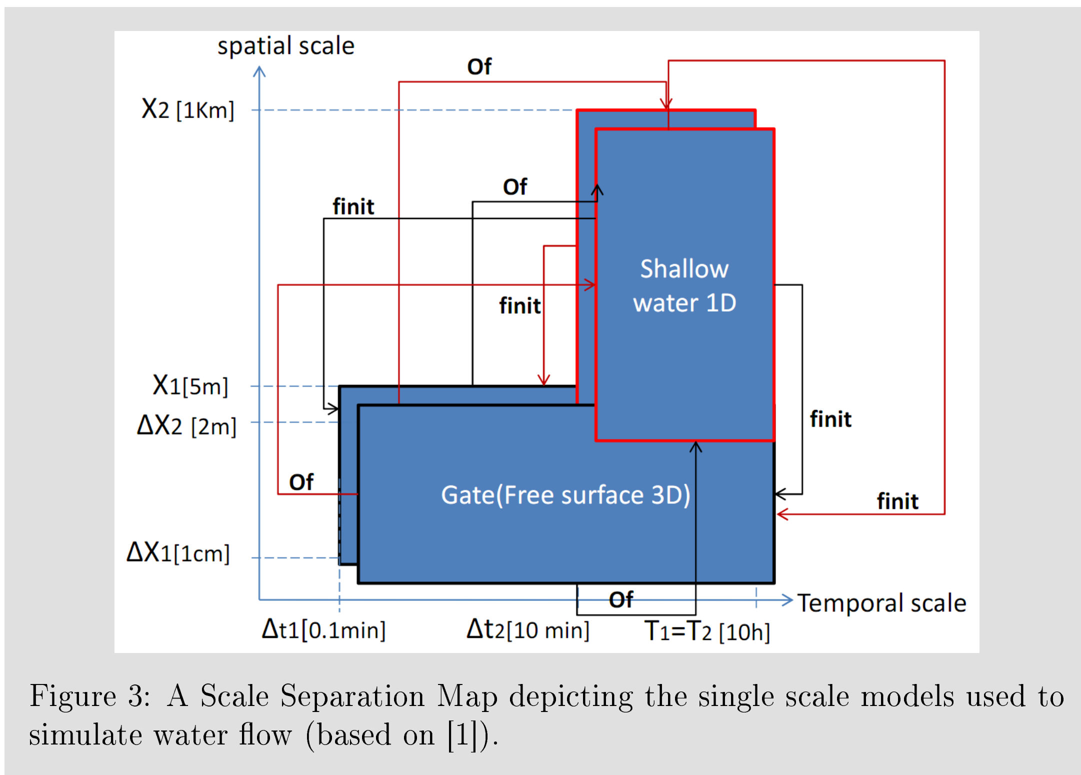


Figure 3: A Scale Separation Map depicting the single scale models used to simulate water flow (based on [1]).

Flow control model developed by the Université de Geneve in collaboration with the Ecole Supérieure d'Ingénieurs en Systemes Industriels Avancés Rhône-Alpes is another example of a multiscale problem.

Simulating flow of canals and rivers is an important task. It may help to maintain the adequate water levels needed in agriculture, water transport, etc., and, which is even more important, it may help to avoid flooding.

Three models at different scales are involved in simulating the full system of irrigating canals[1, 9]:

1. One-dimensional shallow water equation for simulating the water flow in long canal sections. This model considers the sediment transport which may have a significant influence on the water flow (i.e. irrigation efficiency reduction).
2. Two-dimensional shallow water equation for simulating branching and large water pools.
3. Three-dimensional, free-surface model is used in simulations of a detailed flow of water in gates and/or in descriptions of the sediment transport. This simulation uses Lattice Boltzmann Model for Free Surface Flow[10].

The third model requires supercomputing capabilities. Figure 3 shows a Scale Separation Map which includes 1D shallow water model, 3D free-surface model and scales at which each model is used.

Fusion

Another example of a multiscale problem includes modeling fusion processes proposed as a part of the ITER[11] project. The project focuses on a description of core plasma in a tokamak. A tokamak[1] is a toroidal device using a magnetic field to confine plasma. The range of scales modeled in the application is depicted in Figure 4.

Hydrogen atoms collide in the core of the Sun and fuse into heavier Helium atoms. The fusion of two atoms produces great amounts of energy (as calculated in Einstein's formula $E = mc^2$). The ITER project models fusion processes so that they can be used to produce commercially available energy. To achieve that, complex states need to be modeled.

An example of such a state is an equilibrium describing the reference plasma state and a series of equilibria varying in the edge current and edge pressure profiles. A stability analysis needs to be performed on each equilibrium in order to show the stable region and instabilities at its boundaries[1].

Computational Biology

The last example of a multiscale problem presented in this Chapter is the modeling of bile acid and xenobiotic system, performed as a part of the NucSys program[8].

An analysis of interactions between several components of a biological system may help to understand the system as a whole. Efforts have been

[1] Tokamak—Russian *ТОроидальная КАмера с МАгнитными Катушками*—a toroidal chamber with magnetic coils

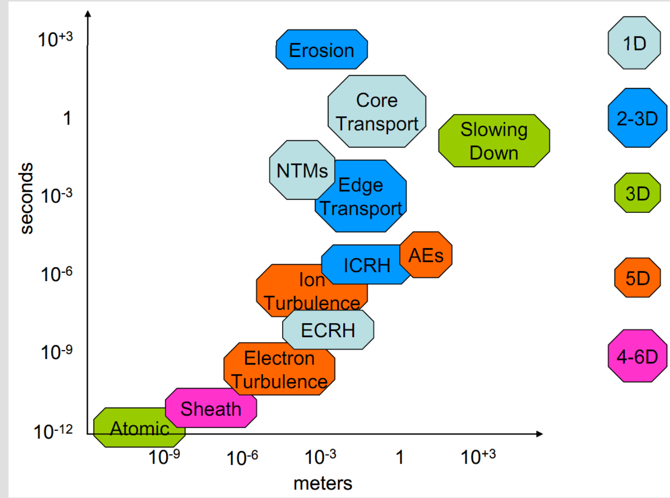


Figure 4: Range of scales modeled in the fusion multiscale application.

made to model processes at different scales, ranging from molecular to organ level and from fractions of a second to months.

An example of such a multiscale process is the previously mentioned modeling of bile acid and xenobiotic system (BAXS). It comprises modeling the processes of metabolism, conjugation and modification and transport phases[1].

The supercomputing capabilities used for modeling this system may result in a better understanding of the model's parameters and the model's reaction to various experimental conditions.

2.4 Summary

In this Chapter, we introduced multiscale applications, their requirements and examples. Studying multiscale applications requirements shows that they require not only supercomputing capabilities to run simulations but also an efficient description language to aid development, cooperation and reusability of models. Popularization of easy to use middleware libraries which enable an automatic code-stubs creation is also vital.

This Chapter presented various examples of multiscale applications. Concurrent (and/or sequential) simulation of different single scale models is used in domains as distant from each other as Flow Control and Fusion processes.

Frequently used and recently proposed model description languages are described in subsequent Chapter. A review of the various single scale models coupling tools is also presented in Chapter 4.

3 Model description languages

This Chapter briefly presents various model description languages, their common usage and collaboration with the MUST tool. In Section 3.1 we focus on the languages used mainly for describing single scale models. Section 3.2 introduces a new language, MML, the main target of which is the modeling of multiscale applications as a whole (including interactions between single scale models). In Section 3.3 we compare the previously introduced types of languages.

3.1 SBML and CellML

Both Systems Biology Markup Language (SBML) [12] and Cell Markup Language (CellML) [13] focus on description of the models themselves (i.e. they do not stress describing connections and interactions between models or simulation details). SBML focuses on modeling physical and chemical phenomena, whereas CellML is primarily used to describe mathematical models of cellular biological function[13]. Physical phenomena are mostly described by differential equations and linear algebra.

SBML

SBML models are hierarchical. An SBML model may contain various child elements: notes, functions, units, compartments, species, parameters, rules, reactions and events. The so-called *compartment* is, in fact, a single model with a certain spatial and temporal scale defined. Unfortunately, compartments can only be contained in another compartment (i.e. no other relation between compartments is possible). *Species* sections may represent chemicals used in the model (ranging from simple ions to complex structures like RNA).

Figure 5 shows an exemplary fragment of an SBML file. Example units, compartments and species are defined.

CellML

CellML models are built from a set of smaller components. A *component* may represent a physical object (i.e. a cell), a physical or chemical reaction or a simple variable.

Besides components, a CellML model may contain unit, group, connection and import sections. The unit section is used for defining complex units (just like a similar section in SBML). The group section describes physical and logical component relations. There are two predefined relations—physical

```
    ...
<Model name="EnzymaticReaction">
  <ListOfUnitDefinitions>
    <UnitDefinition id="per_second">
      <ListOfUnits>
        <Unit kind="second" exponent="-1"/>
      </ListOfUnits>
    </UnitDefinition>
    <UnitDefinition id="litre_per_mole_per_second">
      <ListOfUnits>
        <Unit kind="mole" exponent="-1"/>
        <Unit kind="litre" exponent="1"/>
        <Unit kind="second" exponent="-1"/>
      </ListOfUnits>
    </UnitDefinition>
  </ListOfUnitDefinitions>
  <ListOfCompartments>
    <Compartment id="cytosol" size="1e-14"/>
  </ListOfCompartments>
  <ListOfSpecies>
    <Species compartment="cytosol" id="ES"
     initialAmount="0" name="ES"/>
    ...
```

Figure 5: SBML file fragment example.

containment and encapsulation (representing a logical component hierarchy). The connection section is used to connect variables between components (so that a change in a particular model may affect another model). The import section may be utilized to reuse the previously defined compartments and units.

Figure 6 shows a fragment of a CellML file. The units are defined in a fashion similar to SBML. An example compartment with several variables is also shown.

SBML models may be described on a higher level of detail than CellML models. CellML models are more loosely coupled, therefore they allow slightly easier reusability (by use of *compartments*).

```
...
<Units name="concentration_units">
  <Unit units="mole" prefix="milli"/>
  <Unit units="litre" exponent="-1"/>
</Units>
<Component name="intra_cellular_space">
  <Variable name="Na" units="concentration_units"
   public_interface="out"/>
  <Variable name="Ca" units="concentration_units"
   public_interface="out"/>
  <Variable name="time" units="second"
   public_interface="in"/>
  ...
```

Figure 6: CellML file fragment example.

Both SBML and CellML may describe models from different fields at many scales. Both languages were designed for ease of models reusability. Unfortunately, in CellML the only possible relation between models (or *compartments*) is inclusion. Connections between compartments can be defined. On the other hand, SBML does not allow to define relations or connections between models. Therefore, SBML and CellML are the tools suited for describing a single scale model rather than a set of models at different scales.

There are many tools which support creation (OpenCell, E-Cell, CellML Viewer) and/or simulation (JSim—supporting both CellML and SBML) of CellML and SBML models. Large repositories[2] of both CellML and SBML models exist.

3.2 MML and CxA

Multiscale Modeling Language (MML) [14, 15, 16] is a concept language proposed as a part of the Mapper project. MML can be used to describe models similarly to SBML and CellML. Furthermore, MML can describe the coupling between single scale models (including relations between computational domains and scales), and the types of coupling (coarse graining, scale

[2]Such as `http://physiome.org`, `http://models.cellml.org`, `http://e-cell.org/ecell-models` or `http://www.ebi.ac.uk/biomodels-main/`

splitting, amplification). MML models can be described in the xMML format (XML Multiscale Modeling Language) and depicted using the graphical representation of MML—gMML (Graphical Multiscale Modeling Language).

MML describes accurately connections between models. Various types of sent information are distinguished (e.g. information sent during or after computation, initial conditions, information updating the domains or boundaries, information regarding scales, etc.).

Single scale models in MML are standalone and were designed for ease of reusability. MML introduces *filters*—they can be used as links between models and perform required transformation of the exchanged data (including change of scale, interpolation or decimation).

MML also describes *mappers*—entities controlling the flow of information between models. A *mapper* gathers information from all connected models, then processes and combines it as required and, finally, sends it to receiver models.

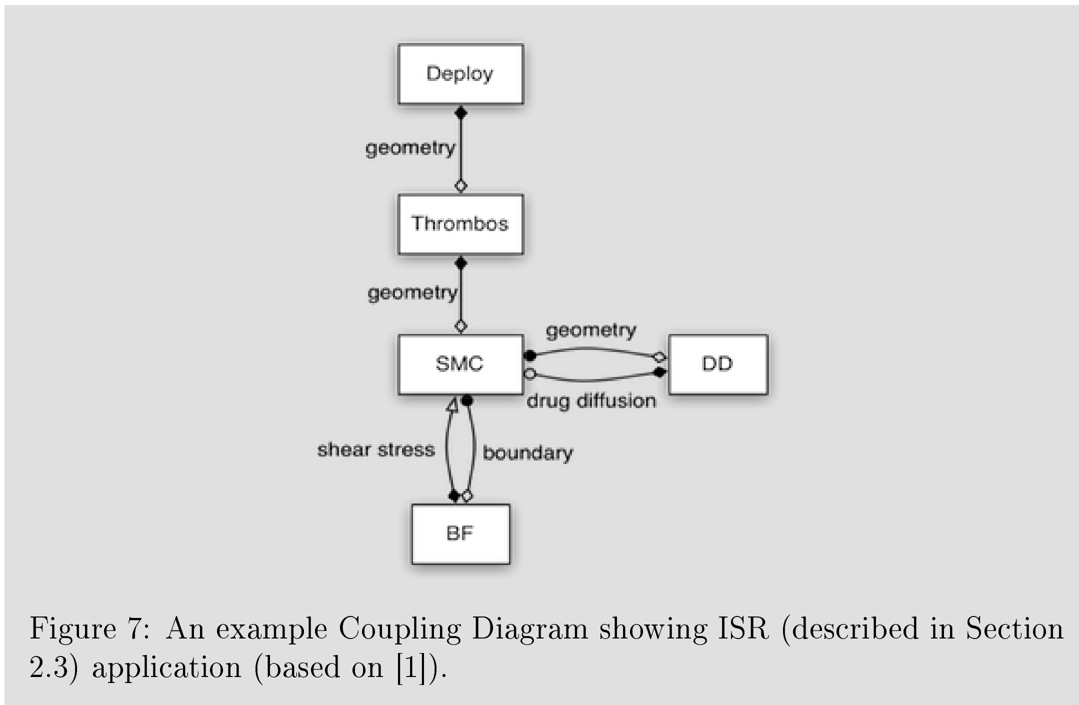


Figure 7: An example Coupling Diagram showing ISR (described in Section 2.3) application (based on [1]).

MML models may be easily depicted using gMML. There are two kinds of diagrams in gMML: Scale Specification Maps (SSM) and Coupling Diagrams. SSM illustrates the coupling of different scale models. It is suited for depicting models with well separated scales. Coupling Diagram which shows interactions between single scale models is best suited for depicting more

complex multiscale applications. It may contain all the elements of MML (including filters, mappers and flow of information).

Figure 7 shows an example of Coupling Diagram. Single scale submodels are shown as rectangular boxes. Couplings between submodels are shown as differently styled connectors. Filled ends of a connector are attached to the submodels sending data and empty ends are attached to the submodels receiving data. Connectors may be labeled to describe data transferred between models. Different connector types are described in detail in [14].

```
# set physical properties
cxa.env["kin_viscosity[m2/s]"] = 4E-6
cxa.env["U_max[m/s]"] = 0.121
cxa.env["rho0[kg/m3]"] = 1000
# ...

# declare kernels
cxa.add_kernel('bf',
   'kernel.flow3d.FlowTestController')
cxa.add_kernel('smc',
   'kernel.smc2d.SMCController')
cxa.add_kernel('smc2bf',
   'cxa.cxa3d.smc2bf.ObsArray2IncrementalLists3D')
# ...

# configure connection scheme
cs.attach('smc2bf' => 'bf') {
    tie('StaticSolid', 'BFObsExit')
    tie('NewSolid', 'BFincSolidExit')
    tie('NewFluid', 'BFincFluidExit')
}
# ...
```

Figure 8: CxA file fragment example.

The xMML format may be used as a base for generating code for coupling libraries such as MUSCLE (described in Section 4.3).

There are several tools using the xMML format, e.g. MAD (Multiscale Application Designer) and MAME (Mapper Memory)[1] developed as a part of the MAPPER project. MAD allows modeling multiscale applications in

a graphical environment. The resulting graphs can be stored as (or loaded from) xMML. Moreover, MAD supports creation of CxA file stubs. The second tool, MAME, allows storing, sharing and reusing various metadata describing multiscale applications (including mappers, filters and even implementations). MAME can also store xMML application descriptions (so that they could be easily accessed by other MAPPER tools).

CxA—MML prototype

The Complex Automata theory (CxA)[17] is a methodology for modelling complex multiscale systems. A Complex Automaton is a set of connected Cellular Automata and agent-based models, every one of them representing a single-scale simulation. Each automaton may consist of several other automata (hierarchical coupling).

The CxA theory is a base for the MUSCLE framework (described in Section 4.3). Each MUSCLE simulation must contain a configuration file, defining *kernels* (each representing a single automaton) and the connections between them.

The configuration file itself is written in Ruby programming language, therefore it is strongly attached to the MUSCLE framework.

Figure 8 shows an example fragment of CxA file. First, the environmental physical properties are set in the example. Then the computing *kernels* are declared and a connection scheme between them is configured.

3.3 Comparison

CellML and SBML are primarily used as single scale model description languages, therefore their possible uses as a primary tool for describing a multiscale application are limited. Although, they may be the best choice if it comes to modeling a single model contained in the application.

MML is a promising format, although not yet an established standard (such as CellML and SBML in their field). Tools such as MAME or MAD[1], which use MML, may help to promote the language. If MML earns well-deserved recognition it may become a leading format in description of multiscale models. The use of MML would certainly facilitate the design and development of multiscale applications. The proposed MML diagrams are also a tool which may ease the transfer of knowledge about legacy applications.

CxA is a successfully used format, although not a model description language per se. Its tight coupling to the MUSCLE framework greatly limits the possibility to use it with other libraries.

The tools allowing conversion from xMML to a library specific description format or even code stubs (as proposed in [14]) would certainly help promote MML as a multiscale model description language.

3.4 Summary

In this Chapter, we presented various model description languages. Languages aiming at detailed single scale models descriptions (SBML, CellML) have been introduced and described. XML code samples were shown and discussed.

The MML language and related diagram types describing multiscale applications as a whole were introduced in Section 3.2. We also presented an example of a CxA file and its relation to the MML language and the MUSCLE library.

This Chapter ended with a short comparison of different types of model description languages.

4 Coupling libraries

This Chapter presents different coupling libraries used for building multi-scale applications. In Sections 4.1—4.3 we introduce three example libraries. Section 4.4 compares the previously introduced tools.

4.1 AMUSE

Astrophysical Multipurpose Software Environment (AMUSE) is a framework for large-scale simulations of stellar systems (dynamics, stellar evolution, hydrodynamics, radiative transfer, etc.).

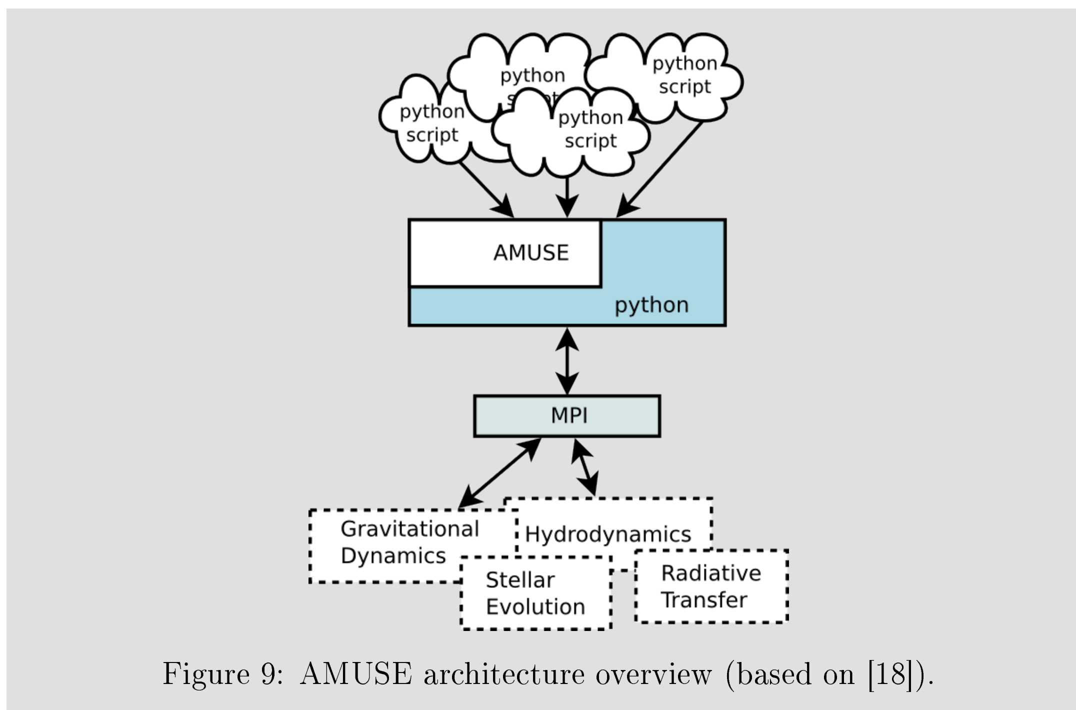


Figure 9: AMUSE architecture overview (based on [18]).

AMUSE is a tree layer framework, the layers being:

1. User Script layer—defines a specific problem and couples two or more codes from the lower layers.

2. AMUSE Code layer—a generic layer providing an object oriented interface on top of the legacy codes.

3. Legacy Codes layer—defines interfaces to the legacy codes, contains existing legacy codes (actual simulations).

Each higher layer adds functionality to the lower layer.

The AMUSE framework is written in Python with use of the Message Passing Interface. The first layer user scripts are written in Python. It is possible to integrate legacy C++ or Fortran code with the AMUSE framework.

Figure 9 shows an overview of the AMUSE architecture. Python scripts can be used to access the underlying AMUSE code layer which enables accessing the underlying codes using MPI.

4.2 MCT

The Model Coupling Toolkit (MCT)[19, 20] is a library for coupling models to form a parallel coupled model. MCT was build to bring together the parallel submodels which form the Community Climate System Model (CCSM). MCT solves problems of transferring data between different parallel programs, allowing for efficient data transfer for demanding interpolation algorithms. The MCT library is scalable and high-performing.

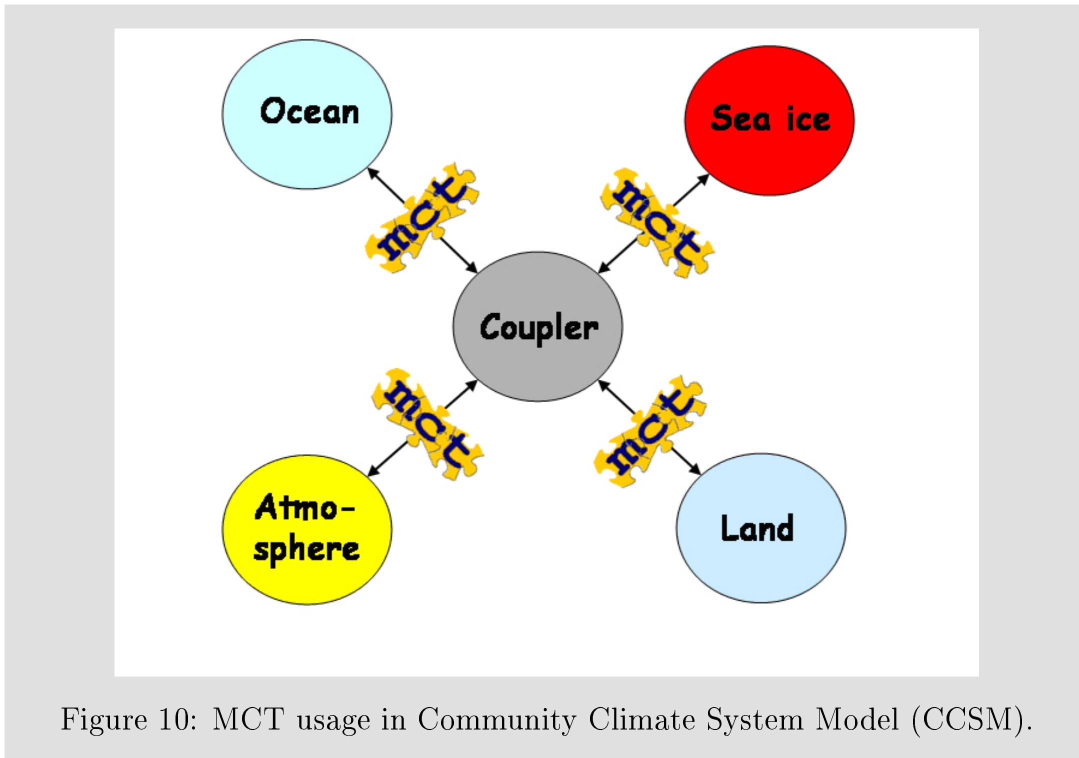


Figure 10: MCT usage in Community Climate System Model (CCSM).

Figure 10 shows cooperation between MCT and various submodels forming the CCSM model. MCT is used to transfer significant amount of data

between submodels.

MCT is available as the Fortran library, but C++ and Python bindings are also available through the external Babel library.

4.3 MUSCLE

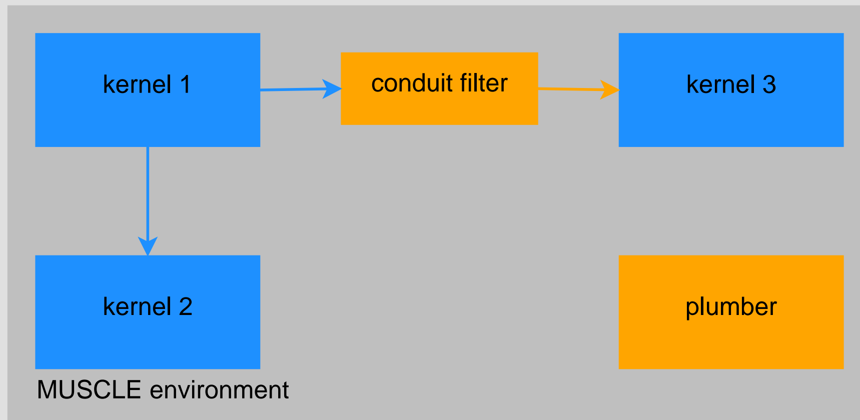
Multiscale Coupling Library and Environment (MUSCLE)[21] is a framework for running multiscale simulations. MUSCLE was primary developed in the COAST project and is currently used in the MAPPER project[4]. MUSCLE allows running simulations based on the complex automata theory (as described briefly in Section 3.2).

A complex automaton is a group of cellular automata and agent-based models. A MUSCLE simulation is a group of independent *kernels*—single scale simulations wrapped into a controller agent communicating with the core kernel (a *plumber*). Optional *conduit filters* may be used to alter data transferred between kernels. A conduit filter may perform simple transformations such as scale altering or coordinate conversion. A MUSCLE simulation can often be depicted using a Scale Specification Map (Section 3.2).

Figure 11 shows an example MUSCLE environment. Kernel 1 communicates with kernels 2 and 3. A conduit filter is used for altering the data transferred from kernel 2 to kernel 3. A plumber kernel is also depicted.

The MUSCLE framework is written in Ruby and Java, based on the Java Agent DEvelopment framework (JADE) [22].

MUSCLE allows developers to write kernels in Java and (using a supplementary library) in native code (C++/C, Fortran). Each kernel must define the scale it is operating at and the *portals* (in/out connections) it is using. Kernels can be connected using a configuration file (CxA, Section 3.2).



(a) A simple MUSCLE environment example.

```
# declare kernels
cxa.add_kernel('kernel1',
   'example.kernel.Kernel1')
cxa.add_kernel('kernel2',
   'example.kernel.Kernel2')
cxa.add_kernel('kernel3',
   'example.kernel.Kernel3')

# configure connection scheme
cs.attach('kernel1' => 'kernel2') {
    tie('OutA', 'InA')
}
cs.attach('kernel1' => 'kernel3') {
    tie('OutB', 'InB',
         Conduit.new("example.filter.ConduitFilter"))
}
```

(b) A CxA file fragment.

Figure 11: A MUSCLE environment example with a CxA file fragment describing it.

4.4 Comparison

Table 1 presents a short comparison of the previously described coupling libraries. The AMUSE and the MUSCLE frameworks seem to operate on a similar level of abstraction, whereas the MCT library solves the lower level problems, more related to parallel programming than to multiscale simulations.

MUSCLE library is more generic than AMUSE (AMUSE is limited to the large-scale stellar simulations).

	AMUSE	MCT	MUSCLE
Level	Multiscale simulations	Parallel programming	Multiscale simulations
Scope	Stellar simulations	Generic	Generic
Supported languages	C/C++, Fortran, Python	C/C++, Fortran, Python	C/C++, Fortran, Ruby, Java

Table 1: Coupling libraries comparison.

The three libraries compared allow the developer to use similar sets of programming languages (C++, C and Fortran for native code). AMUSE and MCT offer Python bindings, whereas MUSCLE kernels can be written in Java and the configuration files—in Ruby (as the whole framework is based on these two languages).

The MUSCLE framework seems to be the best suited tool for high level multiscale simulations. On the other hand, the MCT library may be the best choice for the parallel applications for which the performance is crucial.

4.5 Summary

In this Chapter, we presented three example coupling libraries (AMUSE, MCT and MUSCLE) which can be used to aid execution of multiscale simulations. After introducing each library, we presented their short comparison.

The MUST tool presented in this Thesis uses the MUSCLE framework based applications. The MUSCLE framework was chosen as the tool best suited for running multiscale applications because it operates on a high level of abstraction (contrary to the MCT toolkit) and is not a problem-specialized tool (as AMUSE). Its close relationship to the MML language was also an advantage, as MML aims at describing single scale models and interactions between them (i.e. whole multiscale applications). Section 3.2 describes MML in detail.

5 Infrastructures

This Chapter presents the grid and the cloud infrastructures. Various definitions of the grid are quoted in Section 5.1, which contains also a list of the most notable grid characteristics. Section 5.2 describes cloud computing: introduces its definitions and characteristics. Section 5.3 compares the grid and the cloud infrastructures, highlighting the differences and similarities between them. Aspects such as programming model, computing model, usability and standarization are compared in detail. Furthermore, both grid and cloud architectures are compared.

5.1 Grid

The term *grid computing* was first used in Ian Foster and Cart Kesselman's work *The Grid: Blueprint for a new computing infrastructure*[23]. It is a metaphor for computing power being as common and easy to access as public utilities such as electricity and water. Since then, many grid definitions emerged.

Definitions

Ian Foster, in his 2002 work *What is the Grid? A Three Point Checklist*[24], defines a grid as a system that:

1. *coordinates resources that are not subject to centralized control*
2. *using standard, open, general-purpose protocols and interfaces*
3. *to deliver nontrivial qualities of service.*

In his definition, Foster highlights the grid's independence from any centralized form of control. A grid system serves many different types of users oriented to various types of resources. A Grid system should also use standard protocol and interfaces to perform such fundamental tasks as user authentication, local and global security policies application, resource management. These qualities should allow the grid system to deliver service of a higher quality than a sum of it parts (e.g. meaning the automation of inter-system communication, single sign on policy, lower response times, higher availability and security levels).

Vaidy Sundernam proposes the following definition of a grid system [25]:

- *A grid system is a paradigm/infrastructure that enables the sharing, selection and aggregation of geographically distributed resources (computers, software, data/databases, people)*

- *depending on availability, capability, cost, QoS requirements*
- *for solving large-scale problems/applications*
- *within virtual organizations.*

Moreover, according to Vaidy Sundernam a grid system is NOT:

- *The Next generation Internet*
- *A new Operating System*
- *Just (a) a way to exploit unused cycles (b) a new mode of parallel computing (c) a new mode of P2P networking*

Sundernam emphasizes the geographic distribution of resources and their structured arrangement within Virtual Organizations. A Virtual Organization (VO) is a logical set of groups/institutions created to solve a common problem using the grid resources. Resource sharing, access restrictions, security policies etc. may be applied on the level of a Virtual Organization.

In [23] yet another grid definition is proposed by Foster and Kesselman: *A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.* This concise definition stresses high quality of grid services and large scale computing capabilities of the grid.

A common vision of a grid system emerges from all the definitions. It is a dependable infrastructure, offering the highest computing (and not only) capabilities by a common effort of many geographically spread institutions. Below, we present the main differences between the grid and a traditional distributed system or a computing cluster.

Grid Characteristics

The most significant grid characteristics which distinguish the grid from other distributed systems and computing clusters are briefly described below:

- Grid systems bring together heterogenic resources (as opposed to a distributed system) from different physical locations. It forces the cooperation with Wide Area Networks (which is rarely the case in computing clusters).
- Cooperation of numerous Virtual Organizations allows solving higher scale problems. Interoperability between VOs (intra-grid) and between

grid systems (inter-grid) is assured by the common open protocols and interfaces (few to none closed proprietary solutions are used).

- Grid systems are more generic and offer diversified resources as opposed to distributed systems and clusters which are often computing power-oriented.

Grid systems are generic tools having wide problem-solving appliance. Although the use of grid-specific tools is often needed but only once employed, it is easy to migrate between different grid systems thanks to unified architecture.

5.2 Cloud

Cloud computing is a metaphor for the abstraction of actual resources it represents. In general, cloud computing is associated with scalable computational resources available on a pay-per-use basis. There are three basic types of services exposed by Clouds.

1. Infrastructure as a Service (IaaS)—basic services (such as computational power, data storage) are available to the customers through abstraction and virtualization (i.e. computational power is abstracted to virtual machine instances, storage—to filesystems). Amazon Web Services (described further in Section 6.3) are probably the largest cloud services offered in such model.

2. Platform as a Service (PaaS) adds an extra abstraction level to IaaS. The service provided is an actual platform on which customer applications may run. This model hides unnecessary details from the customer (i.e. scaling of the application may be transparent for the customer developer). An example of such model is the Google App Engine[28].

3. Software as a Service (SaaS) exposes to users applications running in cloud systems. Such applications may be accessed from devices with small computational power (such as mobile devices, PDAs). Google Docs[29] is an example of SaaS. Although this application does not require large computing power (so it could be run on a PC), it takes advantage of being hosted in the cloud differently—by enabling document sharing between users (such a functionality is more natural since the whole application is run in the cloud).

Clouds may expose services on slightly different levels (theoretically, SaaS could be built on PaaS which could be deployed to IaaS) but there are definitions which try to find intersection between these kinds of services. The definition proposed in [27] is quoted below:

Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized Service Level Agreements.

This definition includes the key cloud features: virtualization, dynamic scalability, pay-per-use (utility-like) model and SLAs application. Other characteristics of the cloud mentioned in [27] are: user-friendliness, located in (available by) the Internet, variety of resources, automatic adaptation and resource optimization.

Some of these features (especially variety of resources, automatic adaption and resource optimization) could characterize also grids. On the other hand, user-friendliness does not seem to be a feature of grids. Doerksen in [30] even goes as far as saying that *Cloud computing is ... the user-friendly version of grid computing*, which is an obvious exaggeration.

5.3 Comparison

In 1990s, Grids were thought of as a technology which would allow consumers to use computing capabilities as a public utility (the term *Grid* itself comes from a comparison to the electrical grid). The growth of Grids was motivated by a good cause—to provide high computing capabilities transparent to everyone, using open protocols and interfaces.

Two decades later, supported by multi-billion dollar budgets of companies like Amazon, Google and Microsoft, Clouds emerged. The aim of those companies was slightly different—to bring computing power to the customers. Clouds use mostly proprietary solutions, they offer simple APIs, but there is little to no support for interoperability between different providers.

Both the Grid and the Cloud serve a similar purpose—to provide large scale computing capabilities for end-users. Table 2 shows a short comparison of the Grid and the Cloud infrastructures in a few different fields.

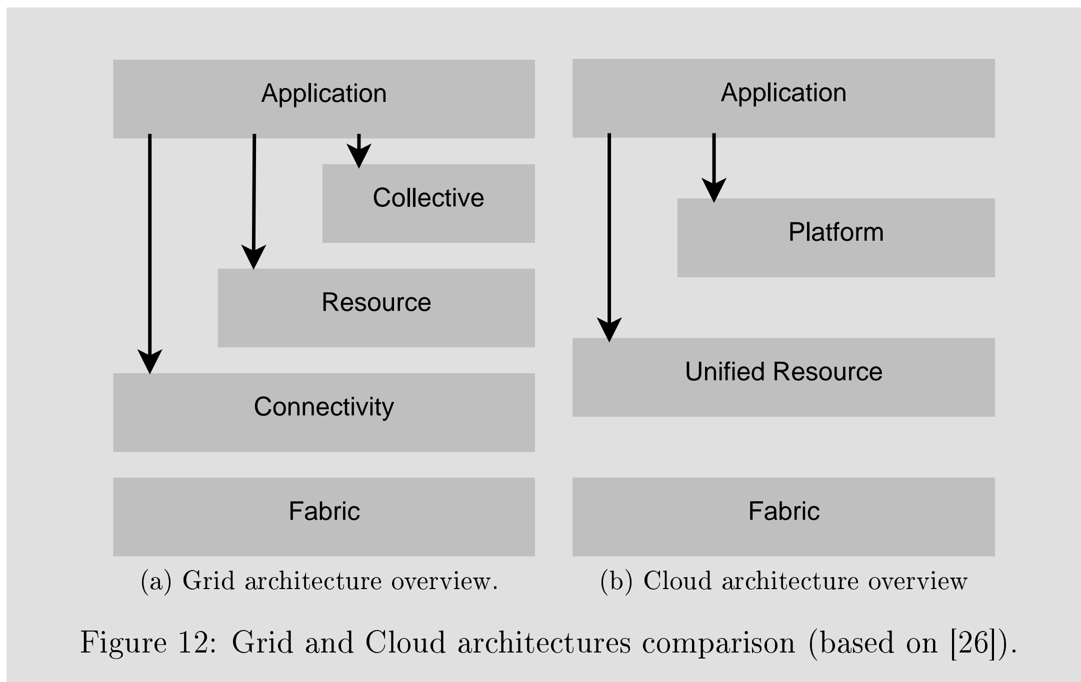
<table>
<tr><th></th><th>Grid</th><th>Cloud</th></tr>
<tr><td>Definitions</td><td>In 2002, Ian Foster created a short checklist defining a Grid:
1. Grid resources are not subject to centralized control.
2. Grid uses standard, open, general-purpose protocols and interfaces.
3. Grid provides non-trivial qualities of service.</td><td>Corresponding checklist describing a Cloud:
1. Cloud resources are often governed by a single organization.
2. Proprietary solutions are used, no interoperability between providers has been established so far.
3. Cloud provides non-trivial qualities of service.</td></tr>
<tr><td>Business Model</td><td>Resources (i.e. CPU hours) are assigned per project. Virtual Organizations use resource trading.</td><td>End-users pay for the resources consumed (a payment model similar to basic resources such as electricity, water etc.).</td></tr>
<tr><td>Compute Model</td><td>Resources are governed by a queuing system (such as PBS). Interactive tasks are rarely supported. However, efforts have been made to lower the latencies to resources.</td><td>Resources are shared by all users at the same time. This model allows latency-sensitive applications to operate. QoS may be difficult to preserve when the Clouds grow in scale and number of users.</td></tr>
<tr><td>Data Model</td><td colspan="2">As the amounts of data to process grow exponentially, the data transfer becomes the bottleneck of large scale computing. This is the concern of Grids as well as Clouds. Efforts have been made to ensure that data is processed in the nearest available location. Nevertheless, assurance of high QoS in such scenarios seems to be a challenge yet to be faced by both Grids and Clouds.</td></tr>
</table>

	Grid	Cloud
Virtualization	Grids do not use virtualization as often as Clouds (although some VOs may do so). However, there are technologies such as Nimbus which provide virtualization in the Grid. Its main purpose is to create virtual workspaces in separation from the physical resources.	Virtualization in the Cloud is used for separating the service from the physical infrastructure. It also helps to provide Service Level Agreement as the virtualized resources are easier to raise back in case of failure.
Programming Model	Grid programming does not differ much from the standard distributed programming, although there are some extra factors which have to be taken into consideration, such as: restricted access to resources, heterogenic resources, difficult exception handling.	Clouds (e.g. Amazon Web Services, Microsoft's Azure) generally offer pre-defined web services API (although integration of applications between different service providers is not a trivial task).
Security Model	Security has always been a fundamental part of Grids design as Grid spans on multiple organizations, each applying their own administration policy. Grids support single sign-on, delegation (user's program may inherit user's access rights) and privacy. User credentials required for login are never transported in an insecure way (i.e. emailed)—this policy is time-consuming but ensures a high level of security.	Clouds generally use slightly lower level of security than Grids. All user credentials may be created/changed online. That may present a potential risk (e.g. emailed passwords). It is in Cloud service providers interest to ensure the customers that their data is treated in a Grid-alike (i.e. redundant data storage, knowledge of physical localization of data, restricted access to data, etc.).
Usability	Although they aim at usability from the beginning, Grids are still not very user-friendly. The developer has to be aware of many middleware tools necessary for using the grid. Applications require a special design to be *grid-runnable*.	Clouds have been designed for usability. There are no special changes or design required for the applications to run in the cloud.

	Grid	Cloud
Standarization	Grids are based on open protocols and interfaces. Interoperability has always been a major concern in the grid design. Well defined standards on every level of communication help to expand, upgrade and maintain the grids.	Clouds are mostly proprietary and the inner mechanisms have been closed for the public. The lack of standards in inner APIs and, for example, virtual machine image format is an inhibitory factor for the global expansion of the Clouds.
Typical Usage	Grids are typically used for demanding batch jobs (spanning hundreds to thousands of nodes) requiring little to none user interaction.	Clouds are usually used for exposing scalable services to the outside.

Table 2: Grid and Cloud comparison (based on [26, 27])

Figure 12 shows a comparison of the Grid and the Cloud architectures. The architectures may seem similar but there are many differences.



(a) Grid architecture overview.

(b) Cloud architecture overview

Figure 12: Grid and Cloud architectures comparison (based on [26]).

The Grid's *fabric* layer provides access to the underlying resources including not only computing power and storage devices but also code repositories,

computed data or organization specific equipment. Resource managers (such as PBS) reside in fabric layer. The above *connectivity* layer ensures stable and secure communication. The *resource* layer is responsible for managing and restricting/providing access to individual resources. The *application* layer uses the Grid resources provided by lower layers APIs.

The Cloud's *fabric* layer represents the actual physical resources used by cloud services (most notably—the compute and storage resources). The above *unified resource* layer exposes and encapsulates the fundamental resources (e.g. in form of a virtual machine or a file system). The higher *platform* level adds necessary middleware and APIs to the lower layers, so that they can be accessed from the outside. The topmost *application* layer contains the actual applications running in the cloud.

The Grid's layers seem to be better separated and more specialized than those of Cloud. Each one of them has a single responsibility. This may be the result of the well defined protocols and interfaces of the Grid.

5.4 Summary

In this Chapter, we presented grid and cloud computing. Firstly, various grid and cloud definitions were quoted and, secondly the characteristics of the Grid and the Cloud were specified. We described also the three basic types of services exposed by cloud computing (Infrastructure as a Service, Platform as a Service and Software as a Service).

Later, grid and cloud computing were compared in detail with various aspects taken into consideration, including programming model, computing model and standarization of both infrastructures. Also, we compared separately the layered architectures.

6 Accessing infrastructures

This Chapter presents middleware tools used for accessing the grid and the cloud infrastructures described in the previous Chapter. Section 6.1 describes a queuing system used to access local PL-GRID resources (on the example of PBS—Portable Batch System). Section 6.2 introduces the GridSpace virtual laboratory—a high level framework allowing access to Grid-based resources. Section 6.3 describes an API used to access cloud computing, storage and messaging resources (on the example of the Amazon Web Services API).

6.1 Local resources—Portable Batch System

One of the main requirements for the MUST tool was integration with locally available PL-GRID resources. Those resources can be accessed using a PBS queue.

The Portable Batch System (PBS)[31] is a tool which allows job scheduling on distributed resources (most notably, cluster environments). PBS consists of both server and client side components. The server component manages queues and jobs, while the client components are mostly the commands allowing the user to handle batch jobs.

The most important PBS features are:

- Running tasks serially or concurrently.
- Scheduler which supports priorities and time restrictions.
- Support for policies that restricting access to certain resources for various jobs.

There are tools which use PBS and expose a higher level API, such as the grid middleware PBS-Globus interface[32]. It allows running jobs on the grid-based resources without using grid-specific job scheduling interfaces.

The GridSpace Virtual Laboratory allows using PBS on the PL-Grid resources through the language specific *gems* (described further in Section 6.2).

Other examples of queuing systems include OGE (Oracle Grid Engine, previously Sun Grid Engine)[42] and LSF (Load Sharing Facility)[43].

6.2 Grid resources—GridSpace

An integration with the GridSpace virtual laboratory was another vital requirement for the MUST tool.

The GridSpace Virtual Laboratory (GS)[44] is a high-level framework which improves and facilitates the usage of Grid-based resources. It allows scientists to develop, share and execute the so-called virtual *experiments*. A GridSpace experiment is a set of scripts (each script is called a *snippet*—available languages include Python, Ruby, Perl and Bash) which can be run on the provided HPC resources.

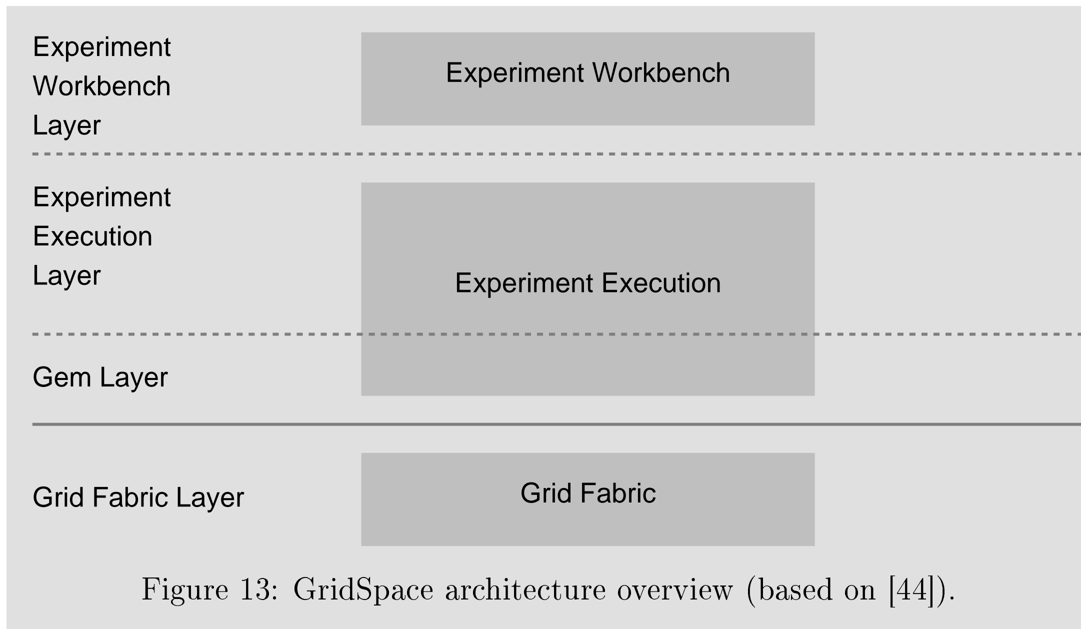


Figure 13: GridSpace architecture overview (based on [44]).

GridSpace has been developed with high usability in mind. The main GS advantages include:

- High-level experiments can be prepared by scientists in isolation from the grid middleware layer.
- All GridSpace features are accessible through a simple to use yet powerful Web 2.0 portal.
- Usage of the Single Sign On authentication mode (e.g. the PL-Grid installation uses the PL-Grid LDAP directory) so that a logged user may use the grid resources without any further authentication.
- Support for collaborative work (sharing experiments) with secure handling of sensitive data such as user credentials or certificates.

Figure 13 shows a simplified overview of the GridSpace layered architecture. The Experiment Workbench (a web browser accessible portal) layer

includes inter alia the Experiment Console and the Credential Manager. The Experiment Execution Layer (EE) contains a plan of the experiment execution and various script interpreters. The EE layer calls the lower *gem* layer. Gems are the small libraries providing APIs for the lower level resources in a particular scripting language.

An example grid fabric middleware accessible through the gem layer includes PBS[31], gLite[33], Unicore[34] or QosCosGrid[35].

The role of GridSpace is similar to that of workflow systems (e.g. Swift[36], Kepler[37], Taverna[38], Triana[39], MyExperiment[40], Pegasus[41], etc.).

6.3 Cloud resources—Amazon Web Services

The Amazon Web Services (AWS) based cloud infrastructure was chosen as the second execution environment for the MUST tool. Ease of access, wide use and availability of many middleware tools providing access to AWS were the main reasons behind the choice of this cloud environment. A wide range of services was another significant advantage.

Amazon provides a web services-based platform allowing users to use the cloud infrastructure. All AWS are accessible through a SOAP API which is a base for many programming languages APIs (including Ruby's RightAWS library[50], AWS Java Library or AWS .NET Library). The most important web services are briefly described below.

6.3.1 Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2)[45] is the core web service providing scalable computing capabilities. A simple API allows developers to run and manage numerous virtual machine instances (types differentiating in configuration of memory, CPU and hard drive are available). Various operating systems (including Microsoft Windows Server and many distributions of Linux) may be used. Virtual machine images with preinstalled specialist software (databases, web hosting software, etc.) are accessible.

A virtual machine (called an *instance*) may be started by means of a SOAP API. It can be accessed via SSH. EC2 has been designed for easy use with other AWS, such as Simple Storage Service (S3), Relational Database Service (RDS) or Simple Queue Service (SQS).

6.3.2 Simple Storage Service

Amazon Simple Storage Service (S3)[46]—allows users to store data online. Stored data may be public or private. Data is stored in so-called *buckets*. A

bucket is accessible through various authentication methods, while an object within a bucket is identified by a unique name. Each object may contain up to 5 terabytes of data. Standard REST and SOAP APIs are available for the data manipulation. The HTTP protocol is the default method of data transfer but a BitTorrent interface may also be used. The data is stored redundantly in a few localizations to minimize the probability of data loss.

6.3.3 Elastic Block Storage

Amazon Elastic Block Storage (EBS)[47] provides storage volumes which can be used with running EC2 instances. Usual usage examples of such volumes are databases, file systems or raw block level storage. The state of EBS volumes persists while detached from running EC2 instance, redundant copies of the data are automatically created (transparently for the end-user). It is possible to create a so-called *snapshot* of an EBS volume, which may serve as a backup copy or the source for a new EBS volume. Amazon hosts public data sets which may be used as an EBS volume (examples of data available in such sets include: Annotated Human Genome Data, US Census Databases, Freebase Data Dump, etc. [48]).

6.3.4 Simple Queue Service

Amazon Simple Queue Service (SQS)[49]—is a simple distributed queue messaging service. SQS may be used for building a workflow whose elements may be on different networks, using different technologies and not even running at the same time. SQS allows creating, reading and deleting messages. A read message may become locked (i.e. not visible) for other machines processing messages from the same queue. If processing fails, message is unlocked again. Otherwise, the message may be deleted so that it is not processed multiple times.

SQS queues may be shared between applications (i.e. multiple applications may have the access to the same queue) but simultaneously access to the queues is restricted and requires the usage of one of the few authentication methods (unless the queue is accessible anonymously).

6.4 Summary

This Chapter presented different middleware tools operating on various levels used by MUST to access and manage resources provided by the Grid and the Cloud infrastructures.

The PBS queuing system which is used by the MUST tool to execute applications on the local PL-GRID infrastructure was described in Section 6.1.

The GridSpace virtual laboratory, whose web interface allows the usage of the MUST tool, was introduced in Section 6.2. Configuration files, startup arguments and input sandboxes may be defined from the GS level. While running in the Grid environment, MUST employs the previously mentioned PBS *pbsdsh* command to allocate nodes and run a slave script on each of them.

Amazon Web Services API described in Section 6.3 is used to communicate with AWS from the MUST sender script level.

Detailed information regarding the usage of the mentioned tools is provided in subsequent chapters.

7 MUST User Support Tool

This Chapter presents MUST User Support Tool. Section 7.1 introduces the MUST tool and lists its requirements. Then, in Section 7.3, various use cases are presented.

Section 7.4 describes the MUST architecture (divided into the grid and the cloud architecture). Sequence diagrams showing time dependencies are also included in this Section.

7.1 Concept of MUST

MUST allows the end-users to run MUSCLE (Section 4.3) based multiscale applications (such as ISR—Section 2.3) in form of a batch job from the GridSpace (Section 6.2) level. It reserves applicable resources and then starts each group of computing kernels on a separate working node. The standard output and error streams are constantly submitted back to GridSpace and displayed live. After the execution, results can be send back to GridSpace for analysis.

MUST supports multiscale applications written using the MUSCLE library. They can be run both on the grid (Section 5.1) and the cloud (Section 5.2) infrastructures.

Running MUSCLE applications manually in distributed infrastructures is very time-consuming. When using a local cluster, it requires requesting the applicable number of working nodes using a PBS queue (Section 6.1), then logging in to each working node and manually coupling the computing kernels (i.e. specifying the group of computing kernels to be run on each node and providing the physical address of the plumber; Section 4.3). The manual process of starting a MUSCLE application using EC2 (Section 6.3.1) is even more complicated—the user needs to start multiple virtual machine instances, manually couple the computing kernels and download the result files (if necessary).

MUST simplifies the process described above. The end-user can start MUSCLE applications in distributed environment using the GridSpace virtual laboratory—there is no need to request computing nodes or to start virtual machine instances, as the computing kernels are coupled automatically by MUST and, when using the EC2 version, the output files are automatically sent to the user's S3 bucket (Section 6.3.2).

MUSCLE-based applications require CxA (Section 3.2) configuration files to define the input/output parameters (which are exchanged between the computing kernels) and the connection scheme of the computing kernels. Various application settings may also be saved in CxA files. MUST allows

the user to edit CxA in the GridSpace virtual laboratory. The edited CxA files are then sent to the cluster's working nodes or running virtual machine instances and used by the MUSCLE application.

7.2 Requirements

Requirements for the MUST tool are described below.

Cooperation with GridSpace virtual laboratory

MUST had to be accessible from the GridSpace level so that users could easily share experiments and results. Moreover, minimum to none changes to the legacy applications are required.

Different infrastructures support

MUST can execute applications on both grid and cloud infrastructures. Applications have to be deployed manually, although there is a basic virtual machine image which can serve as a base for a personalized image (for use on the cloud infrastructure).

Ease of expansion

Support for other infrastructures and different multiscale coupling libraries can be easily added to MUST. Logic concerning different infrastructures is carefully separated. To expand the tool (i.e. add support for a new infrastructure) the developer should implement a few steps (e.g. allocating nodes, launching tasks, collecting output). The implementation details and expansion possibilities are described in Sections 8.1 and 8.2 respectively.

7.3 Use cases

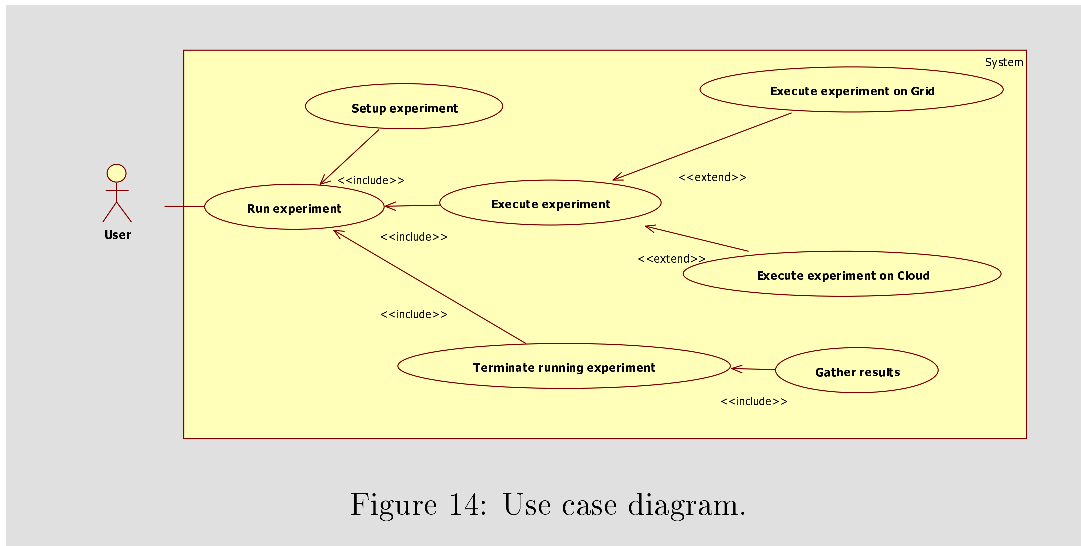


Figure 14: Use case diagram.

Figure 14 shows a use case diagram. The **run experiment** use case is described below:

- **Setup experiment**—The user inputs experiment parameters and may define the input sandbox for cloud use.
- **Execute experiment**—The experiment is started on the chosen infrastructure (grid or cloud).
- **Terminate running experiment**—The running experiment may be terminated. If it is running on the cloud infrastructure, after the experiment has stopped (i.e. it has finished or has been terminated), computed resulting files are gathered.

7.4 Architecture

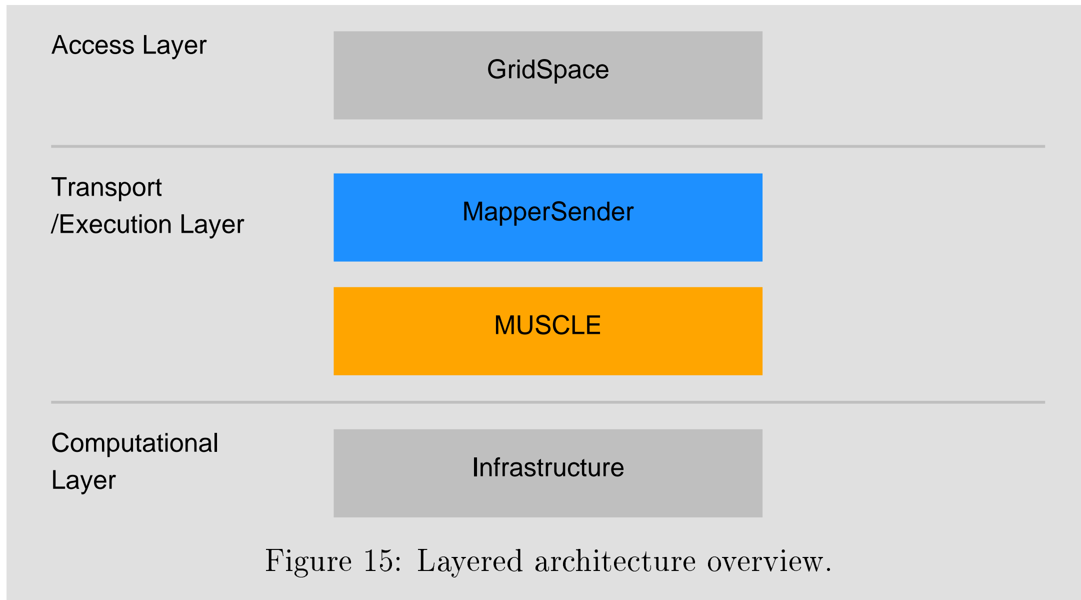


Figure 15: Layered architecture overview.

Figure 15 presents the layered architecture of the MUST environment. Please note that the layers presented on Figure 15 are separated conceptually rather than physically.

The Access layer is responsible for interaction with the end-user (i.e. defining startup scripts, setting up environmental variables, displaying results etc.).

The Transport and Execution layer consists of:

- MUST—responsible for launching the MUSCLE platform in the distributed environment, sending necessary input files and gathering the results,
- MUSCLE platform—which executes the underlying single scale simulations and is responsible for inter-simulation communication.

The Computational layer performs the actual calculations (i.e. runs a single MUSCLE kernel).

7.4.1 MUST Architecture—Grid

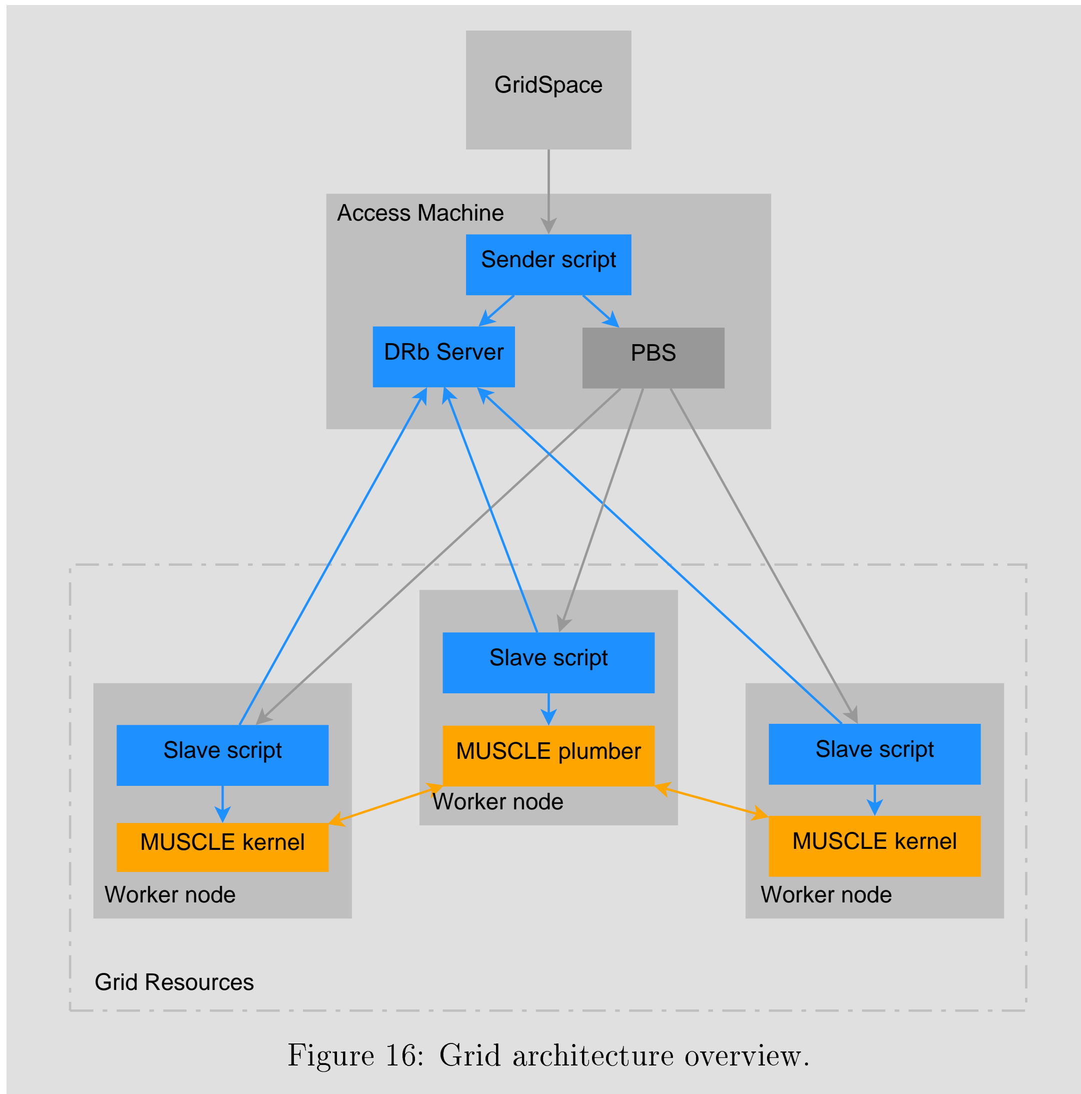


Figure 16: Grid architecture overview.

Figure 16 shows a detailed MUST architecture (with a usage of the grid infrastructure). MUSCLE kernels running on working nodes are depicted using yellow rectangles. Intra-MUSCLE communication is shown in yellow as well. MUST parts are depicted using blue rectangles (sender script and DRb Server on the access machine and slave scripts on the working nodes). MUST messages are shown in blue (scheduling PBS tasks and starting DRb Server, starting MUSCLE on each working node and reporting progress to the DRb Server). Other communication is shown in gray (i.e. GridSpace

communication with the access machine and starting PBS tasks on worker nodes).

Communication on various levels is shown on the Diagrams 16, 17 and 18.

Communication overview

The course of action of the tool on different infrastructures is described below.

First, a sender script is executed on the access machine. It reserves working nodes, starts a DRb server and sends a task description to nodes using the local queuing system.

Then, a slave script is executed on each working node. It connects with the server and waits for a command description. This command description is prepared by the server (i.e. each working node is assigned a different group of kernels to execute). After receiving the description, an appropriate command is executed on each working node. The standard output end error streams are then sent back to the server line by line.

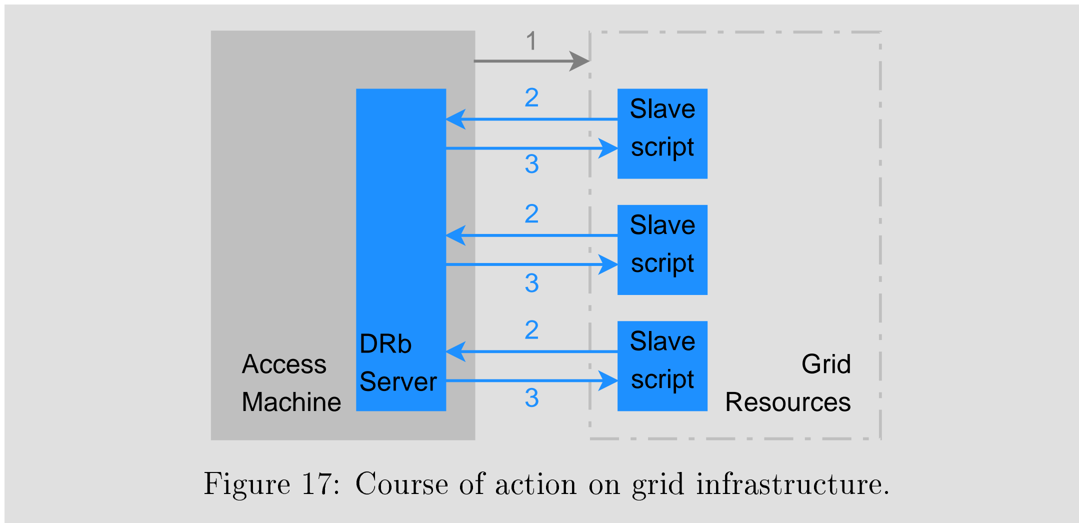


Figure 17: Course of action on grid infrastructure.

A detailed communication process is depicted on Figure 17 and described below:

1. Access machine sender script: the server on access machine is started, the grid resources are reserved using the local queuing system.

2. Nodes slave scripts: each node connects with the server.

3. Server method: The details about groups of kernels are sent to working nodes. Groups of kernels are then started.

Figure 18 shows a sequence diagram describing time cooperation between the entities taking part in the process of running MUST on the grid infrastructure. A detailed description follows:

1.–3. PBSSender creates an instance of PBSTaskManager, a DRbServer is started and PBS environment is initialized.

4.–5. Jobs are submitted and an adequate number of PBSTask instances is started.

6.–8. Tasks register themselves with the task manager and get the command to be executed.

9. MUSCLE kernels are started.

10.–11. Progress is reported by the MUSCLE kernels to PBSTask, and then to the task manager.

12.–15. MUSCLE kernels report having finished computations. PBSTask instances unregister themselves from the task manager. PBSTaskManager and DRbServer are stopped.

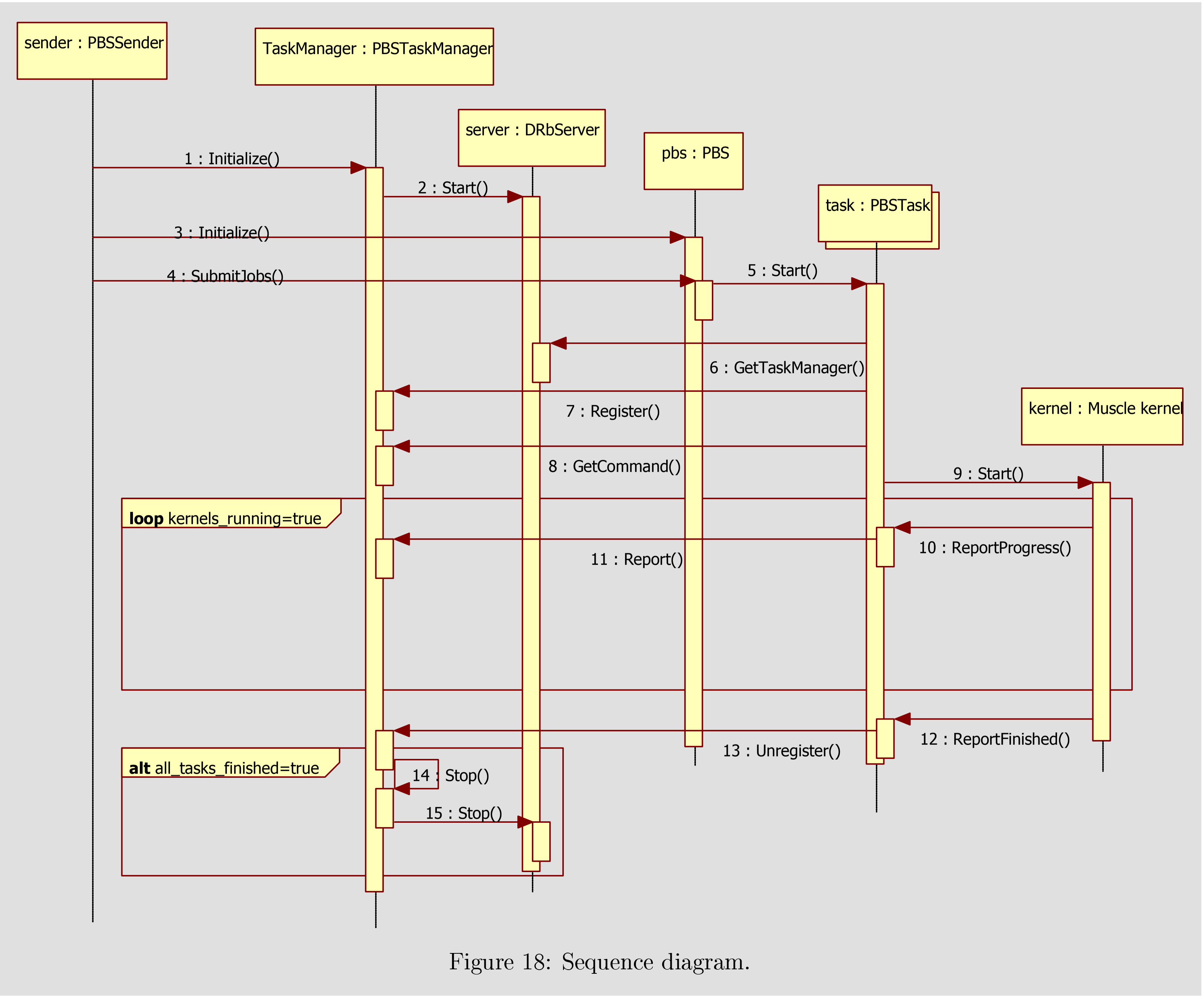


Figure 18: Sequence diagram.

7.4.2 MUST Architecture—Cloud

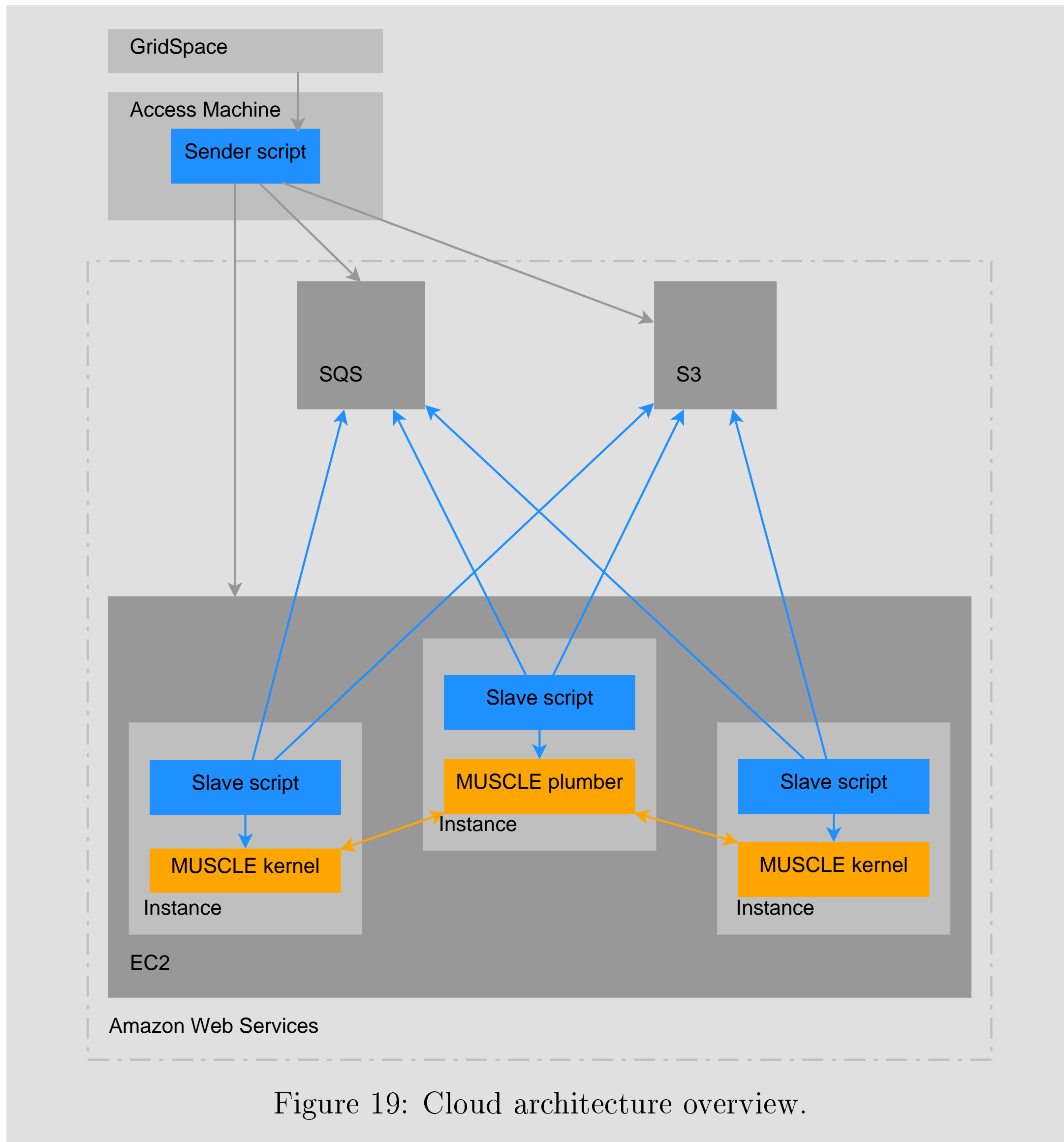


Figure 19: Cloud architecture overview.

Figure 19 shows a MUST architecture overview with use of the cloud infrastructure. The layered architecture is highlighted with different colours. MUSCLE kernels running on the virtual machine instances are shown as yellow rectangles. Intra-MUSCLE communication is shown in yellow as well. MUST scripts are depicted using blue rectangles (sender script and slave scripts executed on running virtual machine instances). MUST communication is shown in blue. Amazon services used are depicted using dark gray

rectangles (SQS, S3 and EC2). Other communication is shown in gray (i.e. GridSpace communication with the access machine and MUST communication with the Amazon web services).

Communication overview

First, a script similar to the one described above is executed on the access machine. It creates SQS queues which will enable communication between the access machine and the working nodes. There are five SQS queues used by the application:

- Control queue—this queue is used for sending the initial message to the virtual machine which will start the MUSCLE plumber, and for sending the commands to be executed to all running virtual machine instances.
- Plumber queue—this queue is used for notifying the virtual machine instances if the MUSCLE plumber has already been started.
- Report queue—this queue is used for sending the output streams from the running virtual machine instances back to the sender.
- Shutting down queue—this queue is used for sending the terminating message if the application has been shut down.
- Results queue—this queue is used for notifying the sender that the result files from each working instance have been sent to the S3 bucket after the execution has ended.

After the queues have been created, a starting message containing groups of kernels is sent to the control queue. Finally, a preset amount of virtual machines is started with the RightAWS API.

The launched machines are based on a preconfigured Amazon Machine Image. After booting, each virtual machine executes a startup script. It reads a message from a specified control SQS queue (or waits if there are no messages). First, a processed message results in starting of a plumber kernel on one of the working nodes. After that, the same node sends messages to the control SQS queue (each containing details about one job to start, i.e. one group of the MUSCLE kernels). The messages are then read by other working nodes and each group of kernels is respectively started.

After the execution, an S3 bucket is created and the generated results are gathered and sent to the bucket.

A detailed communication process is depicted on Figure 20 and described below (sending a live output and gathering results is omitted for clarity):

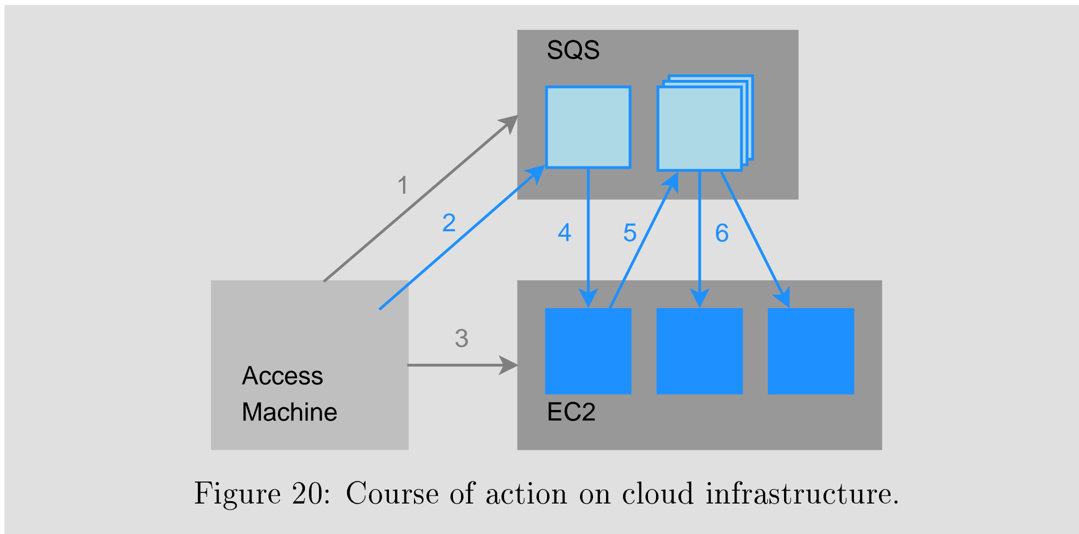


Figure 20: Course of action on cloud infrastructure.

1. Access machine script: SQS queues are created.
2. Access machine script: A message containing information about all groups of kernels is sent to the SQS queue.
3. Access machine script: A required number of virtual machine instances is started.
4. Instance startup script: The message containing information about all groups of kernels is read from the queue by the first instance.
5. Instance startup script: Messages (each containing information about one group of kernels) are sent to the SQS queue.
6. Instance startup script: One message containing information about a particular group of kernels is read by each virtual machine instance. An appropriate group of kernels is then started by each instance.

Figure 21 shows a sequence diagram depicting cooperation and time dependencies of different entities present in the process of running MUST. A brief specification follows:

1.—4. The Sender script creates an SQS queue and sends to it a message containing information about the MUSCLE kernels. Then, an S3 bucket is created and an input sandbox is sent.

5. An adequate number of EC2 instances is started.

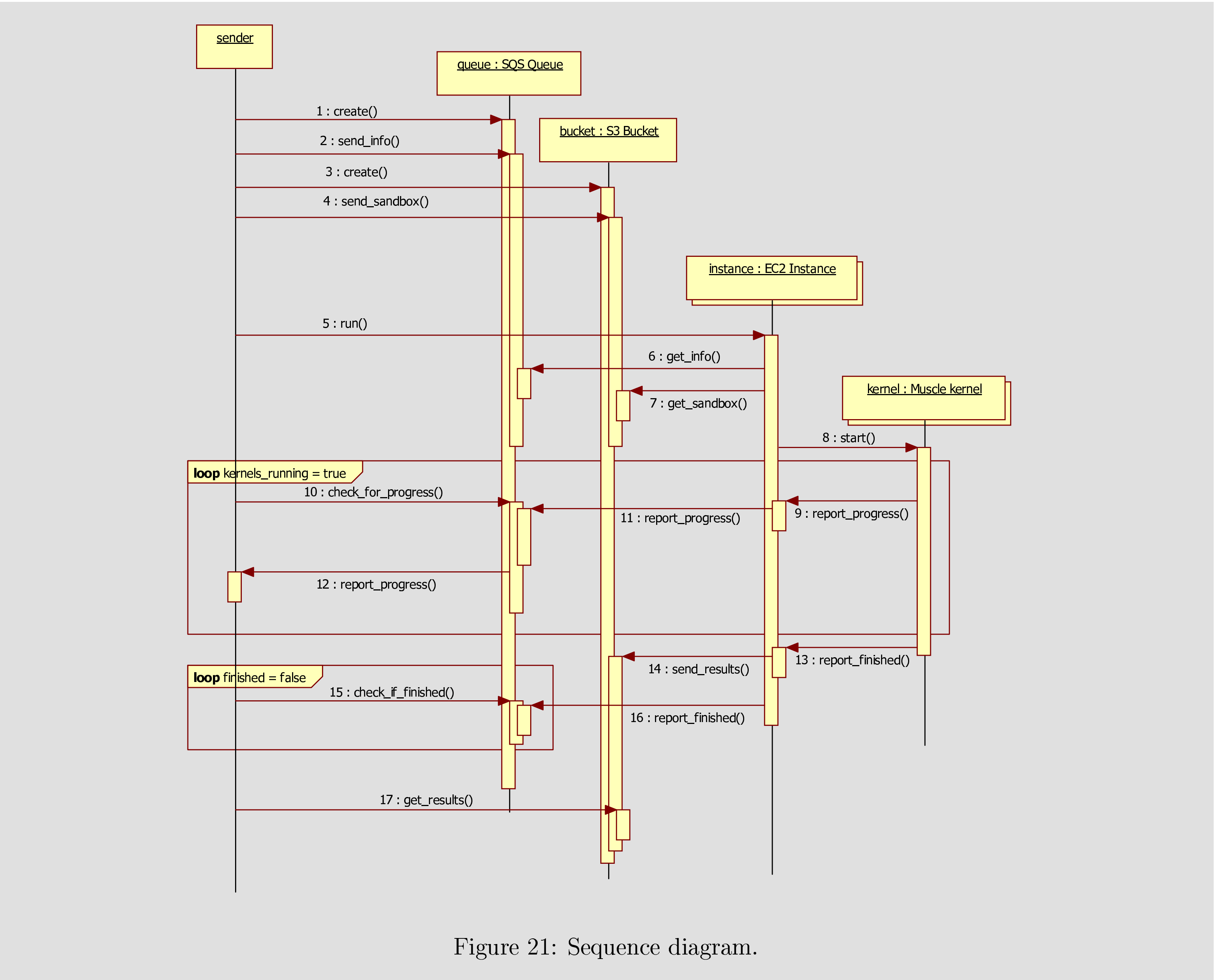


Figure 21: Sequence diagram.

6.–7. The EC2 instances read messages from the SQS queue and download the input sandbox.

8. The MUSCLE kernels are started.

9.–12. A progress is reported by the MUSCLE kernels to the EC2 instances, then to the SQS queue, and finally to the sender script which is continuously checking the SQS queue.

13.–17. The MUSCLE kernels report having finished computations. The EC2 instances gather the results and send them to the S3 bucket. Finally, the sender script downloads the results.

7.5 Summary

In this Chapter we presented the MUST tool and its requirements. MUST allows execution of MUSCLE-based multiscale applications in the grid and the cloud environments. The tool is accessible through the GridSpace virtual laboratory web interface.

We also described in detail the architecture of the MUST tool and presented the sequence diagrams (divided into grid and cloud cases).

The following Chapter shows the implementation details and describes steps needed in order to add support for different execution environments.

8 Implementation details

This Chapter presents the MUST implementation details. Section 8.1 includes a class diagram showing dependencies between different parts of the MUST tool (and its use of external libraries). Section 8.2 describes different possibilities of expansion (i.e. adding support for various execution environments and coupling libraries). Section 8.3 lists and describes the tools used by MUST to cooperate with MUSCLE applications, the GridSpace virtual laboratory and grid and cloud resources.

8.1 MUST implementation

Figure 22 shows the class diagram depicting the structure of the MUST tool and the relationships between its components. The classes are briefly described below (for readability, grid and cloud classes have been separated).

- **ISender**—an interface exposing the highest level methods invocated by the execution scripts (the implementing class is chosen based on the script execution mode). ISender exposes only three methods—*PrepareStart()*, *Start()* and *ReceiveOutput()*.
- **Grid** classes—PBSSender, PBSTaskManager, PBSTask and PBS.
  - **PBSSender** is used to read input arguments to prepare a distributed execution of the MUSCLE kernels in grid environment. *PrepareStart()* method starts an instance of PBSTaskManager and creates a PBSDSH job description file. *Start()* method prepares the PBS environment (using GridSpace's PBS gem) and starts a PBSDSH job (reading the job description from the previously created file). *ReceiveOutput()* method uses PBSTaskManager to print a standard output.
  - **PBSTaskManager** class is used for PBSTask management—it registers and unregisters tasks, receives their output (while there are any tasks running) and creates a command to execute for each task (based on the parameters received form PBSSender). PBSTaskManager uses a Distributed Ruby server to communicate with tasks (server is started when PBSTaskManager is created and its address is passed to the working nodes in the PBSDSH job description file).
  - **PBSTask** class instance is created on each working node. It connects to an instance of PBSTaskManager (using its DRb server

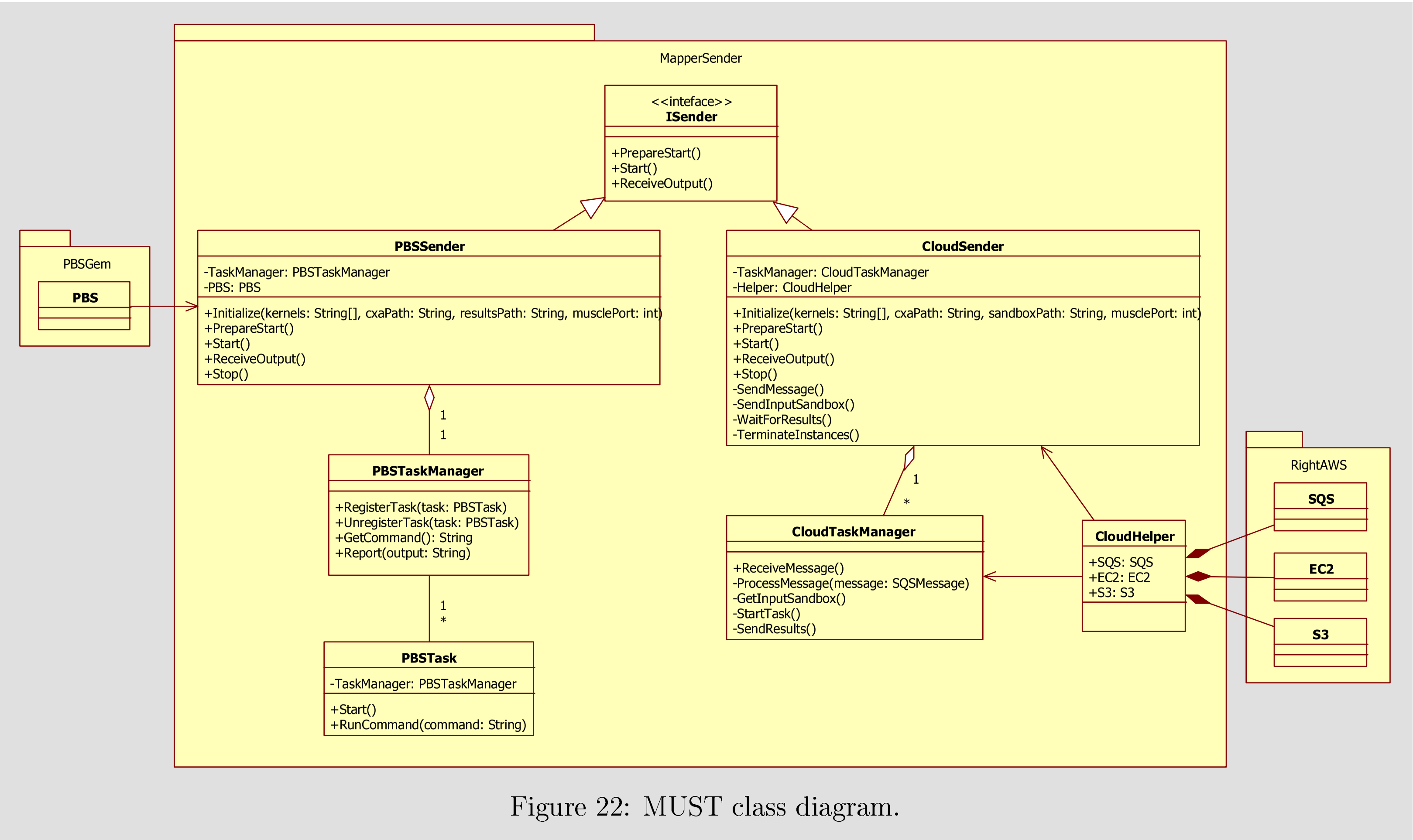


Figure 22: MUST class diagram.

whose address is passed to each working node by PBSDSH). PBSTask runs a group of MUSCLE kernels and sends its output to PBSTaskManager.

  - PBSSender uses GridSpace's **PBS** gem (GridSpace gems are described in Section 6.2) to access the underlying PBS system.

- **Cloud** classes—CloudSender, CloudTaskManager, CloudHelper and RightAWS classes.

  - **CloudSender** class prepares, starts and terminates the distributed execution of the MUSCLE kernels in cloud environments. *PrepareStart()* method creates an instance of CloudHelper, creates SQS queues, sends an input sandbox to an S3 bucket and sends a message containing information about groups of MUSCLE kernels to the previously created coordinating queue. *Start()* method starts Amazon EC2 instances using the previously created CloudHelper class instance. *ReceiveOutput()* method reads messages from the output queue and prints them. After the execution is finished, *ReceiveOutput()* receives output from the S3 bucket and terminates the running instances.
  - **CloudTaskManager** instance is created on each started EC2 instance. It reads information about SQS queues from instance startup arguments, connects with the coordinating SQS queue, reads information about a group of the MUSCLE kernels to run, downloads an input sandbox from the S3 bucket and starts the execution of a particular group of the MUSCLE kernels. The output is transmitted to the SQS output queue and, after the execution is finished, the resulting files are sent to the S3 bucket.
  - **CloudHelper** class creates, initializes and exposes RightAWS library classes allowing access to the Amazon Web Services in Ruby.

A CloudTaskManager instance is created on each running instance, contrary to the PBSTaskManager, because communication uses SQS queues, and each EC2 instance has to connect with the appropriate queues separately.

8.2 Expansion possibilities

There are two main possibilities of expansion of the MUST tool—adding support for new infrastructures and adding support for different coupling libraries.

New infrastructures

In order to add support for a new infrastructure, it would be necessary to create a class implementing the ISender interface (described in Section 8.1) to the MUST tool. The new class could take advantage of the existing execution scripts to integrate with the GridSpace virtual laboratory, read the MUSCLE CxA files and group the MUSCLE kernels. It would be necessary, however, to expand the execution scripts—add new execution mode and pass any infrastructure-specific arguments to a new class implementing the ISender interface.

Depending on the type of the new infrastructure to support, the existing classes could be expanded to allow switching between infrastructures (i.e. cloud classes could be expanded to add support for any non-Amazon-based cloud infrastructure; grid classes could be expanded to allow usage of a queuing system other than PBS).

New coupling libraries

It would be necessary to abstract the execution commands in order to add support for executing multiscale applications based on coupling libraries other than MUSCLE. Execution commands could be switched based on the chosen library mode. Passing execution arguments, however, would present a greater challenge, as the current execution model allows only to pass MUSCLE-specific arguments and a group of kernels to particular working nodes.

8.3 Tools used

MUST uses various tools to cooperate with MUSCLE applications, the GridSpace virtual laboratory, grid resources and the AWS-based cloud infrastructure. The version of each tool used for development and testing is given in brackets where suitable.

The MUSCLE (MUSCLE_2010-01-11_13-51-27)[3]. library is necessary to use MUST.

[3] The latest MUSCLE version can be downloaded from `http://muscle.berlios.de/`

Ruby (1.8.7) and Java (1.6.0) are used to run MUSCLE-based applications. Rubygems (1.3.5) library is also needed by some of the further dependencies.

The right_aws (2.1.0) Ruby gem and EC2 API Tools (1.4.4.1) are used to communicate with the AWS cloud.

The pbs_gem[4] is used to communicate with the local PBS queue from the GridSoace virtual laboratory.

The json (1.5.1) Ruby gem is used by the graphical tool for grouping kernels [2].

8.4 Summary

This Chapter described the MUST implementation details. The usage of external libraries was discussed, including a detailed specification of classes forming the MUST tool.

After the implementation details, we discussed the possibilities of expanding MUST and we also listed the steps necessary to add support for new execution environments and new coupling libraries.

Finally, various tools used by MUST were listed and described.

[4] A gem developed as a part of the GridSpace virtual laboratory.

9 Case study

This Chapter presents the comparison of the performance results of the execution of a scientific application launched using the MUST tool on the grid and the cloud infrastructures. The tests results using different setups and infrastructures are presented and interpreted. We discuss separately all the executed steps and compare them.

9.1 ISR2D performance results

The tests were performed in order to compare performance results on the grid and the cloud infrastructures. An in-stent restenosis 2D application (described in Section 2.3) was executed on both infrastructures with different setups.

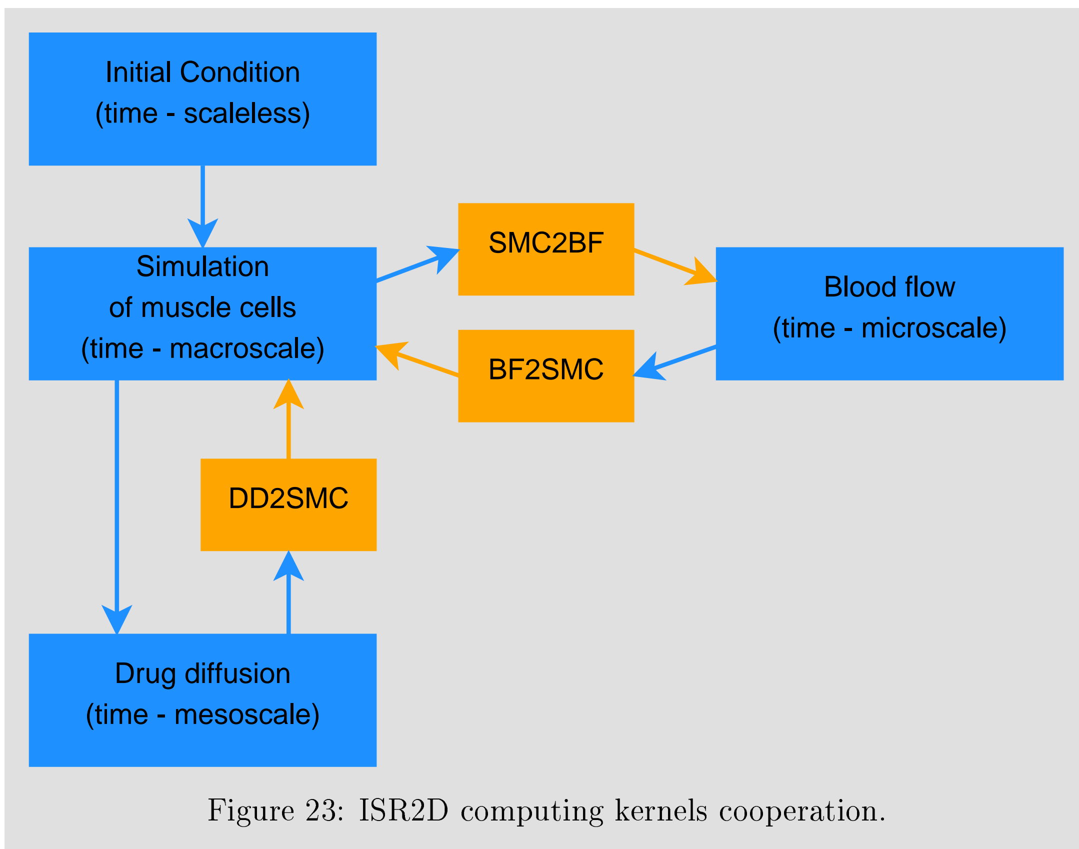


Figure 23: ISR2D computing kernels cooperation.

Figure 23 shows the MUSCLE computing kernels taking part in the in-stent restenosis 2D simulation (with time scales and data flow marked).

There are four main computing kernels (initial condition, blood flow, drug diffusion and simulation of muscle cells) taking part in the simulation. There are also three helper kernels which aid communication between kernels in different time scales.

The in-stent restenosis modeling is started by the initial condition kernel which holds information about initial cell placement and metadata. In each iteration, the simulation of muscle cells kernel (SMC) processes the current information to check whether any thrombus is formed due to the back flow of the blood. Updated conditions are then passed to blood flow and drug diffusion kernels (using helper kernels). New information about blood flow and drug diffusion are then sent back to the SMC kernel, which can start the next iteration[7].

The local PL-Grid resources (Zeus cluster) have been used for the grid tests. Specification: Zeus, Cluster Platform 3000 BL 2x220 with Xeon X5650 6C 2.66 GHz processors, connected with Infiniband, hosted at Cyfronet, Kraków, number 88 on the November 2011 Top 500 list[5].

The Amazon Web Services Cloud resources have been used for the cloud tests. Two different setups were used and compared:

- m1.xlarge instances (High-CPU Extra Large)—20 EC2 Compute Units[6], 7GB memory, 1690 GB local storage, 64 bit platform.
- m2.4xlarge instances (Cluster Compute Quadruple Extra Large)—33.5 EC2 Compute Units, 23GB memory, 1690 GB local storage, 64 bit platform.

Table 3 shows the performance results on both grid and cloud infrastructures. The columns of Table 3 are described below.

- *Iterations* column shows how many ISR2D iterations were executed in the test.
- *Infrastructure*—indicates the type of infrastructure (grid/cloud).
- *Instance type*—various instance types were used while testing on the cloud infrastructure (m1.xlarge and m2.4xlarge).
- *Submission* column lists the submission time in seconds. It is comprised of:

[5] `http://i.top500.org/system/177388`

[6] One EC2 Compute Unit is the equivalent of 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

- Grid—executing sender script, sending a PBS job to the queue, waiting in the queue and launching the MUSCLE environment.
- Cloud—executing sender script, creating the SQS queues needed, sending starting messages, sending an input sandbox to an S3 bucket, launching EC2 instances, downloading an input sandbox on each launched instance and launching the MUSCLE environment.

- *Execution* column lists the actual MUSCLE kernels execution time in seconds (starting after launching the MUSCLE environment and ending after receiving the last line of kernels output).
- *Total* column is the sum of the submission and the execution time in seconds.
- *Sending output* column lists the time needed to send the output files to an S3 bucket in seconds (the step performed only while executing in the cloud infrastructure).
- *Total + sending output* column is the sum of the total and the sending output times in seconds.

Table 3 shows the minimum, average and maximum execution times and their standard deviations of each step for each setup.

	iterations	infrastructure	instance type	submission [s]	execution [s]	total [s]	sending output [s]	total+sending output [s]
min	15	grid	-	10	166	198	0	198
average	15	grid	-	96,7	195,1	291,8	0	291,8
max	15	grid	-	363	211	562	0	562
σ	15	grid	-	-	16,8	-	0	-
min	15	cloud	m1.xlarge	73	195	278	82	382
average	15	cloud	m1.xlarge	81,3	247,2	328,5	101,8	430,3
max	15	cloud	m1.xlarge	91	259	345	116	447
σ	15	cloud	m1.xlarge	6,58	18,6	19,68	10,06	18,7

	iterations	infrastructure	instance type	submission [s]	execution [s]	total [s]	sending output [s]	total+sending output [s]
min	15	cloud	m2.4xlarge	69	181	259	102	374
average	15	cloud	m2.4xlarge	80,36	187,18	267,55	128,09	395,64
max	15	cloud	m2.4xlarge	101	191	289	189	448
σ	15	cloud	m2.4xlarge	10,28	3,19	10,07	24,7	22,06
min	150	grid	-	6	1381	1394	0	1394
average	150	grid	-	12	1512,46	1524,46	0	1524,46
max	150	grid	-	15	1708	1719	0	1719
σ	150	grid	-	-	131,83	130,09	0	130,09
min	150	cloud	m1.xlarge	67	2048	2123	95	2218
average	150	cloud	m1.xlarge	72,6	2068,07	2140,67	121,07	2261,73
max	150	cloud	m1.xlarge	83	2107	2178	152	2312
σ	150	cloud	m1.xlarge	4,21	15,35	15,80	18,11	29,74
min	150	cloud	m2.4xlarge	69	1521	1592	11	1614
average	150	cloud	m2.4xlarge	73,64	1526,5	1600,14	110,71	1710,86
max	150	cloud	m2.4xlarge	82	1534	1608	213	1809
σ	150	cloud	m2.4xlarge	4,18	3,80	4,85	60,59	60,23

Table 3: ISR2D—minimum, average and maximum execution times and their standard deviations in different setups.

9.2 Performance results interpretation

The results show that the cloud infrastructure is slightly slower when executing a larger number of iterations (1524s total time on the grid vs 1600s total time on the cloud—including the *after* step in comparison is purposeless, as the step is not executed on the grid infrastructure because of a common file system).

The total average execution time is only about 5% longer on the cloud—it is caused primarily by the extra steps performed (i.e. sending and downloading an input sandbox and booting virtual machine instances). The virtualization overhead on the cloud seems negligible, based on the results of the tests performed.

All the tests show that the standard deviation of execution and submission times is much lower on the cloud infrastructure. The execution and submission times on the grid seem much more dependent on instantaneous load of the execution nodes. The execution times range from 1394 to 1719 seconds on the grid (23% difference) and from 1592 to 1608 seconds on the cloud (only 1% difference).

The submission times on the grid infrastructure depend primarily on the number of jobs currently queued in the PBS system. This is clearly visible when comparing the 15 and 150 iterations test results, as the tests were performed separately (96,7s average for 15 iterations and 12s average for 150 iterations—805% difference).

9.3 Summary

This Chapter presented the performance results of the execution of an ISR2D multiscale application. Numerous tests were performed, differing in terms of the environment setup and the iterations executed.

The test results showed the submission and the execution times in different environments. The results gathering (i.e. sending the output files to S3 buckets) was also measured on the cloud infrastructure.

In Section 9.2, the test results were discussed and interpreted. The execution times were generally slightly shorter on the grid environment. On the other hand, the grid environment performance was significantly more dependent on the instantaneous load of the working nodes.

10 Summary

In this Thesis we proposed a user support tool that facilitates the execution of multiscale applications in distributed environments. Development of the MUST User Support Tool was the major goal of this Thesis. MUST meets the requirements established in Chapter 1. The subsequent Section 10.1 describes how the goals of the MUST User Support Tool were achieved in detail.

The second main goal of this Thesis was a throughout comparison of the grid and the cloud computing environments. All the areas of comparison assumed in Chapter 1 were taken into consideration. Section 10.2 sums up the research results and conclusions of the former Chapters.

10.1 MUST User Support Tool

The main requirements of the MUST tool were (presented originally in Section 1.2): support for distributed execution of multiscale applications, support for both grid and cloud distributed infrastructures and access from the GridSpace virtual laboratory level.

The main requirements of the proposed tool were fulfilled. The MUST tool proposed in this Thesis facilitates the execution of multiscale applications in distributed environments. Key features of the MUST tool are listed below:

- Ability to execute MUSCLE-based multiscale applications in the local cluster environment.
- Ability to execute MUSCLE-based multiscale applications in the Amazon Web Services cloud environment.
- Accessibility from the GridSpace virtual laboratory.
- Ability to send sandbox files to the Amazon Web Services cloud environment which may be used for automatic deployment of MUSCLE-based applications.
- Ability to send output files from the EC2 virtual machine instances to S3 buckets.

To create the above-described tool, we studied multiscale applications, model description languages, coupling libraries and both grid and cloud distributed infrastructures.

In Chapter 2, we discussed multiscale applications in general, their main requirements and some example multiscale problems (e.g. an ISR2D application, used later for the MUST performance testing). Later, various model description languages which may be used for describing both single scale models and multiscale applications as a whole were presented in Chapter 3. Next, coupling libraries used for development of multiscale applications were presented in Chapter 4.

Based on the research and conclusions of Chapters 2—4, MUSCLE environment was chosen as the most suitable middleware tool. The MUST tool supports execution of MUSCLE-based multiscale applications. MUSCLE is an applicable tool for executing multiscale applications, as it operates on the adequate level of abstraction (computing kernels which can be used as single scale models), it supports implementing filters facilitating communication between single scale models. MUSCLE is closely related to MML (Multiscale Modeling Language, described in Section 3.2), the best suited language for describing multiple models and their relationship. The MUSCLE environment was presented in detail in Section 4.3.

In Chapters 5 and 6, we presented the grid and the cloud computing architectures and dealt with various tools used for accessing them. The GridSpace virtual laboratory was described in Section 6.2. Detailed description of the MUST tool and its accessibility from the GridSpace virtual laboratory were presented in the subsequent Chapters, 7 and 8. Layered architecture of both grid and cloud-based versions of the MUST tool were also presented, as well as the implementation details relating to accessing both grid and cloud distributed environments.

10.2 Grid and cloud comparison

Main areas of comparison of both grid and cloud distributed infrastructure were (presented in Section 1.2): theoretical comparison, performance results and ease of access, difficulty of installation and amount of changes required in legacy applications.

The grid and the cloud computing infrastructure were compared in detail in Chapter 5. This Chapter discussed their various aspects, ranging from definitions, business, computing and data models, through virtualization, usability, programming and security models to standarization and typical usages. The grid and the cloud infrastructure architectures were also described and compared.

Chapter 9 presented the performance results based on the execution of the ISR2D multiscale application on both grid and cloud distributed infrastructures using various setups. The tests performed showed that the average

execution time was about 5% longer on the cloud infrastructure. On the other hand, the submission and execution times are much more dependent on instantaneous load of the execution nodes.

Chapters 7 and 8 discussed MUST tool architectures with the usage of both grid and cloud environments. Implementation details relating to accessing both environments were also presented, as well as the possibilities of expansion, including adding support for new environments. Appendix B shows various examplary usages of the MUST tool on both the grid and the cloud infrastructures.

List of Figures

1 The same model shown as a single multi-scale model and decomposed to the set of single-scale models (based on [1]). . . . 19
2 A Scale Separation Map depicting the dependencies between single scale models forming the simulation of In-Stent Restenosis (based on [6, 5]). 21
3 A Scale Separation Map depicting the single scale models used to simulate water flow (based on [1]). 22
4 Range of scales modeled in the fusion multiscale application. . 24
5 SBML file fragment example. 28
6 CellML file fragment example. 29
7 An example Coupling Diagram showing ISR (described in Section 2.3) application (based on [1]). 30
8 CxA file fragment example. 31
9 AMUSE architecture overview (based on [18]). 35
10 MCT usage in Community Climate System Model (CCSM). . 36
11 A MUSCLE environment example with a CxA file fragment describing it. 38
12 Grid and Cloud architectures comparison (based on [26]). . . . 47
13 GridSpace architecture overview (based on [44]). 50
14 Use case diagram. 57
15 Layered architecture overview. 58
16 Grid architecture overview. 59
17 Course of action on grid infrastructure. 60
18 Sequence diagram. 62
19 Cloud architecture overview. 63
20 Course of action on cloud infrastructure. 65
21 Sequence diagram. 66
22 MUST class diagram. 70
23 ISR2D computing kernels cooperation. 75

List of Tables

1 Coupling libraries comparison. 39
2 Grid and Cloud comparison (based on [26, 27]) 47
3 ISR2D—minimum, average and maximum execution times and their standard deviations in different setups. 78

References

[1] *Mapper Project*, http://www.mapper-project.eu/

[2] Paweł Pierzchała, *Multiscale applications in the GridSpace virtual laboratory.* Kraków, Poland, in preparation.

[3] Katarzyna Rycerz and Marian Bubak *Building and Running Collaborative Distributed Multiscale Applications*, in: W. Dubitzky, K. Kurowsky, B. Schott (Eds), Chapter 6, Large Scale Computing, J. Wiley and Sons, 2012.

[4] Mission of Coast – complex automata. http://www.complex-automata.org/

[5] Peter M. A. Sloot, Alfons G. Hoekstra. *Multi-scale modelling in computational biomedicine.* Briefings in Bioinformatics, 2010: 142-152

[6] Evans DJW, Lawford PV, Gunn J, et al. *The application of multiscale modelling to the process of development and prevention of stenosis in a stented coronary artery.* PhilTrans R Soc A 2008;366:3343–60

[7] Joris Borgdorff, Carles Bona-Casas, Mariusz Mamonski, Krzysztof Kurowski, Tomasz Piontek, Bartosz Bosak, Katarzyna Rycerz, Eryk Ciepiela, Tomasz Gubala, Daniel Harezlak, Marian Bubak, Eric Lorenz, Alfons G. Hoekstra *A distributed multiscale computation of a tightly coupled model using the Multiscale Modeling Language.* International Conference on Computational Science, ICCS 2012

[8] NucSys, Marie Curie Research Training Program http://www.uku.fi/nucsys/

[9] Litrico X, Fromion V. *Modeling and Control of Hydrosystems.* Springer, 2009.

[10] C. Korner and M. Thies and T. Hofmann and N. Thurey and U. Rude. *Lattice Boltzmann Model for Free Surface Flow for Modeling Foaming.* J. Stat. Phys. 121:179 (2005).

[11] http://www.iter.org/

[12] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, and the rest of the SBML Forum, *The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models*, Bioinformatics, 9(4):524–531, 2003.

[13] A. Cuellar, P. Nielsen, M. Halstead, D. Bullivant, D. Nickerson, W. Hedley, M. Nelson, C. Lloyd, *CellML 1.1 Specification*, http://www.cellml.org/specifications/cellml_1.1, 2003.

[14] J. L. Falcone, B. Chopard, A. Hoekstra *MML: towards a Multiscale Modeling Language*, Procedia Computer Science 00 (2010) 1–8, 2010.

[15] J. Borgdorff, J. L. Falcone, E. Lorenz, C. Bona-Casas, B. Chopard, and A. G. Hoekstra 2011a. *Foundations of Distributed Multiscale Computing: Formalization, Specification, Analysis and Execution.* Journal of Parallel and Distributed Computing, submitted, 1–31.

[16] J. Borgdorff, J. L. Falcone, E. Lorenz, B. Chopard, and A. G. Hoekstra 2011b. *A principled approach to distributed multiscale computing, from formalization to execution.* In Proceedings of the 7th IEEE International Conference on e-Science. IEEE Computer Society Press, Stockholm, Sweden.

[17] A. G. Hoekstra, B. Chopard, P. Lawford, R. Hose, M. Krafczyk, and J. Bernsdorf *Introducing complex automata for modelling multi-scale complex systems. In Proceedings of European Complex Systems Conference. European Complex Systems Society*, Oxford, UK, 2006.

[18] AMUSE project, http://amusecode.org/

[19] Larson, Jacob, Ong *The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models.*, 2005: Int. J. High Perf. Comp. App.,19(3), 277-292.

[20] Jacob, Larson, Ong *MxN Communication and Parallel Interpolation in CCSM3 Using the Model Coupling Toolkit.* 2005: Int. J. High Perf. Comp. App.,19(3), 293-307.

[21] Multiscale Coupling Library and Environment (MUSCLE), http://muscle.berlios.de/

[22] Java Agent DEvelopment Framework (JADE), http://jade.tilab.com/

[23] I. Foster, C. Kesselman *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufman, 1999.

[24] I. Foster *What is the Grid? A Three Point Checklist.*, GRIDToday, July 20, 2002.

[25] Vaidy Sunderam, *Programming Metasystems* http://www.cyf-kr.edu.pl/crossgrid/Seminars-INP/Sunderam-05Sep03.ppt

[26] Ian T. Foster, Yong Zhao, Ioan Raicu, Shiyong Lu. *Cloud Computing and Grid Computing 360-Degree Compared.* CoRR, 2009.

[27] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, Maik Lindner, *A break in the clouds: towards a cloud definition*, ACM SIGCOMM Computer Communication Review, v.39 n.1, January 2009.

[28] *Google App Engine* http://code.google.com/appengine/

[29] *Google Docs* http://docs.google.com/

[30] Jeremy Geelan, *Twenty one experts define cloud computing.* Virtualization, August 2008. Electronic Magazine, http://virtualization.sys-con.com/node/612375

[31] Albeaus Bayucan, Robert L. Henderson, James Patton Jones, Casimir Lesiak, Bhroam Mann, Tom Proett *Portable Batch System External Reference Specification* http://teal.gmu.edu/lucite/manuals/PBSPro5.0/pbs_ers.pdf, 2000

[32] Albeaus Bayucan *Grid-enabled PBS: the PBS-Globus Interface, Globus retreat 2000,* Pittsburgh, PA, August 2000 http://www.pstoolkit.org/Documentation/globus_PBS-abstract.pdf

[33] gLite—Lightweight Middleware for Grid Computing http://glite.cern.ch/

[34] UNICORE (Uniform Interface to Computing Resources) http://www.unicore.eu/

[35] QosCosGrid—Distributed Computing Middleware http://www.qoscosgrid.org/trac/qcg

[36] Swift parallel scripting language http://www.ci.uchicago.edu/swift/main/

[37] The Kepler Project https://kepler-project.org/

[38] Taverna Workflow Management System http://www.taverna.org.uk/

[39] Triana—The Open Source Problem Solving Environment http://www.trianacode.org/

[40] myExperiment Virtual Research Environment http://www.myexperiment.org/

[41] Pegasus Workflow Management System http://pegasus.isi.edu/

[42] Oracle Grid Engine http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

[43] Platform Load Sharing Facility http://www.platform.com/workload-management/high-performance-computing/lp

[44] Eryk Ciepiela, Daniel Harężlak, Joanna Kocot, Tomasz Bartyński, Marek Kasztelnik, Piotr Nowakowski, Tomasz Gubała, Maciej Malawski, Marian Bubak, *Exploratory Programming in the Virtual Laboratory.* http://www.proceedings2010.imcsit.org/pliks/183.pdf, 2010

[45] *Amazon Elastic Compute Cloud* http://aws.amazon.com/ec2/

[46] *Amazon Simple Storage Service* http://aws.amazon.com/s3/

[47] *Amazon Elastic Block Storage* http://aws.amazon.com/ebs/

[48] *Amazon Public Data Sets* http://aws.amazon.com/publicdatasets/

[49] *Amazon Simple Queue Service* http://aws.amazon.com/sqs/

[50] *Right Scale* http://rubyforge.org/projects/rightscale

[51] K. Rycerz, M. Nowak, P. Pierzchala, M. Bubak, E. Ciepiela and D. Harezlak, *Comparision of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations.* In Proceedings of The Seventh IEEE International Conference on e-Science Workshops, Stockholm, Sweden, 5-8 December 2011. IEEE Computer Society, Washington, DC, USA, 81-88 (2011).

A Glossary

AMI—Amazon Machine Image—a snapshot of the operating system which can be started as a virtual machine using the Amazon Elastic Computing Cloud.

AMUSE—Astrophysical Multipurpose Software Environment—a framework for large-scale simulations of stellar systems.

AWS—Amazon Web Services—a platform providing computing resources, storage, messaging services, etc. through the web services interface.

CxA—the Complex Automata—a theory used to model complex multiscale systems based on a set of connected cellular automata and agent-based models. The CxA theory is used by the MUSCLE environment as the base for a file format which describes the connection between computing kernels.

EBS—Elastic Block Storage—Amazon web services-based storage. It provides storage volumes which can be used with running virtual machine instances. Such volumes may contain databases, file systems or raw block level storage.

EC2—Elastic Computing Cloud—the core Amazon web service providing scalable computing capabilities.

GridSpace—a high level framework facilitating the usage of Grid-based resources. It allows development, sharing and execution of virtual experiments.

GridSpace experiment—a set of scripts forming a logical experiment which can be executed using the GridSpace virtual laboratory.

ISR—In-stent restenosis—an example physiological multiscale problem. The ISR application is used for MUST's performance testing.

Kernel—a single scale simulation wrapped into a controller agent, a part of the MUSCLE framework simulation.

MUST—a user support tool proposed in this Thesis. It allows execution of MUSCLE-based multiscale applications in distributed environments.

MCT—Model Coupling Toolkit—a low level library used for building parallel

coupled models.

MML—MultiScale Modeling Language—a concept language which may be used to describe multiscale applications. It is designed to describe both single scale models and whole multi model applications, including the cooperation (data flow) between single scale models.

MUSCLE—Multiscale Coupling Library and Environment—a framework for building and running multiscale applications. MUSCLE allows running simulations based on the CxA theory.

Plumber—a core kernel of the MUSCLE-based multiscale simulation. It coordinates the communication between single scale simulations forming a multiscale application.

S3—Simple Storage Service—Amazon web service providing storage services. The data stored may be public or private and is organized into logical buckets.

S3 bucket—a logical set of data accessible by Amazon S3 web services. Buckets may be created through the web services API. Data (files) may be uploaded to and downloaded from buckets.

SQS—Simple Queue Service—an Amazon web services-based distributed queue messaging service. SQS allows to create, read and delete messages. Messages may be used by the systems created in various technologies, running on various networks and not even running at the same time.

Virtual machine instance—a running virtual machine started by means of Amazon EC2. Virtual machine instances may be controlled using SOAP API and accessed via SSH.

B Examples

This Appendix presents various examples of the usage of MUST. Section B.1 presents a complete exemplary GridSpace experiment. Section B.2 describes a MUST command line usage with examples. A short installation guide is included in Section B.3.

B.1 Example GridSpace experiment

An example of a GridSpace experiment is shown below. The MUST tool is installed as an interpreter in the GridSpace virtual laboratory. The interpreter configuration is included in the *<interpreter>* section of the experiment. The experiment also includes the whole CxA configuration (in the *<code>* section) which is passed as a standard input to the MUST tool.

```
<?xml version="1.0" encoding="UTF-8"?>
<experiment>
  <serializerVersion>0.7</serializerVersion>
  <metadata>
    <name></name>
    <author>plgmarcnowa</author>
    <description></description>
    <manual></manual>
    <creationDate>2011-02-09</creationDate>
    <comments/>
  </metadata>
  <interpreters>
    <interpreter cmd="source /etc/profile; module add intel;
            module add ruby/1.8.7-p249.sl4;
            exec ruby /people/user/pbs_drb/sender.rb"
            interactive="false" name="Muscle" prompt1="" prompt2="">
      <envvar name="GEM_PATH"
            value="/people/user/gems:/usr/lib/ruby/gems/1.8/"/>
      <envvar name="JAVA_HOME" value="/usr/lib/jvm/java-1.6.0"/>
      <envvar name="CLASSPATH"
            value="/people/user/lib/jade/lib/jade.jar:.:
                    /people/user/lib/jsr-275-1.0-beta-2.jar:
                    /people/user/lib/colt-1.0.3.jar:
                    /people/user/lib/xstream-1.2.jar:
                    /people/user/muscle/build/muscle.jar"/>
      <envvar name="PATH"
            value="$PATH:/people/user/gems/bin:/people/user/bin"/>
      <envvar name="RUBYLIB"
            value="/people/user/pbs_drb:/people/user/gems/lib:."/>
```

```
      <envvar name="GEM_HOME" value="/people/user/gems"/>
      <envvar name="MUSCLE_HOME" value="/people/user/muscle"/>
    </interpreter>
  </interpreters>
  <snippet id="1" interpreterName="Muscle">
    <code><![CDATA[
# =========================================
# ISR_CXA_TEMPLATE: template for
# configuration file for a MUSCLE CxA
# A Caiazzo, version 14.05.09
# NOTE: to be used as basic template only.
#       copy and modify for personal tests.
# ==========================================
# configuration file for a MUSCLE CxA
abort "to be used with the MUSCLE bootstrap utility" if __FILE__ == $0

# add build for this cxa to system paths (i.e. CLASSPATH, LD_LIBRARY_PATH)
m = Muscle.LAST
m.add_classpath "/people/user/isr2d/build"
m.add_libpath "/people/user/isr2d/build"

# configure cxa properties
cxa = Cxa.LAST

#
# ic file locations

cxa.env["flow_para_path"] =
        "/people/user/isr2d/kernel/flow3d/param/BF_Re=120_dx=1e-2_150_148/"
cxa.env["ic_stage3_path"] = File.dirname(__FILE__)+"/"
#
# bf parameters
cxa.env["max_timesteps"] = 3110400*2
cxa.env["bf_max_iter"] = 100
cxa.env["coarsest_dt"] = 1
cxa.env["finest_dt"] = 1
cxa.env["cxa_path"] = File.dirname(__FILE__)
# cxa domain
cxa.env["length[mm]"] = 1.5
cxa.env["lumen_width[mm]"] = 1.0
cxa.env["tunica_width[mm]"] = 0.12
cxa.env["total_width[mm]"] = 1.24
#
# discretization
cxa.env["flowdt[s]"] = 0.0001
cxa.env["flowdx[mm]"] = 0.01
cxa.env["dd_dx[mm]"] = 0.01
```

```
cxa.env["xmin[mm]"] = 0.0
cxa.env["ymin[mm]"] = -0.62
cxa.env["zmin[mm]"] = 0.0
# bounding box
cxa.env["xmin_bb[mm]"] = 0.0
cxa.env["ymin_bb[mm]"] = -0.75
cxa.env["zmin_bb[mm]"] = 0.0
cxa.env["width_bb[mm]"] = 1.5
#
# cell properties
cxa.env["smc_mean_rad[mm]"] = 0.0151934
cxa.env["smc_sigm_rad[mm]"] = 0.000
cxa.env["ec_mean_rad[mm]"] = 0.00426935
cxa.env["ec_sigm_rad[mm]"] = 0.000
#
# strut properties
cxa.env["no_of_struts"] = 4
cxa.env["strut_side[mm]"] = 0.18
# test_hannan -- changing deployment depths
cxa.env["max_deploy[mm]"] = 0.09
#
# physical properties
cxa.env["kin_viscosity[m2/s]"] = 4E-6
cxa.env["U_max[m/s]"] = 0.121
cxa.env["rho0[kg/m3]"] = 1000
#
#
# smc model properties
#
# 1 for hexagonal packing of smcs, 0 for "realistic" distributions
cxa.env["ic_smc_hex"] = 1
# fixed run time of physical solver, internal units
cxa.env["solver_fixed_run_time"] = 2000
# relative hoop stiffness in compressive regime
cxa.env["compressive_hoop_stiffness"] = 1.0
# width of png image produced
cxa.env["png_width[px]"] = 800
# strut representation: boundary_element or obstacle
cxa.env["strut_rep"] = "obstacle"
# biology
# threshold (hoop) strain for smc apoptosis/necrosis
cxa.env["smc_max_strain"] = 1.0
# threshold stress for smc apoptosis/necrosis
cxa.env["smc_max_stress"] = 1.0
# threshold (longitudinal) strain for iel breaking
cxa.env["iel_max_l_strain"] = 0.05
# threshold (hoop) strain for iel breaking
```

```
cxa.env["iel_max_h_strain"] = 0.05
# threshold stress for iel breaking
cxa.env["iel_max_stress"] = 1.0
# drug concentration threshold for smc proliferation
cxa.env["smc_drug_conc_thresholdL"] = 1.0
cxa.env["smc_drug_conc_thresholdH"] = 1.0
cxa.env["smc_OSI_thresholdL"] = 0.45
cxa.env["smc_OSI_thresholdH"] = 0.5
# wss_max threshold for smc proliferation
cxa.env["smc_wss_thresholdL"] = 2.76
cxa.env["smc_wss_thresholdH"] = 3
cxa.env["ci_threshold_count"] = 4
cxa.env["ci_weight_smc"] = 1
cxa.env["ci_weight_iel"] = 3
cxa.env["ci_weight_obstacle"] = 1
cxa.env["ci_range_factor"] = 1.1
cxa.env["adventitia_width[mm]"] = 0.00
# parameters used by equilibration code
# maximal number of iterations without radial force; set to 1
cxa.env["equilibrate_max_iter_1"] = 10
# png plotting interval
cxa.env["equilibrate_png_iter"] = 1
# quantity to plot
cxa.env["equilibrate_png_quantity"] = "stress"
cxa.env["equilibrate_png_min_value"] = 0.0
cxa.env["equilibrate_png_max_value"] = 0.01
# parameters used by stent deployment code
# number of iterations to resolve deployment
cxa.env["deploy_max_iter_1"] = 150
# number of iterations after deployment
cxa.env["deploy_max_iter_2"] = 1
# png plotting interval
cxa.env["deploy_png_iter"] = 1
# quantity to plot
cxa.env["deploy_png_quantity"] = "stress"
cxa.env["deploy_png_min_value"] = 0.0
cxa.env["deploy_png_max_value"] = 0.01


# declare kernels
cxa.add_kernel('ic', 'kernel.ICgenerator.ICgenerator')
cxa.add_kernel('bf2smc', 'cxa.cxa3d.bf2smc.BFPulsBoundary2SMCStress')
cxa.add_kernel('bf', 'kernel.flow3d.FlowTestController')
cxa.add_kernel('smc', 'kernel.smc2d.SMCController')
cxa.add_kernel('dd', 'kernel.DrugDiffusion.DiffusionController')
cxa.add_kernel('dd2smc', 'cxa.cxa3d.dd2smc.DrugDiffusion2SMCController')
cxa.add_kernel('smc2bf', 'cxa.cxa3d.smc2bf.ObsArray2IncrementalLists3D')
```

```
# configure connection scheme
cs = cxa.cs

cs.attach('smc2bf' => 'bf') {
        tie('StaticSolid', 'BFObsExit')
        tie('NewSolid', 'BFincSolidExit')
        tie('NewFluid', 'BFincFluidExit')
}

cs.attach('smc' => 'smc2bf') {
        tie('CellPositionsBF', 'ObsArrayExit',
        Conduit.new(
                "muscle.core.conduit.AutomaticConduit",
                ["cxa.cxa3d.smc2bf.Cell2ObsFilter3D", "timeoffset_1"]))
}

cs.attach('bf' => 'bf2smc') {
        tie('BFCoordEntrance', 'FlowBdNodesExit')
        tie('BFLinkEntrance', 'FlowBdLinksExit')
        tie('BFPressEntrance', 'FlowPressureExit')
        tie('BFShearEntrance', 'FlowStressExit')
        tie('BFOsiEntrance', 'FlowOsiExit')
        tie('BFMaxShearEntrance', 'FlowMaxShearStressExit')
        tie('BFAbsShearEntrance', 'FlowAbsShearStressExit')
}

cs.attach('smc' => 'bf2smc') {
        tie('CellPositionsBF2SMC', 'CellPositionsExit')
}

cs.attach('ic' => 'smc') {
        tie('InitialSMCList')
        tie('InitialECList')
}

cs.attach('bf2smc' => 'smc') {
        tie('CellOSIEntrance', 'CellOSI')
        tie('CellMaxStressEntrance', 'CellMaxStress')
}

cs.attach('dd2smc' => 'smc') {
        tie('DrugConcentrationCells', 'CellDrugConcentration')
}

cs.attach('smc' => 'dd') {
```

```
        tie('CellPositionsDD', 'NodeTypes',
        Conduit.new(
                "muscle.core.conduit.AutomaticConduit",
                ["cxa.cxa3d.smc2dd.Cells2NodesTypeFilter3D",
                "timeoffset_1"]))
}

cs.attach('dd' => 'dd2smc') {
        tie('LatticeCoordinates')
        tie('LatticeConcentration', 'DrugConcentrationLattice')
}

cs.attach('smc' => 'dd2smc') {
        tie('CellPositionsDD2SMC', 'CellPositions')
}

$KERNELS_GROUPS = [["ic","smc"],["bf2smc","bf","smc2bf"],["dd","dd2smc"]]
]]></code>
  </snippet>
</experiment>
```

The experiment launches an in-stent restenosis (Section 2.3) multiscale application in the grid environment. The MUSCLE kernels are launched in three groups in this particular example. The line responsible for grouping the kernels is:

```
$KERNELS_GROUPS = [["ic","smc"],["bf2smc","bf","smc2bf"],["dd","dd2smc"]]
```

A description of the graphical tool which allows kernel grouping and the performance results depending on different kernel groupings may be found in Paweł Pierzchała's Master's Thesis[2].

B.2 Example usage

MUST can be run as a command line tool. These are the arguments which may be passed to the command line version:

- –**mode**—the mode in which the program will be run. It can be either *pbs* or *cloud*.
- –**results**—the path where the results will be stored. This folder will be sent to the S3 bucket when using the *cloud* mode.
- –**sandbox**—the sandbox folder which will be sent to each EC2 instance and uncompressed in the path where the tool will be executed (may only be used in the *cloud* mode).
- –**port**—the port on which TaskManager's DRb server will listen when using the *pbs* mode.

- –**cxa**—the temporary path where CxA will be saved (MUSCLE expects CxA as an input file, but MUST accepts CxA configuration as a standard input to cooperate with the GridSpace virtual laboratory, so the input is saved to a temporary file in order to cooperate with the MUSCLE environment).
- –**endline**—MUST will terminate all running working nodes or EC2 instances after receiving this output line. The argument may be used for testing and debugging purposes.
- –**no-submission**—the PBS job will not be submitted or EC2 instances will not be started. The argument may be used for testing and debugging purposes.

An example usage for the *cloud* mode could be (assuming *conf.cxa.rb* is a valid CxA configuration file):

```
$ruby sender.rb --mode cloud --sandbox Sandbox --results Res < conf.cxa.rb
```

An example usage for the *pbs* mode could be:

```
$ruby sender.rb --mode pbs --results Res --port 12345 < conf.cxa.rb
```

Assuming that we would like to stop the ISR2D application after the 150th iteration for testing or performance measuring purposes, this is the line which could be used:

```
$ruby sender.rb --endline "iter=   150" --mode cloud < conf.cxa.rb
```

B.3 Installation guide

This Section lists the MUST prerequisites and shows exemplary MUST configurations for both *pbs* and *cloud* modes (described in Section B.2). The MUST source code may be downloaded from the online GitHub repository[7].

B.3.1 Prerequisites

The following prerequisites need to be installed in order to start MUST. The versions used for tests are listed in the brackets.

- Ruby (1.8.7).

[7]MUST GitHub repository—`https://github.com/wrozka/mapper_sender`

- rubygems (1.3.5). The following Ruby gems need to be installed:
  - right_aws (2.1.0),
  - json (1.5.1),
  - pbs_gem[8].
- Java (1.6.0).
- EC2 API Tools (1.4.4.1).
- MUSCLE (MUSCLE_2010-01-11_13-51-27)[9].

MUSCLE needs to be accessible with the *muscle* command. An example output of the *muscle –version* command is shown below:

```
$muscle --version
This is the Multiscale Coupling Library and Environment (MUSCLE) from
2010-01-11_13-51-27 running in 64-bit mode, native library available
(built at Oct 23 2010 16:27:31, assertions : active, exceptions : active,
_DEBUG : not defined, DEBUG : not defined, NDEBUG : not defined)
```

B.3.2 Access machine and nodes configuration

The following environmental variables need to be set in order to run MUST.

```
EC2_HOME=$HOME/ec2-api-tools/
PATH=$PATH:$HOME/gems/bin:$HOME/ant/bin:$HOME/bin:$EC2_HOME/bin
JAVA_HOME=/usr/lib/jvm/java-1.6.0
MUSCLE_HOME=$HOME/muscle
GS_PBS_SERVER=batch.grid.cyf-kr.edu.pl

#line broken for clarity
#some of the items in the classpath are required by ISR2D application
CLASSPATH=$CLASSPATH:.:
          $HOME/lib/jade/lib/jade.jar:
          $HOME/lib/jsr-275-1.0-beta-2.jar:
          $HOME/lib/colt-1.0.3.jar:
          $HOME/lib/xstream-1.2.jar:
          $HOME/muscle/build/muscle.jar

ACCESS_KEY_ID={AWS access key ID}
ACCESS_KEY={AWS access key}
AMI_ID=ami-7f4c8f16
```

[8]A gem developed as a part of the GridSpace virtual laboratory.

[9]The latest MUSCLE version may be downloaded from `http://muscle.berlios.de/`

After installing the prerequisites and setting up environmental variables on both access machine and computing nodes, MUST can be run in the *pbs* mode.

B.3.3 EC2 instance configuration

To be able to start the AWS Cloud version of MUST, the following environmental variables must be set in the running instance.

```
JAVA_HOME=/usr/lib/jvm/java-6-openjdk

#line broken for clarity
#line required only by ISR2D application
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
                /opt/intel/lib/intel64/:
                /opt/intel/compilerpro-12.0.2.137/compiler/lib/intel64/

MUSCLE_HOME=$HOME/muscle
PATH=$PATH:$HOME/gems/bin:$HOME/ant/bin:$HOME/bin
PWD=$HOME/gems/bin
CLASSPATH=$CLASSPATH:*
ACCESS_KEY_ID={AWS access key ID}
ACCESS_KEY={AWS access key}
```

The following MUST startup lines need to be included in the startup scripts (e.g. in the */etc/rc.local* file).

```
cd $HOME/mapper_sender >> $HOME/log 2>&1
ruby cloud_slave.rb >> $HOME/log 2>&1
```

The example above will work, assuming that the MUST files have been put in the *$HOME/mapper_sender* directory.

An example AMI image (id *ami-7f4c8f16*) may be used as a starting point for building a customized AMI. The example AMI image has MUSCLE environment, the MUST tool and the ISR2D example application configured.

B.4 Summary

This Appendix showed various examples of the usage of MUST. Section B.1 included a complete GridSpace experiment with comments. The experiment may be used for launching the ISR2D application. Section B.2 showed various examples of command line usage and described the possible command line options. Section B.3 was a short installation guide. It included all the MUST prerequisites and showed exemplary MUST configurations in both grid and cloud environments.

C Publication—Comparison of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations

The Comparison of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations paper (by K. Rycerz and co-authors M. Nowak, P. Pierzchala, M. Bubak, E. Ciepiela and D. Harezlak)[51] was submitted to the Distributed Multiscale Computing 2011 conference[10]. The article was presented by K. Rycerz on the DMC 2011 conference. The whole article is presented below.

[10] http://www.computationalscience.nl/dmc2011/

Comparison of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations

Katarzyna Rycerz*‡, Marcin Nowak*, Paweł Pierzchała*, Marian Bubak *†, Eryk Ciepiela ‡ and Daniel Harężlak ‡
*AGH University of Science and Technology
Instutute of Computer Science
al. Mickiewicza 30,30-059 Krakow, Poland
Email: kzajac|bubak@agh.edu.pl and marcin.k.nowak|pawelpierzchala@gmail.com
† University of Amsterdam,
Institute for Informatics, Science Park 904,
1098XH Amsterdam, The Netherlands.
‡ AGH University of Science and Technology
Academic Computer Centre – CYFRONET,
Nawojki 11,30-950 Kraków, Poland
Email: e.ciepiela|d.harezlak@cyfronet.krakow.pl

***Abstract*—In this paper we present and compare a support for setting up and execution of multiscale applications in the two types of infrastructures: local HPC cluster and Amazon AWS cloud solutions. We focus on applications based on the MUSCLE framework, where distributed single scale modules running concurrently form one multiscale application. We also integrate presented solution with GridSpace virtual laboratory that enables users to develop and execute virtual experiments on the underlying computational and storage resources through its website based interface. Last but not least, we present a design of a user friendly visual tool supporting application distribution.**

I. Introduction

Multiscale modeling is one of the most significant challenges which science faces today. There is a lot of ongoing research in supporting composition of multiscale simulations from single scale models on various levels: from high level description languages [1], through dedicated environments [2], [3], [4] to the efforts of exploiting European Grid e-Infrastructures such as Euforia [5], MAPPER[1] or the UrbanFlood.eu [6] projects.

In this paper we present a support for programming and execution of MUSCLE-based multiscale applications in the variety types of infrastructures – namely we comparatively evaluate the performance of local HPC cluster approach with cloud-based solutions (i.e. performance of their resource management mechanism). Recently, there is a lot of ongoing effort in fulfilling high performance computational requirements on cloud resources, which general advantage over classical clusters is ad-hoc provisioning (instead of using long queues in batch queue systems) and pay-as-you-go pricing (instead of large investment in dedicated, purpose-built hardware). The goal of this work was to use some of these solutions to build a support for multiscale applications and compare it with a classical HPC. There are some other affords in this direction - such as VPH-Share[2] project that aims at using clouds for multiscale simulations from Virtual Physiological Human research area. However, up to our best knowledge, there is no any particular mature solution yet.

We investigate and integrate solutions from: virtual experiment frameworks, such as the GridSpace Platform [7][3], tools supporting multiscale computing such as MUSCLE [2][4] and Cloud and HPC infrastructures. Additionally, we present a design of a user friendly interface, suitable for scientists working on multiscale problems (computational biologists, physicists) without computer science background.

The paper is organized as follows: In Section II we present background of our work, in Section III we describe characteristic and requirements for chosen MUSCLE-based multiscale applications, in Section IV we outline overall architecture of our environment. In the next two Sections we present details of our solution: in Section V we describe Kernel Graph Editor - a visual tool aiding a user in distribution of application modules (kernels) and in Section V we show details of supporting HPC cluster and Cloud execution. The use case of the example multiscale medical application is presented in Section VII and the preliminary results are shown in Section VIII. Conclusion and future work can be found in Section IX.

II. Background

Multiscale simulations are of the great importance for a complex system modeling. Examples of such simulations include e.g. blood flow simulations (assisting in the treatment of in-stent restenosis) [8], solid tumor models [9], stellar system simulations [4] or virtual reactor [10]. The requirements of such applications are addressed by numerous partial solutions.

MML [1] developed in the MAPPER project is the language for description of multiscale application consisting of different

[1]http://www.mapper-project.eu

[2]http://uva.computationalscience.nl/research/projects/vph-share
[3]http://dice.cyfronet.pl/gridspace/
[4]http://muscle.berlios.de

singe-scale modules. The Model Coupling Toolkit (MCT) [3] is a tool capable of simplifying construction of parallel coupled models that applies a message passing (MPI) style of communication between simulation models and it is oriented towards domain data decomposition. The Astrophysical Multi-Scale Environment (AMUSE) [11] is a software environment for astrophysical applications where different simulation models of stars systems are incorporated into a single framework using scripting approach. The Multiscale Coupling Library and Environment (MUSCLE)[2] provides a software framework for constructing multiscale application from so-called single scale kernels connected by uni-directional conduits. MUSCLE has its roots in the complex automata theory [12], but can be used for multiscale applications in general. Because it is already designed for distributed execution of kernels, it has been chosen as a basic tool for applications supported in our solution. In the future, we also plan to extend our solution also to other, not MUSCLE-based, applications.

We extended the capabilities of the GridSpace (GS) platform developed as a basis for the Virtual Laboratory in the ViroLab project [5] and currently further developed in MAPPER project. GS is a framework enabling researchers to conduct virtual experiments on Grid-based resources and HPC infrastructures. Additionally, the preliminary experiments using GridSpace with cloud computing have been described in [13]. An experimental environment supporting building and execution of multiscale applications consisting of HLA-based components in the Grid environment can be found in [14].

In this paper we present the further results towards building a support for multiscale simulations running on Clouds in a GridSpace environment.

III. Supported Multiscale Applications Characteristic and Requirements

Multiscale applications implement models of multiscale processes [15], [16]. We focus on such multiscale applications that can be described as a set of connected single scale modules i.e. modules that implement models of single scale processes. Therefore, a typical multiscale application consists of:

- software modules simulating certain phenomena in certain time or space scale (scaleful); usually this modules are computationally intensive, could require HPC resources, often (but not always) are implemented as parallel programs,
- software modules that convert data from one scaleful module to another; usually these modules do not have demanding computational requirements; however, to avoid additional communication, they often required to be executed "close" to the scaleful modules they are connecting; they can even be implemented in the same process as one of the scaleful modules.

In this paper we focus on peer to peer type of computation where all application modules are executed concurrently

[5]http://www.virolab.org

CxA file
```
cs.attach(kernel1 => kernel2) {
  tie(entrance, exit)
}
```

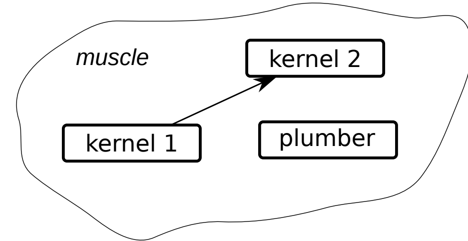


Fig. 1: Example MUSCLE application. Entrance of the kernel 1 is connected with exit of kernel 2.

and exchange data in usually asynchronous fashion; example is part of MAPPER In-stent Restenosis application [8], Canals [17] and Fusion [5] applications; during the course of execution, applications often pass many synchronization points (the number can be static or dynamic); therefore, this type often requires mechanism of efficient communication.

As a supporting communication environment we have chosen MUSCLE communication library that connects tightly coupled simulation modules (MUSCLE kernels). The library allows to concurrently run all modules of the simulation that communicate directly using message passing paradigm. MUSCLE API is specifically designed for Complex Automata (CxA) simulation model and allows a user to specify connection ports (called Exits and Entrances). The MUSCLE communication is based on actor-based concurrency model i.e. asynchronous sending, synchronous receiving. Exits and Entrances are connected using external configuration mechanism (implemented as ruby script called CxA file) for specifying connections between modules and their parameters. The example architecture of MUSCLE application is shown in the Fig.1. The two kernels are executed in Muscle environment and are managed by so-called plumber that assures that all kernels are properly started and joined.

Current MUSCLE implementation is using Java Agent DEvelopment Framework (JADE) framework[6] Kernel communication is performed at JADE agents level, which uses JICP protocol based on TCPI/IP.

IV. General Architecture of Proposed Environment

The general architecture of the solution is shown in the Fig.2 In our research we have combined solutions from: the GridSpace Experiment Workbench, tools supporting multiscale computing such as MUSCLE and Cloud and HPC infrastructures. The GridSpace Experiment Workbench is a Web 2.0-based tool supporting joint development and execution of virtual experiments by groups of collaborating

[6]http://jade.tilab.com/

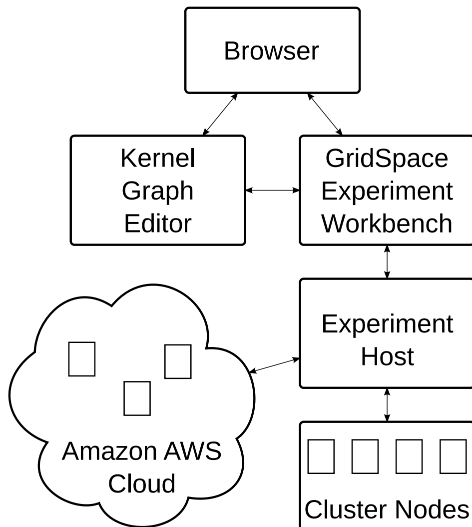


Fig. 2: General Architecture of Proposed Environment

scientists. GridSpace experiments consist of scripts which can be expressed in a number of popular languages, including Ruby, Python and Perl as well as domain specific languages. The framework supplies a repository of gems enabling scripts to interface various resources (e.g external Web Applications, local resource management queues, etc.)

For the purpose of this paper, we have extended GridSpace Environment by adding support for experiments consisting also of CxA connection specification. This was done by designing and implementing additional set of CxA interpreters (different for each infrastructure), launched from GS Workbench. We have also built graphical editor showing connections between MUSCLE kernels and supporting grouping them for execution purposes (called Kernel Graph Editor) that is accessible from GridSpace as external Web Application. After the connections between MUSCLE kernels are specified, the actual application is executed on a chosen infrastructure (local HPC cluster or AWS Amazon cloud [7]) dependent on chosen interpreter. Thanks to user friendly design of GS Experiment Workbench switching between interpreters (and therefore choosing the infrastructure) is very easy and does not require any changes from MUSCLE application developer. The support for both types of infrastructures is described in more detail in the next section.

The detailed use case of the proposed environment is as follows:

1) User logs to chosen access machine (called Experiment Host) using GridSpace Experiment Workbench. The actual connection is done using ssh mechanism.
2) User creates or loads (from Experiment Host) CxA connection scheme to the GS Experiment Workbench. The scheme describes how to join MUSCLE kernels (that are available on the Experiment Host as software packages). A Simple example of such scheme is shown in the Fig.3.
3) CxA scheme is parsed and sent to Kernel Graph Editor that displays connections.
4) Gridspace prompts user with the Kernel Graph Editor, which aids the user in joining kernels in groups that should be executed at the same host.
5) Depending on user preference the application is performed on HPC Cluster or AWS Amazon Cloud.

[7]http://aws.amazon.com

V. Kernel Graph Editor

Grouping is needed to achieve good performance by reducing the volume of network communication between computational and converter type kernels (see Section III). Kernel Graph Editor is an external web application that enables graphical modification of application structure.

Once CxA script is created in GS Experiment Workbench, it is parsed and application connection scheme is sent to Kernel Graph Editor. The editor processes the message and renders it inside user web browser using GridSpace gem called Webgui. Once a user decides about final connection scheme and grouping of kernels, the final scheme is sent back to the GridSpace CxA Interpreter for execution. The communication between Kernel Graph Editor and GridSpace CxA interpreter is done using simple POST of HTTP protocol. The application structure and information about kernel groups are described in JavaScript Object Notation (JSON) format. The Kernel Graph Editor server is implemented in Ruby using the Sinatra framework[8]. The client is written in JavaScript with the use of library InfoVis[9].

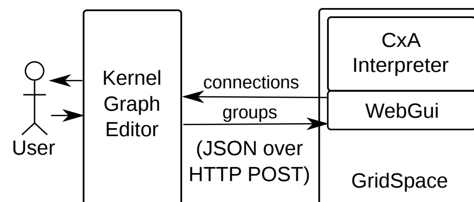


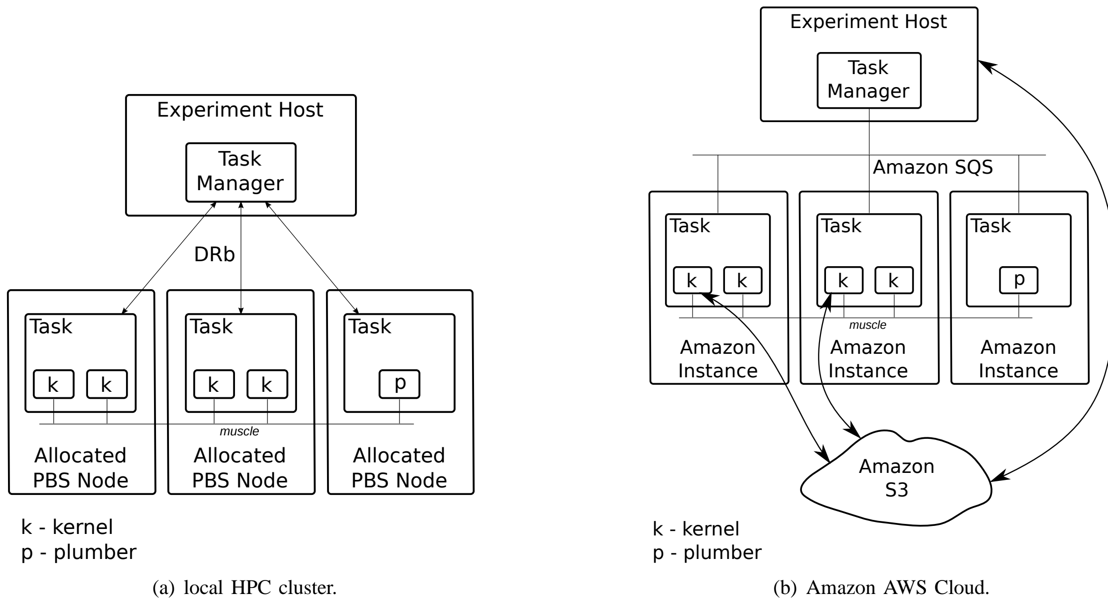
Fig. 3: Loading of CxA connection scheme in the Kernel Graph Editor

VI. Applications Distribution Support in Different Infrastructures

After preparation of MUSCLE application as described in Section IV, it is executed on the chosen infrastructure. As described in Section III, MUSCLE application is a peer to peer type of computation where all kernels are executed concurrently. The execution is controlled by a plumber that once started, registers all kernels, connect them according to the CxA schema and initiates execution.

When using plain legacy MUSCLE software, plumber and groups of computational kernels are started manually on different computing nodes (each kernel group needs information

[8]http://www.sinatrarb.com/
[9]http://thejit.org/



(a) local HPC cluster.

(b) Amazon AWS Cloud.

Fig. 4: Setting up MUSCLE application on various infrastructures

about plumber localization). To automatically control running such an application in a distributed environment we decided to apply a general Master - Slave architecture. Master is responsible for distributing computational tasks (plumber or group of kernels for one node), synchronization and standard output/error gathering. Slaves asks Master for a job to execute and then redirect its standard output and error streams. Once started by Slaves, actual kernels communicate with each other using MUSCLE. This scheme is generally used for both types of infrastructures (local HPC and Cloud) described in this paper. The detailed solutions differ with the chosen technology (used according to actual infrastructure) as described in the next subsections and shown in the Fig.4.

Usually, apart from sending control messages to the output/error streams, scientific applications produce quite a lot of data that are stored in files. This type of output is also treated differently regarding if the computation took place on a host with local or remote file system. The details are described in the next two subsections.

A. Local HPC solution - distributed Ruby and PBS queue system

In case of local HPC solution we have chosen Portable Batch System (PBS) local management system for allocating resources and Distributed Ruby (DRb) for communication between master and slaves. This communication requires a number of short control messages as the actual connection between kernels is done using MUSCLE mechanisms and main simulation output is saved in the files. The detailed architecture of our solution is shown in the Fig.4(a). Master algorithm is as follows:

1) Proper number of nodes is allocated through PBS. This is done as one singe allocation (by using pbsdsh tool).
2) The TaskManager is started.
3) On each of the assigned nodes a Task process (Slave) is started (via pbsdsh tool) that connects to TaskManager using DRb.
4) As asked by a Task, TaskManager sends request to start the plumber
5) As asked by a Task, TaskManager sends requests to start appropriate group of kernels
6) TaskManager prints the received Task's output to the screen.

Slave algorithm is as follows:

1) Task connects to Task Manager using DRb and asks it for a job description
2) Task receives a job description (request for staring a plumber or the kernels in a single group)
3) Task redirects the output and error streams to the Task Manager

In a case of local HPC resources the computational nodes share filesystem with the Experiment Host, so the output files are seen immediately by File Browser which is a standard part of GS Experiment Workbench.

B. Amazon AWS cloud solution

In case of Amazon AWS cloud infrastructure we have used standard mechanisms for launching virtual instances (one instance for one group of kernels). The images used by instances were based on a preconfigured Amazon Machine Image (AMI) with added MUSCLE installation. For communication between Master and Slave we have used Amazon SQS[10] queues. The detailed architecture of our solution is shown in the Fig.4(b). Master (Task Manager) algorithm is as follows:

[10]http://aws.amazon.com/sqs/

1) Amazon SQS queues (one for control messages and one for output and error streams) are created
2) messages to start the plumber and kernels are sent to the control SQS queue.
3) CxA file and Input sandbox with kernels input and implementation are sent to Amazon S3 storage [11]
4) Proper number of virtual instances are started through Amazon EC2 Ruby API. On each virtual machine the Task process is started automatically after the booting.
5) the received (by SQS queues) output and error messages are printed to the screen.

Slave (Task) algorithm is as follows:

1) Tasks fetches the input sandbox from S3 and unpacks it.
2) Task connects to Task Manager using control SQS queue. The first Task fetches job description (request for staring a plumber and group of kernels). The other Tasks wait if there are no messages.
3) The Task that fetched the job description starts the plumber and distributes rest of jobs (name of kernels in each group) to the other Tasks using SQS control queue.
4) Each Task sends the output and error streams to the Task Manager using appropriate SQS queue
5) After the job executes, the files with simulation output are sent to S3.

This scenario assumes that the kernel implementation is lightweight and portable (e.g. in form of simple java jars). If some of the kernels needed more sophisticated dependencies (e.g native libraries), it would require to prepare AMI accordingly before the execution. This process is, however; often more convenient as a user has a full access to virtual instance, in comparison to local cluster resources, when he has to ask administrator for additional installation of packages.

As the computational nodes do not share filesystem with the GS2 user access machine (Experiment Host), the output files have to be fetched from Amazon S3 storage to be seen by GridSpace File Browser.

VII. Use Case - Instent Restenosis

As an example of multiscale application we have used the Instent Restenosis Application (ISR) [8] that simulates treating of recurrent stenosis of artery after surgical correction. We have used 2D version of the simulation. More information about the application can be found in [18]. As shown in Fig. 5 the application consists of three modules of different time scale: simulation of blood flow (BF), simulation of muscle cells (SMC), and drug diffusion (DD). The application includes also scale-less transformation modules connecting ones which feature a scale (scaleful) and initial condition module. All modules are implemented as MUSCLE kernels. The BF, SMC and DD modules are synchronized and perform around 1700 iterations in total (around 70 hours wallclock time). They exchange about 10MB data during each iteration.

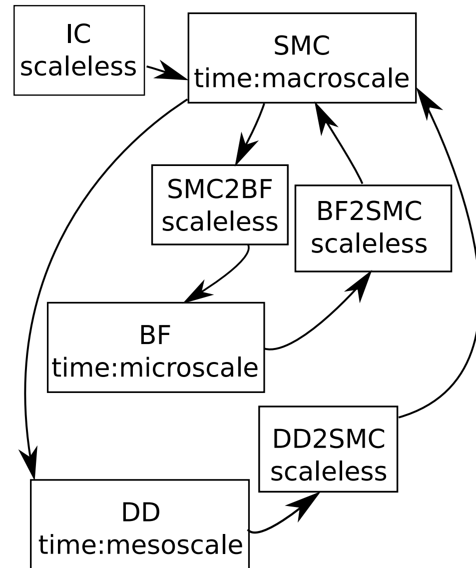


Fig. 5: Simulation of In-stent restenosis - 2D version.

The example screenshot of the kernel graph editor showing connections for in-stent restenosis application is shown in the Fig.6.

VIII. Local HPC vs Cloud - Performance Results

To compare local HPC and Cloud approach we have performed preliminary tests of ISR application. As the total execution time of the application is very long (3 days for 1700 iterations), we present tests for a partial execution (for 15 and 150 number of iterations). The demo of running the application from GridSpace is available online[12].

The local HPC cluster used was the HP Cluster Platform 3000 BL 2x220, connected with Infiniband hosted at ACC Cyfronet, Krakow. The machine is number 81 on the June 2011 Top 500 list. When using Cloud we compared following types of instances (please note that One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor):

- High-CPU Extra Large (m1.xlarge) Instances with 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 1690 GB of local instance storage, 64-bit platform. Benchmark results of network parameters between Amazon instances can be found in [19]
- Cluster Compute Quadruple Extra Large (m2.4xlarge) Instances 23 GB memory, 33.5 EC2 Compute Units, 1690 GB of local instance storage, 64-bit platform, 10 Gigabit Ethernet.

In case of the local HPC cluster the *setting up* phase included waiting in a PBS queue and dispatching tasks using DRb as described in Section VI-A, the *execution* phase

[11] http://aws.amazon.com/s3/

[12] http://www.youtube.com/watch?v=3S9-kljyXIw

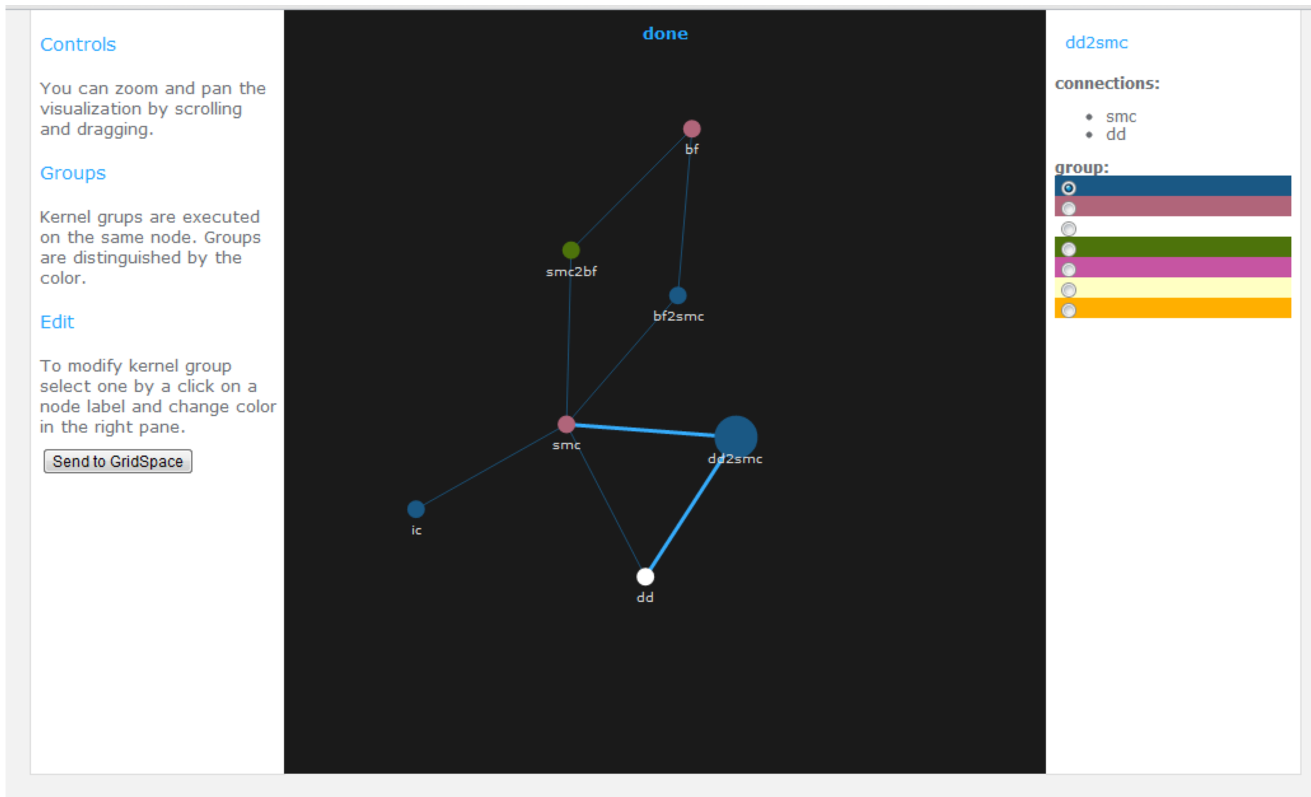


Fig. 6: Screenshot of Kernel Graph Editor for Instent Restenosis 2D application

TABLE I: Setting up and execution times of sample MUSCLE application on Local HPC Cluster and AWS Amazon Cloud - comparison

ISR 2D 15 iterations								
Infrastructure	**Setting up**		**Execution**		**Sending Output**		**Total**	
	min - max (sec)		avg (sec)	σ			min - max (sec)	
Local HPC Cluster	6 - 363		190	16	N/A		196 - 553	
	avg(sec)	σ	avg (sec)	σ	avg(sec)	σ	avg(sec)	σ
AWS Cloud								
m1.xlarge	81	6	250	20	100	10	430	20
m2.4xlarge	80	10	187	3	130	20	400	20
ISR 2D 150 iterations								
Infrastructure	**Setting up**		**Execution**		**Sending Output**		**Total**	
	min - max (sec)		Avr (sec)	σ			min - max (sec)	
Local HPC Cluster	6 - 363		1500	130	N/A		1506 - 1863	
	avg(sec)	σ	avg (sec)	σ	avg(sec)	σ	avg(sec)	σ
AWS Cloud								
m1.xlarge	72	4	2068	15	120	20	2260	30
m2.4xlarge	74	4	1526	4	110	60	1710	60

included actual application execution (including MUSCLE environment start-up).

In case of the AWS Cloud *setting up* phase included all steps required to set up the application as described in Section VI-B: creation of SQS queues, sending input sandbox to S3, booting instances, dispatching Tasks by SQS queue and fetching input sandbox by Tasks. Additionally, we separately included time of sending output to S3 for permanent storage (this step is not necessary in local cluster case with the shared filesystem). The amount of output is around 1MB (for 15 iterations) and 3MB (for 150 iterations).

As mentioned before, in ISR application the amount of communication between legacy MUSCLE kernels was 10MB per iteration. During actual execution the amount of communication between TaskManager and Tasks is much lower and includes transferring single lines of diagnostic output of order of kilobytes (information about iteration number etc.)

As can be seen in the Tab.I, the results show that using

cloud resources is more predictable. For most of results, we show average (avg) from 10 application runs and σ indicates standard deviation. PBS queue waiting time depended on frequency of scheduler execution and of a number of waiting jobs of other users and vary significantly from one run to the other, therefore we present only maximum and minimum values. The time of setting the application up on the cloud is more stable and is much lower that actual application execution. The the application execution time is comparable on both infrastructures, especially when using Quadruple Extra Large instances dedicated for HPC applications.

IX. Summary and Future Work

In this paper we presented and compared the two approaches for using computing infrastructure for MUSCLE-based multiscale applications: local HPC cluster and Amazon AWS Cloud. Both types of infrastructures were integrated with GridSpace Experiment Workbench. Additionally, we have introduced visual Kernel Graph Editor for setting up connection scheme of multiscale application based on MUSCLE and CxA approach.

The preliminary results have shown that setting up multiscale application in a Cloud environment is comparable to its submission on a classical PBS-based HPC cluster. The detailed comparison was summarized in Tab.II.

TABLE II: Local HPC Cluster and AWS Amazon Cloud - comparison summary

Local HPC	**Cloud**
requires hardware investment	pay for what you use
shared and persistent file system	file system is not persistent, requires additional time to stage data in and out from/to external storage (e.g. S3)
often requires contact with administrator for additional installation of packages	a user has administrative access to a virtual instance
variable PBS waiting time depending on number of other users' jobs	constant and predictable virtual instances booting time
using DRb requires setting up point to point socket connections, hosts and ports have to be explicitly known	using SQS more convenient: high level API to shared message queue, communicating entities are not visible to each other.

In a future we plan to perform more sophisticated tests with different number of kernels and different type of Amazon instances, we also plan to test the presented solution on other cloud stacks (e.g. Eucalyptus on FutureGrid resources [13]). Additionally, we plan to build a multiscale application skeleton framework for creating various parametrized MUSCLE application skeletons for further testing.

Acknowledgment

The authors wish to thank Alfons Hoekstra, Joris Borgdorff and Eric Lorenz from UvA for discussions on ISR2D, CxA and MUSCLE and our colleagues from DICE team for input concerning GridSpace, especially Maciej Malawski and Jan Meizner for discussions about cloud computing. The research presented in this paper was partially supported by MAPPER project – grant agreement no 261507, 7FP UE and the AGH grant 15.11.120.090 Access to the Amazon EC2 cloud was supported by an AWS in Education grant.

[13]https://portal.futuregrid.org/

References

[1] J.-L. Falcone, B. Chopard, and A. Hoekstra, "MML: towards a Multiscale Modeling Language," *Procedia Computer Science*, vol. 1, no. 1, pp. 819 – 826, 2010, ICCS 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050910000906

[2] J. Hegewald, M. Krafczyk, J. Tölke *et al.*, "An Agent-Based Coupling Platform for Complex Automata," in *ICCS '08: Proceedings of the 8th International Conference on Computational Science, Part II*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 227–233.

[3] J. Larson, R. Jacob, and E. Ong, "The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 3, pp. 277–292, 2005.

[4] S. Portegies Zwart, S. Mcmillan, S. Harfst *et al.*, "A Multiphysics and Multiscale Software Environment for Modeling Astrophysical Systems," *New Astronomy*, vol. 14, no. 4, pp. 369–378, May 2009.

[5] B. Guillerminet, I. C. Plasencia, M. Haefele *et al.*, "High Performance Computing tools for the Integrated Tokamak Modelling project," *Fusion Engineering and Design*, vol. 85, no. 3-4, pp. 388 – 393, 2010, Proceedings of the 7th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0920379610000049

[6] B. Balis, M. Kasztelnik, M. Bubak *et al.*, "The urbanflood common information space for early warning systems," *Procedia CS*, vol. 4, pp. 96–105, 2011.

[7] E. Ciepiela, D. Harezlak, J. Kocot *et al.*, "Exploratory programming in the virtual laboratory," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, Wisla, Poland, 2010, pp. 621–628.

[8] A. Caiazzo, D. Evans, J.-L. Falcone *et al.*, "Towards a Complex Automata Multiscale Model of In-Stent Restenosis," in *Computational Science ICCS 2009*, ser. Lecture Notes in Computer Science, G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, Eds. Springer Berlin / Heidelberg, 2009, vol. 5544, pp. 705–714.

[9] S. Hirsch, D. Szczerba, B. Lloyd *et al.*, "A Mechano-Chemical Model of a Solid Tumor for Therapy Outcome Predictions," in *ICCS '09: Proceedings of the 9th International Conference on Computational Science*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 715–724.

[10] V. V. Krzhizhanovskaya, "Simulation of multiphysics multiscale systems, 7th international workshop," *Procedia CS*, vol. 1, no. 1, pp. 603–605, 2010.

[11] S. Portegies Zwart, S. Mcmillan, B. O. Nualláin *et al.*, "A Multiphysics and Multiscale Software Environment for Modeling Astrophysical Systems," in *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part II*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 207–216.

[12] A. Hoekstra, E. Lorenz, J.-L. Falcone *et al.*, "Toward a Complex Automata Formalism for Multi-Scale Modeling," *International Journal for Multiscale Computational Engineering*, vol. 5, no. 6, pp. 491–502, 2007.

[13] M. Malawski, J. Meizner, M. Bubak *et al.*, "Component Approach to Computational Applications on Clouds," *Procedia Computer Science*, vol. 4, pp. 432–441, May 2011. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2011.04.045

[14] K. Rycerz and M. Bubak, "Building and Running Collaborative Distributed Multiscale Applications," in *Large-Scale Computing Techniques for Complex System Simulations Wiley Series on Parallel and Distributed Computing*, W. Dubitzky, K. Kurowski, and B. Schott, Eds. John Wiley & Sons, 2011, vol. 1, ch. 6, pp. 111–130.

[15] A. Hoekstra, J. Kroc, and P. Sloot, Eds., *Simulating Complex Systems by Cellular Automata*, ser. Understanding Complex Systems. Springer, 2010. [Online]. Available: http://springer.com/978-3-642-12202-6

[16] E. Weinan, B. Engquist, X. Li *et al.*, "Heterogeneous Multiscale Methods: A Review," *Communications in Computational Physics*, vol. 2, no. 3, pp. 367–450, Jun. 2007. [Online]. Available: http://www.global-sci.com/openaccess/v2_367.pdf

[17] P. van Thang, B. Chopard, L. Lefèvre *et al.*, "Study of the 1D lattice Boltzmann shallow water equation and its coupling to build a canal network," *Journal of Computational Physics*, vol. 229, no. 19, pp. 7373–7400, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jcp.2010.06.022

[18] E. Lorenz, "Multi-scale simulations with complex automata: in-stent restenosis and suspension flow," Ph.D. dissertation, Universiteit van Amsterdam, November 2010. [Online]. Available: http://dare.uva.nl/en/record/358709

[19] T. Ristenpart, E. Tromer, H. Shacham *et al.*, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687