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

PAWEŁ PIERZCHAŁA

MULTISCALE APPLICATIONS IN THE GRIDSPACE VIRTUAL LABORATORY

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

Paweł Pierzchała

Aplikacje wieloskalowe w wirtualnym laboratorium GridSpace

Opiekun pracy:
dr inż. Katarzyna Rycerz

Kraków 2011/2012

Acknowledgements

It is with immense gratitude that I acknowledge the support and help of my Supervisor Katarzyna Rycerz.

I share the credit of my work with my colleague Marcin Nowak, with whom we have developed the initial version of tool described in this thesis.

I would like to thank Eryk Ciepiela and Daniel Harężlak from GridSpace team, for their invaluable help with integrating G–MUST with GridSpace.

I wish to thank Mirek Woźniak for his patient proofreading of this thesis.

Works presented in this thesis are related to the MAPPPER project which receives funding from the EC's Seventh Framework Programme (FP7/2007-2013) under grant agreement no RI-261507.

Abstract

Multiscale modeling in multiple fields of science allows scientists to create more complex and precise models that weren't possible before. Such models require significant computational power and are often distributed to fully utilize resources of grids and clouds.

Graphical–MUST User Support Tool was developed to address the problem of optimal distribution of multiscale application. In order to successfully distribute an application author's knowledge of hardware requirements is necessary. G–MUST assists users in application distribution.

G-MUST was used to learn the performance characteristic of multiscale in-stent restenosis simulator under distribution. Several configurations were tested in order to find the system bottlenecks.

Chapter 1 gives problem outline and explains the aims and scope of this thesis. In Chapter 2 multiscale problems are introduced and several problems are discussed. Multiscale libraries are discussed and compared. Workflow Managments systems are compared in Chapter 3. The architecture of Graphical–MUST User Support Tool is described in Chapter 4. It is followed by a Chapter 5 on G–MUST implementation details. Chapter 6 containing a case study presents the results of running a multiscale application in various distribution configurations. Last Chapter, i.e. 7 sums up the thesis goals and results.

Appendix A is a glossary of terms used in this thesis. Article [27], which compares cloud and HPC environments using tool described in this thesis, is attached in Appendix B

Contents

List of Figures

List of Tables

1 Introduction

This Chapter elaborates on the aim of this thesis. It presents the problem outline and related works and describes the requirements of the created tool. Finally, it presents the contribution of other authors and gives an overview on the paper organization.

1.1 Problem outline

Mutliscale problems are emerging from multiple disciplines, involving scientists with and without technical background that need to cooperate in an efficient manner to solve problems. Multiscale applications are used to simulate the most complex problems from different fields of science. Solutions are built from various components demanding more computational power every year.

Multiscale problems are recognized and supported by the workflow systems. Those systems allow users to define computations and operations through well defined APIs or GUI interfaces. Created components can be later composed in order to solve more sophisticated problems. Solutions from the complete solvers to the smallest transformations can be easily shared and reused by experts.

Workflow systems provide mechanisms for binding operations to computational infrastructure. Those bindings require different amounts of configuration. More or less complicated tools are available for users to configure execution environments.

From the point of view of this work, the most interesting system with workflow capabilities is the virtual laboratory GridSpace[14]. It is a web based access and execution system for multiple grids and clusters. Users can write and share scripts in various interpreters that can be used to orchestrate experiment flow. As a single point of access service it is a perfect fit for the usage of multiple infrastructures. The web 2.0's nature makes user cooperation extremely convenient and smooth.

As the multiscale applications get more and more profound, their infrastructure requirements grow. It also results in complicated setups, which tend to be done manually. Deployment processes performed by hand are cumbersome and error-prone, thus MUST User Support Tool (MUST)[12] was created, a tool that makes infrastructure configuration simple. It assists the user in both application and infrastructure configuration, and then executes the very application.

Distributed multiscale applications have different hardware requirements, some perform extensive computations on CPU, other require fast communication services. To successfully distribute such application, authors knowledge is necessary. This thesis proposes Graphical–MUST (G–MUST), a tool that helps in distribution configuration.

1.2 Aims and scope

The goals of this thesis are following:

1. Create MUST. Initial version was created as a joint effort with Marcin Nowak. Later, he enhanced MUST with cloud support which he describes in his thesis [12].

2. Create Graphical–MUST (G–MUST) an extension of MUST that assists the user in distribution configuration of a multiscale application. Multiscale application are mostly distributed and may use network or CPU heavily. In order to make its run efficient author's knowledge of application is necessary. We present such tool in Chapter 4.

3. Extend GridSpace. G–MUST needs to be tightly integrated with GridSpace virtual laboratory platform to create an interactive tool for running mutliscale applications. An implemented solution is presented in Chapter 5.1.

4. Examine strategies for running multiscale distributed applications. Performance of multiple configurations of in-stent restnosis application [9] were evaluated to learn the best practices for running multiscale solvers. We present the results in Chapter 6.

5. Comparison of the Workflow systems. As G–MUST extends GridSpace, other workflow systems are compared together to explain why the GridSpace was chosen for extension. This is presented in Chapter 3

1.3 Contribution of other authors

G-MUST was created as a part of MAPPER Project [1]. This project is an initiative funded by European Union to create computational strategies, software and services for distributed multiscale simulations across disciplines, utilizing existing and evolving European e-infrastructure. The project is coordinated by Alfons Hoekstra from University of Amsterdam.

The tool is the effect of works on this thesis and Marcin Nowak's paper [12]. The goal of the former is to aid in examining strategies for running multiscale distributed simulations, while the latter addresses the support of cloud infrastructures. Additionally this thesis contains a survey of workflow systems and results of experiments comparing various distribution strategies, while the second includes an overview of multiscale libraries and results of empirical comparison of cloud and grid environments.

1.4 Thesis overview

This Chapter presents the goals of the thesis and general view on the subject. In Chapter 2 the description of multiscale problems is given. Example multiscale problems and libraries are presented. Next Chapter 3 is an overview of workflow systems. Definitions of relevant terms are introduced. Selected solutions are described and compared. Chapter 4 is a detailed description of Graphical–MUST architecture. Requirements and resulting decisions are discussed. Created tool implementation is described in Chapter 5. Chapter 6 is a performance analysis of In-stent restnosis, an existing application from Mapper-Project. Thesis works are concluded in final Chapter 7.

Appendix A is a glossary of terms used in this thesis. Appendix B is an article that compares grid and cloud environments using G–MUST.

2 Multiscale problems

This Chapter introduces definition of multiscale problems in Section 2.1. Three practical and interdisciplinary problems are discussed in Section 2.2. Selected multiscale libraries are presented in 2.3. Multiscale description languages are covered in Section 2.4.

2.1 Description

The interdisciplinary problems like Climate System Model or In-stent Restenosis require the knowledge from multiple disciplines resulting in the multiscale problems. Multiscale problems are composed from coupled models created by numerous science teams. Each model is working in different spatial and time scale. Models of subproblems are interacting in different scales and inconsistent data formats are possible. Algorithms complexity in multiscale problems is very high, thus heavy computational resources are needed.

An example of multiscale problem is the material modeling. Models used in such simulation are depicted on Fig. 1. Components in multiple scales work and cooperate with solvers in different scales, for example Quantum mechanics model (QM) results are used by a larger scale Atomistic model, allowing different precisions of simulation.

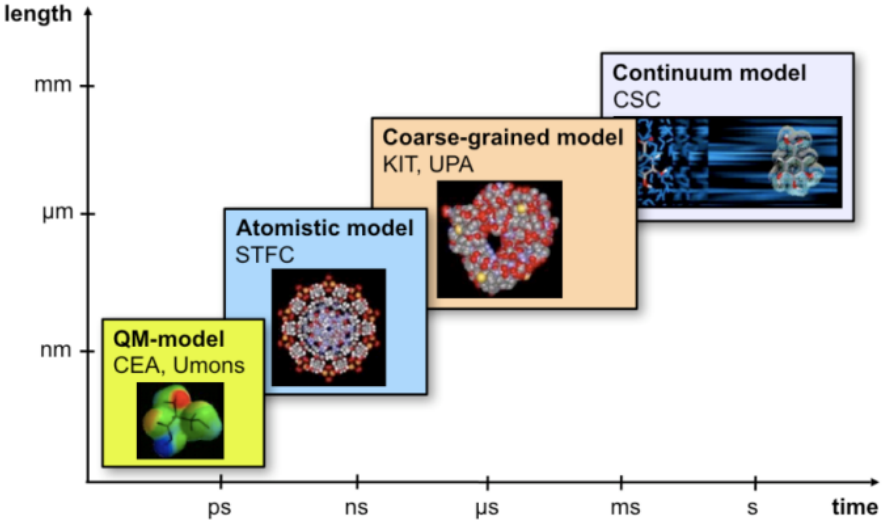


Figure 1: Scale map of Multiscale Material Modelling
https://www.multiscale-modelling.eu/

2.2 Examples

This Chapter presents three examples of multiscale problems. Applications were chosen to cover different disciplines and approaches.

Community Climate System Model [2] is used to model and predict climate changes. Originally developed as Community Climate Model in 1983, is one of the most accurate climate models. In order to predict climate changes, various models of chemical, physical and biological processes have to cooperate. Composition of atmosphere, land surface, ocean and sea ice models results in a climate model.

Complex scientific project like ITER [8] require complex models. Project Fusion was started to address this problem. The goal of Fusion is to cover and simulate many scenarios and aspects of nuclear fusion. The most important simulation is the Transport Turbulence Equilibrium which is a simplified and approximate version of a simulation of the full fusion core in a nuclear fusion reactor.

Another example of a multiscale problem is the in-stent restenosis [9] problem from physiology. The purpose of the model is to study arteries narrowing after stent placement. In-sten restenosis simulates stent deployment and other processes involved. The model is composed from biological and physical models like blood flow, drug diffusion and smooth muscle cell proliferation.

Two dimensional simulation of in-stent restnosis was used for G-MUST case study. Results are presented in Chapter 6.

2.3 Multiscale libraries

Multiscale libraries are written in various languages according to different paradigms meeting different requirements. Following Sections will discuss Amuse, a library for dense stellar systems simulations; MCT, a toolkit for creating parallel coupled models using message passing and MUSCLE, a framework for building applications according to Complex Automata theory.

Multiscale problems are usually too profound to compute them on a single PC in feasible time thus high performance infrastructures are used. Grids, clusters and clouds are used, subproblems are computed on different machines and many types of communication schemes are used, from the low level interfaces like MPI to higher level Agent communication protocols like JADE.

2.3.1 AMUSE

AMUSE is an Astrophysical Multipurpose Software Environment [3]. According to the website, it is a „software framework for large-scale simulations of dense stellar systems, in which existing codes for dynamics, stellar evolution, hydrodynamics and radiative transfer can be easily coupled, and placed in appropriate observational context."

The whole framework is implemented in Python on top of MPI utilities. User scripts are usually written in Python, however it is possible to integrate Amuse with Fortran or C++ code.

2.3.2 MCT

The Model Coupling Toolkit [4], [5] is a parallel coupling library. Problems are decomposed into component models which may use distributed-memory parallelism and occupy the processing elements. The message passing layer is provided by MPEU, Message Passing Environment Utilities is a toolkit created by NASA to support their parallel operational data assimilation system. The Model Coupling Toolkit was created as a part of improvements in Community Climate System Model (CCSM).

The basic MCT package is available as a Fortran library, but Python and C++ bindings are available through Babel interface. Programming model is very similar to MPI one.

2.3.3 MUSCLE

The MUSCLE [6] is a framework for building multiscale applications according to the complex automata theory. The library is a platform independent solution working on top of Java Agent DEvelopment Framework. The core of the library is written using a mix of Java and Ruby code. Developers are allowed to create computational kernels in pure Java or native code like C++, C or Fortran. The name of the library stands for Multiscale Coupling Library and Environment. The Muscle library was created as part of Mapper-Project [1].

Complex Automata (CxA) [7] is a paradigm to model multi-scale systems. The idea behind CxA is to decompose a multi-scale system into several cellular automates (CA) working in different spatial and time scales. Those cellular automates are represented on Scale Separation Map (SSM) which is a two dimensional map where vertical axis is a spatial space and horizontal one is a temporal scale. Every CA is working on its own scales, time steps are known in respect to the global scale. Edges of automates exchange data using information about each other's scale provided by the Scale Separation Map.

2.4 Multiscale description

Multiscale problems' solutions are usually applications decomposed into multiple subproblems, thus there is a need for a formal description of the connections on computations method. Each one of the previously described frameworks has its own description, although there are unifying descriptions like MML [11].

Muscle projects are using cxa description files, which are describing computational modules and connections between them. Current Muscle implementation stores cxa description in plain Ruby files.

The Multiscale Modeling Language (MML) is a language describing the architectures of multiscale applications. Architecture is defined by:

- submodels that are performing actual computations
- filters which know about scales of connected submodels and are responsible for performing necessary conversions.
- mappers are used when more then two submodels are connected. They receive and combine information from all connected subproblems and send it back to proper recipients

Multiscale Modelling Language variant based on XML is known as xMML [10].

More detailed information about multiscale description languages is available in Marcin Nowak's thesis [12].

2.5 Summary

With the still growing computer power more complex problems will be solved. However, many new multiscale problems, even more complicated than the ones previously described, will emerge.

Multiscale problems require complex software solution, as a result frameworks and libraries were created to aid professionals in their work. Packages like MCT were results of works on the Climate System Model, Amuse was created to support large scale simulations of dense stellar systems. General purpose libraries like Muscle are also helping in modeling the multiscale phenomena.

The current version of G-MUST supports applications created in Muscle library. Rationale behind this choice is that Muscle library is not bound to any specific kind of problem.

3 Workflow Management

This Chapter introduces the workflow concept in Section 3.1. Several workflow managements systems are discussed in Section 3.2 and compared in Section 3.3.

3.1 Introduction

Nowadays scientific work involves experiments in-silico, which are getting more popular every year due the increase of the computer power and decrease of its price. However not every scientists have a information technology background needed for development of the software. The problems being researched are getting more and more multidisciplinary, thus being composed from independent modules developed by separate teams.

Workflow management systems make the process of experiment development easier, in some cases by providing graphical tools, in other cases by exposing easy to use high level APIs. The idea behind workflow systems is to broke applications into smaller jobs, that will be easier to develop and maintain independently. Workflow unit is a multi-step computational task, that might perform the actual computations or reformat the data to meet different interface or get the input from an external storage. The experiment is created by ordering and connecting those units together.

Created workflows can be shared between the users, thus minimizing the duplicate work and maximizing the computer resources to human time ratio. Sharable and reusable workflows are bringing the in-silico experiments to non IT scientists like chemists and biologists.

3.2 Workflow systems

Following Section brings the author's pick of the commonly used workflow systems.

3.2.1 Pegasus

Pegasus[21] is a planning framework for mapping abstract workflows for execution on the Grid. Pagasus changes abstract workflows into concrete ones by associating execution resources with each operation in workflow and adding tasks for transferring input and output data. Concrete workflows can be executed using grid middlewares like DAGMan or Condor-G [23]. Pegasus can execute workflows in a number of different environments from desktops to grids and clouds. Installation of this tools might be complex and not every grid user has necessary privileges for that. To address this problem, Pegasus team released Pegasus Portal [24].

Fig. 3 presents an audio analysis workflow implemented in Pegasus. The DART (Distributed Audio Retrieval using Triana) uses Pegasus to run nine hours lasting experiments of music information retrieval. Multiple audio files are processed by a series of sub-alghoritms (depicted as D1 – Dx in Fig. 3) that determine pitch.

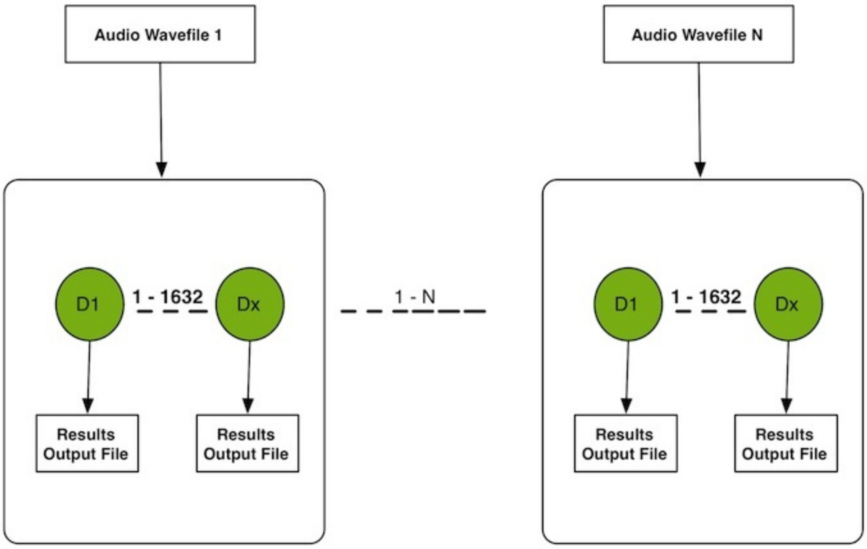


Figure 3: Example of Pegasus Workflow: DART Audio Processing

3.2.2 Kepler

Kepler is an open source workflow management system under lead development of University of California Davis, Santa Barbara and San Diego. It is a mature project witch aims to solve problems in a broad range of scientific end engineering disciplines.

It is a multiplatform solution written in Java based on Pytolemy II [25] that allows users to create workflows using built-in GUI or programming interfaces. Thanks to workflows' independent nature it is possible to write them in tools such as R and MATLAB.

GUI is designed in a similar manner to programming IDEs. It is used to develop, compose and run workflows. Lotka Vloterra simulator implementation is depicted in Fig. 4. It is a pair of differential equations that describe dynamics of biological system in which predator and prey interact. Kepler workflow in Fig. 4 is composed from actors that represent predator and prey equations, integrals and their plotters.

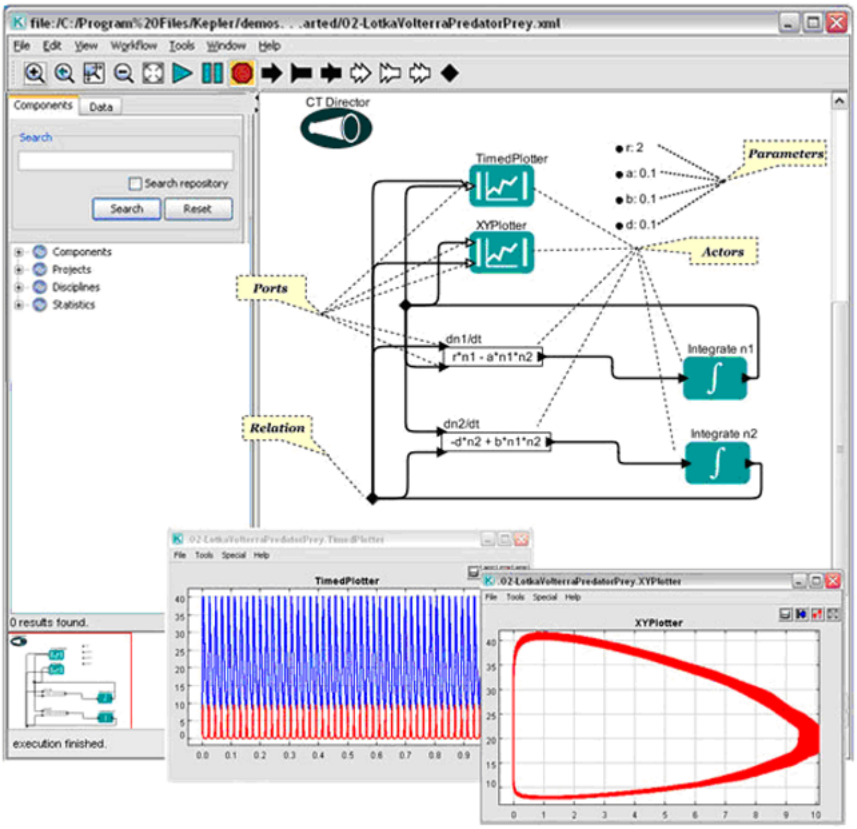


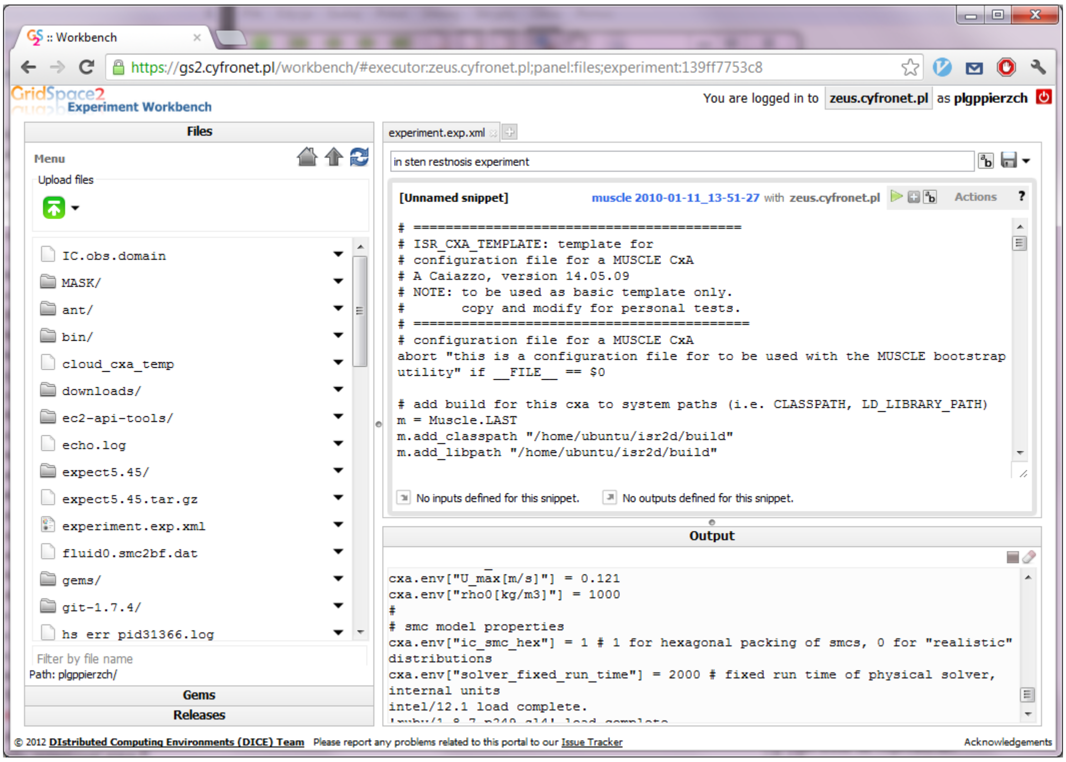
Figure 4: Example of Kepler workflow: Lotka-Volterra

3.2.3 GridSpace

GridSpace [14] is a new and unique web–based computing and data access platform for scientific problems. It is suited to ease up the access to high–throughput and high performance infrastructures. It was originally developed in the ViroLab project [15]; currently it is part of MAPPER Project.

GridSpace user work on experiments which are created in form of scripts called snippets in GridSpace. Experiments may be composed from many snippets written in various languages like Ruby, Python etc. or domain specific tools like G–MUST.

G–MUST extended the GridSpace with the capability of running multi-scale applications. Screen shot in Fig. 5 shows GridSpace running G–MUST while it executes the code in a grid environment.


:: Workbench
https://gs2.cyfronet.pl/workbench/#executor:zeus.cyfronet.pl;panel:files;experiment:139ff7753c8
GridSpace2
Experiment Workbench
You are logged in to zeus.cyfronet.pl as plgppierzch
Files
Menu
Upload files
IC.obs.domain
MASK/
ant/
bin/
cloud_cxa_temp
downloads/
ec2-api-tools/
echo.log
expect5.45/
expect5.45.tar.gz
experiment.exp.xml
fluid0.smc2bf.dat
gems/
git-1.7.4/
hs err pid31366.log
Filter by file name
Path: plgppierzch/
Gems
Releases
experiment.exp.xml
in sten restnosis experiment
[Unnamed snippet]
muscle 2010-01-11_13-51-27 with zeus.cyfronet.pl
Actions
?
No inputs defined for this snippet.
No outputs defined for this snippet.
Output
© 2012 DIstributed Computing Environments (DICE) Team Please report any problems related to this portal to our Issue Tracker
Acknowledgements

```
# ==========================================
# 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 "this is a configuration file for 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 "/home/ubuntu/isr2d/build"
m.add_libpath "/home/ubuntu/isr2d/build"
```

```
cxa.env["U_max[m/s]"] = 0.121
cxa.env["rho0[kg/m3]"] = 1000
#
# smc model properties
cxa.env["ic_smc_hex"] = 1 # 1 for hexagonal packing of smcs, 0 for "realistic"
distributions
cxa.env["solver_fixed_run_time"] = 2000 # fixed run time of physical solver,
internal units
intel/12.1 load complete.
```

Figure 5: G–MUST running a mutliscale application in GridSpace

3.3 Comparsion

All the previously described tools are used to design and execute experiments in-silico. Some of them use more abstract workflow formats like Pegasus and Kepler, other, like GridSpace, use groups of scripts to form an experiment.

Kepler and Pegasus are standalone applications and can execute workflows on local machines or remote resources like grids or clouds. GridSpace is a SaaS application that is used to access grids, although clouds can still be reached from the grid resources.

Pegasus, GridSpace and Kepler are not bound to any specific scientific domain, they are generic purpose workflow management solutions. They offer different interfaces, standalone GUIs or APIs in case of Pegasus and Kepler, web based in case of GridSpace, to solve various heavy computational problems.

Workflow system are compared in table 1. It gathers information about infrastructure characteristic like type of installation or supported environment. Moreover, workflows types and formats are compared and support for multiscale applications is discussed.

Table 1: Workflows comparison

	Pegaus	Kepler	GridSpace
Supported workflows	Formalized XML based.	XML Based.	In form of source code snippets.
Support for multiscale applications	Multiscale applications can be modeled in Pegasus using standard Workflow mechanisms. Focused on DAG workflows.	Kepler supports multiscale via regular workflows. That may expose different types of concurrency according to underlying Pytolemy II framework.	GridSpace supports multiscale application via MUST. MUST executes MUSCLE applications.
Workflows editor	Using XML editor.	Graph based GUI ediotr.	Web based source code editor.
Supported infrastructures	Clouds, Grids, Clusters, Local	Grids, Local	Clouds, Grids, Clusters, Local
Type of installation	Standalone, Web based via Pegasus Portal	Standalone	Web based

GridSpace was chosen for an extension for several reasons. Thanks to its web based nature it does not require users to install any software, which makes it a very convenient tool. Source code based notation of workflow enables wide range of multiscale applications to be imported easily, including the existing ones like in-stent restnosis.

4 Graphical–MUST User Support Tool

This chapter presents G-MUST features in Section 4.1. Next Section 4.2 describes the tool concurrency model. Utility requirements are listed in Section 4.3. Implemented use cases are presented in Section 4.4. Architecture of G–MUST modules is explained in Section 4.5.

4.1 Features

G–MUST User Support Tool runs distributed multiscale application in a grid or cloud environment with minimal configuration. It is an extensions of MUST and the created utility is tightly integrated within GridSpace virtual laboratory which removes repetitive tasks in every day scientific work. Outputs of the multiscale application are displayed in real time in GridSpace, both in the form of output streams and files with results.

To utilize environment resources efficiently, end users can configure the distribution scheme. G–MUST adds a web based tool to MUST, that allows authors to use there insights on application details and precisely group computational task for parallel execution.

Both cloud and grid environments are supported by MUST: it automatically allocates and cleans up all the resources needed by an experiment. The tool seamlessly integrates with grid environments through PBS queue system, where cloud is managed via Amazon EC2 API. MUST support for cloud is described in Marcin Nowak's thesis [12].

Developed tool removes repetitive parts from in-silico experiments. Repetitive updates of application configuration, which required remote machine access, are now matter of configuration update through GridSpace web interface. Results are no longer spread across multiple machines – they are collected during the tool clean up phase.

G–MUST was created with usage simplicity in mind, thus does require only minimal changes within configuration. It is integrated with GridSpace to utilize simple access and user friendly interface, enabling rapid result analysis with built–in statistical packages.

There are multiple multiscale frameworks and cloud providers available; it is impossible to support them all at once, however G–MUST was built to be extensible. At the moment of writing only applications using MUSCLE are supported. Application can be deployed in grids supporting PBS and Amazon cloud.

4.2 Concurrency model

The tool can be described as an actor–oriented system for hosted applications which are a process network [26] in terms of Pytolemy II [25]. Pytolemy is a project that studies modeling, simulation and design of actor oriented concurrent systems. It supports multiple classes of communication process networks (PN), discrete-events (DE), dataflow (SDF), synchronous/reactive(SR), rendezvous-based models, 3-D visualization, and continuous-time models.

The process network model describes a concurrent system in which sequential components communicate through unidirectional FIFO channels. Messages send through channels are called tokens, every process can read from a channel but it cannot poll it for presence of a token, in such case the read operation blocks.

4.3 Requirements

G–MUST is a user support tool, that helps in everyday scientific work, thus it has different set of goals than any multiscale framework or application. The following list presents requirements that shaped the architecture:

- Support tool for heavy applications – The tool itself is a set of programs that host other multiscale applications which need to monitor and collect the outputs of supervised applications.

- Cannot introduce visible overhead once hosted application is started. It has to be efficient in output transportation.

- Runs mutliscale application – has to support multiscale frameworks, in the initial version only MUSCLE is supported.

- Distributed – runs application in both grid and cloud environments.

- Accessible – it has to be a highly available tool following usability standards; the web–based approach makes it accessible to anyone with a browser.

- Configurable – G–MUST needs to provide an easy mechanism for the hosted application's configuration.

- Extensible – Process of adding support for new infrastructures should involve minimal overhead. New multiscale frameworks should be possible to add.

4.4 Use cases

Graphical–MUST extends primary set of MUST use cases by the kernel distribution configuration scenario. G–MUST and MUST use cases relations are depicted in Fig. 6. Following use cases are supported by the application:

- **Configure kernel distribution** – user can configure the distribution scheme using G–MUST's graphical tool.
- **Start experiment** – user can start the configured application in a remote environment.
- **Stop experiment** – application can be stopped at any time.
- **Monitor experiment** – user can monitor output of the hosted application in real time using GridSpace web application.

Described use cases help user in efficient tweaking of optimal distribution. After user starts the experiment, they can monitor the output, to determine if performance is satisfying. If the user wants to try other configuration they can terminate the experiment and quickly reconfigure kernel groupings.

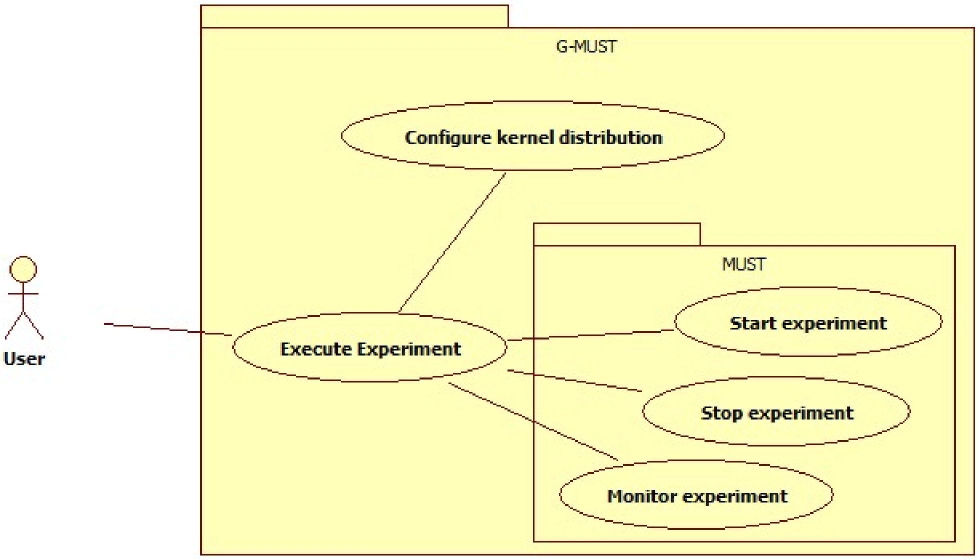


Figure 6: G–MUST Use Cases

4.5 Architecture

To meet the requirement, the application was designed as a set of separated modules, each of which has a particular role in the whole process.

G–MUST can be divided into two layers which are presented in Fig. 7. The Access Layer divided into two modules is responsible for multiscale application configuration. The Grouping module assists the end user in distribution scheme configuration and the GridSpace module allows the end user to edit the hosted application configuration and start the experiment. Execution Layer runs the actual experiment. The Sender module is responsible for communication with the Computational platform and experiment supervision. Modules are described in details in next Sections.

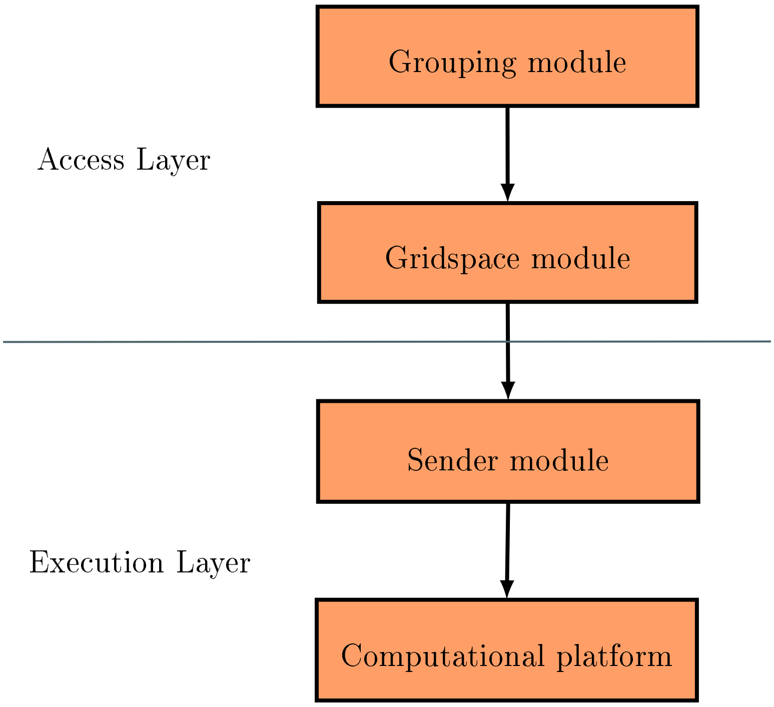


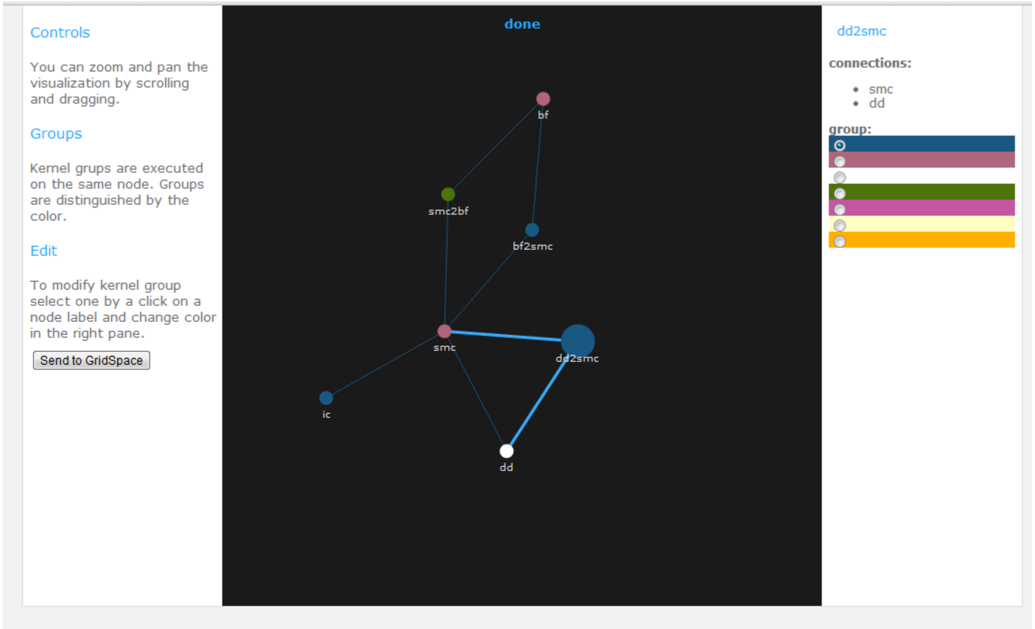
Figure 7: G–MUST tool modules

4.5.1 Grouping module

The Access Layer is divided into two modules. This Section describes Grouping Module. It is G–MUST's key extension of MUST, a web application that visualizes the connection configuration of multiscale kernels. Users can inspect the kernels and group them using the editor for optimal run of application. The visualization application communicates with GridSpace, which then proceeds with execution.

Fig. 8 depicts the in-stent restnosis application kernel configuration with all kernels in separate groups. As it can be seen in the screen shot, kernels are displayed in a graph form, with kernels in the same group rendered with the same color. In order to group tasks, the user has to assign them the same color. When the user is finished with configuration they can submit the result to GridSpace.

Figure 8: Grouping module screenshot visualizing ISR2D kernels



Controls

You can zoom and pan the visualization by scrolling and dragging.

Groups

Kernel grups are executed on the same node. Groups are distinguished by the color.

Edit

To modify kernel group select one by a click on a node label and change color in the right pane.

Send to GridSpace

done

dd2smc

connections:

- smc
- dd

group:

◉ 🟦
○ 🟪
○
○ 🟩
○ 🟪
○ 🟨
○ 🟧

Interactions between GridSpace and Grouping module are presented in the sequence diagram in Fig. 9. Presented sequence consists of following steps:

1. User logs in to GridSpace and starts the experiment.

2. GridSpace returns the URL of Grouping module which is displayed in the browser. It is displayed in a modal in GridSpace; the grouping module receives kernel configuration and session data as a JSON request.

3. User configures the kernels and submits the configuration.

4. Grouping module notifies the GridSpace about configuration changes. Then it sends a post request to GridSpace with configuration in JSON format. Application authenticates with session data received in the initial request.

5. GridSpace continues with the experiment.

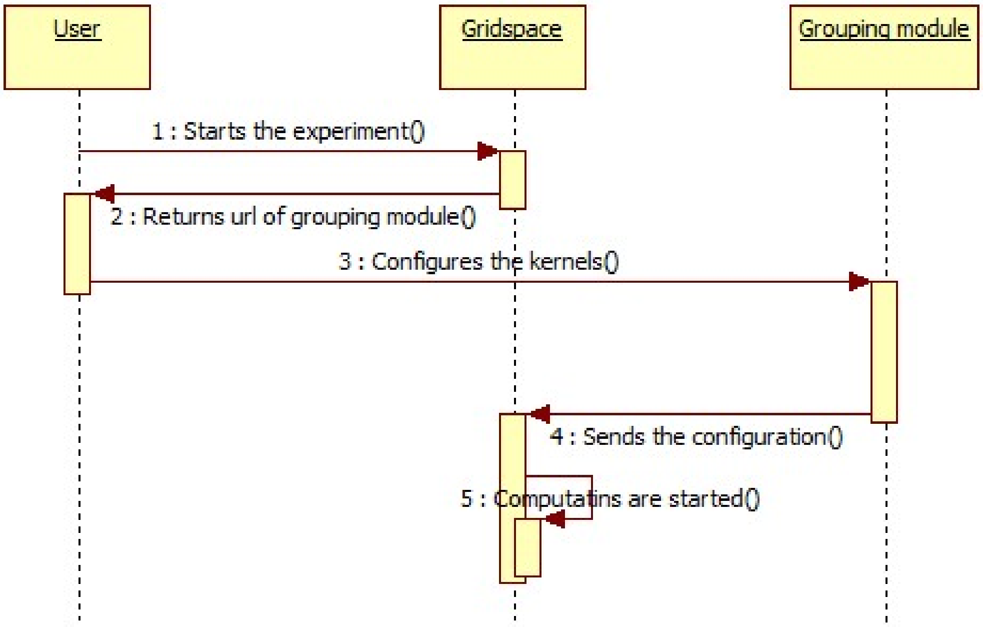

User
Gridspace
Grouping module
1 : Starts the experiment()
2 : Returns url of grouping module()
3 : Configures the kernels()
4 : Sends the configuration()
5 : Computatins are started()

Figure 9: Sequence diagram: Grouping module interactions with GridSpace

4.5.2 GridSpace module

This Section discusses the second Access Layer module, the GridSpace Module. It is the MUST's entry point, after accessing which they can run its multiscale application using a provided interpreter.

GridSpace module is used to configure experiment parameters via configuration files and environment variables. It can start the Grouping module to get additional kernel configuration if needed. It is responsible for sending the start and stop signals to the Sender module. This module can present the Sender live output stream from multiscale application in the GridSpace interface. After the experiment has ended, users can access the application artifacts stored on file system.

4.5.3 Sender module

It is the key module of MUST. Sender communicates and prepares the selected infrastructure and manages the experiment run.

Based on multiscale configuration sent by GridSpace module it allocates computational platform resources. It packages and sends the tool code to allocated machines. It is responsible for the application life cycle – it starts it, monitors the output, and ensures a clean shutdown.

The Sender module follows Master – Slave architecture depicted in Fig. 10. After the Sender allocates resources, through PBS in case of grid, it becomes the Master; allocated machines become Slaves that wait for a list of kernels to execute.

Master is responsible for sending initial kernels configuration to Slaves, output gathering, statistics collection and monitoring the end conditions.

Slaves ask the Master node for a set of kernels to execute. Each slave monitors a single process and sends its output to the Master – after the shutdown signal the monitored process is terminated and slave quits.

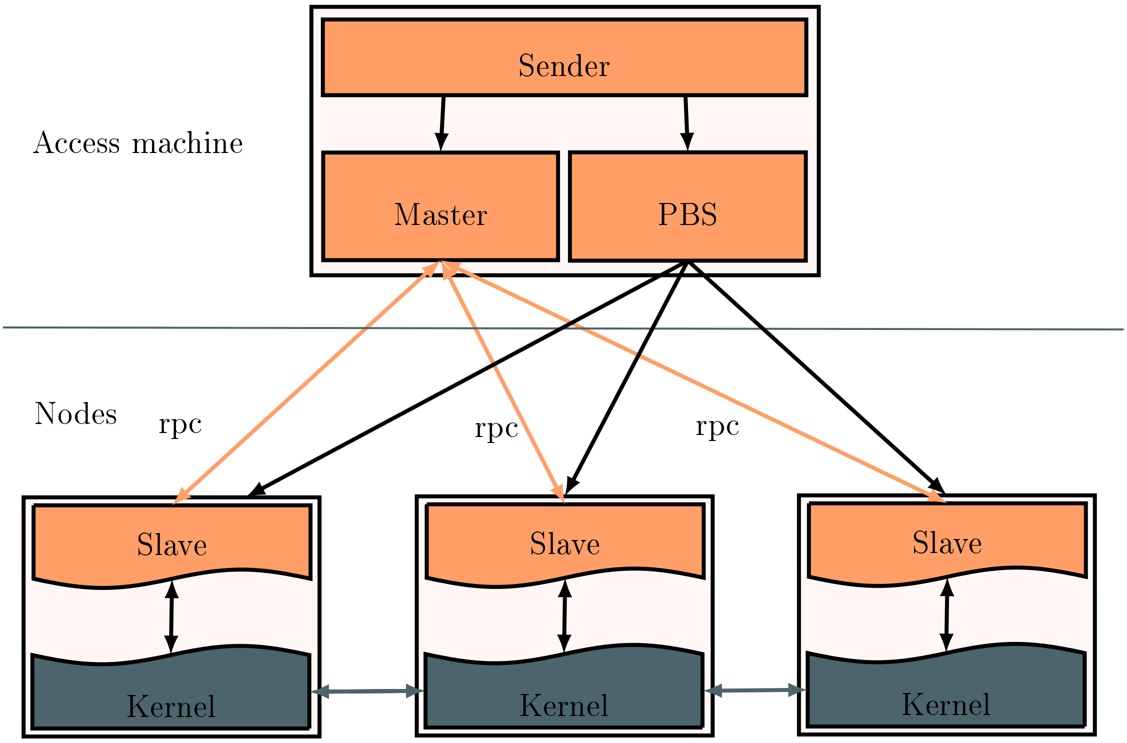


Figure 10: Master – Slave architecture of Sender module

4.5.4 Computational platform

The Computational platform is accessed by the Sender module via a platform-specific communication mechanism. In case of grids it is an adapter for the Portable Batch System. Cloud infrastructure is supported via an adapter on Amazons Web Services client.

4.6 Summary

This Chapter described MUST architecture requirements and the modules that implement it. Design of each application module was discussed.

MUST has a simple two layer architecture. User–facing functionalities are contained in Access Layer. Experiment runner and platform specific code are in Execution Layer. Each layer is divided into modules which follow the single responsibility principle. The most complex module – the Sender, which is responsible for application distribution, is designed as Master – Slave application.

The decision to make the Grouping module a web application and the usage of GridSpace access infrastructure make MUST a easly accessible tool, which after deployment requires only web browser from end users. It provides both text based and GUI based tools for configuration. Separated adapters for different infrastructures make it extensible.

5 Implementation

This Chapter describes the implementation of G–MUST. Each module's implementation details are described in following Sections: Grouping module in 5.1, GridSpace module in 5.2 and Sender module in 5.3. Communication sequence overview is given in Section 5.4. Code structure is the subject of Section 5.5. Possible extensions are discussed in Section 5.6.

MUST is implemented as a set of separate JavaScript and Ruby applications with minor parts written in Bash. Tool modules run on different machines and communicate over network using HTTP based APIs, SSH sessions and distributed Ruby services.

This thesis is concerned with details of grid implementation only, the adapter for cloud infrastructure was a subject of Marcin Nowak's thesis. He implemented and described the cloud module architecture in his work [12].

5.1 Grouping module

Grouping module is a web application that visualizes kernel configurations and enables users to configure the distribution scheme using a simple graph based tool. The purpose of the module is to group the kernels for execution.

The grouping module is a single web page application, frontend is implemented in Javascript and backend code is written in Ruby. The application itself is hosted on heroku cloud: `http://mapper-webgui.heroku.com/`.

Frontend part of application is implemented in JavaScript and the graph manipulation is implemented with the use of InfoVis library [19].

The web server side code is implemented in Ruby using the Sinatra [16] micro framework. It is responsible for serving the frontend code and submitting the result back to GridSpace.

Application is rendered inside user's web browser using the GridSpace webgui mechanism.

5.2 GridSpace module

The GridSpace module is the entry point to the MUST. It is the user facing part of the application, which is a Ruby script that can be run as a GridSpace interpreter. It configures and runs the experiment.

GridSpace module reads the configuration files and allows the end user the edit them using GridSpace interface. It uses the GridSpace Webgui API to show the user a modal with Grouping module. Finally it is used to start the experiment by executing the Sender module.

GridSpace infrastructure allows the user to monitor the mutliscale application outputs in nearly real time. Application artifacts are available immediately after the experiment ends through GridSpace web interface.

The script executes following steps:

1. The configuration is passed by user as a snippet content and saved in a file.

2. The configuration file is parsed and information about computational tasks and their connections is extracted.

3. Extracted information is passed to Grouping module 5.1.

4. When the grouping module sends the connections back, the sender module 5.3 is started.

5.3 Sender module

Sender module is the one responsible for conducting the experiment. It allocates the resources using platform specific mechanism, runs the mutliscale application and monitors its progress. It is an application written in Ruby.

After the Sender program starts working on the access machine it reads the configuration from standard input and saves it in file for the mutliscale application. Current implementation supports only applications written with the MUSCLE library. Based on the configuration it allocates the grid machines using PBS queue system, each allocated machine executes a bash script that starts the slave task.

One type of infrastructure is handled by one computational module. In current release the module for PBS is only available. The module handles allocation of nodes, computational tasks distribution and output redirection. Computational module implements a simple Master–Slave architecture. Master is responsible for distributing computational tasks, synchronization between leader task if necessary and output gathering. Slave asks server for a job to execute and then redirects its output. Both parts are implemented using library Distributed Ruby [17], the library doesn't offer the best performance, but the actual computations are using their own communication facilities and the amount of communication needed to start experiment is limited. The server might wait for a specific task to achieve the desired state, which is checked based on regular expression matched against its output. The Master functionality is implemented in TaskManager class, Slave is implemented in Task class.

Master side module started by GridSpace executes following steps:

1. Proper number of nodes is allocated through PBS [20].

2. The Ruby TaskManager server is started.

3. Each one of the assigned nodes is starting the same bash script which will eventually start a Ruby Task that connects to TaskManager (Slave side).

4. TaskManager executes and prints the received output to the standard output.

Slave side steps:

1. Slave connects to TaskManager and asks Master for a job

2. Received command line is executed on local machine and the output is redirected to Master

5.4 Communication

Fig. 11 shows all modules and the connections outside and inside them. In order to start the tool the user has to log into GridSpace and execute the „muscle interpreter" passing the contents of configuration file to a snippet body.

After the user starts the experiment, GridSpace sends a HTTP request with JSON encoded body to external Grouping module.

Once configuration is submitted back to GridSpace, it starts the Sender module in selected infrastructure using an SSH connection.

The hosted application uses its own communication facilities, in case of MUSCLE it is JADE. Master–Slave orchestrating code communicates using Distributed Ruby Protocol. Slaves monitors kernel output by capturing the kernel standard output and error streams.

5.5 Code structure

The experimental part of this thesis required the author to write small utilities that would share the code base with MUST, the tool code was split into smaller classes. G–MUST is an extension of refactored MUST code base. Shared code was extracted into smaller classes and throughly unit tested. Unit testes were written in RSpec. G-MUST was developed with the use of Guard - a program that monitors files changes and executes configured commands. The source code contains guard configuration that runs tests automatically when the production or test code changes.

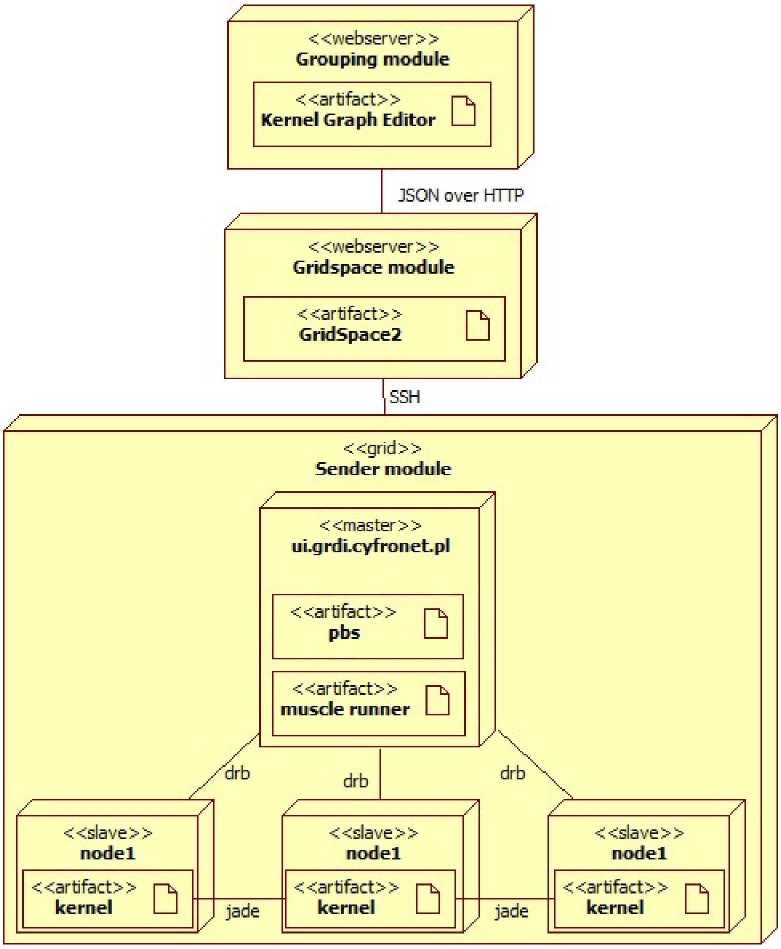


Figure 11: Multiscale tool modules and used protocols

Fig. 12 presents must classes that are used for the grid part. Following classes are used:

- **PBSSender** is the main grid class, it is responsible for infrastructure allocation and start of the TaskManager.
- **Command** wraps common tasks around monitoring of the running program. It starts the command, monitors its output using pipe and can stop the program on given regular expression.
- **TaskManger** is a distributed object that orchestrates the Tasks, sends them jobs to execute and gathers their output.

- **Task** is the slave part of Sender. There is one Task per node in a grid which runs a part of multiscale application and reports its progress to TaskManager.
- **ExecutionTimesLogger** Responsible for logging execution times to given stream.
- **ISROutputParser** Extracts progress information from the in-stent restenosis output.
- **CsvFormatter** Formats the run time statistics in CSV format.

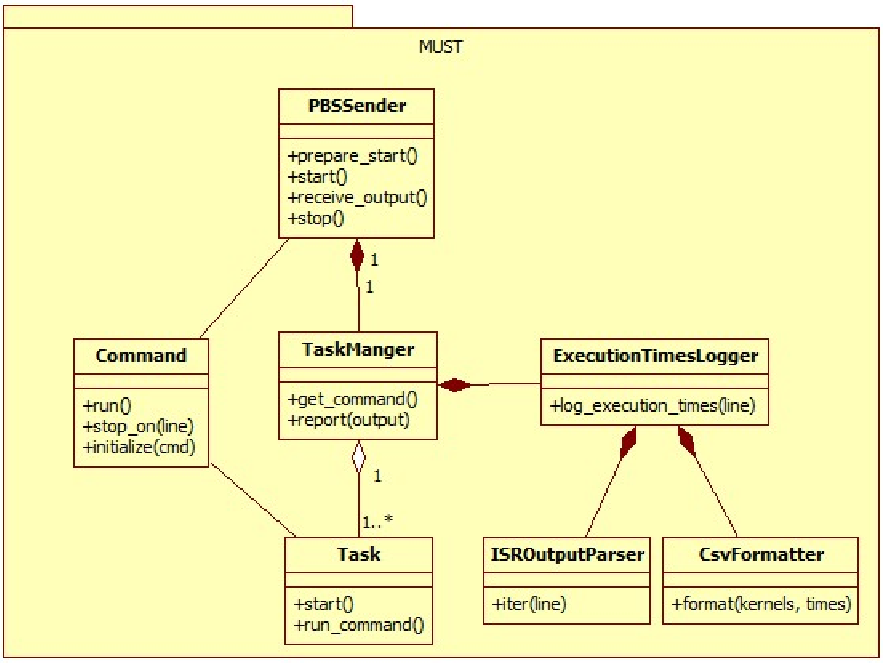


Figure 12: MUST class diagram

5.6 Extensions Possibilities

The logic was separated into several submodules to simplify the process of adding new computational platforms. New platforms have to implement adapter aligning with PBSSender interface. Marcin Nowak's thesis [12] gives a detailed description of the cloud extension.

Complexity of adding support for a new multiscale libraries depends greatly on the library structure. For libraries that use similar application description language, existing code should be fairly easy to adapt, for other, it may be a challenging task.

5.7 Summary

This Chapter presented G-MUST's implementation details. Section 5.1 described the Grouping module's implementation. The GridSpace module was described in Section 5.2. Details of sender implementation are presented in part 5.3. Communication between modules was described in Section 5.4. Detailed information on code structure was presented in Code structure 5.5. Extension possibilities were discussed in Section 5.6.

6 Case study

This Chapter presents the in-stent restenosis application (ISR2D) in Section 6.1 and the rationale behind choosing it as a benchmark application. G–MUST was used to run the application both in grid and cloud environment, results are presented in Section 6.2.

G–MUST is used to learn the in-stent restensosis performance characteristic. Multiscale applications performance will depend on distribution configuration, too many process on one machine with not enough cores will result in CPU contention, on the other hand, too distributed applications may suffer from high network latency.

Successful distribution requires understanding of the application architecture. Created tool provides an useful mechanisms for the user to choose the best configuration. It is also a valuable tool for authors that want to see if the distribution improves overall experiment times.

Measurements and analysis are presented in Section Results.

6.1 In-stent restenosis application

Coronary heart disease is the most common cause of death in Europe. It is treated with stent placement, however sometimes the treatment results in in-stent restenosis, which can cause serious implications, including death.

The In-stent restnosis is briefly described in Section 2.2. ISR2D is multiscale application that models the process of stent restenosis. Multiple modules called kernels operate in various scales and communicate through network. Each of has a different purpose and environment requirements, both in hardware and software.

Application is composed from following kernels:

- **IC** – Initial conditions, reads parameters, calculates boundary conditions and passes them to SMC
- **SMC** – Smooth Muscle Cells simulation, computationally intensive solver written in C++
- **BF** – Blood Flow model implemented using Lattice Boltzmann methods. Heavy computations implemented in Fortran.
- **DD** – Drug Diffusion model written in Java, it is not as computationally intensive as the previous two.
- **dd2scm, smc2bf, bf2smc** – small modules, called connectors, that are responsible for input/output transformation between kernels

Kernel connections are visualized in Fig. 13.

Given that two kernels are computationally intensive and the rest plays a supporting role, machine with three or more cores should be able to efficiently run the application. More complicated version of in-stent restnosis application exists – it is a three dimension model called ISR3D [9]. Three dimensional simulations have more demanding hardware requirements, for example blood flow is simulated using the MPI model, which would require a multi core machine itself. Simpler two dimensional solver was chosen because it was ready at the time the works on this thesis begun.

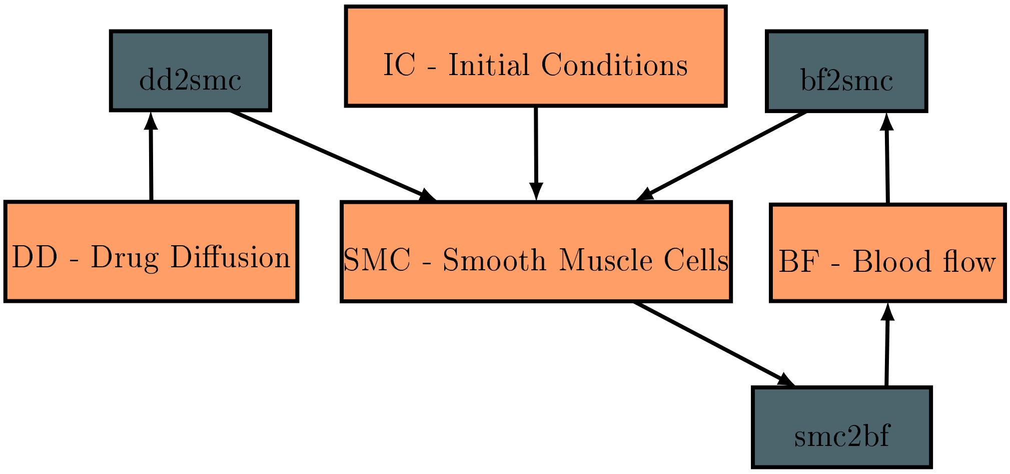


Figure 13: ISR2D Kernels connections

Rationale behind choosing in-stent restnosis as a benchmark application was:

- it is a part of a Mapper-Project
- it is a distributed application with exhaustive computations
- distribution scheme is configurable
- heterogeneous nature of computation makes it a challenging problem

6.2 Results

This Section presents results of ISR2D execution in grid and cloud environments. Grid experiments were performed on PL-Grid machines from ACK Cyfronet AGH, where cloud ones were held in Amazon Elastic Computer Cloud. In grid environment only one type of machine was available, in

EC2 small and large instances were selected to present nontrivial spectrum of runtime scenarios.

The aim of those experiments was to determine the best kernel configuration for every environment. Another important aspect was to prove the usefulness of distribution configuration mechanism.

The same set of kernel configurations was run in each environment and selected type of machines, additional configurations were added in case of EC2 small instances to help understand the performance characteristic in more detail. Following kernels configurations were tested in every environment:

- **[ic,smc,bf2smc,bf,smc2bf,dd,dd2smc]** All the kernels grouped together to show bottlenecks in CPU contention.
- **[ic],[smc],[bf2smc],[bf],[smc2bf],[dd],[dd2smc]** Every kernel running alone to maximize the impact of network throughput and latency.
- **[ic,smc],[bf2smc,bf,smc2bf],[dd,dd2smc]** Blood flow is computationally intensive, it is grouped together with its connectors, to validate network impact.
- **[ic,smc],[bf2smc],[bf],[smc2bf],[dd,dd2smc]** Symmetric case to the previous one, they are both used to measure network impact and the connectors' role in the overall performance characteristic.
- **[ic],[smc,smc2bf],[bf,bf2smc],[dd,dd2smc]** Kernels and connectors together, again to validate connectors' impact.
- **[ic,smc],[smc2bf,bf],[bf2smc,dd],[dd2smc]** Drug Diffusion connector separated to see its impact on performance.

Results are presented in a tabular and graphical form. The tabular data contains kernels configuration, number of iterations performed, execution time average in seconds and standard deviation of time. Each experiment was performed five times. Each table has a corresponding plot of times.

Following Sections cover the results per environment, the last one compares them and summarizes the conclusions.

6.2.1 Grid

Grid execution times are gathered in table 2, Fig. 14 shows the graphical representation.

PL-Grid node Zeus is a powerful computational unit with following spec:

- 24GB of RAM

- 2x Intel Xeon X5650 2,66 Ghz – a 6 core high performance CPU
- InfiniBand network

Due to PL-Grid machines resources differences between particular configurations are not significant. The ISR2D is mostly a CPU bound application with relatively low and synchronized communication between application modules. Given that, there is no considerable difference between running all the kernels on separate machines than on a single one, as long as it has enough cores to take advantage of parallelism. PL-Grid high end network infrastructure and small traffic generated by application take the inter kernel communication out of the picture.

In PL-Grid environment the best choice for running the ISR2D is the configuration with all kernels running one machine, thus the application is isolated from any network issues.

Table 2: Grid results

Kernel groupings	Iterations	Time [s]	δ
[ic,smc],[bf2smc,bf,smc2bf],[dd,dd2smc]	150	1621,67	124
[ic,smc,bf2smc,bf,smc2bf,dd,dd2smc]	150	1629,67	101
[ic,smc],[bf2smc],[bf],[smc2bf],[dd,dd2smc]	150	1646,67	150
[ic],[smc],[bf2smc],[bf],[smc2bf],[dd],[dd2smc]	150	1595,67	111
[ic],[smc,smc2bf],[bf,bf2smc],[dd,dd2smc]	150	1611,00	114
[ic,smc],[smc2bf,bf],[bf2smc,dd],[dd2smc]	150	1644,33	153

6.2.2 Cloud

Cloud experiments were held in the Amazon EC2 infrastructure. It offers multiple instances for different types of computations. From the vast array of standard machines:

- Micro Instance – t1.micro
  - 613 MB memory
  - Up to 2 EC2 Compute Units (for short periodic bursts)
- Small Instance – m1.small
  - 1.7 GB memory
  - 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)

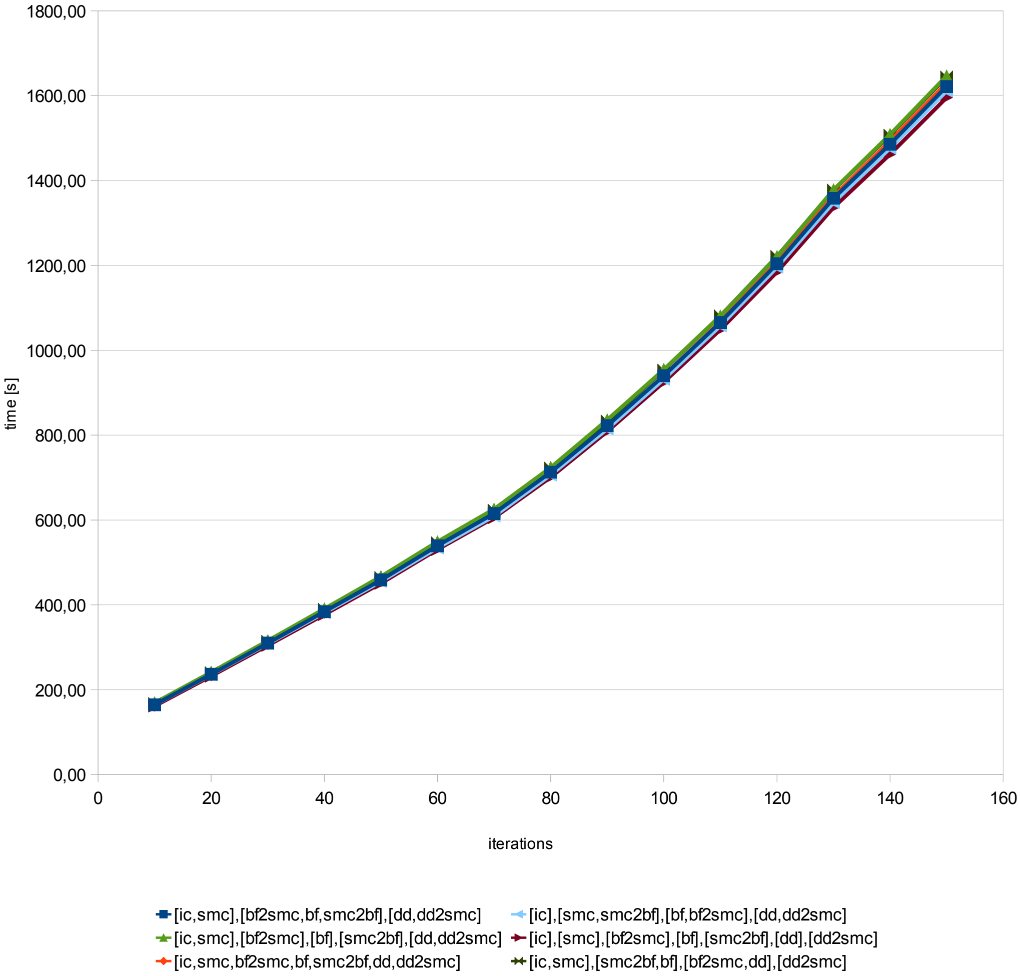


Figure 14: Grid results plot

- Medium Instance – m1.medium
  - 3.75 GB memory
  - 2 EC2 Compute Unit (1 virtual core with 2 EC2 Compute Unit)
- Large Instance – m1.large
  - 7.5 GB memory
  - 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)
- Extra Large Instance – x2.large

  - 15 GB memory
  - 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each)

Two machines were chosen:

- m1.small – because of simple 1 core architecture
- x2.large – for the multicore CPU that can efficiently run ISR2D on one machine

Micro instances were ignored because they are designed for tasks that are stale for the most of the time and in peaks require higher CPU power. Under normal circumstances it operates with CPU power around 0.5EC, under heavy load it can work as 2EC machine. Its performance did not meet the ISR2D requirements, so m1.small instances was chosen, as it provides a constant 1EC of computational power.

For the large instance x2.large was selected, because it offers capacity high enough to run complex problems on one machine.

Following paragraphs present results from two selected instance types.

Small instances

Results of execution on small instances are presented in table 3 and Fig. 15.

Execution times vary from under 4000 to over 18000 seconds, from total distribution to a single machine run. Small instances are not big enough to meet performance requirements of an ISR2D application, while being run in the same instance heavier kernels have to compete for CPU time and result in high execution times. A ingle machine for every kernel resulted in one of the fastest executions because there was no contention for the computational resources between the kernels. The less distributed the application was, the worse the results were.

In addition to baseline configurations, four other kernel configurations were used to measure results on smaller set of machines:

- **[ic,smc,bf2smc,bf],[smc2bf,dd,dd2smc]** Kernels and corresponding connectors are evenly split into two groups to measure impact of the CPU's load.
- **[ic,smc],[bf2smc,bf,smc2bf,dd,dd2smc]** Uneven split chosen as a reference.

- **[ic,smc,bf2smc,bf,smc2bf],[dd,dd2smc]** Another uneven split chosen as a reference.
- **[ic,smc,bf,dd],[smc2bf,bf2smc,dd2smc]** Kernels and connectors are running on separated machines to emphasis to network impact.

The most distributed configuration has one of the best times, what confirms that network efficiency is not a key factor of ISR2D overall performance.

One of the configurations that groups the kernels and corresponding transformations on the same machine would be the optimal one.

Table 3: Cloud results – Small instances

Kernel groupings	Iterations	Time [s]	δ
[ic],[smc],[bf2smc],[bf],[smc2bf],[dd],[dd2smc]	150	4229	174
[ic,smc],[bf2smc],[bf],[smc2bf],[dd,dd2smc]	150	3731,5	639
[ic,smc],[bf2smc,bf,smc2bf],[dd,dd2smc]	150	3797	573
[ic],[smc,smc2bf],[bf,bf2smc],[dd,dd2smc]	150	3909,67	573
[ic,smc,bf2smc,bf,smc2bf,dd,dd2smc]	150	18327	1535
[ic,smc],[smc2bf,bf],[bf2smc,dd],[dd2smc]	150	10076,5	957
[ic,smc,bf2smc,bf],[smc2bf,dd,dd2smc]	150	6820	967
[ic,smc],[bf2smc,bf,smc2bf,dd,dd2smc]	150	14474,5	1457
[ic,smc,bf2smc,bf,smc2bf],[dd,dd2smc]	150	12416	1119
[ic,smc,bf,dd],[smc2bf,bf2smc,dd2smc]	150	15141	1128

Large instances

Tabular data from cloud run on large instances is gathered in table 4. It is plotted in Fig. 16.

Execution times on large instances show only slight differences in run times. It is due to the fact that multicore architecture of x2.large machines allows an efficient run of the whole application on a single machine.

Similar results of distributed and single machine runs confirm that amount of communication between machines is quite small.

The optimal kernel selection for the large instances in EC2 environment is the single machine as it isolates the application from any network issues. It can occur much often than in the HPC Grid environment.

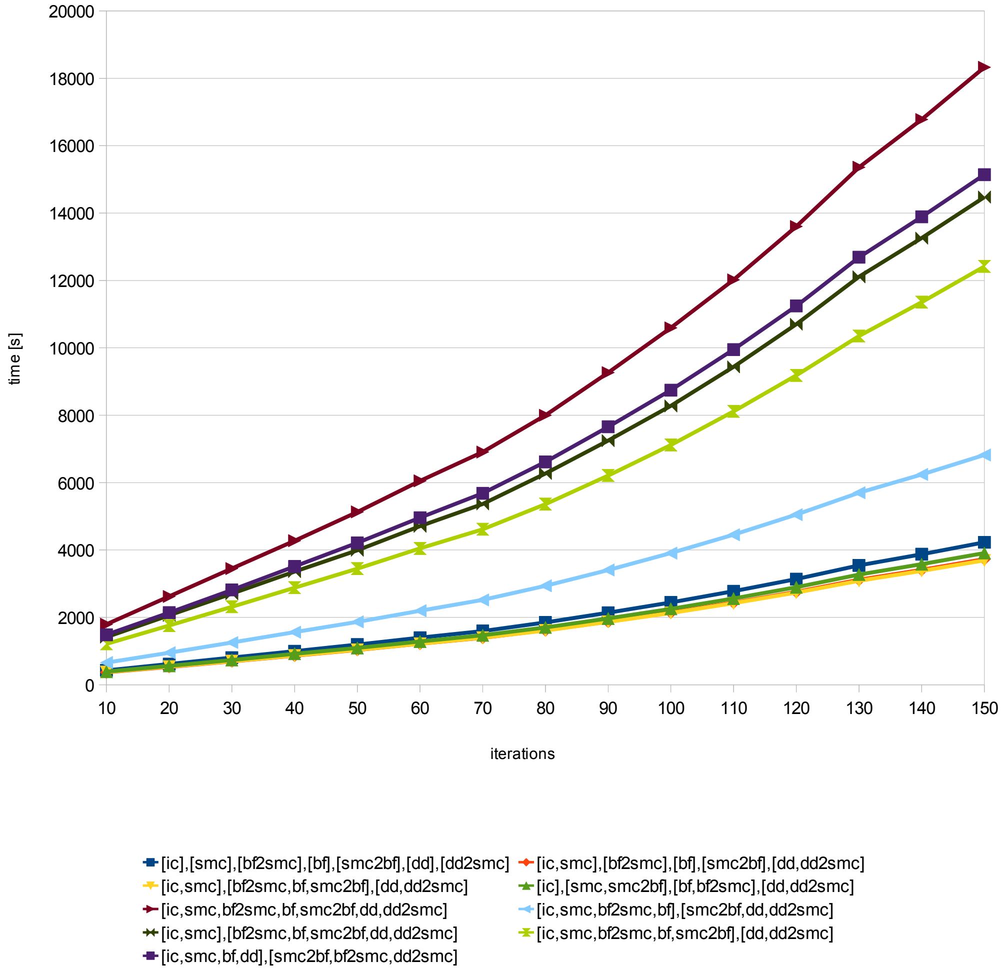


Figure 15: Cloud results plot – Small instances

6.3 Summary

Case study Chapter presented a challenging mutliscale application ISR2D. Application introduction was followed by runtime results analysis of ISR2D executed by G–MUST.

Large cloud instance results are very similar to the grid ones, because both environments offer a high performance environment for multiscale applications. The major difference is the network performance, which in case of EC2 cloud is not a HPC class, although it has no visible influence on the execution times. Again, it is because ISR2D is a CPU, not network IO, bound problem.

Table 4: Cloud results – Large instances

Kernel groupings	Iterations	Time [s]	δ
[ic,smc,bf2smc,bf,smc2bf,dd,dd2smc]	150	2050	81,88
[ic,smc],[smc2bf,bf],[bf2smc,dd],[dd2smc]	150	2043	53,42
[ic],[smc],[bf2smc],[bf],[smc2bf],[dd],[dd2smc]	150	2036	62,57
[ic,smc],[bf2smc],[bf],[smc2bf],[dd,dd2smc]	150	2039	69,40
[ic,smc],[bf2smc,bf,smc2bf],[dd,dd2smc]	150	2041	72,16
[ic],[smc,smc2bf],[bf,bf2smc],[dd,dd2smc]	150	2071	92,02

Experiments on small cloud instances shown that different kernel configurations can have a large impact on application performance.

G–MUST proved to be a valuable tool for learning the performance characteristic of an application and choosing the optimal configuration. Its light weighted configuration and resources allocation automatization allowed the author of this thesis to easily perform the experiments and collect the results.

For the author of mutliscale applications G–MUST offers lightweight way to learn about the application behaviour under distribution. Its interactive nature enables the user to quickly end the inefficient runs, minimizing the time spent on finding the best configuration.

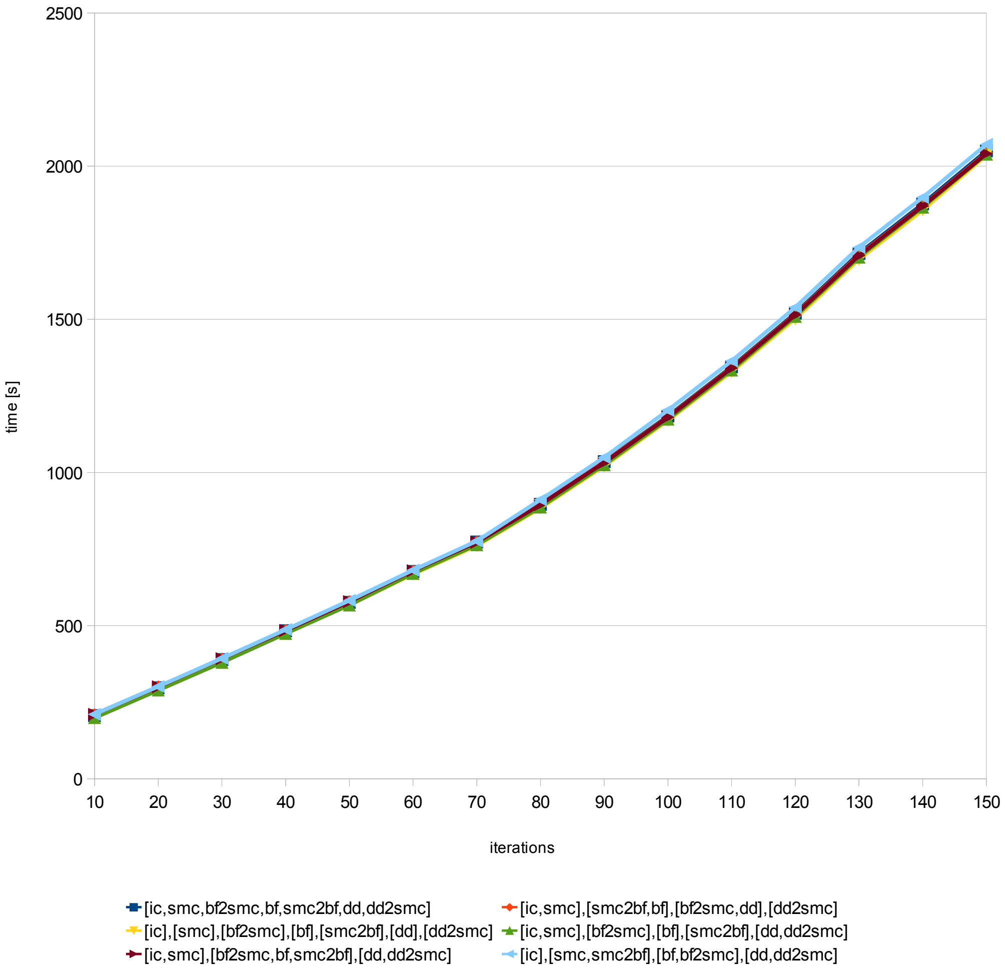


Figure 16: Cloud results plot – Large instances

7 Summary

As a result of the work on this thesis, MUST was created as joint effort with Marcin Nowak. It runs multiscale applications both in grid and cloud environments, relieving the user from error-prone and cumbersome tasks associated with manual execution.

G–MUST, an extension of MUST, was created. It is a tool that assists the user in distribution configuration of a multiscale application. Multiscale application are mostly distributed and may use network or CPU heavily and in order to make its run efficient, author's knowledge of application is necessary.

G–MUST was tightly integrated with the GridSpace virtual laboratory platform to create an interactive configuration tool for distribution scheme of multiscale applications.

G-MUST and MUST are both presented in Chapter 4.

Comparison of two popular workflow systems and GridSpace was presented in Chapter 3. They were compared in terms of their infrastructures support, multiscale problem recognition, workflow types and installation type.

Results from Chapter 6 revealed the nature of performance of two dimensional in-stent restenosis. G-MUST assisted by running multiple distributed configurations of ISR2D in both grid and cloud environments. On HPC grid machines and large instances from cloud the results were very similar no matter how the application was spread across the machines. On the contrary, small instances from cloud resulted in the slowest execution times, while total distribution was one of the fastest configurations and confirmed that ISR2D simulation is CPU-bound.

Case study described in Chapter 6 proved the tool to be useful in distributing the multiscale application, as well in learning the performance characteristic of in-stent restnosis simulation.

References

[1] *MAPPER Project.* http://www.mapper-project.eu/web/guest

[2] *The Community Climate System Model* The National Center for Atmospheric Research http://www.cesm.ucar.edu/

[3] *Amuse* The AMUSE Team, 2009, 2010, 2011, http://www.amusecode.org/

[4] *$M \times N$ Communication and Parallel Interpolation in CCSM3 Using the Model Coupling Toolkit.* Robert Jacob, Jay Larson, Everest Ong, 2005, International Journal of High Performance Computing Applications archive Volume 19 Issue 3, August 2005, Pages 293 - 307 http://www.mcs.anl.gov/research/projects/mct/mxnP1225.pdf

[5] *The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models.* Jay Larson, Robert Jacob, and Everest Ong, 2005, International Journal of High Performance Computing Applications archive Volume 19 Issue 3, August 2005 Pages 277 - 292 http://www.mcs.anl.gov/research/projects/mct/mctP1208.pdf

[6] *Multiscale Coupling Library and Environment (MUSCLE).* research project COAST, http://muscle.berlios.de/

[7] *Towards a Complex Automata Framework for Multi-Scale Modeling: Formalism and the Scale Separation Map.* Alfons G. Hoekstra, Eric Lorenz, Jean-Luc Falcone, and Bastien Chopard, 2007, ICCS '07 Proceedings of the 7th international conference on Computational Science, Part I: ICCS 2007 Pages 922 - 930 http://staff.science.uva.nl/~elorenz/docs/hoekstra_07_towards.pdf

[8] *International Thermonuclear Experimental Reactor.* ITER Organization, http://www.iter.org/

[9] *A distributed multiscale computation of a tightly coupled model using the Multiscale Modeling Language* 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, 2012, Proceedings of the International Conference on Computational Science, ICCS 2012, Volume 9, 2012, Pages 596–605 http://www.sciencedirect.com/science/article/pii/S1877050912001858

[10] *MML: towards a Multiscale Modeling Language* Jean-Luc Falconea, Bastien Choparda, Alfons Hoekstrab 2010, Procedia Computer Science, Vol. 1, No. 1. (May 2010), pp. 819-826

[11] *The xMML format* . `http://www.mapper-project.eu/web/guest/wiki/-/wiki/Main/XMML%20format`

[12] *Multiscale applications composition and execution tools based on simulation models description languages and coupling libraries.* Marcin Nowak, 2011.

[13] *myExperiment.* The University of Manchester and University of Southampton, `http://www.myexperiment.org/`

[14] *The Capabilities of the GridSpace2 Experiment Workbench.* Marian Bubak, Bartosz Baliś, Tomasz Bartyński, Eryk Ciepiela, Włodzimierz Funika, Tomasz Gubała, Daniel Harężlak, Marek Kasztelnik, Joanna Kocot, Maciej Malawski, Jan Meizner, Piotr Nowakowski, Katarzyna Rycerz, 2010 `http://dice.cyfronet.pl/publications/filters/source/papers/CGW2010workbench_abstract.doc`

[15] *Development and Execution of Collaborative Application on the ViroLab Virtual Laboratory* Marek Kasztelnik, Tomasz Gubała, Maciej Malawski, and Marian Bubak, 2008

`http://virolab.cyfronet.pl/trac/vlvl/attachment/wiki/WikiStart/vl07-a02-collab.pdf?format=raw`

[16] *Sinatra.* Blake Mizerany, `http://www.sinatrarb.com/`

[17] *Distributed Ruby.* Masatoshi Seki `http://www.ruby-doc.org/stdlib/libdoc/drb/rdoc/index.html`

[18] *JSON*, Douglas Crockford `http://json.org/`

[19] *JavaScript InfoVis Toolkit.* Nicolás García Belmonte, `http://thejit.org/`

[20] *Portable Batch System.* `http://doesciencegrid.org/public/pbs/`

[21] *Pegasus Workflow Management System* Ewa Deelman, Gaurang Mehta, Karan Vahi, Fabio Silva, Mats Rynge, Jens Voeckler, Rajiv Mayani `http://pegasus.isi.edu/`

[22] *Pegasus : Mapping Scientific Workflows onto the Grid* Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, Miron Livn `http://www.isi.edu/~deelman/pegasus_axgrids.pdf`

[23] *Condor and DAGMan* Condor Team, University of Wisconsin-Madison `http://www.cs.wisc.edu/condor/dagman/`

[24] *Pegasus portal* Gurmeet Singh, Ewa Deelman, Gaurang Mehta, Karan Vahi, Mei-Hui Su, G. Bruce Berriman, John Good, Joseph C. Jacob, Daniel S. Katz, Albert Lazzarini, Kent Blackbur, Scott Korand `http://pegasus.isi.edu/publications/Pegasus_Portal_final2.pdf`

[25] *Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java* Christopher Brooks, Edward A. Lee `http://chess.eecs.berkeley.edu/pubs/655.html`

[26] *Process Networks in Ptolemy II* Mudit Goel `http://ptolemy.eecs.berkeley.edu/papers/98/PNinPtolemyII/pninptII.pdf`

[27] *Comparison of Cloud and Local HPC approach for MUSCLE-based Multiscale Simulations* Katarzyna Rycerz, Marcin Nowak, Paweł Pierzchała, Marian Bubak, Eryk Ciepiela and Daniel Harezlak 2011

A Glossary

AMUSE – an Astrophysical Multipurpose Software Environment. Software framework for large-scale simulations of dense stellar systems.

MCT – The Model Coupling Toolkit

MUST – MUST User Support Tool

G–MUST – Graphical–MUST User Support Tool

ISR2D – Two dimensional in-stent restnosis simulation

ISR3D – Three dimensional in-stent restnosis simulation

Kernel – A single scale simulation from MUSCLE

GridSpace – a web–based computing and data access platform for scientific problems.

MUSCLE – a framework for building multiscale applications according to the complex automata theory.

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

The author of this thesis is also a coauthor of publication [27]. The article is attached below.

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].

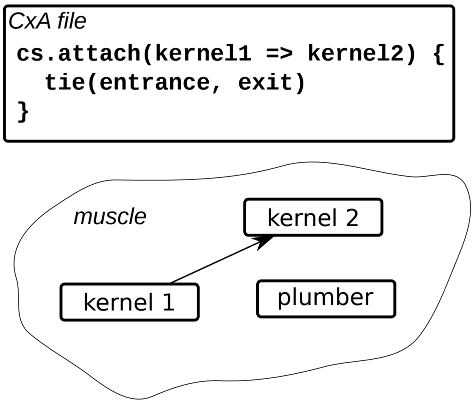
In this paper we present the further results towards building a support for multiscale simulations running on Clouds in a GridSpace environment.

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 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.



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

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

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

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

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

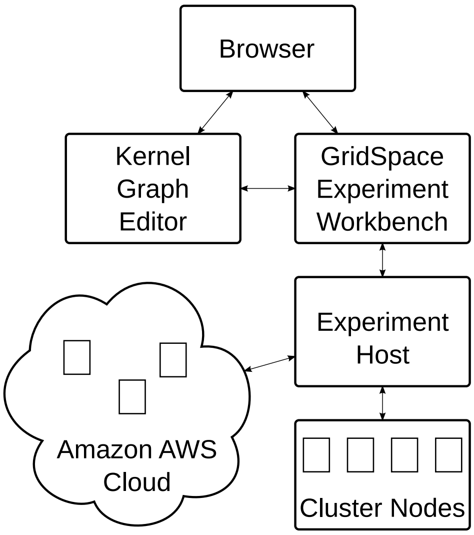


Fig. 2: General Architecture of Proposed Environment

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.

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].

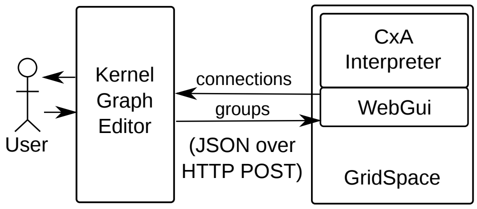


Fig. 3: Loading of CxA connection scheme in the Kernel Graph Editor

VI. Applications Distribution Support in Different Infrastructures

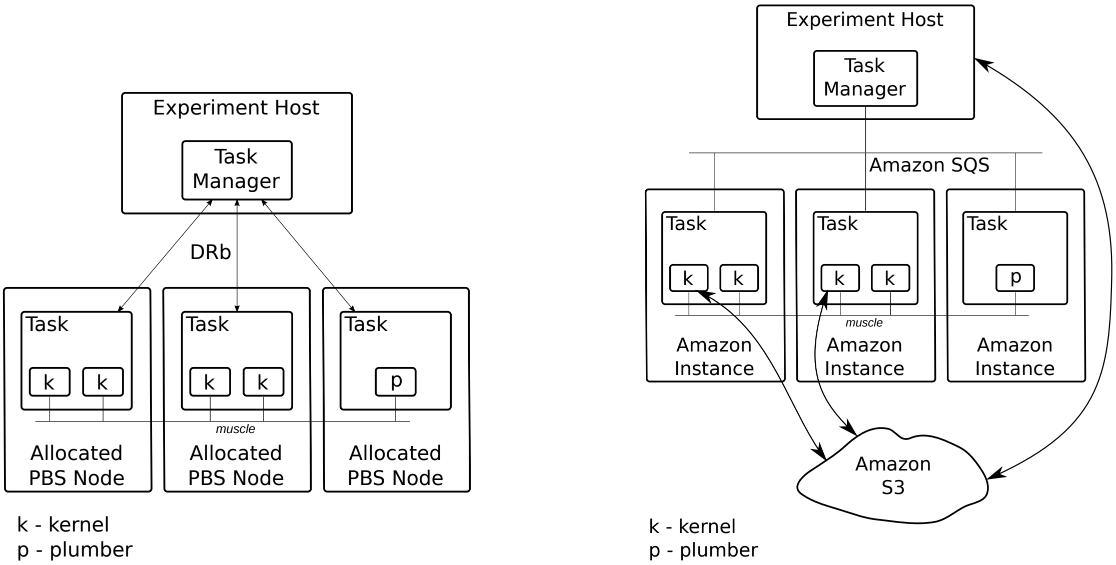
After preparation of MUSCLE application as described in Section IV, it is executed on the chosen infrastructure. As described in Section III, MUSCLE application is a peer to peer type of computation where all kernels are executed concurrently. The execution is controlled by a plumber that once started,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

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

[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.

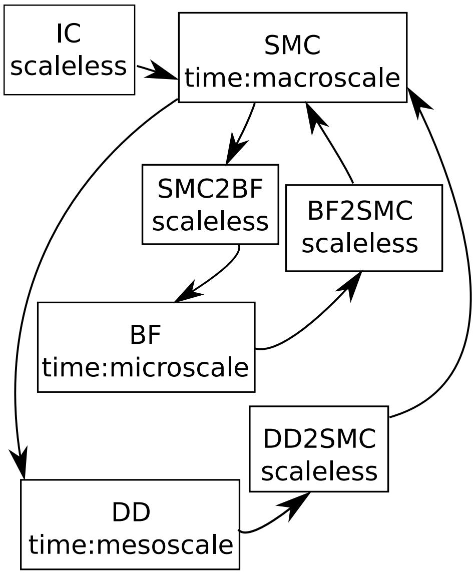


Fig. 5: Simulation of In-stent restenosis - 2D version.

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].

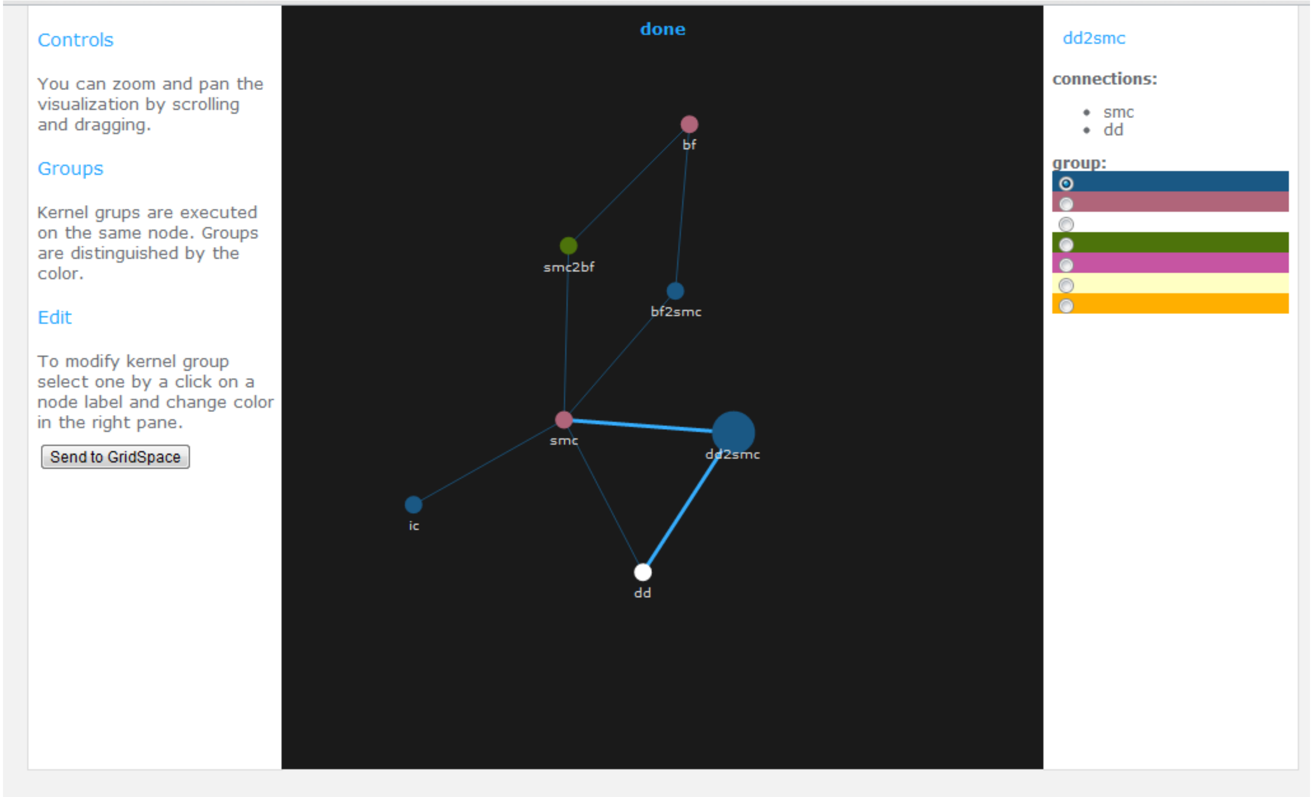
The local HPC cluster used was the HP Cluster Platform 3000 BL 2x220, connected with Infiniband hosted at ACC Cyfronet, Krakow. The machine is number 81 on the June 2011 Top 500 list. When using Cloud we compared following types of instances (please note that One EC2 Compute Unit provides the equivalent 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



Controls

You can zoom and pan the visualization by scrolling and dragging.

Groups

Kernel grups are executed on the same node. Groups are distinguished by the color.

Edit

To modify kernel group select one by a click on a node label and change color in the right pane.

connections:

- smc
- dd

group:

◉
○
○
○
○
○
○

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 m2.4xlarge	 81 80	 6 10	 250 187	 20 3	 100 130	 10 20	 430 400	 20 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 m2.4xlarge	 72 74	 4 4	 2068 1526	 15 4	 120 110	 20 60	 2260 1710	 30 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.

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

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

[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