UNIVERSITÉ DE GENÈVE

Archive ouverte UNIGE

https://archive-ouverte.unige.ch

Thèse 2015

Open Access

This version of the publication is provided by the author(s) and made available in accordance with the copyright holder(s).

Distributed and multiscale computing for scientific applications

Ben Belgacem, Mohamed

How to cite

BEN BELGACEM, Mohamed. Distributed and multiscale computing for scientific applications. Doctoral Thesis, 2015. doi: 10.13097/archive-ouverte/unige:48195

This publication URL: https://archive-ouverte.unige.ch/unige:48195
Publication DOI: 10.13097/archive-ouverte/unige:48195

© This document is protected by copyright. Please refer to copyright holder(s) for terms of use.

UNIVERSITÉ DE GENÈVE
Département d'Informatique

FACULTÉ DES SCIENCES
Professeur Bastien Chopard
Professeur Nabil Abdennadher (HES-SO)

Distributed and Multiscale Computing for Scientific Applications

THÈSE

présentée à la Faculté des Sciences de l'Université de Genève
pour obtenir le grade de Docteur ès sciences, mention Informatique

par
Mohamed Ben Belgacem
de
Gafsa (Tunisie)

Thèse N° 4768

GENÈVE
Atelier d'impression numérique - Repromail
2015

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Doctorat ès sciences
Mention informatique

Thèse de ***Monsieur Mohamed BEN BELGACEM***

intitulée :

"Distributed and Multiscale Computing for Scientific Applications"

La Faculté des sciences, sur le préavis de Monsieur B. CHOPARD, professeur et directeur de thèse (Département d'informatique), Monsieur N. ABDENNADHER, professeur et codirecteur de thèse (Haute Ecole du Paysage, d'Ingénierie et d'Architecture de Genève, Département d'informatique), Monsieur J.-L. FACONE, docteur (Département d'informatique), Monsieur A. HOEKSTRA, professeur (Institut of Informatics, Faculty of Science, Amsterdam, The Netherlands) et Monsieur R. COUTURIER, professeur (Université de Franche-Comté, Institut Universitaire de Technologie Belfort-Monbéliard, Département d'informatique, Belfort, France), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 20 février 2015

Thèse - 4768 -

Le Doyen

N.B. - La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".

Remerciement

Je tiens à exprimer mes plus vifs remerciements à mon directeur de thèse, le Professeur *Bastien Chopard,* et mon co-directeur de thèse, le Professeur *Nabil Abdennadher*, qui m'ont donné l'opportunité de me lancer dans le monde de la recherche scientifique au sein de leurs groupes de recherche. Je remercie Mr. *B. Chopard* de m'avoir fait découvrir le monde de la recheche scientifique multidisciplinaire et pour ses hautes compétences scientifiques, les discussions enrichissantes, et sa vision multidisciplinaire qui ont profondément changé ma vision de la recherche. Je remercie également Mr. *N. Abdennadher* qui m'a introduit dans le monde de la recherche scientifique depuis mon mastère en m'impliquant dans plusieurs projets. Je le remercie aussi de son aide précieuse, ses hautes compétences scientifiques et les discussions enrichissantes tout au long de cette thèse, chose qui m'a beaucoup inspiré.
Je leur suis reconnaissant de leurs précieux conseils et leurs qualités scientifiques et d'encadrement exceptionnelles durant cette thèse.

J'aimerais aussi remercier le Professeur *Alfons Hoekstra*, le Professeur *Raphaël Couturier* et le Docteur *Jean-Luc Falcone* qui m'ont honoré en acceptant de faire partie de mon jury de thèse et évaluer mon trvail.

Je remercie *Marko Niinimaeki* de m'avoir aidé à affronter le monde de "grid-computing". Je remercie *Andrea Parmigiani* de m'avoir initié avec la méthode "Lattice Boltzmann". Un grand merci à *Jonas Lätt*, *Orestis Malaspinas* et *Derek Groen* pour leurs précieux conseils et toutes les réponses à mes questions.
J'adresse toute ma gratitude à tous mes collègues du groupe "Scientific and Parallel Computing" pour le temps qu'on a passé ensemble, les discussions, les pauses café, les blagues, etc.: *Alexandre*, *Andrea*, *Aziza*, *Christophe*, *Daniel L.*, *Federico*, *Gregor*, *Jean-Luc*, *Jonas*, *Kae*, *Orestis*, *Pierre*, *Ranaivo*, *Reto*, *Shalu*, *Xavier*, *Yann T.*, *Yann S.*
Je remercie aussi mes collègues à hepia pour les moments agréables qu'on a passé ensemble: *Régis*, *Marko*, *Davide*, *Dimitri*, *Mikael*, *Eduardo*, *François*, *Khaled*, *Lionel*, *Paul* et *Yassine*. Je remercie aussi tous les membres du personnel administratif et technique: *Anne-Isabelle*, *Daniel A.*, *Lara* et *Nicolas*.

Durant ma thèse, j'ai eu une expérience très enrichissante dans le projet Européen MAPPER, et je tiens à remercier tout le consortium MAPPER. Une pensée particulière va

à *Mariusz Mamonski* (1984-2013) dont les qualités étaient nombreuses et la contribution apportée à ce projet était grande.

Au final, une pensée chalereuse pour mes amis et ma famille, en particulier, mes parents *Abdallah* et *Mahbouba* et mes frères et soeurs: Zouhaier, Hichem, Leïla, Mouna et Ismaïl pour leur soutien moral durant toutes mes années d'études.

Genève, 13 Février 2015

Contents

List of figures vii

List of tables xiii

Summary xvii

Introduction 1

1 A recent overview of the HPC computing landscape 5
1.1 Overview of the evolution of HPC infrastructures . . . 6
1.1.1 From DEISA to PRACE . . . 7
1.1.2 From EGEE to EGI . . . 8
1.1.3 HPCI . . . 10
1.1.4 From TeraGRID to XSEDE . . . 11
1.2 From Grid to Cloud . . . 12
1.2.1 StratusLab project . . . 12
1.2.2 VenusC project . . . 14
1.2.3 EDGI project . . . 15
1.2.4 Magellan . . . 17
1.2.5 ClusterVision: cloud bursting solution . . . 18
1.3 Overview of running distributed applications . . . 19
1.3.1 MPI . . . 20
1.3.2 Shared memory . . . 20
1.3.3 GPGPU . . . 20
1.3.4 Cross-site MPI . . . 20
1.3.5 Distributed computation . . . 21
1.4 Conclusion . . . 23

2 e-Science applications and computational workflows 25
2.1 MetaPIGA . . . 26
2.2 NeuroWeb . . . 28
2.3 Selector . . . 30
2.4 GIFT . . . 32
2.5 CleanCity . . . 33

2.6 Hydrology 35
2.7 Conclusion 36

3 High level paradigms to develop embarrassingly parallel applications 39
3.1 Introduction 39
3.2 New paradigms for the DW e-science applications 41
3.2.1 Pattern 1: Skeleton for handling non persistent jobs 42
3.2.2 Pattern 2: Skeleton for handling persistent jobs 45
3.3 MetaPIGA and NeuroWeb implementation 48
3.3.1 MetaPIGA use case 48
3.3.2 NeuroWeb use case 50
3.3.3 Experiments 52
3.4 Conclusion 54

4 Rationale for multiscale applications design and deployment 57
4.1 Introduction 57
4.2 Challenge in modeling real phenomena at different scales 57
4.3 A brief history of multiscale modeling techniques 58
4.4 The MAPPER Multiscale Framework 59
4.4.1 Modeling 60
4.4.2 Computation Design 71
4.4.3 Multiscale Implementation 72
4.4.4 Execution 79
4.5 Conclusion 84

5 1D-1D coupling of shallow water equation 87
5.1 Introduction 87
5.2 Introduction to the Lattice Boltzmann method 88
5.3 Modeling of an irrigation network with LBM 90
5.4 Coupling through *water junctions* 92
5.4.1 Coupling through a gate junction 92
5.4.2 Coupling through a spillway junction 94
5.5 Towards a multidisciplinary and multiscale computational science 96
5.6 Conclusion 98

6 Evaluation of the MMSF for 3D Lattice Boltzmann based applications 101
6.1 Introduction 101
6.2 Performance model for the MMSF approach 101
6.3 Experiments on EGI HPC clusters 106
6.3.1 Local execution 107
6.3.2 DMC execution 107
6.4 Experiments on universities HPC clusters 108
6.5 Experiments on Cloud clusters 111

6.5.1 Monolithic execution . . . 111
6.5.2 DMC execution . . . 112
6.6 Conclusion . . . 116

7 Impact of optimization techniques on DMC simulation 117
7.1 Boundaries in stencil based applications . . . 117
7.2 Optimization techniques . . . 118
7.2.1 The overlapping communication-and-computation technique . . . 118
7.2.2 Load-balancing technique . . . 121
7.3 Conclusion . . . 124

8 Conclusion and perspectives 127
8.1 Conclusion . . . 127
8.2 Limitations . . . 129
8.3 Perspectives . . . 130

Publications 131

Bibliography 146

List of Figures

2.1 MetaPIGA computation process 26
2.2 Distributed workflow of MetaPIGA 27
2.3 Neuroweb computation algorithm 28
2.4 Splitting of the matrix of the neuronal activities DNM^i 29
2.5 Workflow of NeuroWeb 30
2.6 Numerical map of the Eastern Asia used for the simulation [48]. Each cell has a surface of $40'000$ km^2. The geographic repartition of the selected population is marked by colors. Green (resp. red) cells correspond to a selection of population from the North (resp. South) East of Asia. Red and green arrows represent the starting cells of the migration phenomena. . . 31
2.7 Simulation of a settlement scenario [48]. 32
2.8 Distributed workflow of Gift 33
2.9 A 3D model of a city that takes into account chimneys and roofs [80]. . . . 34
2.10 partition of the 3D geometry domain of the city in several blocks. 34
2.11 The *La Bourne* irrigation canal (*Valence* in France). 35
2.12 The *Rhone* application. 36
2.13 Classification of the scientific applications based on their computational workflow type and targeted computing systems. 37

3.1 Circles are nodes that represent slave jobs. Rectangles represent controllers (master jobs). White circles are non persistent jobs while gray ones are persistent jobs. Arrows between persistent jobs represent data transfer and communication. 41
3.2 Decisional process in a DW applications 41
3.3 Computation workflow with convergence condition. 42
3.4 The computation workflow of pattern 1 43
3.5 Computation workflow of pattern 2 47
3.6 Execution time of MetaPIGA on the Amazon EC2 and the Microsoft Azure cloud platforms. `AWS` stands for Amazon resources. 52
3.7 Execution of MetaPIGA on Amazon EC2: performance comparison with the GC3Pie programming model. 54

4.1 The different steps to model, design, implement and run a multiscale application with the MMSF. 60
4.2 An illustration of the multiscale coupling: two phenomena A and B modeled by submodel A and submodel B having a grid based domain. . . 61
4.3 The *SSM* respresentation of the sedimentation-transport model 62
4.4 Example of a coupling template describing two coupled submodels. Arrows represents the data transfer. 65
4.5 the coupling topology of submodel instances. the number of submodel instances can be fixed in advance before execution ($1..n$) or dynamic during runtime ($1..*$). Similarly, synchronization between the submodel instances can fixed or dynamic. 66
4.6 Example of *filters* applied to *conduits* 67
4.7 Coupling *conduit*s type [60]. The *undefined* marker is used when coupling *mappers* to other submodels. 71
4.8 Modeling step 71
4.9 ScreenShot of a submodel in the MAD interface. `SW1D` is the name of a submodel modeling a rectangular water section with two boundaries. The submodel has two input ports and three output ports. The input port (a triangle in a green circle referring to an arrow marker as described in Fig. 4.7) corresponds to the **B** *SEL* operator used to receive data, in order to update a boundary side of the water section. The two output ports with a black circle marker correspond to the $\mathbf{O_i}$ operator. Each output port sends out an intermediate observation of the corresponding boundary. The output port with a black diamond marker corresponds to the $\mathbf{O_f}$ operator and sends out a final observation at the end of the submodel execution. . . 72
4.10 Screen-shot of the MAD web interface 73
4.11 Computation design step 74
4.12 MUSCLE2 communication design. A Local Manager starts in parallel an instance of its corresponding submodel. Submodels start computation, send data and block until new data are available. Local Managers handle the send/receive operations. Communication between Local Managers is represented by the orange arrow annotated with the communication protocol. 80
4.13 Example of how to run manually a simple MUSCLE2 simulation over two machines: a first command line starts the submodel S_1, Local Manager and the simulation Manager on *machine1*. A second command line start the submodel S_2 and Local Manager on *machine2*. Both Local Managers connect to the Simulation Manager (IP address IP1 and port 5000) and then communicate. 82
4.14 MTO communication protocol. Internal ports ranges in all clusters must be disjoints. MTO public ports (for example ports 5000 and 6000) should be publicly reatchable. 82

5.1 illustration of the collide and Stream operations in D1Q3. 89
5.2 Illustration of the Lattice Boltzmann 1D Shallow water model (SW1D) . . 89
5.3 Illustration of the 3D free-surface Lattice Boltzmann model (FS3D) 90
5.4 The known ($f_1(A)$, $f_0(A)$, $f'_0(B)$ and $f'_2(B)$) and unknown ($f_2(A)$ and $f'_1(B)$) distribution functions at the connection lattice site A and B. 90
5.5 A coupling example of two canal sections and a Gate junction with a boundary condition. The figure on the top describes the real phenomena to simulate. The figure on the bottom represents the MML description of the problem. The two sections are identified as submodels and the gate and the wall are identified as *mapper*s. The wall connected to each section imposes a bounce-back boundary condition. 91
5.6 The known ($f_1(A)$, $f_0(A)$, $f_0(B)$ and $f_2(B)$) and unknown ($f_2(A)$ and $f_1(B)$) distribution functions at the connection lattice site A and B. $f_1(M)$ and $f_2(N)$ are determined by applying a boundary condition. 93
5.7 Simulation using a gate: convergence of the flow Q_A (black dots) and Q_B (white dots) to the imposed discharge $Q = 0.044294$. Other parameters: $\tau = 0.6$, $\alpha = 0.2m$, $h_A = 0.12m$, $h_B = 0.1m$. 95
5.8 Simulation using a spillway: convergence of the flow Q_A and Q_B. 96
5.9 Simulation snapshots of two water sections connected through a spillway. The left section is name `SW1D1` and the right section is named `SW1D3`. `RateFixer_lightKernel` is the name of the *mapper* binary that fixes the discharges Q in the inlet boundary of the left section and the outlet one of the right section. 97
5.10 Building an irrigation canal by coupling three different numerical models: 1D, 2D shallow water and 3D free surface with sediments transport. 98

6.1 Uniform parallelization of MPI based 3D problems into k sub-domains over p cores, where $p_k = p/k$ 102
6.2 MUSCLE2 execution of two connected submodels. Blue block is the boundary data to exchange. The *gather* task collects the boundary data and sends it to the submodel neighbor. The *update* task receives boundary data and updates the computing domain of the corresponding submodel. 103
6.3 The performance of the three execution scenarios of the cavity 3D model (on *Gordias* cluster), for different numbers of grid points per meter N. . . 109
6.4 The runtime of different operations in the local and distributed cases during one iteration, as described in pseudo-code 5: total time [T_{total}]; `Compute()` [$T_{Compute}$]; `getBoundaryData()` [T_{Gather}]; `sendReceiveBoundaryData()` [$T_{Transfer}$]; and `updateBoundaryData()` [T_{Update}]. 110
6.5 Monolithic execution of the cavity 3D flow problem 111
6.6 Definition of the different time slots characterizing a distributed execution over two connected clusters. 112

6.7 The runtime of different operations in the distributed computations cases, during one iteration, as described in algorithm 5 and Fig. 6.6. Figure 6.7a and Figure 6.7b present the results of the same execution using *Scylla* and EC2-Cluster-1. T_{Total}, $T_{Compute}$, T_{Gather}, T_{Update} correspond to the times of the operations `compute()`, `getBoundaryData()`, `updateBoundaryData()`, respectively. T_{Send} and T_{Wait} are defined as the time required to only drop data in the network and $T_{Receive}$ as the time required for the data to move from the sender to the receiver. 113

6.8 The runtime of different operations in the distributed computations cases, during one iteration, as described in algorithm 5: Figure 6.8a and Figure 6.8b present the results of the same execution using *Scylla* and EC2-Cluster-2. T_{Total}, $T_{Compute}$, T_{Gather}, T_{Update} correspond to the times of the operations `compute()`, `getBoundaryData()`, `updateBoundaryData()`, respectively. T_{Send} and T_{Wait} are defined as the time required to only drop data in the network and $T_{Receive}$ as the time required for the data to move from the sender to the receiver. 114

6.9 Timeline chart of the runtime execution over 50 cores. $T_{PreCompute}$ stands for the time required to only compute the border domain in order to get the border data ready to be sent. The “compute()” operation (represented by $T_{Compute}$) is performed in parallel with the “gatherBoundaryData()” and “sendReceiveBoundaryData()” operations (represented by T_{Gather}, T_{Send} and $T_{Receive}$). 115

7.1 Illustration of an example of a stencil based application: 2D Lid-driven cavity flow problem. Green arrows represent the communication between neighboring cells. A red block represent the domain to process by one machine core. Red arrows represent the inter-process communication. . . 118

7.2 Illustration of the overlap zone and the extended boundary when coupling two submodels in a stencil based application. 119

7.3 Timeline chart of the runtime execution before (Fig. 7.3a) and after (Fig. 7.3b-7.3c) applying the overlapping communication-and-computation optimization strategy over 50 cores between Scylla and EC2-Cluster-1. For the optimization case, the “compute()” operation (represented by $T_{Compute}$) is performed in parallel with the “gatherBoundaryData()” and “sendReceiveBoundaryData()” operations (represented by T_{Gather}, T_{Send} and $T_{Receive}$). 122

7.4 Timeline chart comparing the runtime execution using the the communication-and-computation optimization strategy using either 50 or 100 cores on each machines. Only the behavior of the Amazon EC2-Cluster-1 is shown. The “compute()” operation (represented by $T_{Compute}$) is performed in parallel with the “gatherBoundaryData()” and “sendReceiveBoundaryData()” operations (represented by T_{Gather}, T_{Send} and $T_{Receive}$) 123

7.5 Simulation of a problem with length size of $30Km$ using 50 cores for each section. The simulation is run with a load-balancing of 55% of the full problem size on Scylla and 45% on EC2-Cluster-1. 124

List of Tables

1.1 Comparison between HPC infrastructures. 12

2.1 Execution time of MetaPIGA. 27
2.2 Execution time of CleanCity test case on different resources: simulation of the *Quartier des Banques* city in Geneva (1200 iterations). 35

3.1 Characteristics of the VM instance type. 53
3.2 Comparison with the GC3Pie programming model. 55

4.1 Some of the MUSCLE2 methods for `C` programming. 75
4.2 MUSCLE2 data type. 75

5.1 Simulation parameters using the gate and spillway junction 94

6.1 Configuration of clusters 106
6.2 Grid size based on Δx and boundary data size 107
6.3 Local efficiency ϵ^{local} and distributed efficiency ϵ^{grid} (2000 iterations) . . . 108
6.4 1 iteration time (sec). 110
6.5 1 iteration time (sec) of the scenario 3. T_{mpi} is the average time of a monolithic execution on 100 cores of EC2-Cluster-1. θ^{grid} is the overhead defined in eq. 6.12. 115

7.1 Performance measurement after applying the overlap optimization technique. Low resolution corresponds to $N = 0.5$. High resolution corresponds to $N = 4$. Execution times are expressed in seconds. The overhead θ is computed based on eq. 6.12. 120
7.2 Measurements using 50 cores for each machine with a resolution $N = 4.0$ and applying the overlap and load-balancing optimization strategies. In the column *Size of sections*, the size of each section is expressed as a percentage value relatively to the overall size of the problem. The overhead θ^{grid} is computed based on eq. 6.12. 121

Summary

Large-scale scientific applications often involve multi-physics and multi-scales components and become indispensable to the scientific research. Modeling, coding and running these applications require a high level abstraction of modeling and an extensive use of heterogeneous computing resources. This thesis investigates a methodology to design, implement and run high performance scientific applications on widely distributed infrastructures. First, two workflows categories are defined for scientific applications: *embarrassingly-parallel* and *communication-intensive*. Then, a methodology to model and run scientific applications on distributed infrastructures was proposed for each category. For the *embarrassingly-parallel* workflow category, a programming model was proposed for the applications where the number of tasks may not be known in advance and the computation is based on a non-deterministic convergence condition. As a use-case, a scientific application (MetaPIGA) has applied this model and run on two different cloud platforms (Amazon EC2 and Microsoft Azure). For the *communication-intensive* workflow category, a Multiscale Modeling and Simulation Framework (MMSF) was introduced to model, develop and run multiscale applications across distributed high performance infrastructures. This framework consists of a Multiscale Modeling Language (MML) and a runtime environment. This formalism has been applied to an irrigation canal use-case: water sections and junctions are modeled as single scale models, described and coupled based on MML and run using the MMSF runtime environment. This use-case showed that MMSF gave a flexibility to build complex multiscale simulations with an excellent reuse of components. Then, the MMSF performance was evaluated through a three-dimensional intensive computing application. Distributed computation has been performed and evaluated across coupled computing clusters from different infrastructures: EGI, Universities and cloud resources. The performance of the MMSF distributed computations is low compared to the monolithic one without a proper adjustment of the CPUs power, bandwidth of computing machines and the problem size. MMSF enabled us to access to unprecedented computing resources by coupling several supercomputers and clusters; Once the resources are adequately chosen, an good load-balancing can bring a very good speedup, even though the coupled machines are linked with a modest and fluctuating bandwidth. The MMSF is an attractive approach that enables users to change the way of doing computational science.

Key words: High performance computing, distributed computing, programming paradigm, multiscale modeling, Lattice Boltzmann coupling.

Résumé

Les applications scientifiques multi-échelles sont de plus en plus utilisées dans la recherche scientifique. Ces applications impliquent souvent des composants multi-physiques et multi-échelles. Leur modélisation, implémentation et exécution nécessitent une modélisation de haut niveau d'abstraction et une utilisation intensive de ressources de calcul souvent hétérogènes.

Cette thèse propose une méthodologie pour modéliser, implémenter et exécuter des applications scientifiques à haute performance sur une infrastructure de calcul distribuée.

En premier lieu, deux catégories d'applications scientifiques sont considérées : *parallélisme embarrassant* et *communication intensive*.

Pour la première famille *parallélisme embarrassant*, un modèle de programmation est proposé pour les applications dont le nombre de tâches et la condition de convergence ne sont pas connus à priori. Ce modèle a été validé dans le cas d'une application scientifique nommée MetaPIGA. L'exécution de cette application a été effectuée sur deux plateformes Cloud différentes (*Amazon EC2* et *Microsoft Azure*).

Pour la seconde famille (*communication intensive*), un système de modélisation et de simulation muti-échelle (MMSF) est proposé. Ce modèle permet de modéliser, développer et exécuter des applications muti-échelle sur une infrastructure distribuée de haute performance. Le système proposé est composé d'un langage de modélisation multi-échelle (MML) et d'un environnement d'exécution. Ce formalisme a été validé dans le cas d'une application hydrologique de haute performance : des canaux d'eau et des jonctions sont modélisés comme des sous-modèles à une seule échelle. Ils sont décrits et couplés en se basant sur le langage MML et déployés grâce l'environnement d'exécution de MMSF. Cette application a montré que MMSF offre une flexibilité pour simuler des phénomènes complexes et multi-échelles. MMSF garantie aussi une réutilisation des composants développés. La performance de l'approche MMSF a été ensuite évaluée sur une plateforme de calcul distribuée composée d'infrastructures hétérogènes : ressources appartenants à EGI, universités et Cloud.

La performance du système MMSF pour le calcul distribué est faible en comparaison avec une exécution monolithique sans ajustement approprié de la puissance des processeurs, de la bande-passante et de la taille du problème à simuler. MMSF a permis l'accès à une puissance de calcul sans précédent en couplant plusieurs super-calculateurs et serveurs ; Une fois les ressources choisies, un bon équilibrage de charge peut améliorer les performances de calcul même dans le cas où les machines sont connectées avec une

faible bande-passante. MMSF est un système attractif qui permet aux utilisateurs de mieux concevoir et développer des applications multi-échelles.

Mots clefs : Calcul à haute performance, calcul distribué, modèle de programmation, modélisation multi-échelle, couplage avec la méthode Lattice Boltzmann.

Introduction

High-performance scientific applications often involve multi-physics and multi-scales components, are complex to implement and run on large distributed computing infrastructures. These infrastructures are commonly hybrid and usually have different computing architecture. Besides, in the last decades, there has been an urgent and increasing need for computing power since our society and living style is somehow driven by the progress of the technology. We speak here about computational sciences that use high performance computing (HPC) resources to study and predict complex phenomena, for instance, climate changes, computational chemistry, medical images processing, etc. However, the way to develop applications changed from simple codes to complex ones developed in collaboration between several teams. This often results in difficulties to design, maintain, couple together these applications with a view to have them run on distributed computing resources. New challenges appear for computer scientists to provide new high level and multiscale frameworks to relieve end-users from these difficulties, as the majority of them have never had access to HPC resources.

This thesis investigates a methodology to design, implement and run high performance scientific applications on a hybrid widely distributed computing infrastructure. The main core of this thesis is driven by the idea to find a tandem between the "jungle" of the existing computing resources and the variety of the existing scientific applications.

Motivation

The HPC machines keep growing in speed and complexity and reached nowadays the petaflop level while we found that scientists are still using scientific computing legacy codes, which are usually connected using one hand-written script and difficult to maintain. Based on several use-cases, we have found that there is a lack of common vision on how to port scientific applications on the computing infrastructures. Besides, new programming models and smart implementation approaches are needed to deal with the heterogeneous architectures of computing resources.

Also, we should note that very often such heavy computations can not be afforded by the limited number of resources available in the same institution. Indeed, to get more

accurate results, computations need more computing power than what a local center can offer. For example, two weeks of simulation of a physical flow problem ($13Km$ of the *Rhone* river in 3D with a spatial resolution of $25cm$) using 4096 cores on a supercomputer is equivalent to 10 hours of real time flow. This computation speed is therefore unlikely to be fast enough to predict and react to hazardous events for example. As a response to this resources limitation, new infrastructure types has been emerged like Grid, Desktops and cloud computing. For instance, it would be very interesting for the programming methods to allow bridging these platforms in order to use their physically distributed and underused resources as if they are part on the same machine. In fact, running simulations across multiple computing sites has been done in previous works. Even though these solutions provide user friendly APIs and convenient grid job submission, they remain not generic enough to be applied to large uses-cases, limited in term of data transfer performance between sites and do not follow a full methodology based on theoretical concepts and a formalism.

An other important deal is that face to this *jungle* of available resources, interoperability and portability aspects of porting applications on computing infrastructures are now being really important. The portability aspect concerns the development and execution of applications regardless the used computing infrastructures. The interoperability aspect will enable an efficient connection between computing resources, a feature which will be specially advantageous for multiscale-applications. This may be considered as an important future trend in computing that may change the face of computational science.

This thesis tries to find an appropriate solution to these problems. After dealing with the porting of different types of scientific applications, we present a solution which we consider appropriate to establish a tandem between the urgent need of computing power and the heterogeneity of the available computing resources.

The point of view adopted in this thesis is to provide two new generic methodologies to deploy scientific applications on large distributed systems. We started our work based on the experience we acquired by porting and deploying the following scientific applications: *NeuroWeb*, *GIFT*, *MetaPIGA*, *Selector*, *CleanCity* and *Hydrology*. Then, we extended our methodologies in the frame of the MAPPER [101] project and its new approach for deploying multiscale applications. We applied our methodologies on real application use case where we studied also the computational fluid dynamic problem: the 1D shallow water model and the 3D Lattice Boltzmann one.

Content of the thesis

The content of this thesis is organized as follows:

- **chapter 1** gives a recent overview about the HPC computing landscape (HPC, Grid,

Desktops and cloud systems) through a description of a set of HPC infrastructures and scientific projects.

- **Chapter 2** presents a set of applications that we have ported and worked with: *NeuroWeb*, *GIFT*, *MetaPIGA*, *Selector*, *CleanCity* and *Hydrology*. In this chapter, we focus on the applications specificities rather than their classification.
- **chapter 3** proposes a high level paradigm to develop *embarrassingly parallel* applications having a convergence condition.
- **chapter 4** introduces a rational to design, develop and run multiscale applications on distrusted HPC systems.
- **Chapter 5** studies the coupling of 1D shallow water based on the Lattice Boltzmann (LB) method and the multiscale modeling approach presented in chapter 4.
- **Chapter 6** applies the multiscale approach on a 3D intensive application.
- **Chapter 7** presents the used optimization techniques to reduce the overhead time of a distributed simulation and evaluate their impact.
- Finally the thesis concludes and presents the future works.

The chapters core is derived from scientific papers published during this thesis. Chapter 2 is derived from the work [106, 23, 105, 107]. Chapter 3 is based on [14, 15]. Chapter 4 is derived from [19, 26]. Chapter 5 is derived from [22, 21]. Chapter 6 is partially derived from [19, 18]. Chapter 7 is derived from [18, 87].

1 A recent overview of the HPC computing landscape

This chapter gives an overview about the current High Performance computing (HPC) landscape and the distributed computing infrastructures which have been used to develop and run large scale distributed applications.

Currently, computing resources are evolving faster than the methods used to develop the scientific applications. On one side, new platforms have been recently emerged like Desktop-grid and cloud computing. On the other side, the methods and formalisms to develop scientific parallel applications remained unchanged. This is more visible especially for the case of “multiscale” applications that require heterogeneous computing resources as the different subsystems may require any computing power, from a simple PC to a supercomputer [70].

Given this recent changes in the HPC landscape this chapter consists of a history of the evolution of HPC infrastructures through a description of a recent set of computing infrastructures and research projects. This set is not complete but we believe that it is illustrative for this purpose. The section 1.1 of this chapter describes a set of HPC infrastructure projects. Section 1.2 deals with the interconnection between resources of different HPC infrastructures. It describes the Grid development and the transition from the Grid to the cloud concept. Section 1.3 discusses the “ways” of running distributed and concurrent scientific applications.

Section 1.1 is a description of a set of research projects. Each project description includes the goal, period of time and the project scope during which it has being operating. Then, this description is extended further to cover the technical aspects of the computing resources so that the reader can know which range of applications can run on which infrastructure. In addition, we highlight the deployment mechanisms that can be used to run applications: for instance, submission of jobs are done using batch systems, command line tools, grid middlewares, reservation mechanisms, etc.

1.1 Overview of the evolution of HPC infrastructures

In this section, we present a recent history of the evolution of six international HPC infrastructures: DEISA, PRACE, EGEE, EGI, HPCI and XSDED.

Before starting presenting the infrastructures, it is worth to define the technical terms that are related to the High-performance computing domain: a supercomputer, HPC-cluster, grid and Desktop-grid.

- **Supercomputer**: is a machine equipped with hight-end devices to run massively parallel codes at an extremely high speed. In general, a supercomputers is composed of several nodes. Each node is composed of many CPUs, memory and local storage space. Nodes are grouped together into the so called *Rack*. The node of the same racks have a collective network linking and storage system. So, a supercomputer is a collection of *Racks* connected with each other through an high-speed and efficient network architecture (such as Three-dimensional Torus networks). Several companies built their supercomputer systems, for instance, *BlueGene/Q* [24] (IBM company), *CrayXC* [44] (Cray company). Currently, the *Tianhe-2* supercomputer located at the *Sun Yat-sen* University in China delivers ~ 33.86 PetaFLOPS/s and is considered as the world's fastest supercomputer.
- **HPC cluster**: a HPC cluster is composed of homogeneous computing nodes, grouped in general within a single *Rack*. Nodes are connected through a fast network connection (generally through InfiniBand device).
- **Grid computing platform**: This is a general term which means a federation of computing resources together through local/wide area network or Internet. These computing resources are closely-managed and monitored. Most of grid computing systems are a federation of national HPC clusters located at different universities and centers and may have different characteristics. In the grid, the resources are operated by a grid-middleware, a software stack that makes the computing resources available through Internet. For instance, a user can install a grid-client tool on his machine to be able to submit jobs on the remote computing resources of the grid. An example of grid system is the *Nordic Data Grid Facility* (NDGF) [52] which is an scientific infrastructure federating HPC clusters from Nordic countries: Denmark, Finland, Norway, Sweden and Iceland. The NDGF platform is managed by the advanced resource allocation (ARC) [122] grid middleware. ARC is used as grid middleware for the *Swiss Multi-Science Computing Grid* (SMSCG) [130] in Switzerland. SMSCG is a grid distributed infrastructure with the objective to provide computational resources to solve scientific problems.
- **Desktop-grid and Volunteer computing**: It federates Desktop machines through a wide network and Internet. Desktop machines belong to individual and/or Institutes. They are volatile in the sens that they can be switched off at any moment. The aim of the Desktop-grid systems is to use the idle time of those machines to

perform scientific computations. Generally, individuals are encouraged to install a light software daemon on their Desktops to connect them to the grid during night. Several Desktop-grid platforms have been built such as SETI@Home [8], BOINC [25], XtremWeb-CH [16, 106]. The federated power of these Desktop-grid platforms can be used to run a set of distributed applications where jobs are independents and CPU intensive. However, these platforms suffer from several limits. One of the limit is that network bandwidth is limited and unstable. In addition, there should exist a fault-tolerance mechanism to resume an execution in case the corresponding volatile machine is suddenly unplugged from the grid.

1.1.1 From DEISA to PRACE

The Distributed European Infrastructure for Supercomputing Applications (DEISA) [66] is an European project (2004-2008) co-funded through the EU DEISA project. The aim of this project was to build up the first European powerful HPC infrastructure which integrates national Tier-1 supercomputers in order to run challenging applications and projects in term of computing power. DEISA was followed by second project, the EU FP7 DEISA2, from 2008 to 2011 to consolidate and further develop the existing distributed high performance computing infrastructure and its services.

DEISA is built on top of national services. The federated European resources have delivered a computing performance that excessed 2 PetaFlops/s. Supercomputers in DEISA infrastructure are connected through a high speed network (up to 500 Gb/s) offering a high level data sharing and transfer services. The network system was based on the GEANT2 [65] and the National Research and Education Networks (NRENs) network infrastructures.

Between 2010(October)-2011(April), 55% of 122 project proposals to DEISA coming from several domains (astronomy, biology, earth sciences, engineering, materials and physics) were accepted and 91 million core-hours among 15 European HPC machines were allocated to them. These projects involved scientists from many countries: 69% of the projects have involved researchers from more than one country and 25% of them from more than three countries.

Users have access to *GridFTP* [5] service for an efficient transfer of massive data between external resources. Access to computing systems is done through a single-sign-on mechanism. Each supercomputer system is equipped with a local resources management system (batch system) to submit parallel jobs directly from the master node. Computing nodes shared working space based on the IBM file system (*GPFS*) [125]. For Grid submission of jobs, users used *UNICORE* [131] grid midlleware to submit job over heterogeneous systems either through its command lines or workflow management interface.

Starting from May 2011, Tier-1 services offered by DEISA becomes part of the PRACE [121]

(the Partnership for Advanced Computing in Europe) research infrastructure that integrates also the most powerful supercomputers in Europe (Tier-0 machines).

The PRACE program is scheduled over 3 steps: PRACE1.0 (2010-2015), PRACE2.0 (2015-2020) and PRACE3.0 (2020-2025). Currently, PRACE1.0 infrastructure is deployed over 6 supercomputers (Tier-0) that are hosted by four European countries: France, Germany, Italy and Spain. These machines have different architectures and offer 15 *PetaFlops* of computing capacity. Since 2010, PRACE1.0 granted nearly 5 billion hours to 259 projects and around 25 European members are participating. Since 2012, access to PRACE resources were opened to all academia fields and industry. For instance, the French vehicle manufacturer *Renault* received 42 million core hours on the *CURIE* supercomputer: the objective is to anticipate the new security regulation (*EuroNCAP6*) in 2015 by improving the security of vehicles. Such simulation requires multiplying the parameters of crashes simulation which was out of reach using the *R&D* internal computing resources of *Renault*.

PRACE1.0 prepares the next phase of the project PRACE2.0 which intends to guarantee the sustainability of PRACE as the European HPC Infrastructure and becomes the world-leading research infrastructure. The main objective of PRACE2.0 is to provide a powerful HPC infrastructure for scientific research opened to all domains and industry. In addition, it aims to attract the European competence and highly skilled workforce in science and engineering in order to provide a high quality services for scientific and industrial European users.

DEISA succeeded in several points. First, it succeeded in building an operational European infrastructure with a unified access mechanism to support European researches and this is established although the political constraints. Besides, it used the existing network infrastructure of GEANT2. Also, It brought the computing power near to researchers and established a European expert team and a European science community.

A limitation of DEISA is that computing resources are heavily used and there exists no efficient load-balancing and reservation system in all the supercomputers systems. This limitation will be addressed within the PRACE program scope aiming to provide a high quality services to end-users.

1.1.2 From EGEE to EGI

The EGEE (Enabling Grids for e-Science) [61] is a European project aiming to federate university clusters and machines to provide a Grid infrastructure for running data-intensive applications. Resources belong to different computing centers (such as CERN [33]) among several countries world-wide. The project was composed of three phases: EGEE-I(2004-2006), EGEE-II (2006-2008), and EGEE-III (2008-2010). The last phase was co-funded by the European Commission and aimed to extend supporting applications from life science and high energy physic to multiple scientific fields.

EGEE is accessible through a grid middleware layer that allows users to run their applications on the computing resources. gLite [49] is the EGEE grid middleware developed based on a service oriented architecture.

In the EGEE project, users communities are grouped by the so-called virtual organizations (VOs). A virtual organization (VO) "*comprises a set of individuals and/or institutions having direct access to computers, software, data, and other resources for collaborative problem-solving to other purposes*" [95]. The resources usage policy and the computing resources access are defined in each VO. A user can be registered in several VOs and so have access to their computing resources. The roles of users are managed by a service called VOMS (Virtual Organization Membership Service).

EGEE was designed first to support running data-intensive applications. The big data files are stored in Storage Element (SE) machines and registered in a catalog file. A catalog file contains the mapping between the logical files names and their physical locations. The storage service in EGEE infrastructure allows to replicate a given file on several SEs using the same logical name. Several data transfer protocols can be used such as *gsftp* that allows to transfer files efficiently and securely between the computing nodes and the SEs.

Before submitting his application, a user should generate a short term grid certificate (x509) that will allow him the access the infrastructure resources of his VO. In general, this proxy certificate has a short validity duration (ranging from 6 hours to 11 days) and should be renewed when it expires. A job should be described using a job description language (JDL for the case of gLite) that allows users to specify their jobs requirements in a well-defined syntax. A user can submit his job directly to a computing node (cluster or machines) or through the Workload Management Service (WMS) of the gLite middleware. The WMS schedules and dispatches the jobs to the computing resources that belongs to user's VO according to the job requirements and the location of its input files.

The EGI-Inspire project [54](2010-2014) co-funded by the European Commission is started as a continuation of the EGEE project with the aim to establish a sustainable European Grid Infrastructure (EGI) [54] for European research.

The European organizational model of EGI-Inspire project is based on NGIs and EIROs organizations to ensure a long-term sustainability of the HPC infrastructure. NGI stands for National Grid initiative where each country manages its computing resources that are provided to EGI. A NGI represents the only contact point of a given country. EIRO stands for a European Intergovernmental Research Organization such as CERN.

EGI aims also to integrate new computing resources coming from the Desktop-grid and cloud platforms. This will enables users to have more choice to run their scientific applications on the required resources. Several applications have used the EGEE/EGI infrastructure like the experiments of Large Hardon Collidor (HLC). The EGI infrastructure

has adopted the software platform delivered by the EMI (European Middleware Initiative) project. It includes ARC [122, 9], gLite [49], UNICORE [131] and dCache [13] grid middleware in addition to Globus [5]. Even though, there exists an interoperability between these middlewares, the sustainability aspect may require administration efforts to support several versions of them.

1.1.3 HPCI

The Japanese's Hight Performance Computing Infrastructure (HPCI) is a national project promoted by the Japanese ministry of Education, Culture, Sports, Science and technology (MEXT) to build a distributed HPC infrastructure. The project started in March 2011 and entered in production phase in the end of September 2012. It consists in connecting the *K computer* [89], other ten supercomputers sites/universities in Japan, and two high performance shared storages via high-speed networks.

K computer is composed of 864 compute Racks. Each Rack is composed of 24 System-Bord (SB), each SB contains 4 nodes, each node contains 1 CPU and has 16GB of memory. The CPU is developed by *Fujitsu* (SPARC64TM VIIIfx processor) and it operates at a clock frequency of 2.0 GHz and has 8 cores sharing a L2 memory size of 6 MB. It has a high communication performance and fault-tolerant network based on 6-dimensional mesh/torus network (each node has 10 link of 5GB/s with 2 bandwidths). In total, *K computer* has 82944 nodes (663552 cores), it delivers a peak of performance of 10.62 PetaFlops and has a total memory capacity of 1.27 PetaByte. It was the fastest supercomputer in the world in 2011. Currently, the *Tianhe-2* supercomputer located at the *Sun Yat-sen* University in China is the world's fastest supercomputer according to the *TOP500* ranking [142].

The main objective of HPCI is to establish the Japanese national center of excellence (CEO) for high performance computing [149]. This center is conducting cutting-edge researches in computational science and industry in Japan. This includes the development of large-scale parallel applications which belong to different area like Nano-technology and life science. In addition to that, CEO has the mission to promote a strong collaboration between computational and computer scientists. The computer scientists teams are organized into two sub-teams. The first sub-team focus on the development of softwares to maximize the use of K supercomputers: scalable and high performance file I/O and improvement at the level of operating system with regards to cache, fault-tolerance and power consumption. The second sub-team focus on developing parallel programming language/model and analysis tools to increase the performance of running applications on over the available parallel systems. Computational scientist team is organized per modeling methodologies rather than per applications domains. Its aim is to develop new numerical algorithms and models that will be commonly used by scientists and run over HPCI resources. Five sub-teams were formed: computational Molecular, Material, Biophysics, Climate and Field theory.

User access to HPCI is managed by the single-sign-on system based on Shibboleth [127] and Grid Security Infrastructure (GSI) [140]. A given user should access to the HPCI portal through Shibboleth authentication to get his proxy certificate and download it into his client machine. Then, using a command line, the user can access the frond-end node of each supercomputers where he can submit jobs.

1.1.4 From TeraGRID to XSEDE

The Extreme Science and Engineering Discovery Environment (XSEDE) [147] is a project started at 2011 and founded by the National Science Foundation (NSF) of USA. It is the successor of the previous TeraGRID project [92]. Its aim is to enhance the existing computing infrastructure into more advanced cyber-infrastructure with reliable hardwares, tools and more focus on user support: scientists should easily use and share computing resources, data and expertise.

XSEDE operates on distributed and heterogeneous computing resources. Resources include more than 20 High Performance Computing (HPC) supercomputers, High Throughput Computing (HTC) machines, visualization and data storage services. XSEDE is built with more user oriented vision in order to support the users community (scientists, engineers, and students) by offering them training, consulting, advanced graphical interfaces and on-line services to facilitate the access to the heterogeneous resources and share experience.

In order to access the XSEDE system, a user should request a XSEDE user identity via the XSEDE web portal. This identity consists of an ID and a password allowing the access to many XSEDE functionalities. For instance, XSEDE provides to end-users, who are non-familiar with command-line session, with an interactive login interface to access remotely the resources. It consists of a web based interface which enables the user to list the resources to which he has access to and select one over them. Then, he clicks on a link to establish a login session on the corresponding XSEDE resource. XSEDE comes also with facilities to transparently access and treat files from anywhere using Internet connection. They are implemented into two mechanisms: the XSEDE Wide File System (XWFS) and the XSEDE Global Federated File System (GFFS). XWFS is based on IBM GPFS [125] and is shared among all users that can transparently access to their files. It requires that all users trust each other and share the *root* storage level. GFFS is a federated file system where it is possible to export a rooted directory trees (which can be one of: Windows, Linux, Luster or GPFS file systems) into a remote and secure directory structure. Then access to this directory requires XSEDE user/group credentials. Access can be performed either through command line interfaces, programming API, or graphical interface. For Linux OS case, access can be also performed by mounting locally a GFFS directory structure in read-only mode. Note that these two file access mechanisms are slower than an OS native access because of the Internet network speed.

The common way to submit jobs on XSEDE computing resources is to use the interactive login node feature in order to access the frond-end node of the computing resource and transfer files there. Then, the user uses the local batch system of the resource to submit, monitor jobs and transfer results files. For remote job submission, XSDED provides The Execution Management Services (EMS) which prepares, executes, manages jobs on the XSEDE grid of resources. The EMS implementation is based on *UNICORE 6* [131] and *Genesis II* [108] grid middlewares. To run a computation, one requires to firstly describe the job to submit using the Job Submission Description Language (JSDL). Then, he should obtain a credential to access one computing resource and to submit the job on it. Job submission can be done using grid-client CLI such as (*UNICORE 6*) or the *Genesis II* GUI.

Table 1.1 illustrates a simple comparison between the described HPC infrastructures.

Table 1.1 – Comparison between HPC infrastructures.

Platforms	**Region**	**Start year**	**End year**	**Target community**	**Supported OS**	**Support MPI**	**Support Grid Jobs**	**Authentication**	**Resources**
PRACE	Europe	2011	–	scientists, Industry	Unix/Linux	Yes	Yes	GSI-SSH	supercomputers
EGI	Europe	2010	2014	scientists	Unix/Linux	Yes	Yes	GSI-SSH	HPC clusters
XSEDE	USA	2011	–	scientists, engineers	Unix/Linux	Yes	Yes	XSEDE credentials	supercomputers
HPCI	Japan	2011	–	scientists, Industry	Unix/Linux	Yes	Yes	GSI-SSH	K-computers

1.2 From Grid to Cloud

In this section, we present four research projects: StratusLab, VenusC, EDGI and Magellan. The majority of the involved partners in these projects comes from the grid computing community. This illustrates better the transition from the Grid to the cloud concept. We present also the integration of cloud in the HPC service through the case of *Bright Cluster manager* product.

1.2.1 StratusLab project

StratusLab (2010-2012) [134, 98] is a European project aiming to enhance the work of the OpenNebula [117] framework. Since there was no complete cloud solution before the start of StratusLab project, the objective of this project was to provide a complete “Infrastructure as a Service” (IaaS) advanced solution for cloud computing deployments. This solution can be applied on several computing infrastructures such as EGI. The StartusLab open-source solution includes computing, storage, networking, and Marketplace service facilities. Project partners are from European countries (France, Greece, Ireland, Spain, and Switzerland).

The previous project OpenNebula started as a research project at the university of Madrid (Spain) in 2005. After its first release in 2007, it joined the open-source community *OpenNebula.org* and started to be adopted in several research projects and commercial companies. It is used to manage a virtualized infrastructure such as in data centers or universities clusters. In addition, one can create a private, large and scalable infrastructure by combining local and public-cloud infrastructures. Indeed, OpenNubela enables the administrator to easily build and monitor an elastic cloud infrastructure composed of virtual machines, storage services and networking facilities through a set of services invocable from command lines interfaces and APIs. Thus, it allows to access computing resources without investing in new hardware and infrastructure. StratusLab was conceived with the idea to support running grid services over a cloud infrastructure. In this case, the grid and cloud interfaces should provide the same authentication mechanisms for the end-users.

StrastusLab is used primarily by administrators to build an elastic grid based on cloud infrastructure. Several applications have run ob a StratusLab based infrastructure [98]: Astrophysics, Machine Learning, High Energy Physics, Meterology. Among these applications, several ways of using StratusLab have been adopted. For parallel applications, StratusLab is used to build HPC clusters composed of virtual machines with a shared NFS storage space to run MPI codes. For applications having independent jobs, a user starts a set of virtual machine using StratusLab command line and launches manually a binary file on each virtual machine. For grid applications, StratusLab was used to create a clean Globus based grid infrastructure of virtual machines offering all the Globus services runtime.

In commercial sector, private company like SixSq [128] has adopted StrastusLab through its product *SleepStream*. It consists of a software used to easily automate the deployment of a cloud runtime environment and the configuration its virtual machines which can belong to different cloud providers. Hence, through adopting StratusLab, *SleepStream* supports local and national cloud providers which makes it attractive for companies, especially for those that do not like to use public cloud resources to treat their critical data.

StratusLab has proven to be valuable as a complete cloud solution. It has formed the basis for several projects[98]: i)*SleepStream* demonstrated the ability to automatically runtime environment of the ground control system of the European Space Agency (SCOS-2000), ii) DS-Cloud ReadyPack is an collaboration initiative with SixSq, IBM and Darest [47] to provide a “ready-to-use” private cloud system destined to run on IBM machines and having *SleepStream* as software stack. and iii) “Helix Nebula” is an initiative to provide cloud infrastructure based on public/private partnership to run large scale scientific simulation from CERN, EMBL, and ESA.

1.2.2 VenusC project

VENUS-C (2010-2012) [144] is a European project mainly funded by Microsoft with the purpose to provide a new friendly-user cloud solution for the scientific research domain. The objective of VENUS-C project is to make it possible for researchers' community to run easily their applications on a large cloud computing infrastructure in order to accelerate their scientific researches. VENUS-C targets computing resources at the Microsoft Windows Azure, the Royal Institute of Technology (KTH, Sweden) and the Barcelona Supercomputing Center (BSC, Spain).

Technically, the Venus-C is an interface between the Cloud providers and the end-users. It aims to provide a *Platform as a Service* (PaaS) with a set of tools and APIs to easily develop e-sciences applications and execute jobs that requires an execution coordination and a platform elasticity features.

The VENUS-C project came up with the a Cloud Data Management Interface(CDMI) Proxy, COMPSs and Generic worker (GW) frameworks. The CDMI Proxy offers a standard RESTfull interface and Java/Python client libraries to access local and cloud storage systems including several mechanisms of authentication, authorization, logging, accounting, etc. It supports several storage systems for instance Amazon S3, Azure Blob Storage and local file system. Users have the possibility to store the same data in more than one storage system.

COMPSs and GW programming frameworks offer to the end-user a programming model and a runtime environment to facilitate porting scientific applications. They shield the users from technical complexity of the steps to use cloud computing resources. COMPSs uses BSC EMOTIVE Cloud [56] and *OpenNebula* to create an elastic cloud infrastructure to run jobs. The GW is an intermediate layer between Azure (The Microsoft cloud platform) [104] and the end-users. The main role of the GW is to facilitate the creation of the virtual machine instances on the Azure cloud infrastructure [104] and executing the users applications.

COMPSs runtime environment automatically increases the number of virtual machines when a low number of submitted jobs is detected or when there exits no virtual machines configuration that matches a given job requirement. Besides, when a low number of submitted jobs is detected, some of the virtual machines instances are shutdown and their resources are released.

When a job is submitted to the COMPSs platform using a client program, a virtual machine is created. Once done, the job package (binary, COMPSs runtime scripts) is downloaded on the corresponding machine. Then, the input files of job are downloaded from their remote location to the local hard disk. Input files can be located on different sites and staged-in using the CDMI Proxy API. Once th job is finished, its output file is uploaded on a storage system as specified in the job description.

A VENUS-C application using th GW concept can be composed of a workflow of jobs, where dependencies are based on input files: a job can not be started unless its input files exist in the Azure storage domain. Each running virtual machine contains a GW instance that handles the execution of a given job. All the submitted jobs are stored in an Azure database table, and, then, scheduled by a service monitoring to the available GW instances. Periodically, each GW instance reads the database and retrieves its scheduled job. To execute the job, the GW instance should check the existence of its job's input file in the Cloud storage domain, then, loads them with the binary and any necessary libraries files to its local machine disc. Accordingly, the status of the job is tracked during its execution in the database. When the execution ends, the GW instance stages out the job result in the user cloud storage. Besides, the GW provides a web service interface that allows the user through its client program to perform the scaling, notification and job management services. It worth noticing here that the number of the GW instances can be efficiently scaled on demand through the client program.

15 applications has been ported on the VENUS-C platform spanning various domains: spanning biology, bioinformatics, chemistry, earth sciences, maritime surveillance, mathematics, medicine and healthcare, physics and social media [144].

GW framework does not support heavily parallel application (such as MPI based) and can only run on windows machines on Azure resources even though Microsoft Azure provides Linux virtual machines.

1.2.3 EDGI project

The FP-7 project European Desktop Grid Initiative (*EDGI*) [51] (2012) is a follow up on the previous project *EDGeS* [50]. It aims to build a European Desktop-grid federation through creating a Bridge between grid middlewares (ARC [122, 9], gLite [49] and Unicore [131]), Desktop-grid systems (BOINC [25, 7], the XtremWeb-CH [148] and XtremWeb-HEP [78]) and cloud platforms. The grid system supports running data-intensive applications while the Desktop-grid system with an extension on the cloud enable to offload urgent computing of scientific simulations.

EDGI infrastructure federates geographically distributed machines, where most of them are university, institutional Desktop grids and EGI resources. The Desktop resources are connected by the BOINC, XtremWeb-HEP and XtremWeb-CH middlewares, therefore, they are largely heterogeneous and support Linux/MacOs/windows operating systems. The EGI grid resources are machines and clusters composed of homogeneous nodes.

Desktop Grid infrastructure can be public or local like The SZTAKI Desktop Grid [138] and the University of Westminster Local Desktop Grid (WLDG) platform. WLDG connects more than 1600 Dual-core desktop machines distributed over four university sites. The SZTAKI Desktop Grid is BOINC based platform installed on the Hungarian Academy of

Sciences. Grid users can extend the computing resources of their Virtual organizations (VOs) using the Desktop ones (such as BOINC). The *EDGI* infrastructure provides bridges to connect EGI resources to Desktop ones allowing grid users to run simulation across Desktop resources. Besides, a cloud bridge allows Desktop grid users to use cloud resources for their simulations.

Security issues related to executing malicious code on the machines of providers have been addressed in this project. Each Desktop grid platform provides its own security strategy. The main security strategy in local and private Desktop grid platforms is the mutual trust between the providers and users. On one side, applications code should be validated before uploading them on the applications repository to avoid damaging the providers resources. On the other side, the providers have a full control on their machines and, thus, end-users have the possibility to replicate the same job execution over several resources to avoid a possible tapering of results. For public Desktop platforms virtualization techniques enable to securely run untrusted codes inside a virtual machine but present an additional performance overhead.

Users from EGI and NGI communities can access the services of the *EDGI* infrastructure. *EDGI* supports running parameters sweep based applications where jobs are *embarrassingly parallel* and do not communicate. An application is considered suitable to be executed on Desktop platform if it satisfies the following condition [3]:

- In its simple form, a job consists of input files, binary, command line parameters and output files.
- Computing resources do not support MPI runtime environment.
- Jobs can not communicate during execution.
- Jobs do not share a data storage.
- Transfer of input and output results of jobs is done through a file server.
- Jobs can run for long period with small input and output files size (100 MBytes).

The submission of simulations over the *EDGI* infrastructure is done either through the gUSE (grid User Support Environment) web portal or command line interface (CLI). The gUse web portal allows to design, submit and monitor workflow execution and hide the low level access to Grid systems. The CLI facilitates the porting of application on a Grid system for grid users [139].

There exist two types of applications: native and legacy applications. Native applications use native APIs calls which are proper to a given Desktop platform. Legacy applications consists in a binary file and a set of input files and does not require much effort to be ported on different Desktop platforms. Face to the diversity of the Desktop Grid system, a meta-API (called DC-API) is provided as a native solution to develop application destined to be executed on CONDOR [141], XtremWeb and BOINC platforms. A virtualized

wrapper called GBAC ("Generic BOINC Application Client") is provided as a generic solution to port legacy application on several Desktop platforms.

1.2.4 Magellan

Magellan [123] (2009-2012) is a project funded by the U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research (ASCR) program. The goal of Magellan is to investigate whether it is useful to offload data-intensive and mid-range computing workloads of the DOE on cloud resources. Mid-range applications are codes that require 10 to 100 CPU cores to run.

To address this objective, a distributed testbed infrastructures were deployed at the Argonne Leadership Computing Facility (ALCF) and the National Energy Research Scientific Computing Center (NERSC). For instance, the testbed installed at *Aragon* was composed of diverse collection of hardware resources: 504 IBM *iDataPlex* servers (each node has 2 Intel Quand-Core Hehalem CPUs with 25GB of memory, 40Gb InfiniBand and 1GB Ethernet connection), 133 GPU servers, 15 large-memory nodes (each has 1TB and Intel quad-core processors), 200 Active Storage servers. Resource were connected through Infiniband (4X QDR, 40Gb) and 1 GB Ethernet. In total, the *Aragon* testbed delivers 150 TeraFlops of computing capacity with 8240 cores, 42TB of memory, 1.4 PB of storage space, and a single 10 Gb external network connection [123].

Magellan had the purpose to explore different programming models and cloud service models (batch system, EC2-model, Hadoop [76] for Map-Reduce model, etc.). To do so, the software stack in both sites used the same tools and softwares as in traditional HPC clusters. In order to have an easily customizable environment, a secure use of resources, and an on-demand resource provision a virtual stack solutions have been adopted in the Magellan project. For instance, Open source projects like Eucalyptus [58] and OpenStack [118] have been tested. These tests were made to evaluate the stability of the virtual stacks solutions to run the scientific applications and the applicability of cloud programming models used by the scientist users. The Eucalyptus provides a compatible API with Amazon EC2 [6] to manage virtual machines and create a private virtual platform for private users. Tests of managing virtual machines revealed that Eucalyptus had a limit of 800 of instance running simultaneously. Besides, operations such as running instances seemed to be sequential and thus resulting on a considerable overhead when adding new instances. Unlike Eucalyptus, OpenStack scaled much better and it was easier to deploy and manage. So, it was turned out that OpenStack offered better solution to run large scale application on the Magellan testbeds. Several applications have been run on the Magellan cloud resources; these applications were throughput-oriented with embarrassingly parallel tasks, have large data size and require complex softwares installation.

Several finding of the Magellan project were pointed out [91]. First, the cloud computing approaches enabled through the visualization feature the users to define and customize their computing runtime without an administration overhead. Furthermore, several applications can take advantage from cloud programming models such as Hadoop. For example, the high-through applications are often serial which makes them not suitable to run on ordinary HPC centers. Instead, that may benefit from the scaling up feature of the cloud resources with combination to the Map-Reduce programming model of the Hadoop system. Some technical difficulties are to consider since scientists may require programming skills to be able to port their scientific applications on cloud platforms. In the end, there is a potential benefit to enable the cloud features on HPC system to ensure both the flexibility and performances when running scientific applications.

1.2.5 ClusterVision: cloud bursting solution

ClusterVision [41] is a rapidly growing company specialized in HPC solutions in Europe. Its main activities are the design, implementation and support of small and large HPC clusters. The company offers high quality hardware (CPU/GPU, fast interconnect solution, efficient storage system, etc) and a full software stack solution to easily use and manage HPC cluster machines.

Bright Cluster manager [30] is the home-grown software product of ClusterVision. It proposes both a friendly-user GUI and a command line interface to easily manage and use a cluster without the need to be a HPC expert. It supports machines with complex architecture and scale up to thousand of nodes on a supercomputer. Administrators can easily extend a given cluster computing capacity by adding extra nodes and/or *Intel Xeon Phi* [84] co-processors. Since from 2011-2012, *Bright Cluster manager* allows extending HPC clusters into the Amazon EC2 cloud [6] (this is called also "cloud bursting" strategy). Currently, two scenarios of cloud bursting are catered: *Cluster-On-Demand* and *Cluster Extension*.

The *Cluster-On-Demand* scenario consists of building instantly an entire HPC cluster on the Amazon EC2 cloud for any duration of time. The created cluster has a front-end node and several computing nodes which can communicate in an isolated private network. Instance type of the cluster nodes can be configured based on the provided EC2 instance types. The *Cluster-On-Demand* feature is ideal for users without HPC facilities and which need running temporary parallel computations.

The *Cluster Extension* scenario concerns administrators of cluster sites who want to address the periods of peak-demands of computation. This can be done by offloading some part of workload on the cloud. *Cluster Extension* feature extends the number of regular nodes of a given cluster by adding extra computing nodes from the EC2 cloud data-centers, including GPU nodes. The front-end of the cluster remains outside the

cloud and one can use *Bright* GUI to manage all the computing nodes as if they are part of the local cluster.

User uses the command-lines of the local batch system to submit and monitor parallel jobs on the local nodes of the cluster. Jobs destined to be run on the cloud resource should be submit using the `cmsub` command. This command is installed on the front-end node of the cluster. The cloud nodes are grouped within queues. User needs to precise the target queue in the `cmsub` command to submit a given job: this allows the job to be considered for running of the cloud nodes. Data transfer of the input files of a given job into the cloud storage system is transparent for the user. The Bright system ensures that the data is available before the job starting in the cloud nodes and that the job output files are retrieved back.

Bright cloud solution is one of the promising step towards democratizing the use of cloud in HPC services. For many workloads types, this is an interesting solution for administrators of HPC centers and for industry. Currently, Amazon EC2 offers interesting *cluster compute Instance* types to meet the HPC community requirement in terms of CPUs power and I/O disk access speed. However, the only difference between the cloud-cluster mounted by Bright and an ordinary HPC-cluster is the network. For example, the 10Gb Ethernet offered by EC2 can not compete with the native InfiniBand speed for example. For communication-intensive applications, processes may have to wait to receive data because of a modest network performance, which results in a waist of CPUs time and resources.

Bright intends to extends its cloud solutions to other cloud platforms. Currently, the private cloud platforms that have gotten the most attention are *Google-App Engine* (GAE) [68] and *Microsoft Azure cloud* [104]. EC2 and Azure are considered as Platforms as a Service (PaaS). Both GAE and Azure offer a high-level services such as an integrated runtime environment that automatizes the deployment of web based applications on virtual machines and that scales them up during peak of traffic. Users don't have a full control on the virtual machines. In addition, GAP offers an efficient solution of storage system on the cloud for big data. Amazon EC2 is seen as a Infrastructure as a Service (IaaS): a user can start a set of virtual machines and have a full control over them. Azure has the particularity also to offer this feature to the end-user. Interaction between the Cloud providers and the users is done either through rich web based interfaces or programming APIs.

1.3 Overview of running distributed applications

In this section, we discuss the distributed and concurrently running applications and how they have evolved over the years. The parallel and distributed applications can run over parallel and distributed resources provided that there are mechanisms to coordinate

communication between the tasks in an efficient way.

1.3.1 MPI

One of the efficient programming model is the Message Passing Interface (MPI). The MPI model is widely used in scientific and engineering domains. It allows efficient parallel programs to run on clusters nodes and multicore machines interconnected with a high speed network connection [73]. There exist several implementations of this model in `C/C++` and `Fortran` language such as OpenMPI [116] and MPICH [109]. However, this paradigm requires high skills in parallel programing where the parallelization and communication operations are explicitly handled and coded by the programmer. Also, it can be only used to be run on cluster or massively parallel machines equipped with high-end and high-bandwidth network.

1.3.2 Shared memory

A shared-memory programming model is also available to take advantage of multicore machines that share the same memory space (see for instance OpenMP [34] and Cilk-Plus [40]). This mode enables a promising and easy programming model for end-users. However, it targets only on multicore machines. Some other works investigated the possibility to move the parallelization paradigm to the compiler (Aka. vectorization), but its efficiency over traditional compilers is only relevant to specific applications [94].

1.3.3 GPGPU

General-purpose computing on Graphics Processing Units (GPGPU) is a promising programming techniques for high performance computations. It consists of a tandem-work between the graphics processing unit (GPU) and the CPU. GPUs are a quite an exciting subject for HPC as they can accelerate an application by executing some intensive portions of its code, while the CPU runs the remaining ones. CUDA [113] (proprietary of NVIDIA [112] company) and OpenCL [110] are two libraries to code applications to be run on GPUs devices. They proposed two different programing interfaces but have similar features.

1.3.4 Cross-site MPI

Face to the expansion and variety of computing resources, new ways to do distributed execution of applications have emerged. The message passing model is applied on distributed computing nodes connected through regular Internet connection: this is known as cross-site MPI runs. Many technical difficulties may be raised for a cross-site

MPI run. For instance, MPI processes are located at different clusters sites and have to communicate. These sites may be placed behind a network Firewall and have strict security policies which limit the attribution of public IP addresses ans opened ports to the computing nodes. Thus, it is a challenging task to provide a transparent mechanism for inter-sites process communication with a lower performance overhead. In addition, computing nodes and network devices may be heterogeneous which requires a high-level programming model to hide this complexity for the programmers.

PACX-MPI [64] is one of the first attempts. It does not requires that all computing nodes have public IP addresses. However, inter-sites data communications are relayed through central servers running on each site.This increases the communication latency and makes PACX-MPI only suitable for applications with non-intensive communications. MPICH-G2 [90] and MPIg [71] are two successful attempts that used the Globus Toolkit services [62] to perform cross-site executions. They enable the user to run parallel code across multiple and heterogeneous clusters sites using the same command as in parallel machine. They mask details about the architectures of nodes, software systems, authentication mechanism, etc. They allow the user through the `mpirun` command to start parallel processes. MPICH-G2 and MPIg requires that all the participating nodes have a public IP addresses to communicate with each others. QCG-OMPI [4] is another tool which uses the QosCosGrid [11] grid-level extension of its runtime environment to address connectivity issues between cluster sites. It supports MPI run across a federation of clusters. Computing nodes do not require to have public IP addresses. QCG-OMPI allows direct communications between processes whenever it is possible. In the case of Firewall restriction, communications are relayed through a proxy service installed at each cluster site. In the same perspective, MPWide [71] is a `C++` light-weight communication library for performing message passing between supercomputers. This API is easy to install and provides a superior communication performance compared to the previous cross-side tools. MPWide connects different MPI applications at compile time, each is installed on a separate site (for example supercomputer or HPC cluster) and runs with a locally recommended MPI implementation. In MPWide, the communication is based on channels. A channel is a bidirectional TCP socket communication between two ports of two different machines. MPWide provides functions that can operate concurrently on multiple channels. For instance, a process can send a data to multiple inter-site processes simultaneously using the `MPW_Send()` function or gather data from them using the `MPW_Recv()` function.

1.3.5 Distributed computation

Many works have been done to give scientists easy tools to perform distributed computations. Several distributed systems have been designed to run bag-of-tasks based application, namely Condor [97], Nimrod-G [31]. These systems are well suited for parameter-sweep applications where tasks are independent and operate on different

input files. Condor federates machines across a campus-wide network and provides an API to easily run a bag-of-tasks on those machines. Condor has been also dapted to the grid through Condor-G [63]. The execution of embarrassingly parallel tasks has been pushed further with the apparition of Grid computing concept. This allowed to enhance the execution performance especially for the transfer of large data and the security to access computing resources. Several Grid-middlewares (Globus [5], gLite[49], ARC [122, 9], UNICORE [131], etc.) were developed to facilitate submitting grid tasks on remote machines. Grid systems are well suited to federate HPC-clusters and run MPI-tasks over them. Tools to transfer large data between computing sites have been developed like gridFTP [5]. In parallel, a new concept called Desktop-grid systems (or Volunteer systems) emerged. This concept targeted running bag-of-tasks on normal Desktop machines belonging to individuals and/or institutes. SETI@Home [8], BOINC [25], XtremWeb-CH [16, 106] are examples of platforms that implemented this concept.

Recently, the Big-Data concept has gained huge attention from researchers, companies and governments. Companies like Facebook [59] is struggling to process the very big data of users that are stored worldwide. New tools to facilitate the process of big data are developed such as Hadoop [76] (based on Map-Reduce model) and DryadLINQ [151]. They are efficient and easy in term of coding and learning efforts and enable processing large data sets across clusters of servers. Recently, the scheduling framework of Facebook which uses Hadoop as back-end reached its performance limits. To resolve this performance issue, Facebook developed a new Hadoop version called Corona [42], aimed to be the next version of Map-Reduce model with better performance.

In addition to the tools, several algorithm skeletons exit for parallel and distributed programming. They have as objective to reduce programming difficulties by reusing predefined templates. The work in [67] proposed a classification of skeletons based on their functionalities:

- Data-parallel skeletons: they operate on data bulk structures. Data bulks are manipulated by operations such Map, reduce, Fork, join, etc. Communication deadlock can not occur in this skeleton category.
- Task-parallel skeletons: they operate on tasks. Interactions between tasks determines the skeleton behaviors. There is no distributed data structure to treat in parallel. Instead, communication mechanism between a set of processes should be established to resolve a given problem. For instance, Task-Farm, Pipe, For, while are example of task-parallel skeletons.
- Resolution skeletons: define certain techniques used to resolve problems. Their usability depends on the nature of problem to resolve. For instance, we can site the Divide-and-Conquer and Brunch-and-Bound algorithms.

1.4 Conclusion

In this chapter, we have presented different infrastructures and research projects individually. We have described the transition from DEISA to PRACE and from EGEE to EGI. Regarding the computing infrastructure, DESIA, PRACE, HPCI and XSEDE can been seen as a federation of distributed supercomputers accessible for the end-users through the same mechanism (single-sign-on). Similarly for EGEE and EGI infrastructures that were destined in the beginning to support data-intensive applications on clusters. Projects like StrastusLab, VenusC, EDGI and Magellan pushed further the adoption of the cloud computing concept towards HPC in the academic domain and industry. Moreover, there is an increased growth of the cloud in the marketplace. We estimate that cloud computing will become more prominent in the future with the rapid growth of data centers and providers.

Several submission mechanisms are supported either through command lines (batch system), grid clients or web/portal interfaces. DESIA/PRACE, EGEE/EGI, HPCI and XSEDE supports both direct submission from the master node (batch systems) or through grid clients tools. Public cloud platforms such as EC2 and Azure provides programming APIs, command line tools and portal web interfaces to run and manage virtual machines. Scheduling and managing of jobs are done on the client side through developing a client program that submits and handles jobs. EC2 and Azure support also building HPC infrastructures (clusters) using cloud their resources.

Linking between HPC infrastructures and research projects is also to consider. For instance, collaboration between PRACE and XSEDE started at 2012 through annual HPC summer schools and it is now exploring an open initiative to support collaborating research teams in Europe and US. A first collaboration consists in establishing an interoperability between the two HPC infrastructure services, which may also includes a possibility for testing a shared resource allocations mechanism. Collaborations between PRACE and other HPC projects such as HPCI will also take place in future.

Classic bag-of-task applications have been run distributed for long time, and tools such Condor have been well equipped to run those applications. Other tools under the umbrella of Map-Reduce model have emerged. Their aim is to facilitate the processing of big data (for instance Hadoop and Corona) by providing efficient and easy programming environments in term of coding and learning efforts.

For concurrent parallel applications, the PVM concept appeared in 1989 is still applied now through MPI. This concept can be seen as a standard for developing and running parallel applications. It consists of homogeneous nodes that are connected with each other: each node has a memory and may be multicores. For large-scale distributed applications case, there is no equivalent to the MPI standard. There have been numerous tools in the past to enable MPI cross-site runs (such as MPICH-G2 and QCG-OMPI). But, without

exception they had major constraints in the deployability and could only be installed on closely-managed or closely-monitored (and usually experimental) infrastructures. MPWide was among the first attempt to provide a more flexible and easily deployable environment at the expense of a bar-bones interface.

This reflects clearly the absence of a common standard to develop and run large scale distributed applications over several distributed infrastructures. One possible solution to this problem is to define a set of modeling and programming paradigms that separates the modeling of applications and their development and deployment. This idea will be further illustrated in the next chapters of this thesis.

2 e-Science applications and computational workflows*

This chapter presents six scientific applications that have been ported and deployed on parallel and distributed platforms. This work was done under this thesis. All these applications are CPU consuming; they belong to different scientific domains. Four of these applications have been deployed on distributed computing environments (Grid, cloud and Desktop grid platforms). The other two applications are massively parallel, multiscale in nature, programmed with MPI and run on parallel HPC platforms (supercomputers and clusters).

The development of these six applications has allowed us to develop expertise in deployment and porting scientific applications and highlight some learned lessons. This accumulated expertise has led us to move from customized solutions of porting applications to more general ones which will be described in details in the next chapters.

This chapter is composed of six sections that describe the six applications. Each section presents the field of use of the application, its global architecture (algorithm) and how the application has been ported on the distributed computing environment.

*This content of this chapter is based on:

- Mohamed Ben Belgacem, Nabil Abdennadher, Marko Niinimaki. Programming distributed medical applications with XWCH2. In *Proceedings of HealthGrid 2010*, volume 159, France, June 2010.
- Mohamed Ben Belgacem, Haithem Hafsi, and Nabil Abdennadher. A Hybrid Grid/Cloud Distributed Platform: A Case Study. In JamesJ.(JongHyuk) Park, HamidR. Arabnia, Cheonshik Kim, Weisong Shi, and Joon-Min Gil, editors, *Grid and Pervasive Computing*, volume 7861 of *Lecture Notes in Computer Science*, pages 162–169. Springer Berlin Heidelberg, 2013.
- Mohamed Ben Belgacem, Nabil Abdennadher, Marko Niinimaki. *Desktop Grid Computing*, chapter The XtremWebCH Volunteer Computing Platform (3). Numerical Analysis and Scientific Computing Series. Chapman and Hall/CRC, June 25, 2012.

2.1 MetaPIGA

MetaPIGA [79] is a phylogenetic application developed at the university of Geneva. It builds the optimal phylogeny tree (tree of life) from deoxyribonucleic acid (DNA) sequences of several biological species to study the evolutionary relationships among them. MetaPIGA uses advanced tree-building methods where nodes are called taxa (populations of organisms). The edges distances between species bring more phylogenetic insights like detecting paralogy relationships among genes, tracing the evolutionary history or relatedness of species, studying molecular characteristics, etc [12].

The objective of MetaPIGA consists in finding the most suitable individual, i.e., the phylogenetic tree that has the greatest probability of having generated the observed sequence of genes of the different taxa (called also the greatest Likelihood).

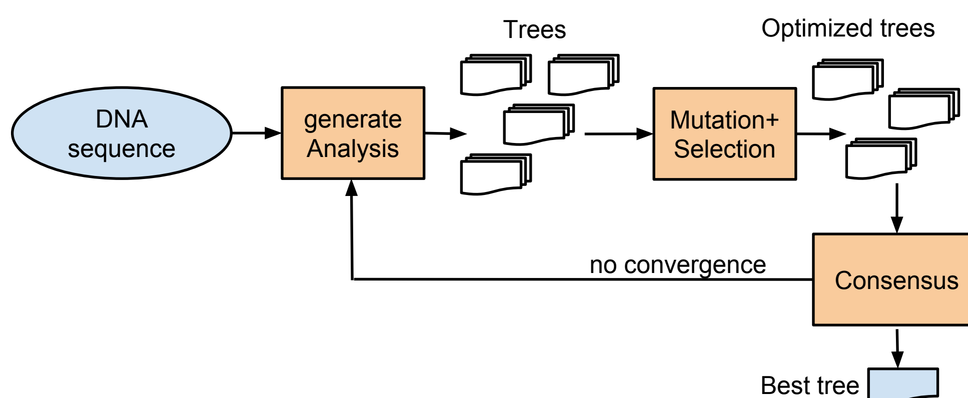


Figure 2.1 – MetaPIGA computation process

Figure 2.1 explains the computation steps of the MetaPIGA application. We start from an input dataset, i.e., a file that contains an alignment of DNA (nucleotides: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T)) sequences from different taxa. Then, the program generates R independent analyses (R is usually > 500). Each analysis is composed of P independent random populations of trees. During computation, these trees will undergo mutation and selection transformations. In the end, a set of solution trees generated from all analyses are merged into a consensus tree. It is worth reminding here that we cannot know in advance the number of analyses needed to obtain the final solution: MetaPIGA will continue generating analyses until the consensus process converges. The execution time of each simulation depends on the dataset size.

MetaPIGA is an embarrassingly parallel application. Its workflow is a Map-Reduce like model. However, the number of jobs and the convergence condition are not known in advance. Each job has its own input dataset and generates as output a set of solution trees. There is no communication between jobs. MetaPIGA has been ported on the XWCH Desktop grid [16, 106], cluster and cloud environments and Figure 2.2 presents

the workflow of the application. Each job (J_i) is assigned to a worker (machine). On a worker machine, a job J_i randomly generates P populations. Each of them is composed of N trees. Then, for each population, the best tree (under the maximum likelihood criterion) is kept while the other $N-1$ trees are subjected to mutation events followed by a selection transformation. This process is repeated until a global optimum is found, i.e., the current optimal solution can no longer be improved. When a job J_i is finished, its output file is downloaded from the computing resource and its result trees are added to the consensus tree (job C_i). Since the number of jobs required to converge can not be known in advance, one can submit simultaneously a very big number of jobs on the computing platform. Then, once the convergence is reached, the remaining jobs will be killed. Because of performance and reliability considerations, it has turned out to be more efficient to progressively submit bags of jobs in order to not overwhelm the computing platform.

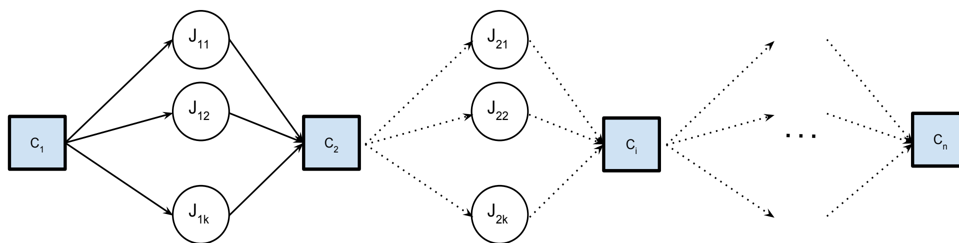


Figure 2.2 – Distributed workflow of MetaPIGA

The main aim of using large distributed systems is to accelerate the computation analysis. We present here some experiments outlined through comparisons with a respect to a sequential execution on a single machine, dedicated multi-core machine and a Desktop grid platform (XWCH [16, 106]).

We started from a small dataset composed of 15 DNA sequences (1314 nucleotides). For the Desktop grid computation case, we have launched the execution with a number of jobs ranging between 1000 and 10000. Execution times are presented in Table 2.1.

Table 2.1 – Execution time of MetaPIGA.

	Cores number	Execution time
Desktop grid (XWCH)	300	50 min
Multicore machine	16	180 min
1 machine	1	1050 min

The number of used machines on the XWCH Desktop grid platform is around 300. The corresponding execution time took 50 minutes. This is therefore more than 20 time faster than an execution on 1-core machine (Intel core 2 duo E8200 2.66GHz with 3GB as memory). On a 16 cores machine, the execution of the same dataset took 180 minutes which is 6 time faster than an execution on a 1-core machine. Here we can observe that

even with 16 cores machine we didn't have a linear speed in execution time. On one hand, if we consider a medium size of input data, each single machine in the Desktop grid platform needs around 1.5GB of memory to perform an analysis. This memory size is very common today. On the other hand, a multi-core machine will need around 24GB for a big dataset size, which is very expensive to obtain, and consequently, it will be outperformed. Compared to a cluster, the Desktop grid platform provides an easier and cheaper way to run computations with comparable performance.

2.2 NeuroWeb

The NeuroWeb application is developed at the university of Applied Sciences-Western Switzerland. It aims to build neuronal maps of brain activity using non-invasive measurements [106, 17]. It is based on "extracting" the activity of the different regions of neurons from captors attached on the scalp of the patient. Measurements of the internal cerebral activity around the scalp are obtained, for example, from a *Magnetoencephalography* (MEG).

Neuronal activities, called generators, are electromagnetic fields captured by the MEG captors. These fields are extremely noisy and disturbed by the cell tissue. In essence, we have a lot of generators ($\sim$60.000) but very few captors ($\sim$200). The difficulty, and hence the computational requirement effort, is to construct a robust and efficient estimator for a neuronal map.

The estimated images of brain activity are a solution to a high-dimensional, non-differentiable optimization problem. The size of the solution space is so large (about 10^7) and requires several days of intensive calculations. Finding the solution to this mathematical problem can be done efficiently using distributed computing. The practitioners will then be able to virtually cut the cortex and identify sources of brain "abnormal" activities such as epileptic crisis, Parkinson, Alzheimer, etc.

NeuroWeb is assumed to produce a dynamic neural map, DNM, in which each row represents the electromagnetic activity of one region of neurons. With 60.000 regions and 1500 measures (assuming that the duration of the experiment is 1.5 second and one measure is taken every 1 millisecond), the size of DNM is greater than 240 MB.

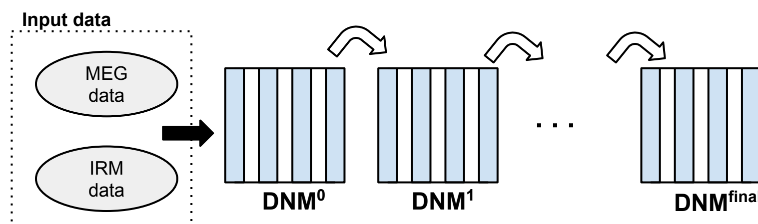


Figure 2.3 – Neuroweb computation algorithm

The application starts by choosing a random map called DNM^0. At each iteration i, a new matrix DNM^i is constructed based upon DNM^{i-1}. The iterations stop when, the DNM^i matrices can no longer be improved, i.e., when convergence is attained.

NeuroWeb has been ported on the XWCH Desktop grid platform using a dedicated XWCH API. The distribution version of NeurWeb consist in splitting the DNM matrix into several blocks B_j as depicted in Fig 2.4. Each block is processed by a daemon process, called "persistent serve" (PS), which executes an iterative asynchronous algorithm. During its execution, a PS, processing a block B, receives input data from its PS neighbors (PSs that process blocks adjacent to B). However, it is worth reminding here that the generation of a block B by a PS can take place even if no input data are received from the neighbors of PS. Indeed, PS is an iterative asynchronous algorithm, which continues improving its block B even if it does not receive any data from its neighbors. In this case, a PS will take more time to converge and thus computation time will increase accordingly.

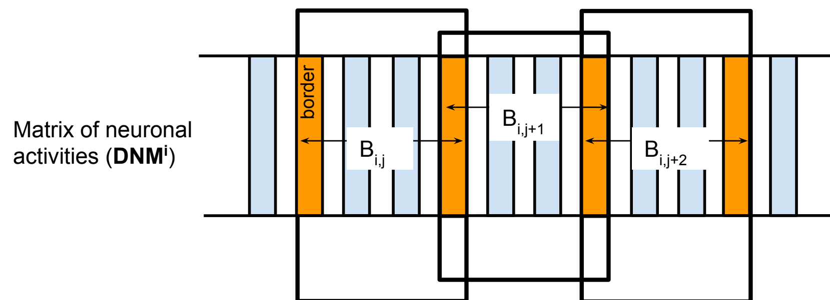


Figure 2.4 – Splitting of the matrix of the neuronal activities DNM^i

On the other hand, a central persistent server (CPS) is deployed in order to gather blocks from all PSs and decide if convergence is reached or not. If the convergence is not yet reached, the CPS launches a new iteration. The workflow computation of NeuroWeb is illustrated in Figure2.5. The tasks G_i are processed by the CPS. During each step, each block B has to be processed by the same PS (the same worker machine) and all the tasks $T_{i,j}$ exchanges border data with its neighbors.

It can be deduced from what is presented above that:

- The number of steps and the workflow of NeuroWeb cannot be fixed before the execution,
- Persistent servers (PS) are daemons processes which must continue their execution and reside on main memory of the worker machines until convergence is attained,
- During its execution, a given PS_j executes an iterative asynchronous algorithm and exchanges data with its neighbors PS_{j-1}, PS_{j+1} and with the CPS.

In this context, the XWCH platform is used to:

– Create the persistent servers (PS_i) on the workers
– Feed the PS_i with necessary data,
– Feed the CPS with data to decide if convergence is reached or not
– Stop the PS_i and CPS.

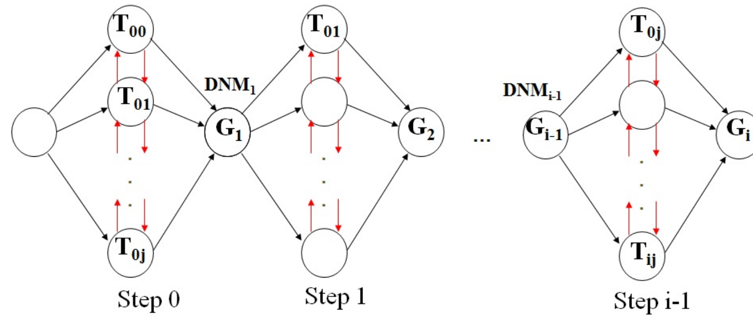


Figure 2.5 – Workflow of NeuroWeb

A first implementation of NeuroWeb is detailed in [16]. In this version, the submitted jobs generate one output file per PS. When the CPS receives an output file from a given PS, it decompresses it and decides to which PSs the contained files should be dispatched. This version overloads the CPS.

In order to overcome this problem, we developed a new “decentralized” version where jobs linked to a given PS, generate several output files each of them is intended to one PS. Contrarily to the first version, jobs do not send the whole output files to the CPS, but only their reference (name of the file, worker and warehouse identifiers where the file is stored).

2.3 Selector

The Selector [48] application is a C++ based application developed at the university of Geneva. It is used for the reconstruction of modern humans’ prehistoric migrations in a given geographic area. This is performed through simulating the migration of individuals among time, as well as their genes. These simulations take into account several factors such as effects of demographic growth, geographic barriers and natural selection. For instance, Selector can be used to estimates the migration of people in East Asia. Based on of the observed “human leukocyte antigen” (HLA) genetic data, Selector can validate one of these two hypothesis: i) the population of South and North East Asia originate from one population; ii) they originates from two different populations.

Analyzing the HLA data observed in the East Asiatic population enables to describe in details the genetic data of this wide region. Despite that, scientists need further informations to explain the evolution of the human settlement. Several factors like natural

selection and demography can influence the evolution of HLA genes. For instance, an intensive genetic flow between populations could maintain a high genetic diversity while a rapid genetic drift or purifying selection could lead to a loss of genetic diversity [77]. The challenges remains to determine the effect of each factors on the migration history of the settlement of this region. Study based on statistical analyzes of observed genetic from contemporary settlement is not enough to explain the migration history of the population. For that, new approach based on numerical models is used for this purpose.

The Selector application consists of two successive steps: simulation and estimation steps (Figure 2.7). The simulation step launches n computations $\{c_i\}_{1 \leq i \leq n}$ based on a numerical settlement model. First, each computation c_i deals with a demographic scenario which uses different parameters values of that settlement model. Then, each computation c_i generates a genetic simulated data D'_i. The estimation step uses all the simulated data to determine the optimal values of the model's parameters that may have generated the real genetic data D (the observed genetic data of a given population at a given region). To do so, all the simulated data $\{D'_i\}_{1 \leq i \leq n}$ are first merged in a structured text file. Then, only a fraction of best simulations data are selected by comparison to the observed data D. These selected data are converted to statistical data and used to estimate the optimal values of the model's parameters (estimation step in Figure 2.7).

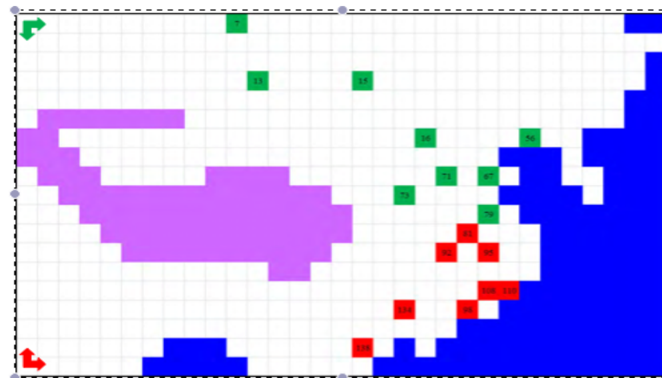


Figure 2.6 – Numerical map of the Eastern Asia used for the simulation [48]. Each cell has a surface of $40'000$ km^2. The geographic repartition of the selected population is marked by colors. Green (resp. red) cells correspond to a selection of population from the North (resp. South) East of Asia. Red and green arrows represent the starting cells of the migration phenomena.

The first simulation step is a CPU and time consuming task. Consider applying the Selector application on the population of the *Tibetan Plateau* in the Eastern Asia. In order to prepare this simulation step, a numerical map of the *Tibetan Plateau* is first prepared. The map is split into a lattice of cells. Each cell is assumed to be populated by number of residents (individuals) that reflects the demographic maximal density over a surface of $40'000$ km^2 (Fig 2.6). Each computation c_i is independent and runs the settlement model on the numerical map. A computation consists of an iterative algorithm which

starts from the cells located in the west or the both sides of the *Tibetan Plateau*. Each iteration produces a generation, where the individuals in each cell are replaced by their descendants allowing a dynamic growth of the population. During each iteration, each individual has the ability to move from his cell to the neighbors cells.

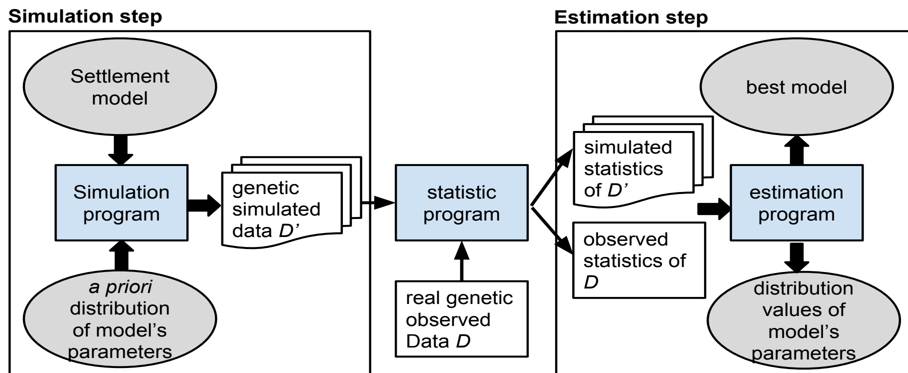


Figure 2.7 – Simulation of a settlement scenario [48].

This numerical simulation approach requires a large number of computations (between $100'000$ and $1'000'000$ for each hypothesis). For instance, running all the computations $\{c_i\}_{1\leq i\leq n}$ on one machine (2GB of memory, and 2.2 GHz CPU) may take between 100-200 days of execution. In this case, a high performance computing platform is thus highly necessary. To do so, the Selector application has been ported on the XWCH Desktop Grid platform. Since the computations are independent, the workflow of the distributed version of Selector follows a simple Map-Reduce architecture. Determining the optimal parameters of one settlement model requires n computations, where $100'000 \leq n \leq 1'000'000$. Given that the average time of one computation can be averaged to ~ 300 seconds on a local machine, each submitted grid job J_i will process m computations where $m = \frac{n}{k}$ and k is the number of used Desktop machines of the XWCH platform.

We have run $68'000$ computations over 100 Desktop machines. The execution time of the application was $276'769$ seconds (~ 22 days): this is therefore 74 time faster than a simulation on one single machine.

2.4 GIFT

The Gnu Image Finding Tool (GIFT) [132] is an application that was developed at the university of Geneva. It consists of a content-based image indexing and retrieval package. It enables users to submit image queries on a specific images databases to retrieve the ones having a similar content. Processing these queries relies basically on the images content (extracted visual features) and thus avoids the need to annotate them before. GIFT comes also with an indexing tool that allows extracting several images features like texture and colors and store them into an inverted file database. This enables a fast

searching and queries processing.

The GIFT application has been adapted to the medical context [132]. This can importantly helps doctors to find similar medical cases of certain diagnostics based on visual informations like in dermatology. However, the amount of produced medical images in hospitals is quite large. For instance, statistics done in 2007 shows that the radiology department of the Geneva university hospitals produces about 70'000 images per day [132]. Given that a typical desktop computer requires 2 seconds to process one image, it is therefore clear that the required time to extract features in a large database images would be not reasonable without parallelization.

Indexing images does not requires considerable work to parallelize tasks. Thus, the workflow computation follows the map-reduce model. Simply, indexing an image collection C can be done by splitting it into n sub-collections $\{c_i\}_{1 \leq i \leq n}$. Then, each sub-collection c_i will be processed independently on a machine or CPU. In the end, the outputs are combined in the same database as described in the Figure 2.8.

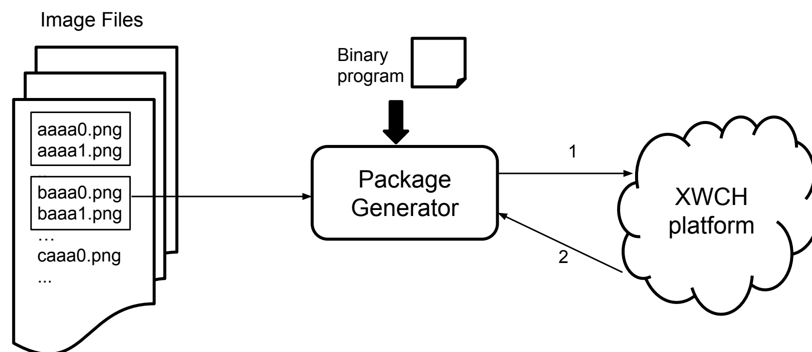


Figure 2.8 – Distributed workflow of Gift

2.5 CleanCity

CleanCity is a collaborative project between the university of Applied Sciences in Switzerland and the *territorial information system* administration in Geneva. Its aim is to to provide decisional helping software in the domain of “urban climat”. For instance, this application enables the urban engineers to predict the impact of several factors (such as the air quality, dispersion of pollution, temperature, etc) on the urban cities. To do so, the application performs Computational Fluid Dynamic computation in order to simulate the dispersion of pollution in the air. The *Quartier des banques* city in Geneva is taken as a testbed.

A numerical simulation of a city undertakes several steps. First, a 3D geometry model of the urban city is built (Figure 2.9) using image from small drones. This 3D geometry is used as input file to define the spatial domain of the numerical simulation. The

Figure 2.9 – A 3D model of a city that takes into account chimneys and roofs [80].

application code is parallelized and run on grid of clusters. In addition to the results of simulations, a real 3D mockup of the city is build of a geometry scale of $\frac{1}{1000}$ and a real simulation is performed in a wind tunnel. Comparison between numerical and real simulation allows to verify the accuracy of the obtained results. A numerical simulation of an urban city is difficult to perform since it requires an adequate initialization of boundary conditions and uses a big number of HPC resources to run.

Since simulation requires supercomputer capability (like PRACE [121] and EGI [53] machines) that can not be afforded in all research institutes and administration, the idea is to slip the 3D model of a given city into several block $\{B\}_{1\leq i\leq k}$ (Figure 2.10). Each block B_j is an MPI code which will run on a cluster machine and exchange boundary data with its neighbors during each iteration of the simulation.

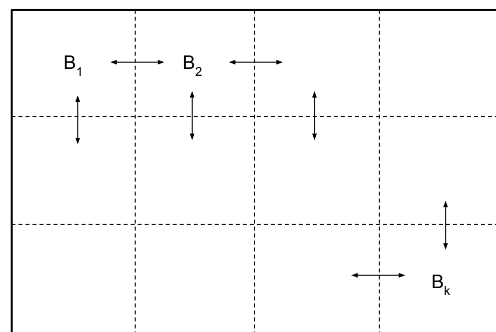


Figure 2.10 – partition of the 3D geometry domain of the city in several blocks.

CleanCity has been ported on HPC grid of clusters and cloud platforms. The application was developed based on the *OpenFoam* [115] software. The runtime environment of OpenFoam has been enabled on two clusters based on *ARC* middleware [122, 9]. Only simulations of separated blocks without communication have been carried out (see Table 2.2).

Table 2.2 – Execution time of CleanCity test case on different resources: simulation of the *Quartier des Banques* city in Geneva (1200 iterations).

Resources	Configuration	Execution time (hour)
Gordias cluster	28 nodes (244 cores) , 8 GB pf memory per node	9.8h
Multicore machine	24 cores, 100 GB of memory	2.64h
Amazon-EC2	1 node with 32 cores, 64 GB of memory	4h

2.6 Hydrology

Canals or rivers are important in populated area as they ensure an adequate supply of water for agriculture and are a key component of electricity production or transportation. An optimal management and the control of such resources can be a critical issue for long term planning or to react to natural hazards. The major challenge is to define appropriate actions (e.g. opening and closing gates) that need to be taken in order to guarantee an adequate water supply throughout the canal system, whatever the external demands or perturbations can be, and respecting constraints such as water height.

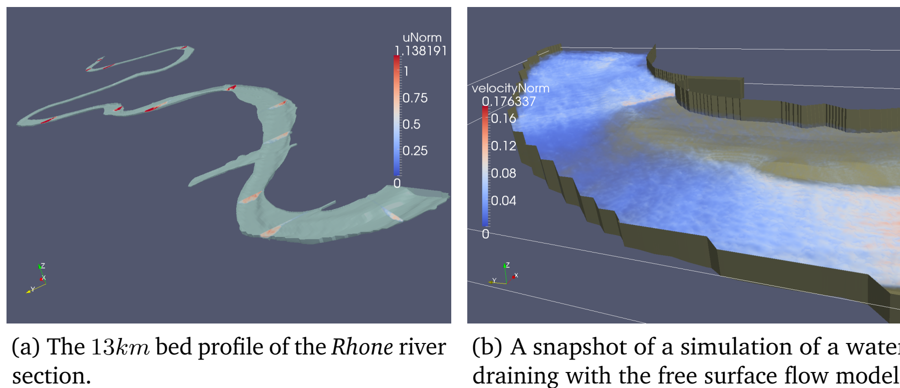
The first hydrology application is a model of the canal de la Bourne irrigation network (see Figure 2.11)

Figure 2.11 – The *La Bourne* irrigation canal (*Valence* in France).

This canal was built in $19\ th$ century and it is still in use now. It is composed of several parts along $\sim 400km$ of length. This canal includes several *water junctions* such as gates, spillways, pumping station, reservoirs, etc. The main goal is to to define appropriate actions (e.g. opening and closing gates) to ensure an adequate supply of water to the population and farmers. In addition, a real-time-control is required to optimize the water

exploitation.

The second hydrology application is to simulate a section of the Rhone river (13 km), from the Geneva city down to Verbois water dam. In collaboration with the Geneva electricity company, we are investigating the possibility to study specific questions, namely, the way to flush sediments in some critical area by changing the water levels.



(a) The $13km$ bed profile of the *Rhone* river section.

(b) A snapshot of a simulation of a water draining with the free surface flow model.

Figure 2.12 – The *Rhone* application.

This class of applications can be addressed through numerical optimization methods. Usually, these methods require the simulation of different scenarios with the canal network subject to different parameters. Therefore numerical methods able to simulate the water flow in a full irrigation system are needed. In order to allow canal operators to respond to real-time events, these methods should compute fast enough, with good accuracy. Due to the size of an irrigation network and the large variation in the flow complexity across different sections, a multi-scale, multi-model computational approach is needed. For instance, some parts of the system can be described with simple numerical methods, using the shallow water equation, whereas other sections need a 3D, free-surface hydrodynamic model (3DFS) to properly capture the flow properties. This application will be described in more details in chapters 5 and 6.

2.7 Conclusion

This chapter presented six scientific applications that have been ported on a large distributed computing system. These use-cases show clearly the need to come up with more general approach in order to design a distributed computation, especially to support the non-deterministic convergence and the multiscale computational aspects.

Some works had proposed classifications of parallel and distributed scientific workflows. For instance, [150] proposed a classification of the scientific workflow systems in the context of grid computing based on four criteria: *workflow design*, *workflow scheduling*, *fault tolerance* and *data movement*. From the *Workflow design* perspective, distributed and

parallel workflows can be classified into two categories: a Directed Acyclic Graph (DAG) or a non-DAG. Tasks in a DAG and non-DAG based workflow can be either sequential, concurrent or iterative (i.e. a section of a workflow tasks are allowed to be repeated). The work in [124] proposed a classification based on several scientific applications coming from several scientific domains. The classification proposed the following patterns:

1. *sequential*: the workflow tasks run in sequential,
2. *parallel*: the workflow tasks run at the same time,
3. *parallel-split*: one task of the workflow feeds the other tasks,
4. *parallel-merge*: multiple tasks merge into one task,
5. *parallel-merge-split*: a combination of *parallel-split* and *parallel-merge*,
6. *mesh*: tasks run in an interleaved manner.

Based on our experience and contribution in the development of the six applications described in this chapter and of other HPC applications, we propose in the next chapters a classification of the scientific applications relying on their computational workflow type and targeted computing systems.

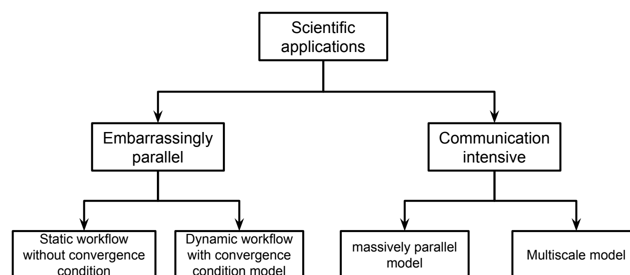


Figure 2.13 – Classification of the scientific applications based on their computational workflow type and targeted computing systems.

In this context, we pointed out two different classes of applications as depicted in Figure 2.13. The first class is the “embarrassingly parallel” applications characterized by loosely coupled jobs. These applications can be executed on off-the-shelf resources like cloud and Desktop resources. We find in this class two categories. The first category concerns the applications that have a static workflow such as Map-Reduce and Task-Farm models. In this category, the number of tasks is fixed in advance and the computation is not handled through a convergence condition: Selector and Gift are good examples of this family of applications. The second category concerns the applications that have more complex features. For instance, the number of tasks composing the workflow may not be known in advance and the computation can be based on a non-deterministic convergence condition: MetaPIGA and NeuroWeb are good examples of this family of applications. The second class is the “communication intensive” applications where jobs communicate in a

tightly coupled way and require parallel environment installed on computing resources. We find in this class two categories. The first category concerns the massively parallel applications which require HPC resources to run such as multicores machine, cluster and supercomputer. The second category concerns the multiscale applications: the CleanCity and Hydrology applications. These applications are multiscale in nature and thus require sophisticated methodology to design, program and deploy them on distributed HPC platform.
These two classes of applications reflect nevertheless challenges in the conceptual and implementation phases. In the conceptual phase, difficulties are related to the degree of complexity to model a distributed version of a monolithic application and run it on different distributed infrastructures without heavy modifications. In the technical challenges, the concept “write once, execute anywhere” is difficult to achieve given the lack of interoperability and portability features among the existing computing infrastructures.

The methodology of porting applications belonging to the “embarrassingly parallel” class will be presented in the next chapter. The one belonging to the “communication intensive” class will be detailed starting from chapter 4.

3 High level paradigms to develop embarrassingly parallel applications*

3.1 Introduction

This chapter focuses on the *embarrassingly parallel* category of scientific applications (Figure 2.13). Specifically, it targets the applications having a dynamic workflow with convergence condition.

Currently several parallel programming models and paradigms exist to port applications on HPC infrastructures such as Message Passing Interface (MPI) [73], shared-memory model [34], vectorization [94], etc. Another interesting alternative to port applications on distributed parallel infrastructures is the "programming skeleton algorithm". It enables users to develop programs by composing a set of skeletons [67]. There exist several skeleton libraries that aims at writing efficient parallel programs, for instance, *Skandiuam* [96] and *Cilk Plus* [40]. GC3Pie [126, 43] is a High-Throughput framework to run parallel and large distributed application over HPC systems. It is a library of Python classes used to remotely run and monitor jobs on diverse batch executing systems (e.g., *Torque* [143], *Slurm* [129]), *ARC* grid midlleware [122] and cloud platforms (e.g., OpenStack [118] and Amazon-EC2 [6]). The main objective of GC3Pie is to relieve the complexity of executing and monitoring of jobs on HPC resources and move the decision making logic to the application level. Using GC3Pie, one can easily write a Python program that generates a dynamic computational workflow through using loops and conditionally branch executions for example.

*This content of this chapter is based on:

- M. Ben Belgacem and N. Abdennadher. Skeleton paradigm for developing e-science applications on distributed platforms. In *The 6th IEEE International Workshop on Multicore and Multithreaded Architectures and Algorithms (M2A2 2014)*. IEEE Computer Society, 2014.
- M. Ben Belgacem and N. Abdennadher. Towards a high level programming paradigm to deploy e-science applications with dynamic workflows on large scale distributed systems. In *The 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (IEEE/ACM CCGrid 2015)*. IEEE Computer Society, 2015. Accepted (10 January 2015).

Even with the existing algorithmic skeleton, it seems to be difficult without the help of experts to easily write an application workflow and handle the convergence verification. In what follows, we note these applications as “dynamic workflow (DW) e-science applications” which stands for applications having the following features:

1. An iterative model based on a master/slave or Map/Reduce architecture.
2. The number of iterations and slave jobs per iteration are not known in advance.
3. The master job submits slave jobs, provides them with input data, retrieves their results and checks if the convergence condition is reached.
4. Slave jobs receive input data from the master job, process them and return back the results.
5. Slave jobs could be persistent or non persistent. Persistent jobs remain running after sending their results to the master job. Persistent jobs continue generating new results even if they do not receive new input data. Communications between two persistent jobs can occur in order to exchange boundary data, whereas non persistent jobs stop running after sending their results. At this stage, it’s worth noting that persistency, in our context, is not an implementation detail. It is tightly related to the "business logic" of the application.

In previous works, we developed a proprietary API (XtremWeb-CH Client API) [16, 106] which has been used to port several scientific applications: Phylip [2], NeuroWeb [106], GIFT [132], Cyclone [57], MetaPIGA [103, 79], Selector [48], SolarEnergy, CiTaTiON and SWAT [137]. The XWCHClient API is built on top of the volunteer computing platform XtremWeb-CH. This API provides all requested functionalities to monitor jobs, express parallelism, manage precedence rules and data dependency. However, the use of this API for non-expert computer developers is not trivial: source codes using the XtremWeb-CH Client API are often too verbose and difficult to understand and maintain.

The main drawback of the XtremWeb-CH Client API is its complexity. Non-expert developers (in particular in the field of parallel and distributed computing) might be unable to take advantage of the functionalities supported by the API.

This chapter proposes two programming paradigms to write DW e-science applications. The proposed paradigms hide the complexity of parallel and distributed programming and are very close to the "business logic" of scientists. In addition, they target different infrastructures, for instance, grid and cloud. The two paradigms are tested and validated in the concrete case of the MetaPIGA and NeuroWeb applications.

3.2 New paradigms for the DW e-science applications

We consider a computational dynamic workflow as defined in the introduction of this chapter: the number of iterations and the number of jobs per iteration are not known in advance and the convergence condition relies on the in-time results of jobs. For a given application, jobs can be persistent or non persistent.

We propose two patterns (distributed programming paradigms) depicted in Fig. 3.1 which target the DW applications with (i) non persistent and (ii) persistent jobs.

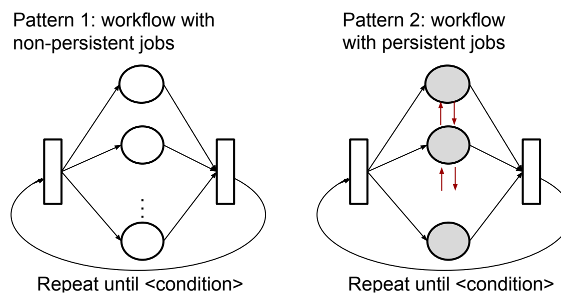


Figure 3.1 – Circles are nodes that represent slave jobs. Rectangles represent controllers (master jobs). White circles are non persistent jobs while gray ones are persistent jobs. Arrows between persistent jobs represent data transfer and communication.

In the first workflow, jobs are "embarrassingly parallels", i.e. there is no communication between them. The process of the convergence checking is launched each time a job is finished. In the second workflow, jobs are considered as daemons (persistent jobs). In this case, jobs keep running on the resources, they communicate with their neighbors (exchange boundary data) and deliver intermediate computation results. Each job, (i) receives some input data and delivers results, (ii) updates and processes its internal data. The non-reception of the input data is tolerated, i.e. step (ii) can be performed even if the input data are not available. This delays the convergence time. Note that the number of persistent jobs at each cycle does not vary.

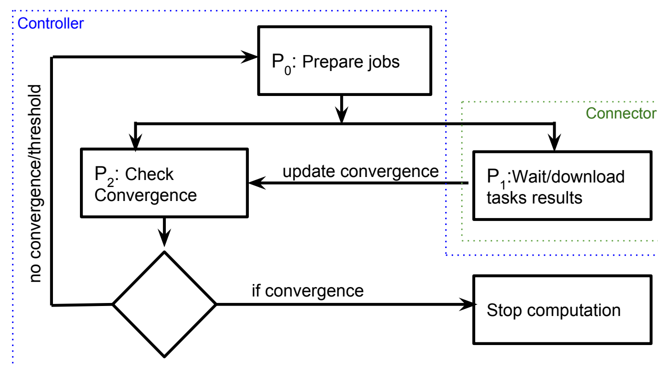


Figure 3.2 – Decisional process in a DW applications

The convergence condition of such dynamic workflows is modeled by a set of processes running on the end-user program as depicted in Fig. 3.2. In the first step, a process P_0 prepares a given number of jobs. Then, two parallel processes P_1 and P_2 are launched. The process P_1 waits for the slave jobs to finish in order to download their results. The process P_2 periodically checks the convergence condition of the computation. If the convergence is not reached, process P_2 prepares *n* additional jobs to be submitted (assuming that the number of running jobs is larger than a given threshold *ts*). When a job is finished, the process P_1 notifies P_2 and the convergence verification is re-launched. Note that P_1 (*connector*) is tightly linked to the infrastructure on which its jobs are running. Also, the targeted platform may be composed of several heterogeneous computing infrastructures (on premise, private cloud, public cloud, etc.).

The objective of this model is to simplify the implementation of DW e-science applications. This model distinguishes two types of nodes within the same workflow: computing (slaves) and controller (masters) nodes. Computing jobs are submitted to remote resources. A *controller* node is a process that handles jobs submission and implements the two processes P_0 and P_2. The process P_1 is illustrated through the *connector* component which handles submission, wait and local download of jobs' output files.

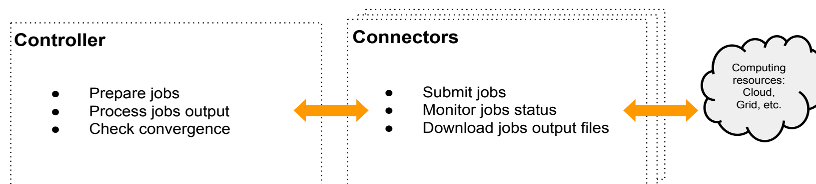


Figure 3.3 – Computation workflow with convergence condition.

The global architecture of the whole system is illustrated in Figure 3.3. Section 3.2.1 presents the pattern 1 including the detailed architecture of the *controller* and *connector* components where the jobs are non persistent. Section 3.2.2 presents the pattern 2 where the jobs are persistent.

3.2.1 Pattern 1: Skeleton for handling non persistent jobs

An application performing parameters sweeping with convergence condition is an example of our first pattern, pattern 1 (Fig.3.4). In this pattern, the first step consists in preparing a list of jobs (with their given input files and parameters) to submit. These jobs are inserted into a job queue. This operation is specific to the application and should be implemented by the programmer.

The *connector* module submits jobs to the target computing platform, monitors their status and retrieves their output files. The coordination between the *controller* and the *connector* modules is as follows. The *controller* takes care of checking the convergence

during all the computation whereas the *connector* performs the runtime execution on the target computing platforms. The *controller* module receives a message from all *connectors* notifying it that a set of jobs have finished and their output files, if any, were downloaded locally. Then it proceeds to post-process the output files and checks the convergence condition. If the convergence condition is not reached (and assuming that the number of running jobs is lower than a given threshold), the *controller* prepares a batch of jobs and insert them into the job queue. Otherwise, the *controller* terminates the computation, cleans the job queue and notifies the *connectors*.

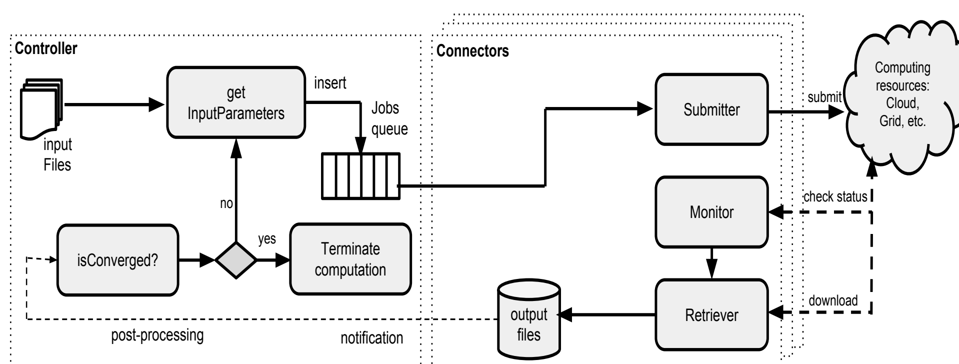


Figure 3.4 – The computation workflow of pattern 1

Hence, a DW application with non persistent jobs can be modeled by this expression

$$W = (c, J, X)$$

where c is a *controller*, J is a set of k jobs and X is a set of n *connectors*.

According to the specification of pattern 1, the code of a DW application can be expressed in object oriented style as follows:

```
1  Nodes jobs=new Nodes("jobs:1:k");
2  jobs.setPrecedence(new BusinessPrecedence());
3  Workflow wk=new Workflow(jobs);
4  Controller ctr=new EndUserController(user_parameters);
5  Connector con1= new Infrastructure_X1_Connector(userCredentialsX1);
6  Connector con2= new Infrastructure_X2_Connector(userCredentialsX2);
7  ctr.setConnectorsList(new ArrayList().add(con1).add(con2));
8  wk.setController(ctr);
9  wk.setDispatcher(new BusinessSplitProcess(jobs, wk.getConnectors()));
10 wk.run();
```

The jobs $\{job_i\}_{1 \leq i \leq k}$ composing the workflow are declared in line-1. The initial number of jobs in the workflow can be extremely large and thus complex to define in the code. A method to reduce the definition of jobs is to assign a range of numbers to jobs as shown in line-1. Moreover, dependencies between jobs can be expressed using a *BusinessPrecedence* object in the method *setPrecedence* as shown in line-2. This method is abstract ans

should be implemented. If there is no real precedence, this line is not required. An object to handle the dynamic workflow is created in line-3. A *controller* and *connector* components are instantiated in lines 4-5-6. The former implements processes P_0 and P_2, i.e., it checks the convergence and takes the decision of whether to run or not additional jobs. The latter submits jobs to the targeted computing infrastructures $X1$ and $X2$ (such as Microsoft Azure, Amazon EC2, etc.) and implements the process P_1, i.e., the handling of the jobs' progress and the download of their outputs. Line-7 associates the two *connector* components to the *controller* $ctrl$ which can handle the computation (line-8). Line-9 dispatches jobs among targeted infrastructures (connectors) according to a predefined policy : available resources, minimization of execution cost (in case of Cloud), minimization of execution time, etc.

It is worth noticing, that the *controller* is a component that should be implemented by the programmer since it is specific to the application, whereas the *connector* depends on the resource providers and is implemented independently of the application.

In order to implement a *controller* class for a given application, the programmer should extend a predefined *Controller* class and implement the following abstract methods:

- getInputParameters(i): This method returns the input parameters of the next job i to be submitted: job name, executable (binary), input files, command line, output files, final (a boolean variable which indicates if the job output must be downloaded).
- isConverged(): it returns a boolean status of the computation convergence condition.
- getInitialJobsNumber(): it fixes the initial number of jobs to submit when starting the computation.
- getSubmissionThreshold(): it returns the threshold value "ts".
- getAdditionalJobsNumber(): it fixes the number of additional jobs to submit in case the convergence is not reached and the total number of submitted jobs did not exceed the threshold value.

The predefined *Controller* class has two methods *initialize()* and *run()* methods. The *initialize()* method starts the computation by sending the initial jobs. The *run()* method is periodically activated and uses the *Connector* component as described in the algorithm 1.

Moreover, in the class *BusinessSplitProcess*, a method called *getJobDispatching()* which should be implemented by the programmer, returns a mapping from the set of jobs J and the *connectors* X.

In order to implement a *Connector* class for an application, the programmer must extend a predefined class and implement the abstract method *postProcessing(Job j)*. The predefined *Connector* class has the following methods:

Algorithm 1: run() method of the *Controller*

```
Data:
Connector[] consList;
1 begin
2   if (not isConverged()) then
3     for i ← 1 to getAdditionalJobsNumber() do
4       Connector con=getConnector(i);
5       con.submitJob(getInputParameters(i));
6   else
7     for i ← 1 to consList.size() do
8       consList[i].stopComputation();
```

- submitJob(Job j): submits a job to the target computing platform.
- getRunningJobs(): returns a list of running jobs.
- getFinishedJobs(): returns a list of finished jobs.
- stopComputation(): stops all running jobs and clean up the execution environment.
- postProcessing(Job j): processes a finished job referenced by the parameter j and notifies the *controller* to verify the convergence condition.
- run(): verifies periodically the submitted job status and calls the *postProcessing* method each time a job is finished.

3.2.2 Pattern 2: Skeleton for handling persistent jobs

A DW application with persistent jobs can be modeled by this expression

$$WP = (pc, J, X)$$

where pc is a controller of a set of k persistent jobs J and X represents a set of n *connectors*. Note that unlike the first pattern, k is fixed by the programmer. A DW application with persistent jobs can be expressed in a object oriented paradigm as follows:

```
1 Nodes jobs=new Nodes("jobs:1:k");
2 Workflow wps=new Workflow(jobs);
3 PController ctr=new EndUserPController(user_parameters);
4 Connector con1= new Infrastructure_X1_Connector(userCredentialsX1);
5 Connector con2= new Infrastructure_X2_Connector(userCredentialsX2);
6 ctr.setConnectorsList(new ArrayList().add(con1).add(con2));
7 wps.setController(ctr);
8 wps.setDispatcher(new BusinessSplitProcess(jobs, wps.getConnectors()));
9 wps.run();
```

The computation steps are as follows:

1. the problem domain is split into several blocks. Each block will be assigned to one persistent job.
2. Once started, each persistent job deploys a proxy interface to communicate (exchange boundary data) with its neighboring jobs.
3. Each persistent job receives boundary data, updates its block and sends new boundary data back to its neighboring jobs. It asynchronously processes its data block even if no boundary data is received.
4. All intermediate results are merged and the convergence of the whole domain is checked.
5. If the convergence is not reached, go to step 3). Otherwise, all persistent jobs receive a notification to clean and terminate the computation.

As explained in these steps, the persistence aspect often implies communications between neighboring jobs. Hence, this skeleton pattern should propose an efficient communication pattern that replaces the jobs precedence rules. So, unlike in pattern 1, pattern 2 does not require a "BusinessPrecedence" object. Because persistent jobs are a general program running on a distributed infrastructure, synchronizing the communication between them is not straightforward. To this end, a well defined paradigm to exchange data is needed. This is achieved through *Messages*. In its simple form, a *Message* is an object that encapsulates information related to the sender (a job identifier), the destination (a job identifier), a message tag and a body. The latter can simply be a file, an URL or an embedded data.

Furthermore, exchanging messages have to be done through a uniform communication interface which should be written by the programmer. We therefore consider a *Communicator* class with two methods *receive* and *deliver*. This class is implemented within the persistent job. The *receive* method feeds the persistent job with a list of *Messages* while the *deliver* method delivers a list of *Messages* that are retrieved by the *controller* and routed to their corresponding destination.

Communication between the *controller* and persistent jobs is depicted in Fig. 3.5. The *controller* receives two types of *Messages*: *Messages* destined to persistent jobs and *Messages* used by the controller to check the global convergence of the computation. *Messages* are inserted into the corresponding queues. There are k+1 queues : one for each persistent job and one for the global convergence *Messages*. Another process retrieves the *Messages* from jobs queues, prepares the jobs and submit them to the remote resources on which the corresponding persistent job is running.

Messages received by a persistent job are used to accelerate the computation whereas those destined to the *controller* are used to check the global convergence of the computation.

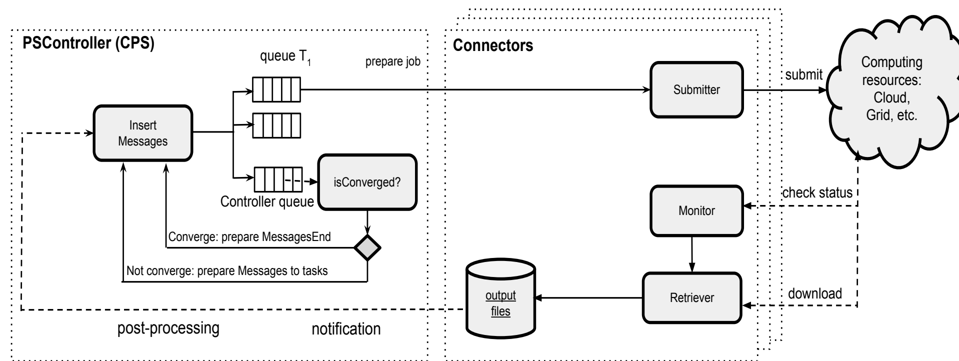


Figure 3.5 – Computation workflow of pattern 2

Reception of *Messages* and jobs submission can be executed in parallel. To shield the programmer from handling the synchronization between all these processes, the application controller object must be extended using a *PController* abstract class and the following abstract methods must be implemented:

- getPersistentJobs(): splits the problem domain into several blocks and returns a list of *Job* objects to be run on the remote resources. It is worth reminding that the binary file of each persistent job must implement the *Communicator* class which will be run in the background on the remote resources.
- isConverged(): reads all *Messages* destined to the *controller* and returns the status of the convergence condition as a boolean value.
- getIterationsThreshold(): returns a threshold value *ts*. This threshold is useful when the computation diverges. In this case, *ts* defines so the maximum number of iterations above which the application stops computing whatever the convergence condition status is.

The predefined class *PController* has additional methods, namely, *initialize()* and *run()*. *initialize()* calls the getPersistentJobs() method, submits the list of persistent jobs to the remote resources (using the corresponding connectors) and waits for their deployment. Once finished, it retrieves the output of each persistent job.

The *run()* method is activated periodically to check for the convergence, it reads the queues, submits the jobs and retrieves their output (*Messages*) as described in the pseudo-code of the algorithm 2. The convergence is checked in line 2. If there is no convergence, the list of messages for each queue is selected (lines 5-6) in order to be sent to the corresponding persistent job. The method *prepareJobToPSJobs()* prepares a job (lines 8-9) with the "right" communication interface parameter. The *connector* object sends the job to the same computing resource where the persistent job is running (line 10). Note that the submission of jobs is not a blocking operation. In case the convergence is reached, the computation is terminated by calling the method *stopComputation()*.

Algorithm 2: run() method of the *PSController* class.

```
Data:
Queue[] queuesList;
1  begin
2    if (not isConverged()) then
3      for i ← 1 to queuesList.size()) do
4        Queue q←queuesList[i];
5        List<Messages>listMsgs =q.element();
6        int receiver = listMsgs.get(0)).getReceiver();
7        Job Tps←getCorreponsdingJob(receiver);
8        Job j←prepareJobToPSJobs(listMsgs,Tps);
9        Connector con←Tps.getConnector();
10       con.submitOnSameResource(j, Tps);
11   else
12     stopComputation();
```

3.3 MetaPIGA and NeuroWeb implementation

In this section, we propose to apply the two high level programming paradigms detailed in section 3.2 to two DW e-science applications : MetaPIGA and NeuroWeb (described in chapter 2). Our aim is to provide an "object oriented" inspired API with a set of predefined classes. The major difference compared to the other environments is that the scientist/programmer can define a convergence condition within a dynamic distributed computation. Consequently, the readability of the code should be improved and the code should be extensible enough to easily support different types of computing infrastructures within the same computation.

The first paradigm will be used for MetaPIGA (non persistent jobs) and the second one for NeuroWeb (persistent jobs).

3.3.1 MetaPIGA use case

The Listing 3.1 shows the pseudocode of the "MetaPIGAController", the *controller* class of the MetaPIGA application. Note that although there are portions of code details which are not shown here, the code includes all relevant details about the structure of the class.

Listing 3.1 – MetaPIGAController Java class implementation

```
1 class MetaPIGAController extends Controller{
2   double MIN_SCORE=0.01; int counter=0;
3   public long getInitialJobsNumber(){return 300;}
4   public long getsubmissionThreshold(){return 10000; }
5   public getAdditionalJobsNumber(){return 100; }
6
```

```
7     public Job getNextJobParameter(){
8       Job param= new Job();
9       param.jobName="T_"+counter;
10      param.executable="metapiga2.jar";
11      String inputFile=createInputFile(counter);
12      param.cmdLine="java -jar -Xms256m -Xmx512m metapiga2.jar noupdate" +"silent
13   "+inputFile+" > metapigaout.log 2>&1";
14      param.compressedOutPutFile="gridoutput.nex.zip";
15      param.isFinal=false;
16      ++counter;
17      return param;
18    }
19
20    public boolean isConverged(){
21    List<Job> finishedjobs= this.con.getFinishedJobs();//gets the new finished jobs
22    for (Job job: finishedjobs){
23      Tree tree=generateTreeFromOutputFile(job.compressedOutPutFile);
24      merge(tree);//merge the tree to the existing ones
25      Tree treeBest=selectBestTree();//compute the current best tree
26      if (treeBest.score> this.MIN_SCORE){//check the score of the best tree
27        return true;//convergence is reached
28      else return false;//convergence is not reached
29    }}
30
31    private void merge(Tree tree){/*merge the tree*/}
32    private Tree selectBestTree(){/*return the best tree over the genearted trees set*/}
33  }
```

A minimal effort is required to implement an application-specific *controller* class. Basically, a small effort is needed to implement *getNextJobParameter()* and *isConverged()* methods as shown in Listing 3.1. Technically, preparing jobs consists in creating a different input file for the same *metapiga2.jar* binary file. Then, the output result for each input file will be compressed in *gridoutput.nex.zip*, downloaded by the *connector* and processed in order to generate a phylogenetic tree. Regarding the computing infrastructure, the programmer has the option to either write their own *connector* or select one from the predefined *connectors* classes such as *AzureConnector* for Microsoft Azure or *AmazonEC2Connector* for Amazon EC2. Writing a *connector* class requires mainly re-implementing the interface of the modules "submitter, watcher and retriever" (Fig. 3.4) since the scheduling back-end of jobs is hidden for the programmer.

The code of the MetaPIGA computation can be written in few lines of Java code. Note that in the line-1 of Listing 3.2, the tasks range should be compliant to what the *getInitialJobsNumber()*, *getsubmissionThreshold()* and *getAdditionalJobsNumber()* methods return. As shown in the code, the total number of tasks should not exceed the threshold 10000. Moreover, there is no precedence rules between tasks.

Listing 3.2 – MetaPIGA main code

```
1  Nodes jobs=new Nodes("T:0:1000");
```

```
2  jobs.setPrecedence(new DefaultBusinessPrecedence());
3  Workflow wk=new Workflow(jobs);
4  Controller ctr=new MetaPIGAController(user_parameters);
5  Connector con1= new AzureCloudConnector(azure_user_credentials);
6  Connector con2= new AmazonCloudConnector(Amazon_user_credentials);
7  ctr.setConnectorsList(new ArrayList().add(con1).add(con2));
8  wk.setController(ctr);
9  wk.setDispatcher(new DefaultBusinessSplitProcess(jobs, wk.getConnectors()));
10 wk.run();
```

3.3.2 NeuroWeb use case

The Listing 3.3 shows the pseudocode of the *PController* class of the NeuroWeb application. The method *initialize()* creates the persistent jobs and retrieves the information related to their communicator interface. Depending on the available resources and user *BusinessSplitProcess* strategy, the submission is handled by a list of connectors and each job is assigned to a connector.

Listing 3.3 – NeuroWebController Java class implementation

```
1  class NeuroWebController extends PController{
2    public List<Jobs> getPersistentTasks(List<Messages> msgsFromMaster) {
3      this.splitDomain(nbMaxTask, msgsFromMaster);
4      nbMaxTask = msgsFromMaster.size();
5      List<Jobs> listjobs = new ArrayList<Job>();
6      for (int i = 0; i < nbMaxTask; ++i) {
7        Job j = new Job();
8        j.jobName = "T_" + i;
9        j.executable = "pstaskbinary.jar";
10       Directives directiveClient = this.getDirectivesToTask(i);
11       String taskInputzip = util.serialize(listmessagetoworker,
12           directiveClient);
13       j.inputFile = taskInputzip;
14       j.compressedOutPutFile = "serializedMessages_" + i + ".zip";
15       j.cmdLine = "nohup java -cp .:* neuroweb.pstask " + i + " "
16           + nbMaxTask + " " + j.compressedOutPutFile +" "
17           + taskInputzip;
18       param.isFinal = false;
19       listjobs.add(j);
20     }
21     return listjobs;
22   }
23
24   public void initialize() {
25     try {
26       List<Messages> msgsFromMaster= new ArrayList<Messages>();
27       List<Jobs> psTasksList=getPersistentTasks(msgsFromMaster);
28       for (Job psJob: psTasksList){
29         //get corresponding connector
30         Connector con = getConnectorForJob(psJob);
```

```
31            psJob.setConnector(con);//set the job's connector
32            con.submitJob(psJob);
33          }
34          //waiting all Workers jobs
35          waitForJobs(psTasksList);
36          //retrieve communication ports for each persistent job
37          for (Job psJob: psTasksList){
38            Port p=getCommunicationJob(psJob.compressedOutPutFile);
39            // update the persistent job object
40            psJob.setPort(p);
41          }
42        }
43  }
```

The method *isConverged()* reads all messages intended to the *controller* and decides whether or not to continue the computation. If the convergence is not reached, this method populates the queues of the persistent jobs with *Messages* objects containing the required information needed by the *controller*. Otherwise, the *controller* prepares a *MessageEnd* object for each persistent job to inform it that the convergence is reached. If a persistent job receives a *MessageEnd* object, its stops and cleans up the computation.

Listing 3.4 – NeuroWebController run method implementation

```
1  public void run(){
2    if ( ! isConverged() ){
3      while (queueListMessage.size() > 0) {
4        //get the Messages of the current queue
5        List<Messages> listMessage = (List) queueListMessage.element();
6        //retrieve the job object of the corresponding persistent job
7        int receiver = ((Messages) listMessage.get(0)).getReceiver();
8        Job psTask=getCorrepondingJob(receiver);
9        //prepare a job to send to the remote persistent job
10       Job j=prepareJobToPsTask(listMessage,psTask);
11       //retrieve the corresponding connector
12       Connector con=psTask.getConnector();
13       //submit the job
14       con.submitOnSameResource(j, psTask);
15     }
16   }else{stopComputation();}
17 }
```

The main code of the NeuroWeb application is similar to the one shown in Listing 3.2 except for line-4 where the programmer should instantiate *PController* object:
PController ctr=new NeuroWebPController(user_parameters)
This object will handle the persistent aspect of the computational workflow of the NeuroWeb application.

3.3.3 Experiments

This section presents the experiments conducted to evaluate our model. First, our model is applied to the MetaPIGA application (described in chapter 2) where its distributed version run on both Amazon EC2 and Azure cloud platforms. Thereby, we can show that our model has been implemented and tested on real use-case by considering at least two cloud platforms. Then, the execution time of our model is compared to a similar tool: GC3Pie [126, 43] . This comparison considered the execution time of MetaPIGA on the Amazon EC2 cloud.

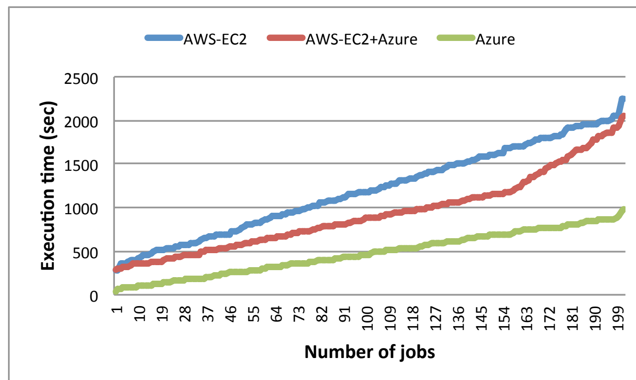


Figure 3.6 – Execution time of MetaPIGA on the Amazon EC2 and the Microsoft Azure cloud platforms. `AWS` stands for Amazon resources.

Deployment on infrastructures

Around 200 MetaPIGA jobs run on both Amazon EC2 (AWS-EC2) and Microsoft Azure cloud platforms. In these experiments, the convergence condition is simply reached once the number of execution jobs exceeds 200. The 20 virtual machines (VMs) used in this experiment are based on the *Small* computing Amazon instance type (see Table 3.1).

The first simulation consists of three scenarios. Scenario 1 (resp. scenario 2) uses 20 VMs of Amazon EC2 (resp. Microsoft Azure) resources. Scenario 3 uses heterogeneous resources from both platforms (10 VMs on each platform). The developed program ran on the client machine and undertook the following operations:

- start the VMs on the cloud platforms,
- upload the input data of each job from the local machine to the corresponding VM,
- run remotely MetaPIGA jobs on the VMs,
- wait and download the output result from the VMs to the local machine.

Figure 3.6 illustrates the execution time of the three scenarios. The execution time on

Table 3.1 – Characteristics of the VM instance type.

Platform	VM type	Memory	CPU	OS	Location
AWS-EC2	m1.small	1.7 GB	Intel-Xeon @2.00GHz	Ubuntu	North Europe
Azure	small (A1)	1.7 GB	Intel-Xeon @2.20GHz	Ubuntu	North Europe

the Azure platform shows better performance than Amazon-EC2. This may be explained by the difference of the VM CPU speed as shown in Table 3.1. Sharing the same number of jobs on a heterogeneous platform composed of Amazon-EC2 and Azure computing resources leads, as expected, to accelerate the execution time compared to Amazon-EC2 simulation. This experiment shows that our model is able to support several Cloud platforms within the same deployment. We tested the portability and interoperability of our model with Amazon-EC2 and Azure. However our classes can be easily extended to other Infrastructures as a Service (IaaS) such as OpenStack and StratusLab [134, 98]. The aim of the next experiment is to compare the performance of our DW model with the GC3Pie tool.

Performance comparison with GC3Pie tool

The GC3Pie (described in the introduction of this chapter) programming model considers mainly the following classes: *Application*, *SequentialTaskCollection* and *SequentialTaskCollection*. The *Application* corresponds to a single job and consists in a generic class that programmers should extend to define their jobs. It enables programmers to describe a job at a high level: they specify the list of input and output files, command line to execute, resource requirements and limits, pre-and-post-processing. Depending on the targeted executing platforms, GC3Pie translates the job description to the adequate format in order to execute it on the corresponding remote resource.

The work-unit in GC3Pie programming model is called a *Task*. An *Application* object is a primary instance of *Task*. A *Task* is a composite: meaning that it can be a single *Task* or a workflow of *Task*s. So, a complex workflow is built by simply composing *Task*s in different ways using the classes *SequentialTaskCollection* and *ParallelTaskCollection*. These two classes are abstract and provide two ways to schedule and execute a collection of *Task*s. The programmer should extend them in order to build a workflow of *Tasks*. The former executes a collection of *Tasks* sequentially. Also, it enables the programmer to perform, after each *Task* execution, a decisional treatment through implementing a given abstract method. A possible decisional treatment can be submitting a new *Task* (which can be a workflow of *Tasks*) of resubmitting a failed one. The latter executes in parallel a collection of independent *Tasks*. By using these two classes one can create a dynamic

workflow. More details about GC3Pie programming model can be found in [126, 43].

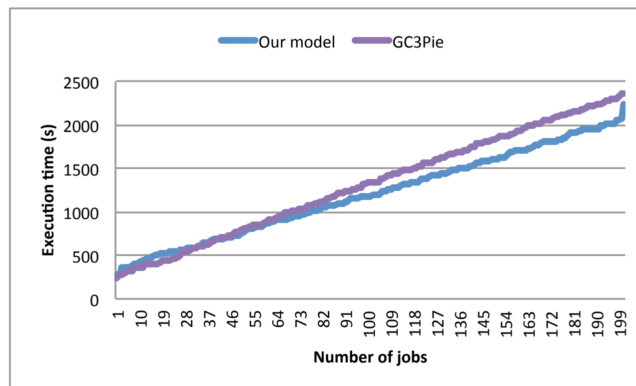


Figure 3.7 – Execution of MetaPIGA on Amazon EC2: performance comparison with the GC3Pie programming model.

We considered the same computing resources for the GC3Pie experiments as in scenario 1. To do so, we mounted a *Slurm* [129] based cloud cluster on Amazon EC2 using the tool elasticluter [55]. This cloud cluster is composed of 20 VMs having the same VMs configuration as for Amazon-EC2 (see Table 3.1). Figure 3.7 illustrates the performance comparison with the GC3Pie programming model. We can see that the execution time is almost the same. Thus, our model is similar to GC3Pie in term of computing performance.

Qualitative comparison with GC3Pie tool

The programmability and the easiness of using the two models (GC3Pie and our model) is not easy to measure. One of the most straightforward evaluation metric is SLOC (Source lines of code), which consists in counting the number of lines of a program source code. Table 3.2 presents a comparison with the GC3Pie though programming criteria. Conceptually, GC3Pie is interesting and provides more flexibility to design composite workflows. In addition, it supports running application on HPC systems (clusters). Our model is easier to use and provides a more human-friendly convergence specification, specially for users with no programming experience. Besides, It supports master/slave architecture where jobs are persistent.

3.4 Conclusion

During the last ten years the Large Scale Distributed Research Group (www.lsds-rg.org) at the University of Applied Sciences, Western Switzerland has been involved in the development, porting and deployment of several e-sciences applications on large scale distributed and heterogeneous infrastructures (cluster, Grid, Volunteer Computing and Cloud) : Phylip [2], NeuroWeb [106], GIFT [132], Cyclone [57], MetaPIGA [103,

Table 3.2 – Comparison with the GC3Pie programming model.

Criteria	Our model	GC3Pie
Convergence condition	easy to program	must be expressed at the application level using loop and condition statements.
Composite workflow	planned	supported
Multiple cloud IaaS	supported	supported
Easiness	easy (main code about 10 lines)	easy, but verbose

79], Selector [48], SolarEnergy, CiTaTiON and SWAT [137]. Thus, our contribution is illustrated through this work and is based on the our experience accumulated not only in porting those scientific applications but also during interaction with other research groups and applications.

In this chapter we presented two high level programming paradigms to develop dynamic workflow (DW) applications and deploy them on large scale distributed and heterogeneous infrastructures. The two proposed paradigms target embarrassingly parallel applications. The first paradigm (pattern 1) addresses scientific applications where jobs are non persistent and their number can not be known in advance. The second paradigm (pattern 2) addresses applications where job are persistent. Both of the two paradigms support handling convergence condition during computation. They were tested in the concrete cases of two scientific applications: MetaPIGA and NeuroWeb.

It is possible to add a layer on top on the two proposed paradigms to easily support other simple programming skeletons. For instance, for applications based on Fork-join skeleton with m jobs, the pattern 1 can be used. To do so, the *isConverged()* method should return *false* and the *threshold* can be simply set to the value m.

The next chapter will deal with the second category of applications (Figure 2.13): the *communication-intensive* applications. It will present the proposed approach to modelise, design and execute these type of applications on large high performance infrastructures.

4 Rationale for multiscale applications design and deployment*

4.1 Introduction

This chapter deals with the second category of scientific applications: *communication intensive* applications (Figure 2.13). We are specifically interested in the multiscale applications having a tightly coupled workflow. These applications involve interaction between several natural phenomena acting at several temporal and spatial scales. The goal of this chapter is to describe the so-called Multiscale Modeling and Simulation Framework (MMSF) developed in the European project MAPPER [102]. It has been successfully applied to seven applications from different research domains: Fusion, Computational biology, bio-engineering, nano-material science and hydrology.

In this chapter we describe the sequence of actions of the MMSF approach, from building to executing multiscale applications on a distributed computing infrastructure, such as EGI [53] and PRACE [121].

4.2 Challenge in modeling real phenomena at different scales

Mathematical equations have been extensively used in sciences to describe and study real and natural phenomena. Mathematical models are used to explain and understand the interactions between different physical processes involved in the same phenomenon based on experimental observations. Most importantly, they are used to predict the behavior of real phenomena under given conditions and constraints. Each physical process may act at different spatial and temporal scales. A scale can be defined as a distance over which

*This content of this chapter is based on:

- Mohamed Ben Belgacem, Bastien Chopard, Joris Borgdorff, Mariusz Mamonski, Katarzyna Rycerz, and Daniel Harezlak. Distributed Multiscale Computations Using the MAPPER Framework. In Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 1106–1115. Elsevier, 2013.

or a temporal interval during which a variable varies significantly.

When building a mathematical model, it is important to define firstly the variables composing the mathematical equations. These variables make it possible to study and describe the behavior of a given natural phenomena. For instance, in order to evaluate the pressure caused by the blood flow on the inner wall of an artery, the model needs variables to describe the velocity of blood and its density. Other variables can be defined to model the changes of the artery geometry during time, for instance, due to an injury. Reciprocally, the deformation of the artery geometry can influence the characteristics of the blood motion. The three-dimensional model of *in-stent restenosis in coronary arteries* (ISR3D) [32] application describes in more details this example.

An other important step when modeling a real phenomena is to choose the adequate temporal and spatial scales to quantify the observation measurement. This depends on the length and time characteristics of the involved physical processes. Many real phenomena can be studied at different magnitudes in space and time. For example, in some physical domains (such as biology, atomic and nuclear physics, etc), studies are done at microscopic scales with a space scale ranging from $10^{-10}m$ to $1m$ and a time scale ranging from $10^{-9}sec$ to months or years. Scientists use generally their experience and knowledge to decide whether to switch to a given scale level to observe the studied process. Taking the previous example of blood motion inside an artery, it is intuitive for scientists to determine that a given physical process may involve cells proliferation, drugs molecules interactions, artery tissue deformation, etc. However, it is difficult to consider within the same phenomena that the geometry deformation process of the artery acts at a time scale of $1hour$ which is larger than the time scale of $1ms$ required to study the blood motion process. Therefore, determining space and time scales of a real phenomena is yet a difficult task.

4.3 A brief history of multiscale modeling techniques

Most real phenomena involve several spatial or temporal scales and interaction between various physical processes. A multiscale application is composed of pieces of complex codes implementing in general numerical solvers. Its main goal is to make interacting phenomena together through exchanging information at different temporal and spatial scales and to perform computational simulation to study them in an accuracy way.

There are rather few frameworks that offer general tools and concepts to propose a methodology to develop multiscale applications. Methodological papers such as [145, 83, 46] are important in this perspective as they propose examples of a conceptual approach. However, there is still a lack of a consensus vision among the many communities dealing with multiscale applications. A recent review [70] shows that existing multiscale projects tend to use a variety of approaches to build and run their applications, mostly connecting

their codes with hand-written scripts and run computation on single sites.

Running simulations across multiple computing sites has been done in previous works, as illustrated for instance in [72, 100, 10]. These approaches offer user friendly API and convenient submission procedures but do not propose a full methodology based on theoretical concepts and a formalism as discussed below.

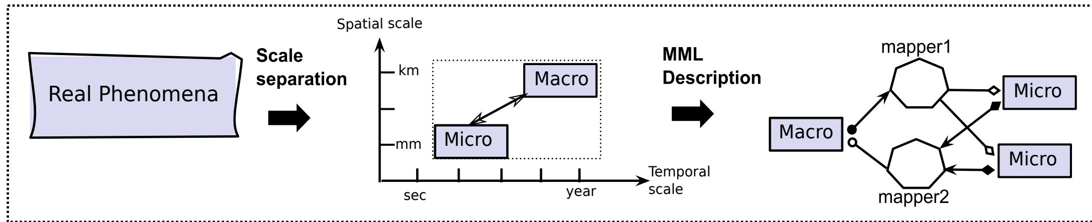
A few years ago, some works are done to develop a multiscale framework, coined CxA for Complex Automata. The initial goal was to couple different Cellular Automata or Lattice Boltzmann models [35] with different spatial and temporal resolutions [1, 81, 82]. This approach has been successfully applied to the so-called *In-Stent Restenosis* biomedical problem [32]. It also gave rise to a Multiscale Modeling Language (MML) [60], offering a powerful way to describe a multiscale problem in terms of its components and their couplings.

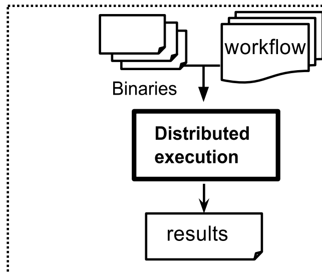
Recently, this approach has been further developed and generalized within the European project MAPPER [102] and has led to the so-called Multiscale Modeling and Simulation framework (MMSF) [27]. The goal was to show that a wide range of applications can be described, implemented and run within the formalism. An important aspect of the approach is the execution phase where the concept of *Distributed Multiscale Computing* (*DMC*) is proposed [86, 27]. The idea is that the different components of a multiscale-multiphysics application can be distributed on several supercomputing facilities and run in a tightly and/or loosely coupled way. *DMC* allows the users to access at once, and for one specific application, much more computing resources than available in a single supercomputing center. In addition, *DMC* allows to run parts of the application on the most appropriate hardware such as GPU.

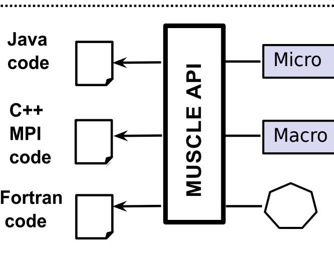
We refer the reader to [60, 82, 86, 27, 37] for a detailed discussion of the theoretical concepts underlying this multiscale approach.
In this chapter, some concepts definitions in the theoretical part will be redefined based on our vision and they may slightly differ from the ones presented in [27]. However, the main idea remains unchanged. We present then the framework from the point of the end-user aiming at developing, or adapting, his application within the proposed framework.

4.4 The MAPPER Multiscale Framework

As illustrated in Fig. 4.1, the MMSF approach is based on four steps: *Modeling*, *computation design*, *multiscale implementation* and *execution*. These four steps are described in the next four sub-sections.







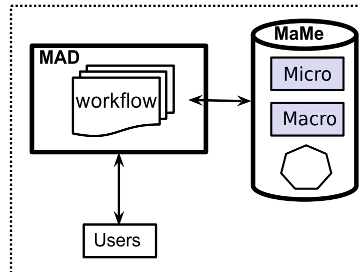


Figure 4.1 – The different steps to model, design, implement and run a multiscale application with the MMSF.

4.4.1 Modeling

A multiscale real phenomena can be defined as a collection of physical processes representing each a part of the given phenomena. These physical processes are interacting with each other at different single scales.

Formally speaking, a real phenomena is decomposed into a set of coupled submodels exchanging data across different scales. A submodel is a program that resolves a physical process at a given scale. A physical process within a real phenomena consists of dynamics acting on an object bounded in space and time intervals. If the dynamics, object and/or scales of that physical process changes, a new submodel should be used to model it. Let's consider the example of the material science domain aiming to discover and design new materials. For example, one can use a submodel implementing the "Newton's Law of Cooling" and apply it on a new material in order to predict the time it will take to cool it down at a given heating temperature. Now, let's consider the conductivity of the new material at electronic level which usually should behave exactly to given electronic specification such as in photovoltaic cells production. In this case, since the level of details is within the atomic level (nano meter), a new submodel implementing a quantum mechanic equations should be used to predict the electronic properties of the material through computer simulation before the manufacturing phase.

Scale Separation Map (*SSM*)

As discussed in section 4.2, the decomposition in physical processes and their couplings is a difficult yet central task. It is usually done according to the specific knowledge that the scientist has of the given problem. This is however facilitated by representing the system on a Scale Separation Map (*SSM*).

A way to identify the different scales involved in a multiscale phenomenon is to represent them in a *SSM*. A *SSM* consists of a 2-axis plot where each single scale process of the phenomenon is represented by a box. Horizontal (resp. vertical) axis corresponds to temporal (resp. spatial) scale. Interactions between processes are represented by annotated edges describing the inter-scale communications, called *scale-bridging techniques*. Each box defines the scales bounds of its corresponding process (*Scale separation* phase in Figure 4.1).

In other words, the *SSM* step offers a simple representation of multiscale phenomena and has the advantage to turn the development of multiscale-multiphysics application into the coupling of submodels. This coupling allows interactions between submodels and thus reflects the multiscale nature of the studied phenomena. Besides, by coupling these submodels in an appropriate way, one expects a good approximation of the original problem with all scales, with yet a substantial reduction of computational needs.

Figure 4.2 illustrates two coupled submodels A and B with a regular grid domain and different scales. For example, let A be a neuronal activity phenomena in a human brain and B a cell proliferation phenomena of a cancerous tumor. B occurs in a relatively small region in the brain compared to A and, consequently, must be studied in more accurate way. Using a multicale numerical simulation may help in predicting and localizing the proliferation of malignant cells and their influence on the patient mental behavior. In term of computation, submodel B is more expensive in resource computation than A since it requires a fine domain resolution. Running a multiscale application composed of coupled submodels, each with a different space scale, is more advantageous in term of computational needs than running a monolithic code that simulates the entire brain with a fine scale as in B.

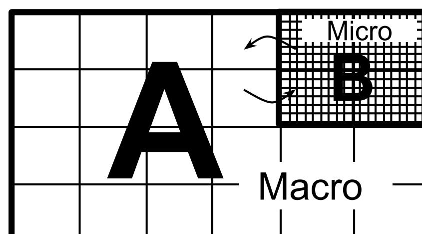


Figure 4.2 – An illustration of the multiscale coupling: two phenomena A and B modeled by submodel A and submodel B having a grid based domain.

To illustrate better this modeling step, Figure 4.3 represents a *SSM* of an application which simulates the sediment transport in a river or an irrigation canal. An example of a daily operation on the irrigation canal consists in opening water gates (for example, during periods of times of 1hours each) which result in a variation of water flow. This flow variation may act on the river bed and lead to an erosion-deposition process. Some solid and fine particles (sediments) are transported within the water flow. Due to the gravity force and their mass, some eroded sediments may deposit (deposition process) resulting in changes in the geometry of the river bed. In return, these changes creates a feedback on the water flow.

The sedimentation-transport model contains three submodels: 3D-FreeSurface (3DFS), 1D-Shallow (1DSW) water, sediment and erosion-deposition models. 3DFS and 1D-Shallow water are Lattice Boltzmann method (LBM) [135, 35] models that simulate the water at a temporal scale of ms. The 3DFS is used to compute the detailed water motion with a high resolution in a specific region and acts at a spatial scale from cm to m. The 1DSW simulates the water surface in long segments of river and acts at a spatial scale from m to km. Each simulation of the water motion is regulated on a period of time of $1hour$ (for daily operations for example) and has a time step of the order of ms . The sediment transport and erosion model is coupled with the water models to study the motion of sediment and the effect of the water on the river bed profile and at a temporal scales from $months$ to $years$. This is an illustrative example and the time scales can be adjusted according to the specificities of the problem to study.

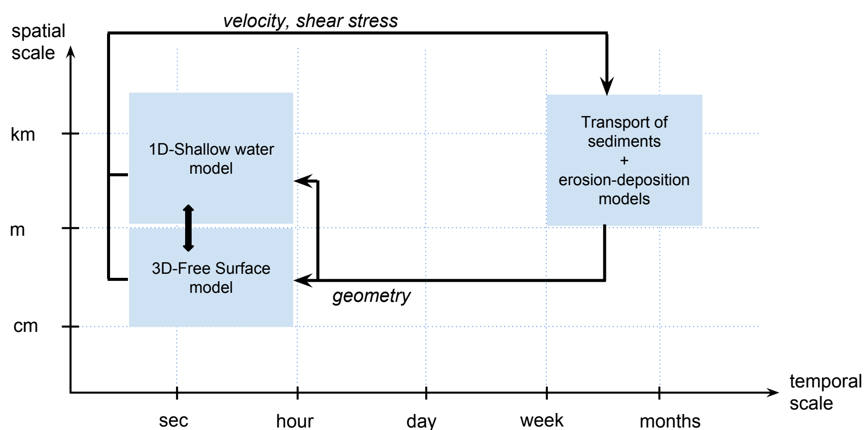


Figure 4.3 – The *SSM* respresentation of the sedimentation-transport model

However, interactions between submodels cannot be fully detailed in an *SSM* and may occur at different stages of the computation. Thus, the Multiscale Modeling Language (*MML*) (presented later) is used here to describe the workflow communication between the involved submodels of the same multiscale application.

Submodel execution loop (*SEL*)

In the *SSM*, interactions between coupled submodels do not convey any information with regards to when interactions happen. These information is important since it reflects the dynamic of data exchanges. More information should be added to the links between submodels in the *SSM* representation to indicate when the date are exchanged. To do so, we need to provide firstly an execution pattern for submodels and, then, formulate submodel interactions.

Exchange of data occur upon a state change of a submodel. The entire coupled system can be modeled by Discrete-Event system since the computation can naturally be described as a sequence of operations. It is worth noticing that other modeling techniques can be used: for example in order to model the behavior of each submodel, a multi-agent system may be the way to go. In a Discrete-Even system, an instantaneous event e models a submodel state change at time $t(e)$ in order to keep track of the simulation time, and a code to simulate e. The time interval between two successive events must be less that the time step Δt of the studied submodel: $t(e_{j+1}) - t(e_i) \ \leq \ \Delta t$. The dynamic of submodel is modeled by an ordered (based on events time) list of all events L and a pending events list P. Each time an event is simulated it will be removed from the pending event list. Following the simulation engine logic of Discrete-Event system, the main loop execution of the coupled system can be presented in algorithm 3. One can use the Discrete-Event system to predict the required execution time of a distributed multiscale system on given resources.

Algorithm 3: Discrete-event execution loop of the coupled system.

```
1  BEGIN
2    Starting time: t ← t0
3    Initialize the system state
4    Put the initial state in the event list
5  while not END do
6      Observe the intermediate state.
7      Set time-stamp to the next event.
8      Do next event and remove it:
         • execute the vent.
         • send out information.
         • receive information.
         • possibly insert new events.
9    Do a final Observation
```

From a practical point of view submodels correspond to numerical solvers, with given spatial and temporal resolution. This can be formalized by the *SEL* of the Algorithm 4,

where f is the submodel status, t_0 is the starting time of the submodel execution, t is the timestamp and Δt is the temporal scale.

Algorithm 4: Submodel execution loop

Input: Starting time t_0 and temporal scale Δt
$t \leftarrow t_0$;
1 $f \leftarrow \mathbf{F_{init}}(t)$;
2 **while** *not END* **do**
3   $\mathbf{O_i}(f, t, t + \Delta t)$;
4   $t \leftarrow t + \Delta t$;
5   $f \leftarrow \mathbf{S}(f, t)$;
6   $f \leftarrow \mathbf{B}(f, t)$;
7 $\mathbf{O_f}(f, t)$;

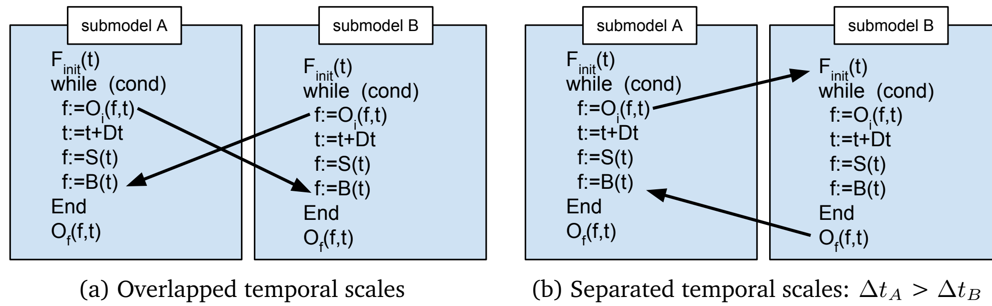
In this *SEL*, a given submodel execution undergoes a few abstract operations (called also operators), named *initialization* ($\mathbf{F_{init}}$), *observation*($\mathbf{O_i}$ and $\mathbf{O_f}$), *solver*($\mathbf{S}$), and *boundary condition* ($\mathbf{B}$). $\mathbf{F_{init}}$ operator is used to initialize the computing domain of the submodel and, its variables and boundary condition, **S** operator is used to solve the problem at each iteration, **B** operator is used to update the domain boundary condition at each iteration and $\mathbf{O_i}$ (resp $\mathbf{O_f}$) operator is used to make an intermediate (resp. final) observation of the submodel status. Practically, these operators are implemented as methods in the submodel code and their execution order can not change. The *SEL* is general enough to be applied to wide range of different phenomena and problems. Indeed, the implementation of operators are done with respect to the specificities of the studied submodel and thereby define its behavior.

Coupling templates

The interactions between the submodels are the basis in order to couple them in a multiscale application. Coupling submodels should reflect the frequency of exchanging data between submodels instances. This is important since it influences the runtime computational behavior of the overall application.

The reason behind using the Submodel Execution Loop (SEL) is to make easy the modeling of coupling submodels. Coupling submodels is thereby defined as data communications between the operators (for instance see Figure 4.4) based on a set of rules: the operators $\mathbf{O_i}$ and $\mathbf{O_f}$ can only send data whereas $\mathbf{F_{init}}$, **S** and **B** ones can only receive data.

In the *SSM*, two submodels A and B can have overlapped or separated time scales. In case of time scale overlap (Fig. 4.4a), two submodels A and B need to exchange data to resolve the computing iterations for each other. This interaction can occur between $\mathbf{O_i}^A$ (reps. $\mathbf{O_i}^B$) and either $\mathbf{S}^B$ (reps. $\mathbf{S}^A$) or $\mathbf{B}^B$ (resp. $\mathbf{B}^A$). If A has a larger time scale



(a) Overlapped temporal scales

(b) Separated temporal scales: $\Delta t_A > \Delta t_B$

Figure 4.4 – Example of a coupling template describing two coupled submodels. Arrows represents the data transfer.

than B, submodel B may perform n iterations per a time step of A: submodel A may send its $\mathbf{O_i}^A$ to B while the latter runs n time steps before sending its $\mathbf{O_i}^B$ to A. This depends only on the coupling topology between submodels A and B. In case of separated time scales and A has a larger time scale than B, A should at each iteration (if needed) instantiate a submodel instance of B : submodel A may send its $\mathbf{O_i}^A$ to $\mathbf{F_{init}}^B$ that will execute and send its $\mathbf{O_f}^B$ to A when it finishes (Fig. 4.4b).

Coupling topology

We distinguish the coupling topology illustrated in Figure 4.5. It consists of a limited set of coupling templates where a given unidirectional communication involves only two operators of two different submodels. These coupling templates reflect the relation between the submodels in the *SSM*. Indeed, coupling topology of a multiscale application can be defined as a weighted graph representation. The nodes are instances of submodels and the weights are the number of required instances of a given submodel. A *SEL* representation with a coupling topology give more insight about the runtime behavior and the dynamic of submodels. The coupling strategy can be classified based on three determined criteria: type of topology graph, number of submodel instances and data transfer model [27].

A graph representation may be cyclic or acyclic. Acyclic graphs are easy to understand and to execute given that the coupled submodels run sequentially. For instance, an instance of submodel A can be coupled acyclically to a fixed or dynamic number of instances of submodel B. Dynamic number of instances can be known only on runtime and, therefore, a given submodel should have a mechanism to spawn dynamically the needed number instances of an other submodels. The dynamic number of instances may add more complexity to run the multiscale application. If the interaction between submodels instances are cyclic then the coupling topology should be cyclic. A cycle may be composite in the sense that it can include several submodels instances coupled to other sub-workflows.

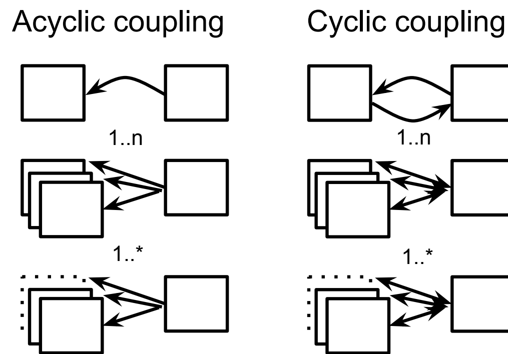


Figure 4.5 – the coupling topology of submodel instances. the number of submodel instances can be fixed in advance before execution ($1..n$) or dynamic during runtime ($1..*$). Similarly, synchronization between the submodel instances can fixed or dynamic.

MML

The Multiscale Modeling Language (*MML*) is a high level language that puts the previous concepts under the same umbrella: scale, submodel definition, coupling of submodels, topology and coupling templates. It bridges the theoretical and design phases and can be seen as a *mock-up* which serves as a *blueprint* for the final development of multiscale applications. To do so, the *MML* requires firstly a general information regarding the submodels implementation such as submodel inputs/outputs and its used scales. Secondly, it requires information regarding the type of data to transfer and how data should be treated. Finally, information about runtime requirements is needed to give an idea where and how submodules can be executed. The *MML* gives in the end a guide-line about the computational architecture of each submodel and how to couple it to other conceptual elements in order build a computation multiscale application.

In *MML*, the coupling between the submodels may be expressed either directly or through components called *mapper*, *conduit* and *filter*.

- *mapper*: is a component that receives data from one or more submodels, performs a computation reflecting the nature of the coupling, and sends updated information to the corresponding recipient submodels (many-to-many relation). In some situations, submodels modeling physical systems such as water junctions (Gate, spillway, etc, in the case of a hydrology application) can be abstracted as *mapper*s through a phenomenological equation or actually simulated with a fully detailed flow model.
- *filter*: is a component used to describe the data-flow links between submodels such conversion of distance unities from *kilometers* to *meters* (one-to-one relation).
- *conduit*: is an abstract notion and represents the link between the *SEL* operators of connected components (submodels, *mapper*s and *filter*).

In *MML* a submodel has input and output ports through which it sends and receives data in the form of messages. For instance, the intermediate observation $\mathbf{O_i}$ is an example of data sent out using an output port. The message is transported through *conduits*. A *conduit* is a unidirectional connection between one input port and output ports of two coupled components. With analogy to the *SEL*, each operator can be associated to an input or output port in a submodel: the operators $\mathbf{O_i}$ and $\mathbf{O_f}$ can use output ports to send data, and $\mathbf{F_{init}}$, $\mathbf{S}$ and $\mathbf{B}$ ones can use input ports to receive data. Thus, a *conduit* between an input and output ports is nothing else than the coupling between two *SEL* operators of two submodels. For example, a cyclic coupling between two submodels A and B may require to coupling $\mathbf{O_i}^A$ with $\mathbf{S}^B$ and $\mathbf{O_i}^B$ with $\mathbf{S}^A$. An acyclic coupling may require to couple $\mathbf{O_f}^A$ with $\mathbf{F_{init}}^B$, for example. Hence, the *MML* gives a powerful way to understand the multiscale workflow just through the coupling between submodels operators.

The *MML* was proposed with the idea that each submodel should not have any concern about the specificities of submodels to which it may be coupled. However, data transferred in a *conduit* should arrive in the right format, understandable by the target submodel. In this case, a *filter* can be applied to a *conduit* in order to convert data to the correct format. It is worth noting here that a *filter* role is to make simple conversion of data as shown in Figure 4.6.

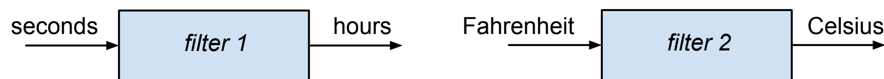


Figure 4.6 – Example of *filters* applied to *conduits*

A *filter* is applied only on a conduit between two submodels. It can not be used when a simple conversion requires receiving data from multiple submodels/*mapper*s or when a data should be transferred at the same time to multiple target submodels. In this case, *mapper* component should be used. A *mapper* can have multiple input and output ports, and can handle different data formats and types. It can be coupled to submodels and also to other *mapper*s. *mapper*s are smart enough to handle multiscale coupling between two submodels. For example, let's consider the case of two submodels A and B having overlapped time and space scales. B should compute faster than A and has a fine-grained domain. So, the *mapper* should ensure that A and B receive the adequate number of message compliantly to their time steps.

At a higher level of description, submodels have no concern how to split/deliver data or to know to how many submodels it may be coupled. The only coupling information they need to know is what kind of data type they should send/receive and to/from which ports. *mapper*s bring more modularity when developing submodels and building the multiscale workflow of the whole application, especially in the case of coupling two submodels with different scales: the knowledge of these scales is in the *mapper*s only, making the submodels autonomous and independent "solvers", blind to the coupling structure.

The *MML* provides both XML and graphical representations, called *xMML* and *gMML* respectively, to describe the coupling of submodels and the data transfer in the multiscale application. The *xMML* is based on *XML* and *gMML* is a graphical implementation inspired from UML which is more human readable. Note that it is easy to switch between the two representation.

xMML

The *xMML* was designed to make the multiscale description portable on machine and expendable. An XML description allows us to rigorously describe each submodel instance (instance identifier, scales, *SEL* ports, data type, etc) and its coupling (connection between ports) with the other submdodels instances. Listing 4.1 presents the *xMML* description of two water segments of river coupled through a gate as depicted in Figure 4.8a (see chapter 5 for more details about the coupling). A water segment is modeled by a *1D-Shallow water* submodel (lines 17-27) called `SW1D` and connected to a gate and a wall. The definition of this submodel is illustrated with the `<ns3:submodel>` XML tag. The gate and wall are *mappers* named `Wall` (lines $9-15$) and `Gate` (lines 29-39), respectively. The definition of these two *mappers* is illustrated with the `<ns3:mapper>` XML tag (line 9).

Listing 4.1 – The *xMML* description of the two water sections coupling

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <ns3:model id="xmml-1410356773143" name="MAD generated XMML" xmml_version="0.4"
     xmlns:ns3="http://www.mapper-project.eu/xmml" xmlns:mad="http://www.mapper-project.eu/mad">
3    <ns3:description>two water sections soupled through a gate</ns3:description>
4
5    <ns3:definitions>
6      <ns3:datatype id="file"/>
7      <ns3:datatype id="HashMap"/>
8
9      <ns3:mapper id="Wall" interactive="no" name="Wall" type="fan-out">
10       <ns3:description>Wall that performs bounce back operation</ns3:description>
11       <ns3:ports>
12         <ns3:in datatype="HashMap" fixedName="false" id="port-id-f_in"/>
13         <ns3:out datatype="HashMap" fixedName="false" id="port-id-f_out"/>
14       </ns3:ports>
15     </ns3:mapper>
16
17     <ns3:submodel id="SW1D" interactive="no" name="SW1D">
18       <ns3:description>This is a shallow water segment without bounce back border. So, use the wall
           junction to couple a border with a BC wall.</ns3:description>
19       <ns3:timescale delta="0" total="0"/>
20       <ns3:ports>
21         <ns3:in datatype="HashMap" fixedName="false" id="port-id-left_in" operator="B"/>
22         <ns3:in datatype="HashMap" fixedName="false" id="port-id-right_in" operator="B"/>
23         <ns3:out datatype="HashMap" fixedName="false" id="port-id-left_out" operator="Oi"/>
24         <ns3:out datatype="HashMap" fixedName="false" id="port-id-right_out" operator="Oi"/>
25         <ns3:out datatype="file" fixedName="false" id="port-id-out_data" operator="Of"/>
26       </ns3:ports>
27     </ns3:submodel>
28
29     <ns3:mapper id="Gate" interactive="no" name="Gate" type="fan-out">
```

```
30              <ns3:description>This is a linear gate used for the irrigation canal application</ns3:description>
31              <ns3:ports>
32                  <ns3:in datatype="HashMap" fixedName="false" id="port-id-left_in"/>
33                  <ns3:out datatype="HashMap" fixedName="false" id="port-id-left_out"/>
34                  <ns3:in datatype="HashMap" fixedName="false" id="port-id-right_in"/>
35                  <ns3:out datatype="HashMap" fixedName="false" id="port-id-right_out"/>
36                  <ns3:out datatype="file" fixedName="false" id="port-id-out_data" operator="Of"/>
37              </ns3:ports>
38          </ns3:mapper>
39      </ns3:definitions>
```

The example consists of two instances *section-1* and *section-2* of the submodel SW1D, two instances *wall-1* and *wall-2* of the *mapper* Wall and one instance of the *mapper* Gate. All the instances of submodels are described in the <ns3:topology> tag (line 40) and each instance description is detailed within the <ns3:instance> tag. The workflow execution of the example is as follows: each submodel starts and initializes its section domain. Then, during each time step, each section computes its domain and send boundary data (water hight and velocity) to the gate and the wall to which it is connected. The gate and the wall wait until they receive the border data, then compute and send the new calculated the new boundary to each section. In the end, each section delivers a final observation.

```
40      <ns3:topology>
41          <ns3:instance id="wall-1" mapper="Wall">
42              <ns3:extra>
43                  <mad:implementations/>
44                  <mad:portMapping>
45                      <mad:portMapping id="id-002" name="f_in"/>
46                      <mad:portMapping id="id-003" name="f_out"/>
47                  </mad:portMapping>
48                  <mad:position x="434" y="617"/>
49              </ns3:extra>
50          </ns3:instance>
51          <ns3:instance id="wall-2" mapper="Wall">
52              <ns3:extra>
53                  <mad:implementations/>
54                  <mad:portMapping>
55                      <mad:portMapping id="id-005" name="f_in"/>
56                      <mad:portMapping id="id-006" name="f_out"/>
57                  </mad:portMapping>
58                  <mad:position x="437" y="145"/>
59              </ns3:extra>
60          </ns3:instance>
61          <ns3:instance id="section-1" submodel="SW1D">
62              <ns3:extra>
63                  <mad:implementations>...</mad:implementations>
64                  <mad:portMapping>
65                      <mad:portMapping id="id-008" name="left_in"/>
66                      <mad:portMapping id="id-009" name="right_in"/>
67                      <mad:portMapping id="id-010" name="left_out"/>
68                      <mad:portMapping id="id-011" name="right_out"/>
69                      <mad:portMapping id="id-012" name="out_data"/>
70                  </mad:portMapping>
71                  <mad:position x="216" y="231"/>
72              </ns3:extra>
73          </ns3:instance>
```

```
74          <ns3:instance id="section−2" submodel="SW1D">
75              <ns3:extra>
76                  <mad:implementations>...</mad:implementations>
77                  <mad:portMapping>
78                      <mad:portMapping id="id−014" name="left_in"/>
79                      <mad:portMapping id="id−015" name="right_in"/>
80                      <mad:portMapping id="id−016" name="left_out"/>
81                      <mad:portMapping id="id−017" name="right_out"/>
82                      <mad:portMapping id="id−018" name="out_data"/>
83                  </mad:portMapping>
84                  <mad:position x="215" y="469"/>
85              </ns3:extra>
86          </ns3:instance>
87          <ns3:instance id="gate" mapper="Gate">
88              <ns3:extra>
89                  <mad:implementations>...</mad:implementations>
90                  <mad:portMapping>
91                      <mad:portMapping id="id−020" name="left_in"/>
92                      <mad:portMapping id="id−021" name="right_in"/>
93                      <mad:portMapping id="id−022" name="left_out"/>
94                      <mad:portMapping id="id−023" name="right_out"/>
95                      <mad:portMapping id="id−024" name="out_data"/>
96                  </mad:portMapping>
97                  <mad:position x="423" y="357"/>
98              </ns3:extra>
99          </ns3:instance>
```

The coupling information about submodel and *mappers* instances are described with in the `<ns3:coupling>` tag where only two ports can be connected (lines 100-108). the ports definition is described within the `<ns3:port>` tag, in which we find the type of operator and the type of data it handles (see for instance line 21).

```
100         <ns3:coupling mad:connectionId="id−025" from="section−1.left_out" mad:fromPortId="id−010"
        to="wall−2.f_in" mad:toPortId="id−005"/>
101         <ns3:coupling mad:connectionId="id−026" from="wall−2.f_out" mad:fromPortId="id−006"
            to="section−1.left_in" mad:toPortId="id−008"/>
102         <ns3:coupling mad:connectionId="id−027" from="section−1.right_out" mad:fromPortId="id−011"
            to="gate.left_in" mad:toPortId="id−020"/>
103         <ns3:coupling mad:connectionId="id−028" from="gate.left_out" mad:fromPortId="id−022"
            to="section−1.right_in" mad:toPortId="id−009"/>
104         <ns3:coupling mad:connectionId="id−029" from="section−2.left_out" mad:fromPortId="id−016"
            to="gate.right_in" mad:toPortId="id−021"/>
105         <ns3:coupling mad:connectionId="id−030" from="gate.right_out" mad:fromPortId="id−023"
            to="section−2.left_in" mad:toPortId="id−014"/>
106         <ns3:coupling mad:connectionId="id−031" from="section−2.right_out" mad:fromPortId="id−017"
            to="wall−1.f_in" mad:toPortId="id−002"/>
107         <ns3:coupling mad:connectionId="id−032" from="wall−1.f_out" mad:fromPortId="id−003"
            to="section−2.right_in" mad:toPortId="id−015"/>
108     </ns3:topology>
109 </ns3:model>
```

gMML

In the graphical version of the *MML*, submodels are represented by boxes connected through edges with markers at their extremities. These markers specify the type of

the corresponding *SEL* operators involved in the data exchanging as described in Figures 4.1 and 4.7. *mapper*s are presented by an hexagonal box and *filter*s are represented by a rectangle with rounded corners. Each component is labeled by a name inside its box.

The edges represent the *conduit*s between the different component instances. Undefined edge markers are used for *mapper*s and *filter*s ports. Edge tail is filled empty and edge head is filled black to indicate the sense of data transfer. A black circle represents the execution start of a component and encircled black circle represents its end of execution.

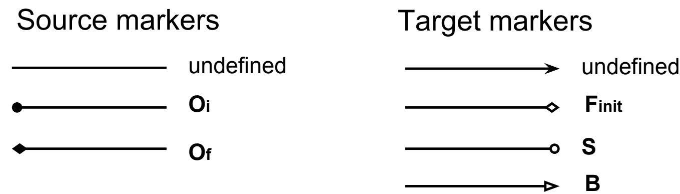
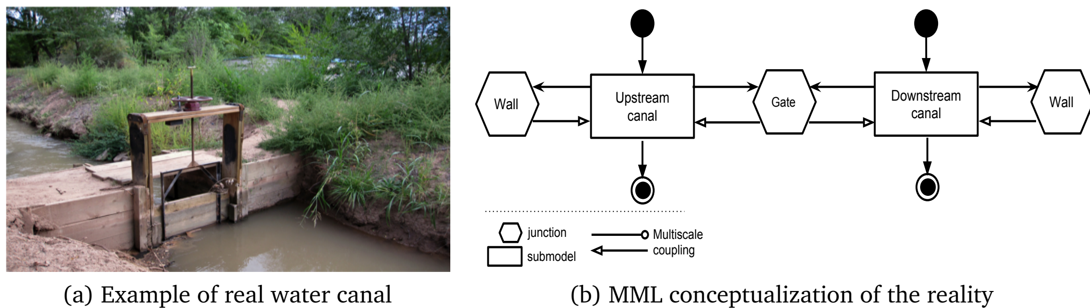


Figure 4.7 – Coupling *conduit*s type [60]. The *undefined* marker is used when coupling *mappers* to other submodels.

Figure. 4.8b illustrates the *gMML* presentation of the system illustrated in Figure 4.8a. However, compared to *xMML*, *gMML* does not include information regarding the scales and the variables of each submodel.



(a) Example of real water canal

(b) MML conceptualization of the reality

Figure 4.8 – Modeling step

4.4.2 Computation Design

As mentioned in section 4.4.1, the *MML* representation enables us to describe the coupling between the submodels within the same multiscale application. It includes also the information about when the data are exchanged during execution. Besides, it gives an overview of the submodel ports, submodel scales and how it should be coded.

There exist tools that realize the *gMML* and *xMML* representations: MaMe (Mapper Memory) and MAD (Mapper Application Designer) [99] developed in the context of the MAPPER project. MaMe is based on a semantic integration technology described in [74] and composed of a database and web based interface that allow end-users to easily

describe submodels, *mappers* and *filters*, to define the operators and conduits operations, and to catalog them into the database. At this stage, a kind of *mock up* of the components is defined: each *mock up* has an name, description, list of input variables, a definition of its *SEL* operators and references to its binary files or modules to load for each platform type. With this, their end-users have the required information to know how to use these components to build his application.

Figure 4.9 – ScreenShot of a submodel in the MAD interface. `SW1D` is the name of a submodel modeling a rectangular water section with two boundaries. The submodel has two input ports and three output ports. The input port (a triangle in a green circle referring to an arrow marker as described in Fig. 4.7) corresponds to the **B** *SEL* operator used to receive data, in order to update a boundary side of the water section. The two output ports with a black circle marker correspond to the $\mathbf{O_i}$ operator. Each output port sends out an intermediate observation of the corresponding boundary. The output port with a black diamond marker corresponds to the $\mathbf{O_f}$ operator and sends out a final observation at the end of the submodel execution.

The MAD tool is also a web interface which reads the MaMe database, displays submodels/mappers *mock-up* as icons (Fig. 4.9) and enables users to drag-and-drop, instantiate them in a graphical design space and couple them together in order to compose a cyclic or acyclic computational workflow (step 2 in Figure 4.1). To illustrate better this concept, Figure 4.10 shows the MAD drag-and-drop feature.

Figures 4.10 and 4.11 illustrates the design step of a multiscale application where a computational workflow is built on the fly using the MAD tool. The connection between components is handled by the defined operators and mappers (arrows in the figure) and inter-scale data are transmitted throughout conduits accordingly to the data type described in the MaMe database. It is worth reminding here that the submodels/mappers can be implemented in different languages (C++, Java, etc.) and have no information regarding the other components, thus, illustrating the “Lego based” aspect of this approach. For instance, the gate can be programmed in `Java` and send/receive an array of double as data type to/from the two water sections that are programmed in `C++` language.

4.4.3 Multiscale Implementation

Multiscale applications involve interaction between different scientific codes which iteratively exchange data at different scale or of different nature. As mentioned earlier,

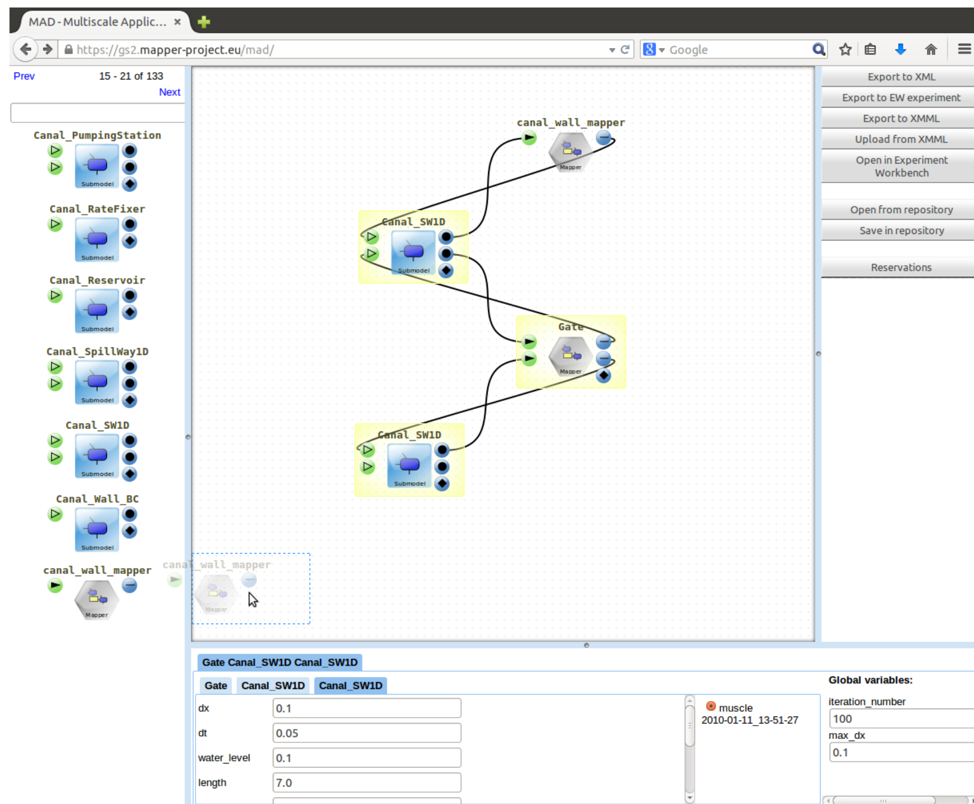


Figure 4.10 – Screen-shot of the MAD web interface

the actual coupling of the submodels can be implemented with the MUSCLE2 framework [26, 111].

MUSCLE2: the Framework

MUSCLE2 is a framework to run time-driven based multiscale applications. It offers a programming library to develop submodels and a runtime environment to couple and run them on local or distributed resources. The library part of MUSCLE2 consists of `C/C++`, `Java`, `Python` and `Fortran` APIs (step 3 in Figure 4.1). They are used to implement the submodels and to handle the data transfer compliantly to the *MML* description of each submodel. Data transfer between submodels is handled mainly by send()/receive() methods calls of the MUSCLE2 library in each submodel. At this programming stage, coupling information between submodels is not needed. However, to prepare the multiscale coupling, the scientists have to envisage implementing the required scale bridging techniques in other *mapper*s and/or submodels compliantly to the *MML* description (see Listing 4.3).

The runtime environment of MUSCLE2 uses the *MML* description to run the multiscale application on distributed infrastructure. It starts each submodel instance on a target

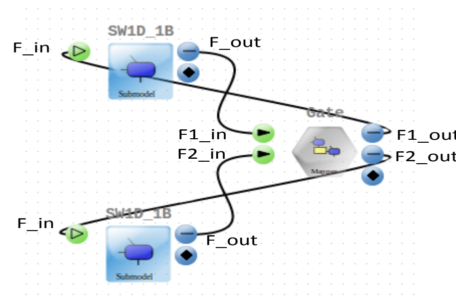


Figure 4.11 – Computation design step

computing resource: desktop machine, cluster or supercomputer. MUSCLE2 runtime environment verifies the coupling between instances ports and ensures that data is sent and received by the right submodel instances. This will be discussed later in section 4.4.4.

MUSCLE2: the library

As presented previously, MUSCLE2 provides `C/C++`, `Java`, `Python` and `Fortran` APIs. Each of these APIs offers the same methods to define the submodels ports and handle the data transfer. Table 4.1 gives an overview of some used methods. For sake of simplicity, only some of `C` methods will be illustrated.

The MUSCLE2 methods are easy to use and do not require any specific programming paradigm. One can easily use an existing or legacy code by adding it a kind of a wrapper in which methods of MUSCLE2 and of the original code are called. MUSCLE2 comes with predefined data types (see Table4.2). They allow data to be transferred between different kinds of computer architectures and operating systems expect for MUSCLE_RAW type which requires the user to handle the endianness when transferring data. Note that the methods MUSCLE_Send and MUSCLE_Receive do not automatically free the data pointers passed in argument for the case of `C/C++` language.

Implementation example

From a programmer point of view, a multiscale application can be seen as follows:

- a multiscale application is composed of several codes coupled together. Each code is a program that can be independently implemented in MPI/c++, Java, Fortran, etc.
- adapting an existing code to the MMSF approach is straightforward.
- codes involved in a multiscale application should be easily replaced without affecting the other codes. They can be also reused in other multiscale applications.

Method	Description
`muscle_error_t MUSCLE_Init(int * argc, char *** argv)`	It initializes the MUSCLE2 runtime environment. It reads the arguments of the main program of the launched submodel instance. This method must be called before using the other methods of the MUSCLE2 API. It returns an error message in case of problem. The parameter `argv` contains the required MUSCLE arguments to initialize the runtime.
`void MUSCLE_Finalize()`	It cleans the MUSCLE2 runtime environment. It must be called in the end of the submodel program.
`muscle_error_t MUSCLE_Send(const char *exit_name, void *array, size_t datasize, muscle_datatype_t type)`	It sends the data pointed by the `*array` variable. Data has the size `datasize` and the type `type` (see Table 4.2). The *conduit* is identified by the port name (parameter `exit_name`) which should exist in the *MML* description.
`void* MUSCLE_Receive(const char *entrance_name, void *array, size_t * datasize, muscle_datatype_t type)`	It listens on the inout port `entrance_name` and receives data pointed by `array` variable. Data has the size `datasize` and the type `type` (see Table 4.2).
`char * get_property(char* name)`	It reads input parameter (identified by `name`) for the corresponding submodel instance from the MUSCLE2 configuration file. The configuration file of MUSCLE2 will be presented later in section 4.4.4

Table 4.1 – Some of the MUSCLE2 methods for `C` programming.

muscle_datatype_t	C/C++ data type	Java data type
MUSCLE_DOUBLE	double *	double[]
MUSCLE_FLOAT	float *	float[]
MUSCLE_INT32	int *	int[]
MUSCLE_INT64	long *	long[]
MUSCLE_STRING	char *	String
MUSCLE_RAW	unsigned char *	byte[]
MUSCLE_COMPLEX	muscle::ComplexData *	any other object

Table 4.2 – MUSCLE2 data type.

- codes can be loosely and/or tightly coupled. This brings a modularity and an easiness when building a multiscale application workflow.
- a multiscale application can run either on a local machine or on distributed infrastructure (MPI cluster and supercomputers): “Design once, run anywhere”.

MUSCLE2 is easy to integrate in a given code. In case of using the MMSF approach, it would be better to implement submodels compliantly to the *SEL*. The following submodel execution loop can be used to implement a given submodel:

```
init();
 while ( notEnd() ){
  intermediateObservation();
  incrementTimeStep();
  solve();
  updateBoundary();
}
finalObservation();
```

The `init()`, `solve()` and `updateBoundary()` methods require reading input data to initialize, solve and update the submodel domain. The `intermediateObservation()` and `finalObservation()` only generate data. It is important that the implementation of a submodel should be independent of the rest of submodels. That is why *mapper*s and *filter*s are needed to convert the generated data during observation of one submodel into the adequate format required by the target submodel during its initialization, solving or updating calls. This proposed execution loop remains indicative and other way to implement submodel can be used. In what follows, we will use Figure 4.11 to describe and explain the implementation and execution steps.

Listing 4.2 illustrates the pseudo-code of the `SW1D` submodel (Figure 4.11), developed in `C` language. In line 5, the MUSCLE2 runtime is initialized. Variables dt and dx (lines 7-8) are read from the MUSCLE2 configuration file (see section 4.4.4). The method `init()` in line 9 initializes the submodel instance. The variable dx and dt are used to define the size of the array of data which represents the grid domain of the section (submodel). They are also used to define the solver variable used for the fluid simulation.

The submodel sends its observation data (array of double) on the port `f_in` using the `MUSCLE_Send` method. This data will be received by the target component connected to `f_in` port of this submodel. The names of ports must be identical to those specified in the configuration file. Note that The MAD tool generates a `Ruby` configuration file (historically, this file was named CxA file in reference to the theoretical concepts of CxA) from the *gMML* or *xMML* workflows. It is the configuration file understandable by MUSCLE2 and it defines the submodels names, scales, binaries, etc, and describes the coupling of all the components as illustrated in Listing 4.4. This CxA configuration file is used by MUSCLE2 during the runtime deployment (see section 4.4.4).

Here, the `SW1D` submodel is not concerned by which submodel, *mapper* or *filter* the data will be received. The `solve()` method (line 14) solves numerically the physical process corresponding to the submodel (fluid simulation) and it is often a CPU and time consuming method. For instance, it can be an MPI based method and, thus, it can run on multicore or parallel machines. After solving the problem domain, the submodel receives on the port `f_out` data required to update its boundary domain (line-16). In the end of simulation, the submodel instance may perform a final observation (line-20) (such as the

boundary water flow) where it can send the simulation results to the next submodel to which it is coupled.

Listing 4.2 – Implementation of the `SW1D` submodel in `C` language

```
#include <muscle2/cmuscle.h>

int main(int argc, char *argv[])
{
  MUSCLE_Init(&argc, &argv); // initialite MUSCLE2 runtime environment
  size_t & sz=10;
  double dt= MUSCLE_Get_Property("dt"); // read time and spatial scales
  double dx= MUSCLE_Get_Property("dx"); // from the config file
  init(dt,dx); //F_init operator
  ...
  while (! isConverge()){
      double* observation =Observation();// get the macroscopic values: (Oi)
      MUSCLE_Send("f_out", observation, sz, MUSCLE_DOUBLE); //send data
      solve();// S operator
      double *dataFromGate = (double *) MUSCLE_Receive("f_in", (void *)0, &sz, MUSCLE_DOUBLE); //receive
      applyBoundary(dataFromGate); //(B) operator
      free_data(observation,MUSCLE_DOUBLE); // free allocated data pointer
      free_data(dataFromGate,MUSCLE_DOUBLE);
  }
  finalObservation();
  ...
  MUSCLE_Finalize();// end MUSCLE
  return 0;
}
```

Let us assume that the upstream and downstream canals have a different spatial and temporal resolutions. In this case, grid refinement techniques must be used for the coupling. Here, the gate can be programmed to handle two different frequencies of send()/receive() operations for each side.
The Listing 4.3 shows the pseudo-code of the *mapper* gate programmed in `Java`. It is implemented as a `Java` class (line 1) that extends a predefined *CAController* class, part of the MUSCLE2 `Java` API. We instantiate *entrance* and *exit* objects (represent the *conduits*) and bind them to the corresponding ports names and data type as described in the *MML* description: `f1_in`, `f1_in`, `f1_out` `f2_out` in lines 8-11.

Listing 4.3 – Implementation of the `Gate` mapper in `Java`language

```
public class Gate extends muscle.core.kernel.CAController {

  private ConduitEntrance<double[]> f_out;
  private ConduitExit<double[]> f_in;
  ...
  @Override
  protected void addPortals() {
      f1_out = addEntrance("f1_out", double[].class);
      f2_out = addEntrance("f2_out", double[].class);
      f1_in = addExit("f1_in",double[].class);
      f2_in = addExit("f2_in",double[].class);
  }
```

The main code of the gate (in the code below) should be implemented in the method *execute()* (line 14). The MUSCLE2 runtime environment ensures that the send/receive operations occur between the right submodels ports compliantly to *MML* coupling configuration. In the `init()` method (line 16), the gate reads its parameters (such as its width, length and angle of opening) from the CxA configuration file. Then, the gate reads the first observation sent on its ports `f1_in` and `f2_in` (lines 17-19). These information encapsulate the time and space scale of each submodels. The variable `round_step` (line-20) is the number of iterations that the fine grain submdodel should perform per one iteration of the coarse grain one. Note that the time scale of the coarse grain submodel should be a multiple (integer) of the time scale of the fine grain one. Lines 22-25 define the input/output ports of the fine and coarse grain submodels based on their time scale.

```
13    @Override
14    protected void execute() {
15    double [] data_tosend;
16      init();// (F_init) operator
17      double [] sub1=(double[]) f1_in.receive();// receive boundary data
18      double [] sub2=(double[]) f2_in.receive();// receive boundary data
19      double dt1=sub1[0]; double dt2=sub2[0];
20      int round_step=1;
21      round_step=(dt1>dt2)? dt1/dt2 : dt2/dt1;
22      fine_port_in=(dt1>dt2)? f2_int: f1_int;
23      fine_port_out=(dt1>dt2)? f2_out: f1_out;
24      coarse_port_in=(dt1>dt2)? f1_int: f2_in;
25      coarse_port_out=(dt1>dt2)? 1_out: f2_out;
```

Thus, during each iteration, the gate knows how many times it should communicate with the fine grain submodel per one iteration of the coarse submodel instance. When the gate should read from both submodels (line-29), it receives from both submodels, computes directly the new water boundary values (line-31) and send them to the submodels (lines 32-33). In the other case, the gate receives data only from the fine grain submodel instance, it performs a grid refinement technique (line 35) and possibly time and scale interpolation to compute the new water boundary only for the fine grain submodel.

```
26      int it=0;
27      while (! isEnd() and it++){
28        double [] dataFineGrid= fine_port_in.receive();
29        if ( it%round_step == 0){//should read from both submodels
30                             double [] dataCoarseGrid= coarse_port_in.receive();
31                             double [][] result=adjusteWaterLevel(dataFineGrid,dataCoarseGrid);
32                             fine_port_out.send(result[0]);//send to the fine grain submodel
33                             coarse_port_out.send(result[1]);//send to the coarse grain submodel
34        }else{ //should read only from the fine grain submodel
35          double [] result=gridRefinementTechnique(dataFineGrid,dataCoarseGrid);
36          fine_port_out.send(result);//send to the fine grain submodel
37        }
38      }
39      saveResults();//(Of) operator
40    }}
```

4.4.4 Execution

The runtime part of MUSCLE2 is designed to be easy to install, configure and use. It supports Unix/Linux and MacOSX operating systems. In what follows, we present the execution phase of the MMSF approach and how to run a multiscale application on a distributed resources.

MUSCLE2: the runtime environment

Let's consider again the example of Figure 4.11. The flow in the canal sections are computed by two instances of the submodel SW1D, hence the names SW1D1 and SW1D2, implementing a Lattice Boltzmann 1D shallow water flow simulation.

Listing 4.4 – Example of a Ruby configuration file.

```
1  # Classes and binary path. Here, CANAL_HOME is an OS environment variable.
2  add_classpath ENV['CANAL_HOME']+'/submodels/bin/'
3  # declare kernels
4  cxa.add_kernel('SW1D1', 'SW1D_binary')
5  cxa.add_kernel('SW1D2', 'SW1D_binary')
6  cxa.add_kernel('Gate', 'Gate_binary')
7  # configure connection scheme
8  cs = cxa.cs
9  cs.attach('SW1D1'=>'Gate') {tie('f_out','f1_in')}
10 cs.attach('SW1D2'=>'Gate') {tie('f_out','f2_in')}
11 cs.attach('Gate'=>'SW1D1') {tie('f1_out','f_in')}
12 cs.attach('Gate'=>'SW1D2') {tie('f2_out','f_in')}
13 #parameters
14 cxa.env['SW1D1:dx']=0.05
15 cxa.env['SW1D2:dx']=0.025
16 cxa.env['SW1D1:dt']=0.025
17 cxa.env['SW1D2:dt']=0.0125
```

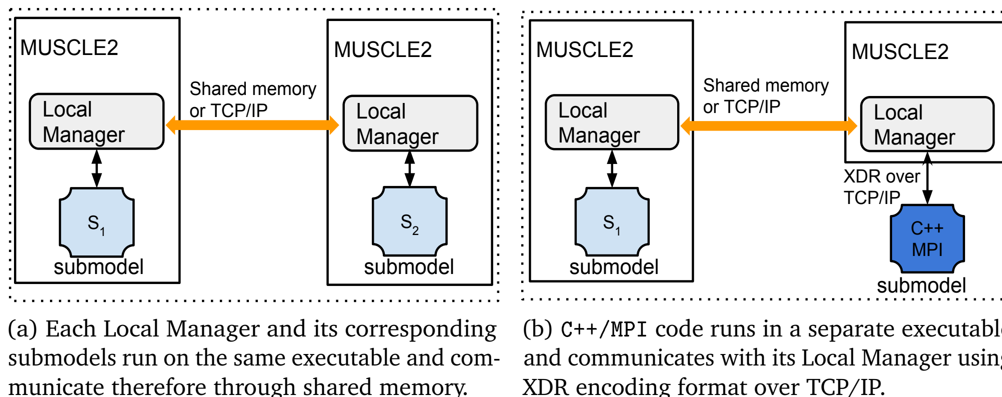
Listing 4.4 illustrates the CxA configuration file of the example. The implementation of the submodel `SW1D` is a `C++` binary file called `SW1D_binary` and the the gate *mapper* is a `Java` binary located in `Gate_binary` binary (lines 4-5). The submodels are declared in lines 4-6. Each component is identified by a name and the binary file that implements it. Lines 9-12 establish the coupling between the ports of each submodel. Here for instance, f_out and f_in are both boundary operators from which the canal section *SW1D1* sends and receives boundary data, respectively. In lines 14-17, we describe the input variables needed by submodels binaries (the solver).

The next step is related to *how one can launch submodel instances on computing resources?*. A submodel instance is launched on a machine by issuing the command line `muscle2` followed by options as shown below:

```
muscle2 -c ruby_file.conf -M -m instanceName
```

The `muscle2` reads the `Ruby` configuration file `ruby_file.conf`. Then, the option `-M` indicates that the current execution is the main execution site (more details will given later). Finally, the option `-m` indicated the name of the submodel instance to launch. This instance name should figure in the `Ruby` configuration file. Note that it is possible to start all the component instances of the application workflow by issuing the `-allKernels` option instead of the `-m` one.

To better understand how a distributed execution can be performed, an overview of the MUSCLE2 runtime environment is needed. Indeed, the MUSCLE2 runtime environment consists of a Local Manager per submodel instance and one Simulation Manager. The Local Manager is launched in parallel to the submodel instance and handles the data transfer and communication. The Simulation Manager is launched automatically each time the option `-M` is specified in the muscle2 command. Once started, The Simulation Manager keeps having a global overview of all running instances of Local Managers and their locations. Figure 4.12 gives an overview of the communication strategy within a MUSCLE2 simulation.



(a) Each Local Manager and its corresponding submodels run on the same executable and communicate therefore through shared memory.

(b) `C++/MPI` code runs in a separate executable and communicates with its Local Manager using XDR encoding format over TCP/IP.

Figure 4.12 – MUSCLE2 communication design. A Local Manager starts in parallel an instance of its corresponding submodel. Submodels start computation, send data and block until new data are available. Local Managers handle the send/receive operations. Communication between Local Managers is represented by the orange arrow annotated with the communication protocol.

The communication between two Local Managers is done through shared memory if they are running on the same machine node and TCP/IP on two separated nodes. The same communication strategy is applied between a Local Manage and its corresponding submodel instance. Thus, it is possible to communicate through shared memory (Figure 4.12a) and through TCP/IP (Figure 4.12b) for the case of submodels coded in `MPI/C++` or `MPI/Fortran`. When the submodel code is a MPI based binary, there is two way to launch it. The first way is to specify "`submodelName:mpiexec_command`" and "`submodelName:mpiexec_args`" lines in the `Ruby` configuration file and the `-native`

option when launching the muscle2 command line:

```
# set up the mpiexec argument for SW1D1 submodel instance (100 cores)
cxa.env["SW1D1:mpiexec_command"] = "mpiexec"
cxa.env["SW1D1:command"] = "-np 100"
```

The second way to do that is possible without adding `mpiexec` arguments in the configuration file; Simply, one can run the binary that was compiled with the MUSCLE2 library as follows:

```
mpiexec -np n_cores MPICode arg1 arg2 -- -M -c ruby_file.conf submodelName
```

Here, when the `MPICode` starts, it initializes the MUSCLE2 runtime environment using command line arguments specified after the `'--'` character.

A MUSCLE2 computation can be launched over local or distributed grid e-infrastructures. The MUSCLE2 library should be installed on all the computing nodes. Its role is to start the submodels described in the `Ruby` configuration file, to establish the communication between them and to handle inter-scale data transfer. In case of local computation, all the submodels can be run on the same machine or distributed on machines within in the same local area network. In case of large distributed execution, involving machines on a wide area network, MUSCLE2 starts the involved submodel instances over different sites (machines). However, It is required to indicate where the main site is located by using the option `-M IP:PORT` in the command line (Figure 4.13): `IP` stands for the ip address of the main execution site where the Simulation Manager is listing on the port `PORT`. When a new submodel instance is launched, it contacts the Simulation Manager that notifies then all the other started Local Managers. Local managers can directly communicates whenever it is possible. Otherwise (for example, there is a firewall between the execution sites), MUSCLE2 library uses the MUSCLE Transport Overlay (MTO), a kind of daemon proxy that allows data transfers between submodels even if they are running on different supercomputers/clusters.

As depicted in Figire 4.14, MTO instances provide access points to the local area networks corresponding to the computing nodes. The configured MTO instances accept only connections coming from remote master nodes and transmit data intended for another supercomputer/cluster to the relevant MTO. The MTO configuration should be installed on each execution site. MTO is a daemon which is enabled interactively on each master node of each execution site and should open a public port. Internally, MTO provides a local ports ranges that allow Local Managers, having local IP addresses and running on the same cluster nodes to communicate internally. When two Local Managers are running on different sites, communication goes through the MTO connection. Note that since the

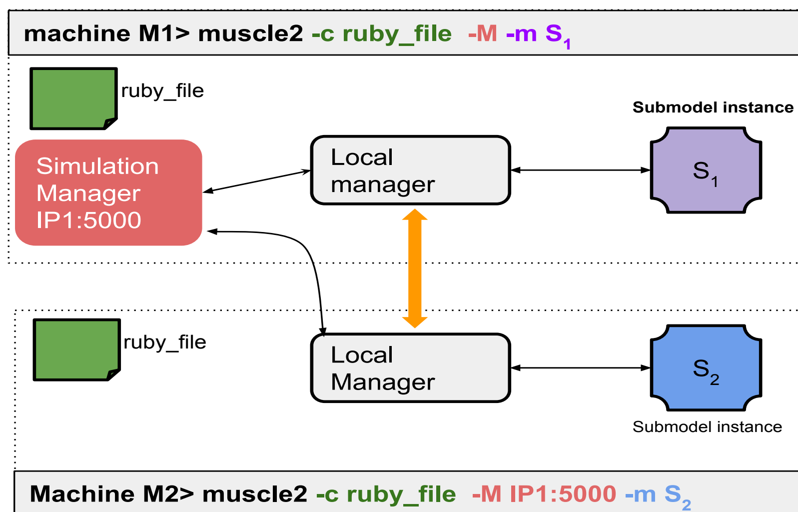


Figure 4.13 – Example of how to run manually a simple MUSCLE2 simulation over two machines: a first command line starts the submodel S_1, Local Manager and the simulation Manager on *machine1*. A second command line start the submodel S_2 and Local Manager on *machine2*. Both Local Managers connect to the Simulation Manager (IP address IP1 and port 5000) and then communicate.

private IPs addresses are not unique, all the running MTO daemon should have a disjoint ports ranges.

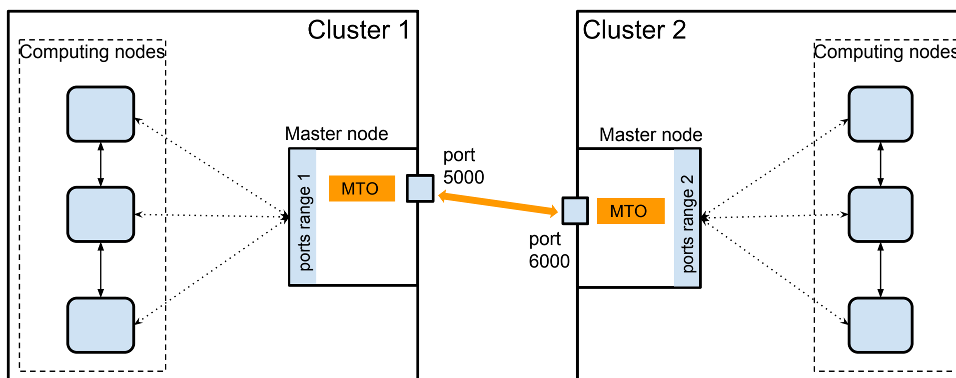


Figure 4.14 – MTO communication protocol. Internal ports ranges in all clusters must be disjoints. MTO public ports (for example ports 5000 and 6000) should be publicly reatchable.

For applications having cyclic communications, distributed simulations require coordination and efficient scheduling of submodels submitted at the same time on the distributed computing sites (supercomputers and clusters). On top of MUSCLE2, the QoSCosGrid middleware is used (QCG) [11] for the submission. It supports an advance reservation mechanism, essential to block several computing resources for the same period of times.

Job submission can be made through the command line or through the GridSpace [38, 39] web based environment. The latter allows the end-user to reserve graphically time slots on the available EGI [53] and PRACE [121] resources, and to export the computation description directly from the MAD tool. Listing 4.5 depicts an example of a QCG configuration file to submit a multiscale grid computation over three machines: two clusters located in Poland (called `inula` and `zeus`) and a supercomputer located in Germany (called `smucm`). In lines 7, 20 and 29, three MPI submodel instances are defined with their resources requirement: LeftSection, MiddleSection and RightSection. The MUSCLE2 configuration file is named `canals.qcg.cxa.rb` (line 63). Reservations of machines should done before and it is specified within the tag `<reservation>` (line 18). The submission of the whole workflow is done through a QCG-client command line and the QCG XML configuration file of the workflow. QCG-client will submit grid jobs on the target machines, deploy remotely the MUSCLE2 runtime and run the tree submodels instances.

Listing 4.5 – Description of a QCG grid workflow

```
1  <?xml version="1.0"?>
2  <qcgJob xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
3  xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance" appId="MAPPER">
4    <task persistent="true" taskId="task">
5      <requirements>
6        <topology>
7          <processes processesId="LeftSection">
8            <processesMap slotsPerNode="16">
9              <processesPerNode>16</processesPerNode>
10             <processesPerNode>16</processesPerNode>
11             <processesPerNode>16</processesPerNode>
12             <processesPerNode>16</processesPerNode>
13             <processesPerNode>16</processesPerNode>
14           </processesMap>
15           <candidateHosts>
16             <hostName>smucm</hostName>
17           </candidateHosts>
18           <reservation type="LOCAL">test</reservation>
19         </processes>
20         <processes processesId="MiddleSection" masterGroup="true">
21           <processesCount>
22             <value>24</value>
23           </processesCount>
24           <candidateHosts>
25             <hostName>inula.man.poznan.pl</hostName>
26           </candidateHosts>
27           <reservation type="LOCAL">plgmohamed.0</reservation>
28         </processes>
29         <processes processesId="RightSection">
30           <processesCount>
31             <value>12</value>
32           </processesCount>
33           <candidateHosts>
34             <hostName>zeus.cyfronet.pl</hostName>
35           </candidateHosts>
36         </processes>
37       </topology>
38     </requirements>
```

```
39      <execution type="mapper">
40        <executable>
41          <application name="muscle2"/>
42        </executable>
43        <arguments>
44          <value>canals.qcg.cxa.rb</value>
45        </arguments>
46        <stdout>
47          <directory>
48            <location
49  type="URL">gsiftp://qcg.man.poznan.pl/~/CANALS/results/${JOB_ID}.output</location>
50          </directory>
51        </stdout>
52        <stderr>
53          <directory>
54            <location
55  type="URL">gsiftp://qcg.man.poznan.pl/~/CANALS/results/${JOB_ID}.error</location>
56          </directory>
57        </stderr>
58        <stageInOut>
59          <file name="rhoneConvectiveScaling.xml" type="in">
60            <location
61  type="URL">gsiftp://qcg.man.poznan.pl/~/grid/data/rhoneConvectiveScaling.xml</location>
62          </file>
63          <file name="canals.qcg.cxa.rb" type="in">
64            <location
65  type="URL">gsiftp://qcg.man.poznan.pl/~/grid/data/canalsmultiple.cxa</location>
66          </file>
67          <file name="canals.pre.sh" type="in">
68            <location
69  type="URL">gsiftp://qcg.man.poznan.pl/~/grid/data/canals.pre.sh</location>
70          </file>
71          <directory name="out" type="out">
72            <location
73  type="URL">gsiftp://qcg.man.poznan.pl/~/CANALS/results/${JOB_ID}.out</location>
74          </directory>
75        </stageInOut>
76        <environment>
77          <variable name="QCG_MODULES_LIST">canals/Mohamed</variable>
78          <variable name="QCG_PREPROCESS">./canals.pre.sh</variable>
79        </environment>
80      </execution>
81      <executionTime>
82        <executionDuration>P0Y0M0DT2H30M</executionDuration>
83      </executionTime>
84    </task>
85  </qcgJob>
```

4.5 Conclusion

In this chapter we have described the new approach of modeling, designing and deploying a multiscale application (the MMSF approach) on a possibly distributed HPC infrastructure. The new methodology is easy and formal enough to be applied to a large range of multiscale applications. Besides, it provides user-friendly tools to hide the technical complexity of running multiscale application on grid infrastructure.

Our contribution to the MMSF can be summarized in two points: (1) propose an approach to support the coupling the two submodels with different time scales, and (2) participate in the improvement of the performance of the MUSCLE2 framework. In the example of coupling two water sections through a gate, we proposed to put the handling of multiscale communication on the *mapper* side. Thus, the submodels remains blind to coupling workflow. Regarding the performance improvement, the previous version of MUSCLE2 did not support an efficient running of native `C++` MPI code on parallel machines. Efforts have been made to push forwards a release of a new MUSCLE2 version which supports intensive computation such as 3D simulation of the hydrology application and to reduce significantly the overhead induced by the MMSF framework.

5 1D-1D coupling of shallow water equation*

5.1 Introduction

Irrigation canals are often man-made and used to carry the flow of water from its source (lake, river, etc.) to farmers for agriculture or to population for daily use. The canal borders are usually delimited by either natural trenches of water or constructed walls. One of the difficulty with an irrigation canal is to provide a sustainable flow of water over a large distance. Some techniques exist to deal with this difficulty: for instance, it is common to build a water dam and separate a long canal section with gates to regulate the water flow and level. In addition, farmers for example can use pumping stations to pump water from the canal to irrigate their soils since they may not be directly connected the water stream.

Hydrology numerical model is of great use to simulate and study the status of an irrigation canal. It is used very often to check planned operations like a dam water-draining to eliminate the accumulated sediment in front of the dam and study its effect. These operations and maintenance of water resources in irrigation canals are crucial especially in populated areas. They involve allocating and distributing water resource equitably among all users (for instance farmers, residents, electricity companies, etc.) and in a sustainable way. Numerical simulation is an interesting method to study and predict the feasibility of such operations.

---

*This content of this chapter is based on:

- Mohamed Ben Belgacem, Bastien Chopard, Latt Jonas, and Andrea Parmigiani. A Framework for Building a Network of Irrigation Canals on a Distributed Computing Environment: a case study. *Journal of Cellular Automata. Special Issue: Cellular Automata Applications for Research and Industry*, 9(2-3):225–240, 2014.
- Mohamed Ben Belgacem, Bastien Chopard, and Andrea Parmigiani. Coupling Method for Building a Network of Irrigation Canals on a Distributed Computing Environment. In GeorgiosCh. Sirakoulis and Stefania Bandini, editors, *Cellular Automata*, volume 7495 of *Lecture Notes in Computer Science*, pages 309–318. Springer Berlin Heidelberg, 2012.

Irrigation canal involves several parts: water sections, junctions (gate, spillway, pumping station, etc) and has a length of several Kilometers. There exist several modeling problems to simulate a given canal. For instance, regions where the water flow is unstable require different techniques of resolution than other regions where the water flow is stable. So, having an irrigation canal with mixed regime of flows is not easy to model and to program. Such solution of this problem requires boundary conditions treatment between the used numerical models. As an example, the water flow around a gate is turbulent and requires a 3D-Free Surface model to study it. Affording a 3D-Free surface model to simulate the entire irrigation is too expensive in time and computing resources. thus, regions where the water level is stable can be modeled by a 1D-shallow water model. This problem is ideally suited to the MMSF approach.

To do so, we study in this chapter the coupling of 1D shallow water model by decomposing an irrigation network into several segments and coupling them through water junctions. The coupling is based on the multiscale approach presented in the chapter 4. Our coupling strategy, based on the concept of Complex Automata (CxA), the Lattice Boltzmann (LB) method and the Multiscale Modeling Language (MML) aims at coupling simple 1D models of canal sections together and preparing a future coupling with 3D complex ones. Besides, this approach enable to build a numerical model that can be run over a distributed grid infrastructure, thus offering a large amount of computing resources. In this chapter, we illustrate our approach by coupling two canal sections in 1D through a gate and a spillway junctions.

5.2 Introduction to the Lattice Boltzmann method

We briefly present the basic equations of the LB method, which is widely used for computational fluid dynamics. Further details can be found in several papers and textbooks (see for instance [135, 35]).

In the LB method, a fluid is described in terms of q density distribution functions $f_i(\mathbf{x}, t)$, $i = 0, \dots q-1$, where $\mathbf{x}$ belongs to a Cartesian grid of spacing Δx and t is the discrete time, which varies by steps Δt. From the f_i, the fluid density $\rho(\mathbf{x}, t)$ and the velocity field $\mathbf{u}(\mathbf{x}, t)$ are obtained as $\rho = \sum_i f_i$ and $\rho\mathbf{u} = \sum_i f_i \mathbf{v}_i$. The $\mathbf{v}_i$ are velocity vectors associated with each f_i, chosen such that $\mathbf{x} + \Delta t \mathbf{v}_i$ is also a point of the computational grid. In LB modeling, the lattice is denoted as DdQq where d is the spatial dimension of the Cartesian lattice and q the number of velocity vectors.

In the so-called BGK based LB methods, the density distribution $f_i(\mathbf{x}, t)$ are computed as a relaxation towards prescribed local equilibrium functions f_i^{eq}: $f_i(\mathbf{x} + \Delta t \mathbf{v}_i, t + \Delta t) = f_i(x, t) + \frac{1}{\tau}\left(f_i^{eq}(\mathbf{x}, t) - f_i(\mathbf{x}, t)\right)$, where f^{eq} depends on the physics of the process, and τ is a parameter called the relaxation time. The above equation can also be expressed as a

succession of collision and streaming steps:

$$\begin{array}{ll}\text{Collision:} & f_i^{out} = f_i^{in} + \frac{1}{\tau}(f_i^{eq} - f^{in}) \\ \text{Streaming:} & f_i^{in}(\mathbf{x} + \Delta t \mathbf{v}_i, t + \Delta t) = f_i^{out}(\mathbf{x}, t)\end{array} \tag{5.1}$$

where $f_i^{in}(\mathbf{x}, t) \equiv f_i$ is the density of particles entering, at time t, site $\mathbf{x}$ with velocity $\mathbf{v}_i$; f_i^{out} is the density of particles with velocity $\mathbf{v}_i$ at site $\mathbf{x}$ after collision. Figure 5.1 illustrates both the collide and stream steps over four lattice sites. The f_i^{in} become f_i^{out} during the collide step on a given lattice site, and then becomes f_i^{in} on the lattice of the neighbor site.

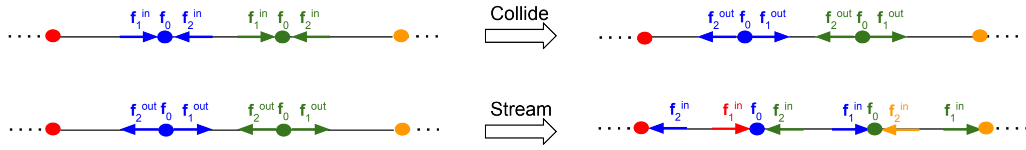


Figure 5.1 – illustration of the collide and Stream operations in D1Q3.

In the shallow water (SW) model, the fluid is described as columns of water with height $h(\mathbf{x}, t)$ and velocity $u(\mathbf{x}, t)$. For solving the SW equation with a LB approach, $f_i(x, t)$ describes the height of the water column at site $\mathbf{x}$ and time t that travels with velocity v_i.

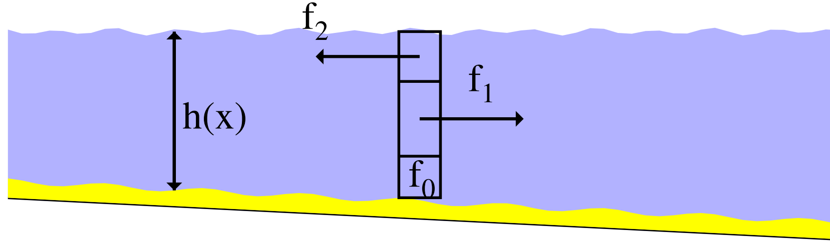


Figure 5.2 – Illustration of the Lattice Boltzmann 1D Shallow water model (SW1D)

The one-dimensional case (SW1D) is illustrated in Figure 5.2. We use a D1Q3 lattice where grid points are separated by a distance Δx. Each lattice site is characterized by three density distributions $f_{i=0..2}(x, t)$, and three velocity vectors $v_{i=\{0..2\}}$: $v_0 = 0$, $v_1 = v$, $v_2 = -v$, with v=$\Delta x/\Delta t$. As suggested in Figure 5.2, the water level h and the velocity u are then defined as $h = \sum_{i=0}^{2} f_i$ and $hu = \sum_{i=0}^{2} v_i f_i$. See for instance [120] for a more detailed description.

The SW1D is sufficient to model a simple canal section where the vertical and perpendicular flow can be neglected. For more complex simulation, a 3D free surface model is needed. An example of 3D free surface model is illustrated in Figure 5.3, but not further described here (see [114, 36] for more details).

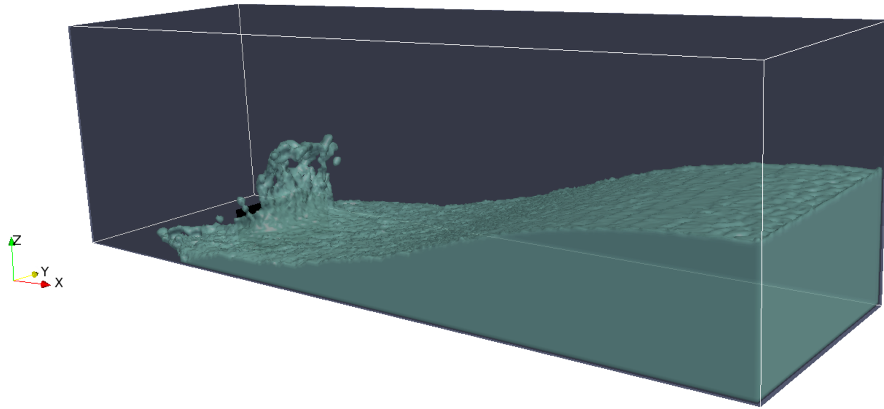


Figure 5.3 – Illustration of the 3D free-surface Lattice Boltzmann model (FS3D)

5.3 Modeling of an irrigation network with LBM

An irrigation network is typically composed of canal sections connected through *water junctions*. They can be gate, spillway, pumping station, reservoir, complex structure, etc. A water section in an irrigation canal which has a rectangular geometry delimited by two walls borders and a bed. The depth of the canal is small compared to its length and the water surface is in contact with the air. So, the shallow water model is adequate to model numerically the simulation of water flow in this situation. The canal sections are described with a Lattice Boltzmann (LB) model whereas *water junctions* are modeled with mathematic equations characterizing the water rate and height around. Thus, coupling two canal sections with a *water junction* amounts to computing for each of them the missing input distribution functions. Their values depend on the characteristic of the *water junction*. Figure 5.4 illustrates the coupling scenario where f_i is the density distributions of the lattice points of the upstream canal and f'_i the density distributions of the down stream canal.

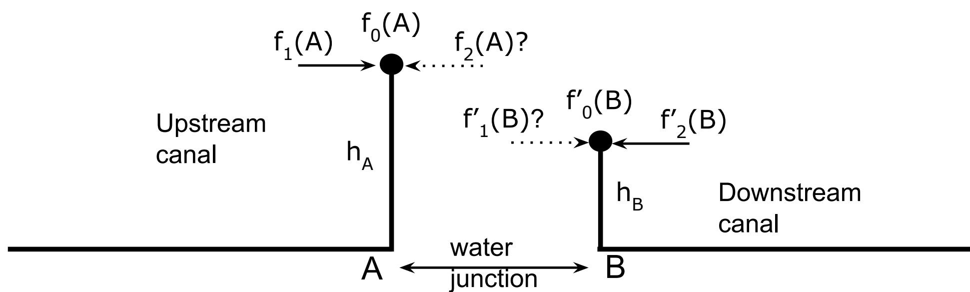


Figure 5.4 – The known ($f_1(A)$, $f_0(A)$, $f'_0(B)$ and $f'_2(B)$) and unknown ($f_2(A)$ and $f'_1(B)$) distribution functions at the connection lattice site A and B.

The unknown distribution functions resulting from such coupling are f_2 for the upstream canal and f'_1 for the downstream one. They are determined by solving a system of equations governed by the LB equations and the physical properties of the junction.

The LB method is an explicit time-driven algorithm which requires computing the unknown distribution functions at each time step of the simulation. In addition, an irrigation canal is composed of several sections connected through *junctions* resulting, therefore, in a tightly coupled system in term of communication.

The MMSF approach, discussed in chapter 4, is well suited in order to develop a numerical application which simulates a real irrigation canal. On one hand, one can consider the canal sections as submodels implementing a given numerical solver. On the other hand, *water junctions* are considered as *mapper*s that handle the coupling operations. What makes the MMSF approach attractive for the canal application is that submodels remain autonomous solvers and independent from the structure of the canal. Hence, one can build an irrigation canal with the possibility to reuse existing submodels (solvers) and couple them with new mappers. Furthermore, by adopting the MMSF approach one can study in more details the flow characteristics in a restricted area by only changing the scale of the corresponding submodel without affecting the communication between the rest of components. In this case the concerned upstream and downstream canals will have two different spatial and temporal resolutions and, therefore, grid refinement techniques must be used for the coupling. As a result the junction must be programmed to handle two different frequencies of send/receive operations for each side. An example of modeling a simple coupling of two canal segments through a gate is illustrated in Figure 5.5. Regarding the deployment, the applications have the advantage to easily run on local or distributed HPC infrastructure illustrating the *DMC* aspect of the MMSF approach.

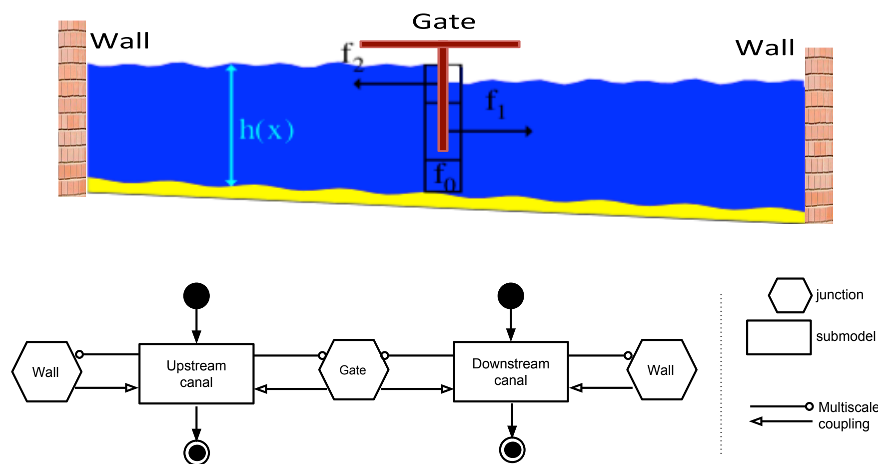


Figure 5.5 – A coupling example of two canal sections and a Gate junction with a boundary condition. The figure on the top describes the real phenomena to simulate. The figure on the bottom represents the MML description of the problem. The two sections are identified as submodels and the gate and the wall are identified as *mapper*s. The wall connected to each section imposes a bounce-back boundary condition.

In what follows we apply the proposed strategy on the 1D shallow water with LB model to

couple two canal sections throughout a gate and a spillway. The problem of this coupling is to find the appropriate boundary condition of the 1D shallow water model. Specifically, this amounts to finding the right values f_2^{in} and f_1^{in} for each section boundary during each time step. At the same time, these values should take into consideration the *water junction* specificities.

5.4 Coupling through *water junctions*

The information that each submodel (canal section) must send to the junction and that it will receive in return depends on the nature of this junction. Here we show which data and which computation are needed to couple two 1D canal sections that are interconnected through a gate and a spillway (the junctions). This coupling is based on the standard relation [69] giving the water flow Q through different junctions, as a function of the water height of the upstream and downstream canals:

$$\begin{array}{lll} Q(gate) & = F(h_A - h_B) & = \gamma_g \sqrt{g(h_A - h_B)} \\ Q(spillway) & = F(h_A - h_s) & = \gamma_s \sqrt{g(h_A - h_s)^3} \end{array} \tag{5.2}$$

where h_A and h_B are the water heights before and after the gate, respectively, $g = 9.81$ is the gravity constant, γ_g is a coefficient depending on the gate geometry and opening, γ_s is a coefficient depending on the spillway geometry and h_s is the spillway height.

In the LB approach to the SW equation (see section 5.2), one has (assuming $\Delta x/\Delta t = 1$)

$$\begin{array}{ll} h_A = f_0(A) + f_1(A) + f_2(A) & Q_A = Q = f_1(A) - f_2(A) \\ h_B = f_0(B) + f_1(B) + f_2(B) & Q_B = Q = f_1(B) - f_2(B) \end{array} \tag{5.3}$$

where $f_i(x)$ denotes the height distribution functions at the point x, and A and B are the points just before and just after the junction, subject to the same discharge Q (see Figure 5.6).

Since A and B are at the extremity of the two canal sections, $f_2(A)$ and $f_1(B)$ are unknown. They have to be provided by the junction model which receives $f_0(A)$ and $f_1(A)$ on its left, and $f_0(B)$ and $f_2(B)$ on its right.

5.4.1 Coupling through a gate junction

After eliminating $f_1(A)$ and $f_2(B)$ from eqs (5.3), one computes $f_2(A)$ and $f_1(B)$ as

$$f_2(A) = \frac{1}{2}\left[h_A - f_0(A) - Q\right] \qquad f_1(B) = \frac{1}{2}\left[h_B - f_0(B) + Q\right] \tag{5.4}$$

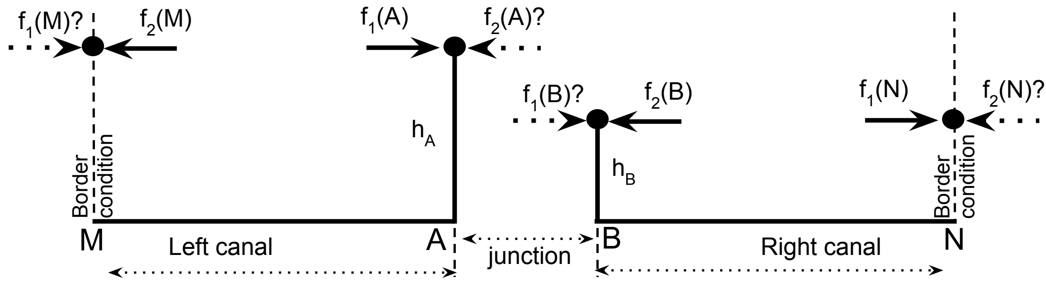


Figure 5.6 – The known ($f_1(A)$, $f_0(A)$, $f_0(B)$ and $f_2(B)$) and unknown ($f_2(A)$ and $f_1(B)$) distribution functions at the connection lattice site A and B. $f_1(M)$ and $f_2(N)$ are determined by applying a boundary condition.

By subtracting these two equations, one gets

$$f_1(B) - f_2(A) = Q + \frac{1}{2}\left[h_B - h_A + f_0(A) - f_0(B)\right] \tag{5.5}$$

If we assume that h_A and h_B are known from the previous time step (which is certainly true in a steady state), then the right-hand side of equation (5.5) is known because Q is a known function of $h_A - h_B$. This gives a first equation for $f_1(B)$ and $f_2(A)$.

A second equation is obtained by requesting that the total amount of water that enters the two canal sections is equal to the amount of water that leaves them. This implies that

$$f_1^{in}(B) + f_2^{in}(A) \equiv f_1(B) + f_2(A) = f_1^{out}(A) + f_2^{out}(B) \tag{5.6}$$

where $f_1^{out}(A)$ and $f_2^{out}(B)$ are known from the last LB step.

Equation (5.6) and equation (5.5) can be solved for $f_1(B)$ and $f_2(A)$

$$\begin{cases} f_1(B) = \frac{1}{2}\left(\Delta + \bar{Q}\right) \\ f_2(A) = \frac{1}{2}\left(\Delta - \bar{Q}\right) \end{cases} \tag{5.7}$$

where

$$\begin{aligned} \Delta &= f_1^{out}(A) + f_2^{out}(B) \\ \bar{Q} &= f_1(B) - f_2(A) \\ &= F(h_A - h_B) + \frac{1}{2}\left[h_B - h_A + f_0(A) - f_0(B)\right] \end{aligned}$$

Thus, the *junction component* of our model receives $f_1^{out}(A)$, h_A and $f_0(A)$ from the upstream canal, and $f_2^{out}(B)$, h_B and $f_0(B)$ from the downstream one. It then computes $f_2(A)$ and $f_1(B)$ according to equation (5.7) and returns each of these values to each of

the canals.

The physical validity of the above coupling can be demonstrated numerically as follows. We enforce the value h_A and h_B in A and B (see Table 5.1) by imposing at each time step the values of $f_1(A)$ and $f_2(B)$ as

$$\begin{aligned} f_1(A, t+1) &= \alpha f_1(A, t) + (1-\alpha)(h_A(t) - f_2(A, t) - f_0(A, t)) \\ f_2(B, t+1) &= \alpha f_2(B, t) + (1-\alpha)(h_B(t) - f_1(B, t) - f_0(B, t)) \end{aligned} \tag{5.8}$$

where α is a relaxation parameter (here chosen as $\alpha = 0.2$) used to smooth the way $f_1(A)$ or $f_2(B)$ reach a value that guarantees the prescribed heights h_A and h_B. Note that equation (5.8) can be seen as a way to impose a boundary condition for the water height at a chosen canal location ($f_1(M)$ and $f_2(N)$ in Figure 5.6).

Table 5.1 – Simulation parameters using the gate and spillway junction

Junction	Simulation parameters		Discharge
Gate	$\gamma_g = 0.1$	$h_A = 0.12m$	$Q = 0.044294$
		$h_B = 1m$	
Spillway	$\gamma_g = 0.106$	$h_A = 0.1m$	$Q_{border} = 1.0\text{E}^{-4}$
		$h_B = 0.075m$	

The simulation result of coupling two canals with a gate junction as illustrated in Figure 5.5 is represented in Figure 5.7. We show the time evolution of the discharge related to the gate junction from arbitrary initial values for $f_2(A)$, $f_0(A)$, $f_1(B)$ and $f_0(B)$. We observe that the quantities $Q_A = f_1(A) - f_2(A)$ and $Q_B = f_1(B) - f_2(B)$ converge rapidly to the imposed value Q. After 20 time steps, Q_A and Q_B are equal to Q within a precision of 10^{-5}. After 40 iterations (not shown), the precision improves to 10^{-8}. After convergence, the value of $\bar{Q}$ is found to be $\bar{Q} = f_1(A) - f_2(B) = 0.035138$, showing that $\bar{Q}$ is clearly different from the imposed discharge Q. Also, after 40 steps, the value $f_0(A) + f_1(A) + f_2(A) - h_A$ is smaller than 10^{-8}, showing the excellent convergence of the present gate model and the present boundary condition used to impose the heights h_A and h_B.

5.4.2 Coupling through a spillway junction

Similarly to the approach used for coupling the gate junction, one has (assuming $\Delta x/\Delta t = 1$) from eq. (5.2) and eqs. (5.3)

$$\begin{aligned} f_2(A) &= f_1(A) - F(h_A - h_s) \\ Q &= f_1(A) - f_2(A) \end{aligned} \tag{5.9}$$

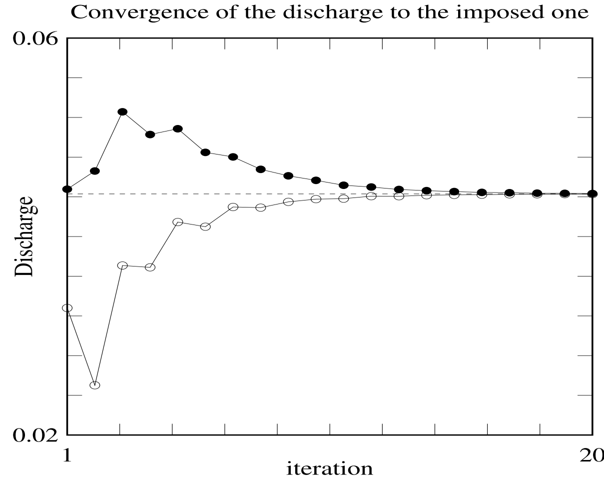


Figure 5.7 – Simulation using a gate: convergence of the flow Q_A (black dots) and Q_B (white dots) to the imposed discharge $Q = 0.044294$. Other parameters: $\tau = 0.6$, $\alpha = 0.2m$, $h_A = 0.12m$, $h_B = 0.1m$.

Note that the amount of the water $f_2(B)$ that leaves the downstream section at point B is bounced back from the spillway. Besides, the downstream section receives water from the upstream one. A second equation is therefore obtained by requesting that the total amount of water that enters the downstream canal section in point B is equal to the sum of the amount of water $M_A = f_1^{out}(A) - f_2(A)$ that leaves point A in the upstream canal and $M_B = f_2^{out}(B)$ that leaves the point B in the downstream canal.

This implies that

$$\begin{aligned} f_1(B) &= M_A + M_B \\ &= f_1^{out}(A) - f_2(A) + f_2^{out}(B) \\ &= \Delta - f_2(A) \end{aligned} \tag{5.10}$$

We obtain from equations (5.9) and (5.10)

$$\begin{aligned} f_2(A) &= f_1(A) - F(h_A - h_s) \\ f_1(B) &= \Delta - f_1(A) + F(h_A - h_s) \end{aligned} \tag{5.11}$$

The physical validity of the coupling through the spillway junction can be demonstrated numerically as follows. Each canal section has 160 lattice points. We initialize the water height in all the lattice points to $h_A = 0.1$ for the upstream canal and arbitrary to $h_B = 0.075 < h_s$ for the downstream canal (see Table 5.1). The initial velocity value is equal to zero in all the lattice points. We enforce the same discharge value $Q_{border} = 1.0\mathrm{E}^{-4}$ as a boundary condition for the two canals (points M and N as

depicted in Figure 5.6) by imposing at each time step the values of $f_1(M)$ and $f_2(N)$ as follows

$$\begin{aligned} f_1(M) &= Q_{border} - f_2(M) \\ f_2(N) &= Q_{border} - f_1(N) \end{aligned}$$

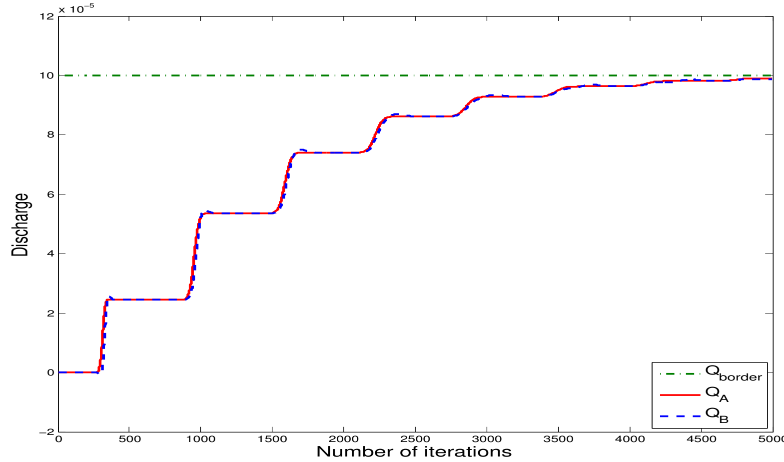


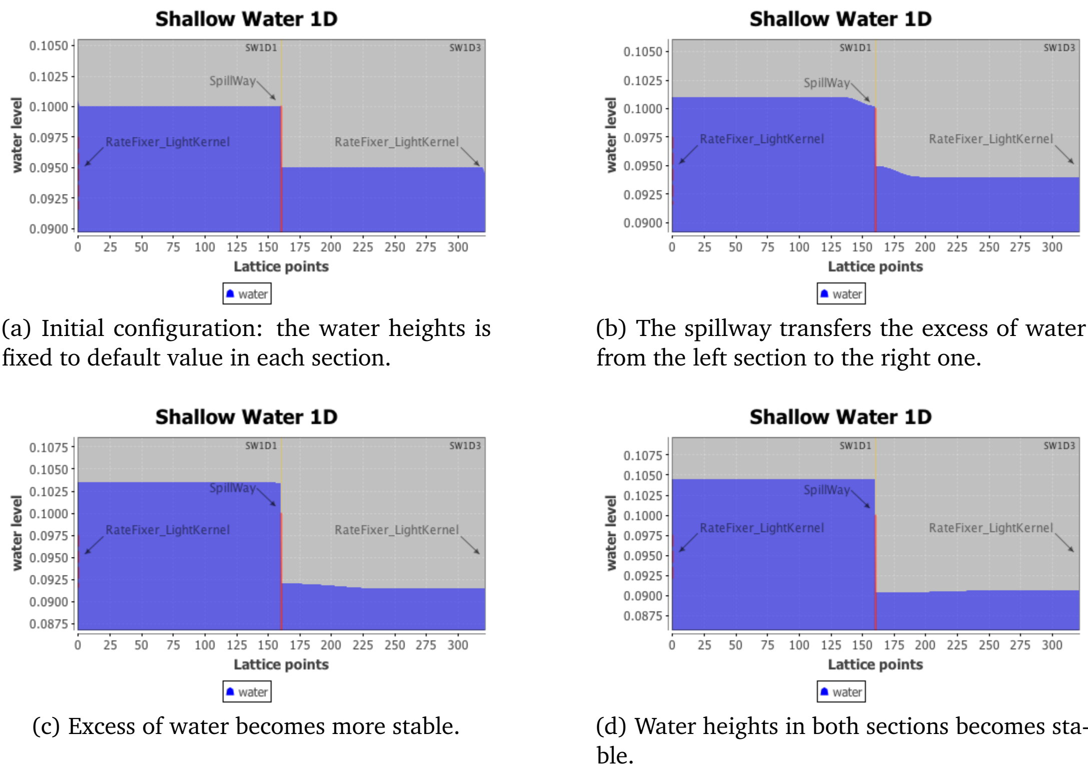
Figure 5.8 – Simulation using a spillway: convergence of the flow Q_A and Q_B.

In Figure 5.8, we show the time evolution of the discharge related to the spill junction from the imposed discharge at the boundary of the two canals. We observe that the quantities $Q_A = f_1(A) - f_2(A)$ and $Q_B = f_1(B) - f_2(B)$ converge to the imposed value Q_{border} after 2880 iterations. After convergence, the value of Q in all lattice points is found to be equal to Q_{border} showing clearly that the simulation reacted the expected steady state. Figure 5.9 illustrates a graphical representation of flow behavior during some simulation steps.

5.5 Towards a multidisciplinary and multiscale computational science

Beyond the well-defined equations and theory of dynamic fluid computation, the great growth of computer power facilitates the shifting to a new paradigm of doing computational science: multidisciplinary and multiscale simulations over distributed resources. In this case of the hydrology application (described in chapter 2), this means the ability to consider a direct numerical approach to simulate the water flow in a long river section, which was out of reach in the past. This large simulation may consider a high spatial resolution in order to simulate the water flow in 3D, with sediment transport over long distance.

For instance, so as to perform a simulation of a complex irrigation canal, one can consider three different models: 1D, 2D shallow water and 3D Free-Surface with sediment



(a) Initial configuration: the water heights is fixed to default value in each section.

(b) The spillway transfers the excess of water from the left section to the right one.

(c) Excess of water becomes more stable.

(d) Water heights in both sections becomes stable.

Figure 5.9 – Simulation snapshots of two water sections connected through a spillway. The left section is name `SW1D1` and the right section is named `SW1D3`. `RateFixer_lightKernel` is the name of the *mapper* binary that fixes the discharges Q in the inlet boundary of the left section and the outlet one of the right section.

transport. River regions where the water flow is turbulent require a full 3D Free-Surface model with a high resolution to capture flow details (for example, to study the water motion around a gate). River regions where the water flow is stable and the water height varies along the width of the river can be modeled using LB-2D shallow water. River regions where the water flow and height are stable can be modeled using LB-1D shallow water model with a low spatial resolution. The final simulation is obtained by tightly coupling all the three models together as depicted in Figure 5.10. This coupling is multiscale in nature since it considers three different models with different dimensions and scales. Furthermore, coupling them with a sediment transport model brings a multidisciplinary aspect of the full problem.

A coupling between the 3D Free-Surface model and 2D shallow water one was presented in [119]. This work studied the feasibility to simulate a river flow with high spatial resolution including a proper physical description. On one side, when modeling the *Rhone* river with a sedimentation process, this work demonstrated the limits of the 2D shallow water model in comparison with the 3D Free-Surface one. Comparison between simulations using these two models was done along a $1\ km$ river section with a

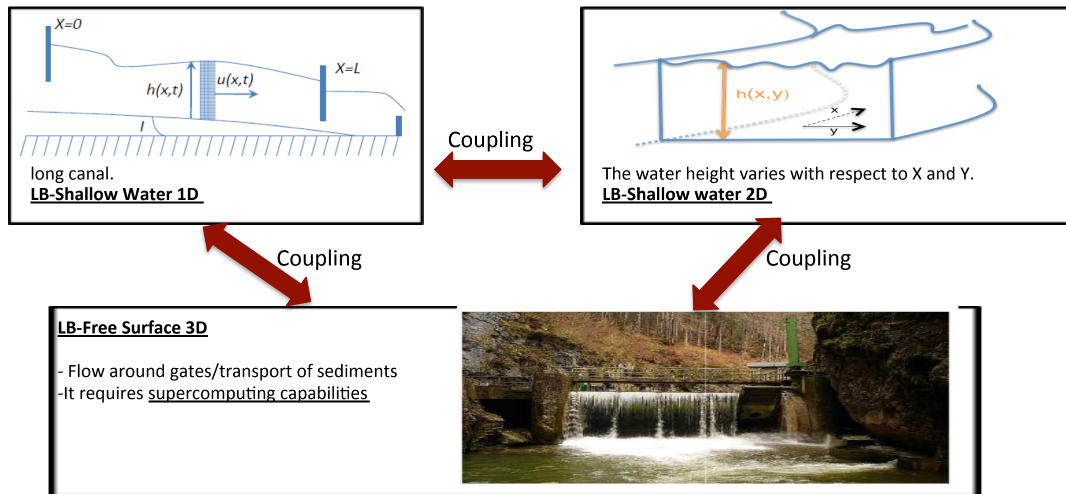


Figure 5.10 – Building an irrigation canal by coupling three different numerical models: 1D, 2D shallow water and 3D free surface with sediments transport.

constant value imposed for the water height and water discharge at the inlet boundary. On the other side, results showed that using 7800 cores of a supercomputer machine (BlueGene/Q), one can estimate that a 3D Free-Surface simulation of 13 km of the *Rhone* river section with 20 cm of spatial resolution would take 1 CPU hours for 60 seconds of real time flow. Let's assume that the computation efficiency is about 50% for strong scale cases, where the number of cores increases and the problem size are fixed. In this case, one would require 1 million cores to be as fast as real time. Computing faster than the real time will greatly assist water companies in the study of several scenarios like the impact of changing the water level on sediment transport or the prediction of such natural hazard.

Given the fact that a detailed simulation may not be required everywhere, distributed multiscale computation can be used to simulate the full system much faster than on one machine.

5.6 Conclusion

We have presented how to use MMSF approach to build an irrigation canal from generic and reusable components, and to simulate the water height and flow among *water junctions*. This is obtained by decomposing the corresponding canal network into different segments and coupling them through junctions. Each canal segment can be resolved separately with either a 1D shallow water with different spatial and temporal resolution.

We have proposed an explicit implementation of a gate and a spillway junctions within the Lattice Boltzmann numerical method. These junctions receive information from both canal sections and return the proper boundary conditions that ensure the right discharge

between the two canals. We have validated the coupling through measurement showing the convergence of our implementation.

In the next chapter we present the experiment that we have conducted for 3D intensive computation over distributed HPC infrastructure.

6 Evaluation of the MMSF for 3D Lattice Boltzmann based applications*

6.1 Introduction

This chapter discusses the performance evaluation of applying the MMSF approach to an intensive computation and running them over a distributed infrastructure. By this we want to evaluate the performance efficiency and the scalability of the MMSF approach for the multiscale applications. The application we target in these experiments is the *Hydrology* application (described in chapter 2). For sake of simplicity, we carried out the experiments using the *Cavity3D-Flow*, a flow problem that is often used for benchmarking new CFD solver and which is similar to the *Hydrology* application in term of computation and implementation.

6.2 Performance model for the MMSF approach

In order to evaluate the MMSF performance in solving 3D parallel problems (such as problems based on the Lattice Boltzmann Method), one can consider a simplified performance model that defines analytically the time required for such execution and the induced overhead. To do so, we consider an MPI based 3D problem of dimensions (in meters): $L_x \times L_y \times L_z$. We define the monolithic execution time of the problem on a HPC cluster and using p cores as:

$$T_{mpi}(N,p) \quad = [\alpha \times \frac{N}{p} + \beta \times N'] \tag{6.1}$$

*This content of this chapter is based on:

- Mohamed Ben Belgacem, Bastien Chopard, Joris Borgdorff, Mariusz Mamonski, Katarzyna Rycerz, and Daniel Harezlak. Distributed Multiscale Computations Using the MAPPER Framework. In Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 1106–1115. Elsevier, 2013.
- Mohamed Ben Belgacem and Bastien Chopard. A hybrid HPC/cloud distributed infrastructure: Coupling EC2 cloud resources with HPC clusters to run large tightly coupled multiscale applications. *Future Generation Computer Systems*, 42(0):11–21, 2014.

where α and β are coefficients depending on the speed of the processors and network bandwidth, receptively. $N = \frac{L_x}{\Delta x} \times \frac{L_y}{\Delta x} \times \frac{L_z}{\Delta x}$ is the problem size (number of sites) depending on the spatial discretization Δx. N' is the amount of sites to exchange between the MPI processes. In case of an uniform parallelization of the problem along the x-axis, where the full domain size is divided into k sub-domains (or blocks) (Fig. 6.1), we have: $N' = \frac{L_y}{\Delta x} \times \frac{L_z}{\Delta x}$. The coefficients α and β of a given cluster can be determined

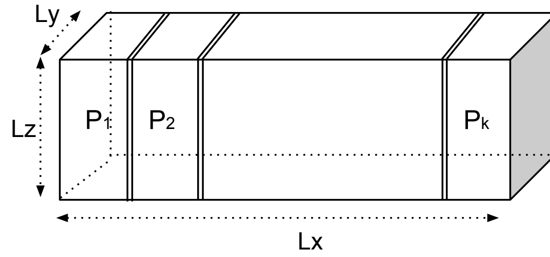


Figure 6.1 – Uniform parallelization of MPI based 3D problems into k sub-domains over p cores, where $p_k = p/k$

from the specifications of the hardware or empirically by running the same problem with different range of cores number and based on eq (6.1).

Parallel code often run on the same cluster and its execution performance is affected mainly by the internal network bandwidth and the number of used cores. A distributed computation involves external network communication and data transfer overhead which could influence drastically the computation time. In our performance model, we address the distributed performance in two levels: the inter-cores communication within the same machine and the across-machines communication time. To measure the overhead induced by the MMSF approach, we focus on MUSCLE2 executions of the same problem size involving k submodels or k sub-domains in our example.

Our benchmark consists of an MPI based lattice Boltzmann (LB) 3D cavity flow problem, developed using PALABOS [114]. The execution of our application is an iterative computation that requires data communication between submodels (sub-domains) during each iteration as shown in this Fig. 6.2. In this experiment, each submodel (sub-domain) is assigned to a different remote parallel cluster.

Compliantly to the *SEL* (Algorithm 4 in chapter 4) in the MMSF approach, Algorithm 5 gives the pseudo-code used by the numerical method. In each iteration, each submodel starts by performing an MPI execution (line 3). Then, it (the *gather* task) collects the boundary data (line 4) and sends/receives it to/from its neighbor submodel (line 5). Upon receiving data, each submodel (the *update* task) updates its boundary domain (line 6). We define the communication cost T_{com} of sending boundary data between two

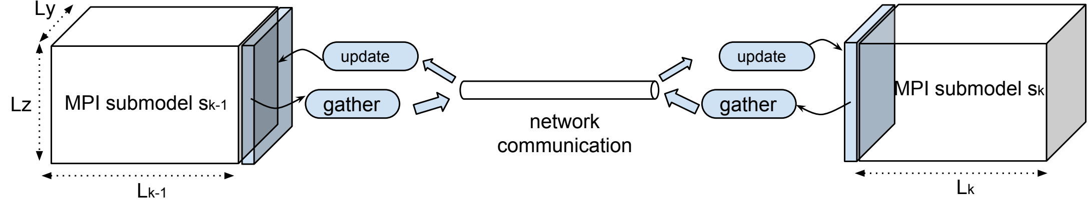


Figure 6.2 – MUSCLE2 execution of two connected submodels. Blue block is the boundary data to exchange. The *gather* task collects the boundary data and sends it to the submodel neighbor. The *update* task receives boundary data and updates the computing domain of the corresponding submodel.

Algorithm 5: Pseudo-code of the cavtiy3D example.

```
1 F_init();
2 while (iterations < max_Iteration_Number) do
3     compute();
4     gatherBoundaryData();
5     sendReceiveBoundaryData();
6     updateBoundaryData();
7 Observation();
```

submodels as:

$$T_{com}(D, p_b) \quad = [T_{gather}(D, p_b) + T_{send/rec}(D) + T_{update}(D, p_b)] \tag{6.2}$$

where D is the size of boundary data distributed over p_b cores, $T_{gather}(D, p_b)$ the time required by a submodel to collect boundary data, $T_{send/rec}$ the time required to send them to its neighbors and T_{update} the time required to update the boundary data.

Collecting or updating boundary data depends on the MPI implementation in each machine, the network bandwidth and the number of cores p_b. We define:

$$T_{gather}(D, p_b) \quad = T_{update}(D, p_b) \quad = \tau_{mpi} + \gamma \times D \times Log(p_b) \tag{6.3}$$

where γ is a coefficient depending on the speed of processors and τ_{mpi} is the startup and synchronization overhead. We assume that τ_{mpi} is negligible.

Send and receive operations have a direct impact on the performance of distributed execution. Very low network bandwidth may result in extremely slow computation. The network communication between two submodels is:

$$T_{send/rec}(D) = \frac{D}{B} + \tau \times m \tag{6.4}$$

where B is the network bandwidth in Bytes/second, τ is the network latency overhead

and m is the number of required messages to send the data D. Here we can assume that the network latency can be ignored compared to the data transmission time $\frac{D}{B}$ on the network.

Let us consider our benchmark application involving n submodels $\{s_k\}_{1\leq k\leq n}$ that run in a tightly coupled way. Each submodel s_k of this application has a computation domain size $N_k = \frac{L_{x,k}\times L_y\times L_z}{\Delta x^3}$ and runs over p_k cores. Note that $p = \sum_{j=1}^{n} p_k$ and $L_x = \sum_{k=1}^{n} L_{x,k}$. We assume that the inter-core communication time is negligible for a big problem size over an adequate number of cores. So, we define $T_{mpi}(N_k, p_k) = T_{serial}(N_k)/p_k$, where $T_{serial}(N_k)$ is the time required to process the submodel s_k domain on one core processor. At each iteration, each submodel s_k exchanges boundary data D_k with its neighbors. Thus, the estimated computation time t_k for a submodel s_k using MUSCLE2 framework is defined as:

$$\begin{aligned}
t_k(N_k, p_k) &= T_{mpi}(N_k, p_k) + T_{com}(D_k, p_b) \\
T_{mpi}(N_k, p_k) &\simeq \tfrac{T_{serial}(N_k)}{p_k} = \alpha \times \tfrac{L_{x,k}.L_y.Lz}{\Delta x^3.p_k} = \tfrac{\alpha_1.L_{x,k}}{p_k} \times \tfrac{1}{\Delta x^3} \\
T_{com}(D_k, \Delta x) &= T_{gather}(D_k, p_b) + T_{send/rec}(D_k) + T_{update}(D_k, p_b) \\
&\simeq \beta' \times \tfrac{L_y.L_z}{\Delta x^2} + \beta'' \times \tfrac{L_y.L_z}{\Delta x^2} + \beta' \times \tfrac{L_y.L_z}{\Delta x^2} \\
&\simeq \beta_1 \times \tfrac{1}{\Delta x^2}
\end{aligned} \tag{6.5}$$

Where α_1 and β_1 are coefficients, independent of Δx, but depending on the speed of the processors, the product $(L_y.L_z)$ and the bandwidth of the network. They can be obtained from technical specifications or by running the application and by fitting the execution time to the above formula (as will be done in section 6.3). As we can see in the eq. (6.5), the $T_{mpi}(N_k, p_k)$ depends on Δx^3 whereas $T_{com}(D_k)$ depends on Δx^2. From eq.(6.5) we obtain

$$t_k(N_k, p_k) = (\tfrac{\alpha_1.L_{x,k}}{p_k} \times \tfrac{1}{\Delta x^3} + \beta_1 \times \tfrac{1}{\Delta x^2}) \tag{6.6}$$

It is obvious that $T_{\mathtt{DMC}}(N, p)$ will be determined by the slowest submodel, i.e:

$$T_{\mathtt{DMC}}(N, p) = \max_{1\leq k\leq n} \{T_{mpi}(N_k, p_k) + T_{com}(D_k)\} \tag{6.7}$$

where DMC stands for *Distributed Multiscale Computing* (chapter 4), $T_{\mathtt{DMC}}(N, p)$ is the estimated MMSF based computational time of the problem with size N over p cores.

For a given cyclic application comprising n submodels, let's now consider an uniform computation domain discretization, where Δx is the same among all the submodels. Also, let us assume that all submodels run on the same type of computing resources. As a result, submodels will exchange the same boundary data size at each iteration and $T_{com}(D_k) = T_{com}$ will be constant. In addition, since $T_{mpi}(N_k, p_k)$ depends on the slowest submodel, a uniform decomposition of the submodels computation domains, for

which $T_{mpi}(N_k, p_k) = T_{serial}(N_k)/p_k$, $L_{x,k} = L_x/n$ and $p_k = p/n = q$ (n is the number of submodels), leads to $T_{mpi}(N_k, p_k) = T_{mpi}(N_k, q)$ for all k, and minimizes $T_{\texttt{DMC}}(N, p)$. Thus, by combining

$$\begin{aligned} T_{mpi}(N,p) &= T_{mpi}(N_k,p_k) = T_{mpi}(\tfrac{L_x}{n},\tfrac{p}{n}) = T_{mpi}(\tfrac{L_x}{n},q) \\ T_{\texttt{DMC}}(N,p) &= T_{mpi}(\tfrac{L_x}{n},\tfrac{p}{n}) + T_{com} = T_{mpi}(\tfrac{L_x}{n},q) + T_{com} \end{aligned} \tag{6.8}$$

with eq. (6.5), we finally obtain

$$\begin{aligned} &T_{mpi}(N_k,q) = \tfrac{\alpha_2}{q} \times \tfrac{1}{\Delta x^3} \\ &T_{\texttt{DMC}}(N,p) = (\tfrac{\alpha_2}{q} \times \tfrac{1}{\Delta x^3} \times (1 + \tfrac{\beta_1 . q}{\alpha_2} \times \Delta x)) \end{aligned} \tag{6.9}$$

In summary, a uniform decomposition of the submodels computation domains, for which $N_k = N/n$, $L_{x,k} = L_x/n$, $D_k = D$ and $p_k = p/n$, enables to have equivalent MPI computation time of each submodels and data communication occurring at the same time. Thus, this leads to minimizes the total estimated time for the full parallel problem. We obtain:

$$T_{\texttt{DMC}}(N,p) = T_{mpi}(\tfrac{N}{n},\tfrac{p}{n}) + T_{com}(D,p_b) \tag{6.10}$$

With this formula, several scenarios can be analyzed. For instance, DMC can offer many more computing resources than locally available to solve a large given problem. One can compare $T_{mpi}(N, p)$ with $T_{\texttt{DMC}}(N, p')$ where $p' >> p$ (will be studied in chapter 7).

In what follows we evaluate instead the overhead of using the DMC approach ($p = p' = \frac{p}{2} + \frac{p}{2}$). This evaluation compares local executions with a distributed ones using different HPC machines located in different geographic regions.

The benchmark scenario of the experiments consists in:

1. Running a monolithic simulation (T_{mpi}) over one cluster.
2. As depicted in Fig. 6.2, splitting the cavity3D problem into two equal sections (left and right) and coupling them compliantly to the MMSF approach. Simulations are first performed on the same single cluster and, then, over two distributed clusters.
3. Measuring the wall-clock time of the computation and the time of data transfer in the distributed simulation.
4. Comparing it with the monolithic execution time which defined as T_{mpi}.

We remind the following definitions:

- DMC: stands for Distributed Multiscale Computing.
- T_{mpi}: is the MPI execution time (called here monolithic execution) of a parallel problem on one cluster using a given number of cores.

- $T_{\mathtt{DMC}}^{local}$: is the execution time of the two submodels on the same cluster.
- $T_{\mathtt{DMC}}^{grid}$: is the execution time of the two submodels on two different clusters.

In what follows, we describe the conducted experiments to evaluate the overhead of the MMSF approach. The conducted experiments followed the same benchmark scenario but conducted over three different environments: on EGI, University and Cloud resources. In the first environment, the MMSF tools was already installed on the used EGI resources. These experiments wants to validate the equation 6.9. In the second environment, we installed manually the MMSF tools on the used University clusters and we used more cores for the simulation. By this, we wanted to evaluate the overhead of the MMSF approach on a local academic environment. In the third environment, we investigated the possibility to couple University cluster with cloud resources and evaluate the MMSF approach on a hybrid environment.

Table 6.1 – Configuration of clusters

Setting	Cloud Amazon EC2		University clusters		EGI clusters	
	EC2-cluster-1	EC2-cluster-2	*Scylla*	*Gordias*	*Reef*	*Galera*
Vcores	128	56	240	224	5892	+5000
CPU type	Intel Xeon E5-2670	Intel Xeon	Intel Xeon 2.4		Intel Xeon E5530	Intel Xeon EM64T
Memory	60.5GB/node	7GB/node	15GB/node	12GB/node	total: 9.6TB	total: 22TB
Network	10 Gbit Ethernet	Ethernet	InfiniBand (20 Gb/sec (4X DDR))		Infiniband QDR	Infiniband DDR
OS	Gnu/Linux, 64 bits					
Batch System	SGE		PBS	SGE	Tandem Torque / Maui	PBS
Location	USA(Northern California)		Switzerland (Geneva)		Poland (Poznan,Gdansk)	

6.3 Experiments on EGI HPC clusters

The EGI resources are two clusters located in Poland: *Reef* and *Galera*. Simulation are submitted using the QCG grid middleware. The clusters configuration is described in Table 6.1.

The total computational domain has a length L_x of 4500 meters, a width L_y of 100 meters and a depth L_z of 25 meters. The Lattice Boltzmann implementation is based on Palabos [114](`C++` MPI) and simulation is carried out with different value of Δx ranging from 0.4(m) to 2(m) with a step of 0.2(m). Regarding the Lattice Boltzmann parameters, each lattice cell contains 19 distribution functions with *double* precision data type (8 bytes). The size of one lattice cell is equal to $19 \times 8 = 152$ Bytes. To process the boundary condition of the problem, we need for each submodel the $e = 3$ boundary cells along the x-axis, including all the corresponding y-axis and z-axis cells.

Table 6.2 shows the different grid sizes and the size of the boundary data D_k used for the Lattice Boltzmann computations as a function of Δx. The quantities $n_x = \frac{L_x}{\Delta x}$, $n_y = \frac{L_y}{\Delta x}$ and $n_z = \frac{L_z}{\Delta x}$ represent the number of sites along the x-axis, y-axis and z-axis of the cavity3D section, respectively.

Table 6.2 – Grid size based on Δx and boundary data size

Δx(m)	0.4	0.6	0.8	1	1.2	1.4	1.6	1.8	2
n_x	11250	7500	5625	4500	3750	3215	2813	2500	2250
n_y	250	167	125	100	84	72	63	65	50
n_z	63	42	32	25	21	18	16	14	13
D_k(KB)	7182	3198	1824	1140	804	590	459	414	269

In this experiment, we used *Gordias* cluster to run the monolithic execution (T_{mpi}) over 20 cores and results are shown in the second column of Table 6.3. In what follows, we perform $T_{\texttt{DMC}}^{local}$ on *Gordias* and $T_{\texttt{DMC}}^{grid}$ on two different clusters (*Reef* and *Galera*).

6.3.1 Local execution

In order to run the local simulation ($T_{\texttt{DMC}}^{local}$) on the *Gordias* cluster, we firstly run the MUSCLE2 manager, a single daemon which manages the registration of the running submodels, in a separate node. Then, we run the left and the right sections over 10 cores each.

The total number of iterations we performed is equal to 2000. For each iteration, we measured the computation clock-time of the operations of Algorithm 5. We repeated the simulation with different Δx values.

Results are shown in the first and second columns of Table 6.3. The difference of the execution clocktime between T_{mpi} and $T_{\texttt{DMC}}^{local}$ for the case Δx=0.4 was 106 seconds over a total elapsed time of 5498 seconds. For the case $\Delta x = 1.4$, it was 141 seconds over a total elapsed time of 370 seconds. This suggests that the time overhead of the `DMC` on the same cluster varies slowly with Δx. This behavior can be explained by the good network speed connecting the nodes of the *Gordias* cluster.

6.3.2 `DMC` execution

For the distributed execution ($T_{\texttt{DMC}}^{grid}$), we kept the same configuration (number of cores, same executable and input data, etc.), but used two separate clusters: *Galera* and *Reef* (Table 6.1). Compared to the local execution $T_{\texttt{DMC}}^{local}$ (one cluster), the distributed execution $T_{\texttt{DMC}}^{grid}$ is mainly influenced by the speed of the network connection between clusters, in addition to the size of the boundary data. This can also be observed on the distributed efficiency values ϵ^{grid}=$T_{mpi}/T_{\texttt{DMC}}^{grid}$ and on the comparison with ϵ^{local}=$T_{mpi}/T_{\texttt{DMC}}^{local}$ as depicted in Table 6.3. Besides, we can observe that we obtain a small time overhead with small values of Δx, and a big time overhead with a big Δx.

Based on the measurements in Table 6.3, we observe that the execution times $T_{\texttt{DMC}}^{local}$ and T_{mpi} follow the performance model proposed in eq. (6.9). Furthermore, these measured

values of $T^{local}_{\texttt{DMC}}$ and T_{mpi} allow us to determine the values of α_2 and β_1. As a result, we can rewrite eq. (6.9) as (for *Gordias* cluster)

$$\begin{aligned} T_{mpi}(\Delta x) &= \tfrac{0.874.10^{-3}}{q} \times \Delta x^3 \\ T^{local}_{\texttt{DMC}}(\Delta x) &= T_{mpi}(\Delta x)(1 + 0.43 \times \Delta x) \end{aligned} \tag{6.11}$$

Table 6.3 – Local efficiency ϵ^{local} and distributed efficiency ϵ^{grid} (2000 iterations)

Δx	T_{mpi}(s)	$T^{local}_{\texttt{DMC}}$(s)	$T^{grid}_{\texttt{DMC}}$(s)	ϵ^{local}	ϵ^{grid}
0.4	5392.54	5498.85	9249	0.98	0.58
0.6	1746.9	1939.85	4111	0.90	0.42
0.8	846.53	1015.09	2229	0.83	0.37
1	490.94	654.78	1403.84	0.74	0.36
1.2	321.81	407.62	1186.74	0.78	0.27
1.4	228.34	369.37	811.129	0.61	0.28
1.6	166.85	255.68	589.5	0.65	0.28
1.8	125.69	246.26	533.12	0.51	0.23
2	97.76	209.47	684.7	0.46	0.14

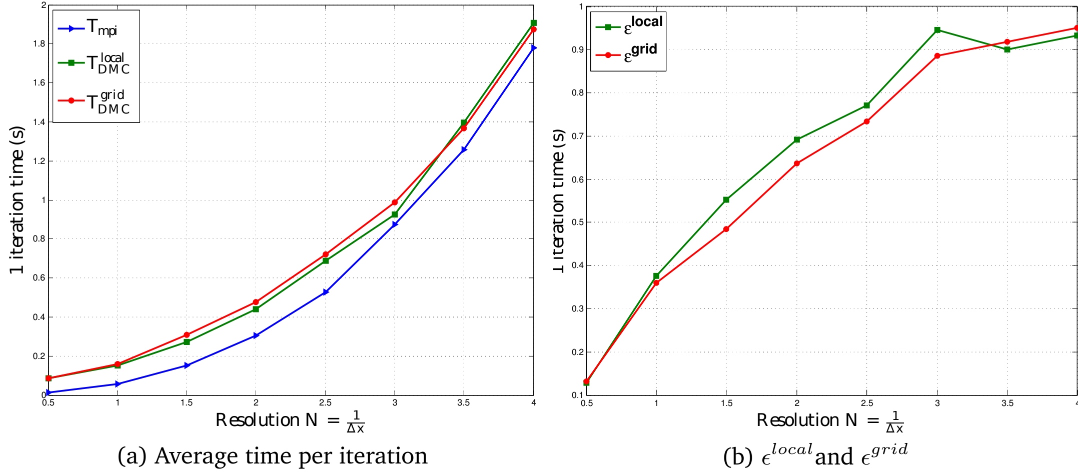
6.4 Experiments on universities HPC clusters

In order to test the scalability of the MMSF approach, we tried to solve a large problem using more number of cores. Based on the same benchmark scenario, we also aimed to evaluate this approach in a local academic environment by using two university clusters: *Gordias* and *Scylla*.

The computational domain of the problem has a length L_x of 13000 meters, a width L_y of 40 meters and a depth L_z of 10 meters. The spatial resolution Δx may vary and will determine the problem size: decreasing Δx implies increasing the domain size as before.

The execution time of these scenarios is indicated in T_{mpi}, $T^{local}_{\texttt{DMC}}$, and $T^{grid}_{\texttt{DMC}}$, respectively. Here, the full section domain (computed in T_{mpi}) is split equally amongst the submodels called left and right. Each simulation carries out 100 iterations, and is repeated three times. We varied the number of grid points per meter $N = \frac{1}{\Delta x}$ from 0.5 to 4, with a step size of 0.5. For the total domain this means varying the problem size from under 820 thousand grid points to over 340 million points.

The monolithic execution is done with 100 cores of the *Gordias* cluster. Likewise, the DMC execution is done with 100 cores, but here the left and right sections run on 50 cores each. The local MUSCLE2 execution is run on *Gordias* whereas the distributed one is executed on two geographically distributed clusters in Geneva: *Gordias* and *Scylla*. On *Gordias* we first ran the MUSCLE2 Simulation Manager (see section 4.4.4 of chapter 4) in a separate node so that it had a fixed address before the job started.



(a) Average time per iteration

(b) ϵ^{local} and ϵ^{grid}

Figure 6.3 – The performance of the three execution scenarios of the cavity 3D model (on *Gordias* cluster), for different numbers of grid points per meter N.

Fig. 6.3a shows the results of the benchmark of T_{mpi}, T^{local}_{DMC} and T^{grid}_{DMC} on *Gordias* cluster. We measured the average time per iteration.

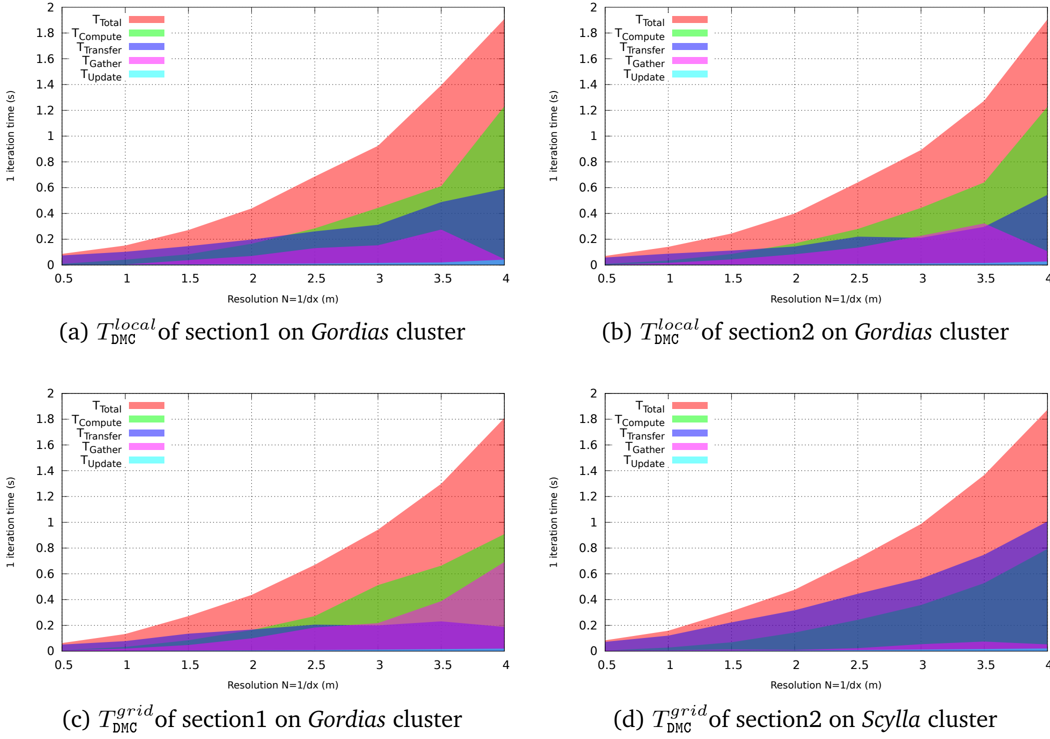
The overhead for this the distributed execution is represented by θ defined as

$$\begin{aligned} \theta &= [T_{DMC}(\tfrac{N}{2},\tfrac{p}{2}) - T_{mpi}(N,p)]/T_{DMC}(\tfrac{N}{2},\tfrac{p}{2}) \\ &= [T_{DMC}(\tfrac{N}{2},\tfrac{p}{2}) - T_{mpi}(\tfrac{N}{2},\tfrac{p}{2})]/T_{DMC}(\tfrac{N}{2},\tfrac{p}{2}) \\ T_{DMC}(\tfrac{N}{2},\tfrac{p}{2}) &= T_{mpi}(\tfrac{N}{2},\tfrac{p}{2})(1+\theta) \end{aligned} \tag{6.12}$$

where $p = 100$ and T_{mpi} is the largest of the two monolithic execution times (given by *Gordias* since *Scylla* being a faster cluster). T_{DMC} is the maximum execution time of the two sections over the two clusters, each using $\frac{p}{2}$ cores.

Results are presented in Fig. 6.3b and Table 6.4. The efficiency values $\epsilon^{local} = T_{mpi}/T^{local}_{\text{DMC}}$– measured by running the two sections on same cluster *Gordias*– and $\epsilon^{grid} = T_{mpi}/T^{grid}_{\text{DMC}}$– measured by using two different clusters (*Gordias* and *Scylla*)– show the same behavior; i.e, we observe a small time overhead with high resolution (small values of Δx), and vice versa.

The runtime of the two sections on the *Gordias* cluster, per operation of the pseudo-code 5, are very similar, as shown in Fig. 6.4a and Fig. 6.4b. In Fig. 6.4c and Fig. 6.4d, we compare the runtime per operation on the *Scylla* and *Gordias* clusters. The overhead induced by the synchronization step when sending and waiting to receive data on the section running on *Scylla* is slightly higher than the section running on *Gordias*. This can be explained by two factors. First, the difference of the network throughput between the two clusters: we measured 42.5 MB/s from *Gordias* to *Scylla* and 10.1 MB/s from *Scylla*



(a) $T_{\texttt{DMC}}^{local}$ of section1 on *Gordias* cluster

(b) $T_{\texttt{DMC}}^{local}$ of section2 on *Gordias* cluster

(c) $T_{\texttt{DMC}}^{grid}$ of section1 on *Gordias* cluster

(d) $T_{\texttt{DMC}}^{grid}$ of section2 on *Scylla* cluster

Figure 6.4 – The runtime of different operations in the local and distributed cases during one iteration, as described in pseudo-code 5: total time [T_{total}]; `Compute()` [$T_{Compute}$]; `getBoundaryData()` [T_{Gather}]; `sendReceiveBoundaryData()` [$T_{Transfer}$]; and `updateBoundaryData()` [T_{Update}].

Table 6.4 – 1 iteration time (sec).

Resolution	T_{mpi}	Local		Grid	
		T_{DMC}	θ	T_{DMC}	θ
0.5	0.011	0.08	6.27	0.08	6.27
1	0.05	0.15	2	0.15	2
1.5	0.14	0.27	0.92	0.3	1.14
2	0.3	0.43	0.43	0.47	0.56
2.5	0.5	0.68	0.36	0.72	0.44
3	0.87	0.92	0.05	0.98	0.12
3.5	1.25	1.39	0.11	1.36	0.08
4	1.77	1.9	0.07	1.87	0.05

to *Gordias*. Second, *Scylla* computes faster than *Gordias* and, thus, it should wait for *Gordias* to finish its *compute()* operation of the current iteration and send its boundary data.

In the following section, we study in more details the synchronization during data transfer and we consider connecting university cluster with a Cloud cluster.

6.5 Experiments on Cloud clusters

In this section we investigate the possibility to couple an University cluster with a cluster mounted on the cloud. To this end, we replaced the *Gordias* cluster of the previous experiments by a cloud cluster mounted on Amazon-EC2 [6] platform and we connected it to *Scylla*.

The cloud cluster is a virtual computing cluster mounted using the package StarCluster [133]. This package enables us to easily build a virtual cluster on Amazon, composed of virtual machines connected with an Ethernet network and running a SGE batch system. Amazon-EC2 provides several types of virtual machine instances with different performance characteristics. The ECU ($EC2\ Compute\ Unit \simeq 1.0 - 1.2\ GHz$) is a unity that defines the CPUs performance. In our experiments, we have built two types of Amazon-EC2 clusters (see Table 6.1):

- *EC2-Cluster-1*: it uses 2 nodes of the *cc2.8xlarge* instance type with 88 ECU and 32 virtual cores (8 cores with hyper-threading). Nodes have 60.5 GB of RAM each, and are connected with a 10 Gigabit Ethernet network.
- *EC2-Cluster-2*: it uses 7 nodes of *c1.xlarge* instance type with 20 ECU and 8 virtual cores. Nodes have 7 GB of RAM each and are connected with a high speed Ethernet network.

6.5.1 Monolithic execution

As described in the section 6.2, the estimated computational time T_{DMC} required for distributed execution is governed by the equation 6.10. The first experiment we carried

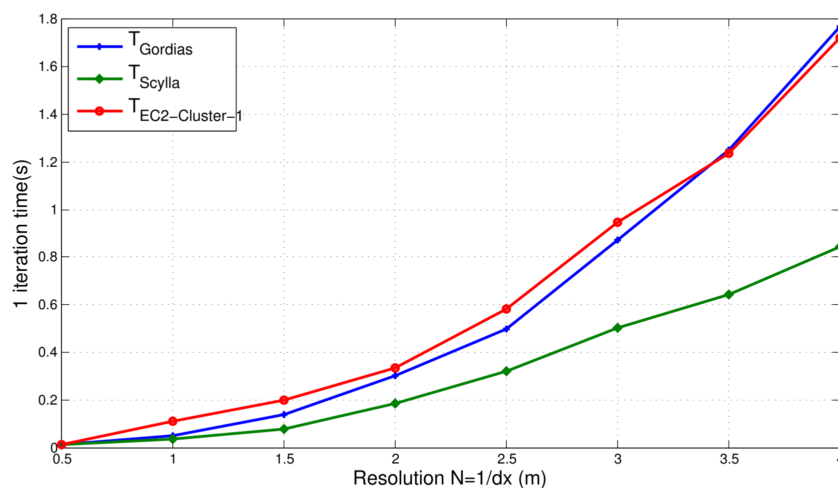


Figure 6.5 – Monolithic execution of the Cavity3D flow problem over 100 cores. The quantity N stands for the number of grid points per meter.

out considers a standalone simulation (without splitting) of the cavity3D problem over the EC2-cluster-1, *Scylla* and *Gordias* clusters. Figure 6.5 compares these monolithic executions over 100 cores ($L_x = 13000m$, $L_y = 40m$ and $L_z = 10m$). We varied the resolution N, which is the number of grid points per meter from 0.5 to 4, with a step size of 0.5. So, increasing the resolution N implies decreasing the basic unit for lattice spacing Δx ($N = \frac{1}{\Delta x}$) and increasing the domain size. The measured CPU time reflects the performance of each cluster. We believe that the difference of execution time for high resolution (for example, N=4) is mainly due to the low bandwidth between the Amazon-EC2 nodes and to their modest processors speed. Also, the *Scylla* cluster has better RAM size and cores distribution than *Gordias*.

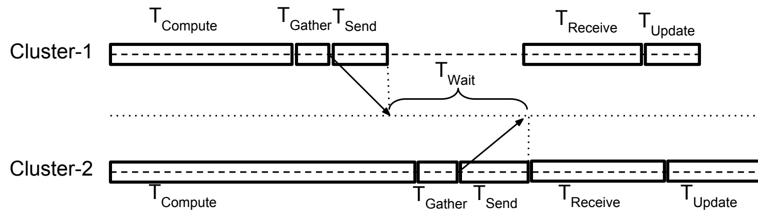


Figure 6.6 – Definition of the different time slots characterizing a distributed execution over two connected clusters.

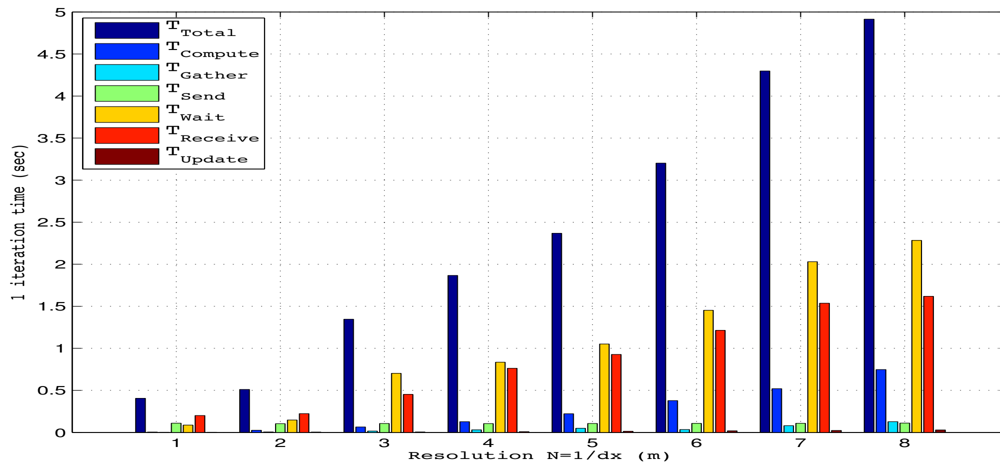
The second experiment we carried out amounts to tightly coupling the two cavity sections using MUSCLE2 and to running them simultaneously as described in the following measurement.

6.5.2 `DMC` execution

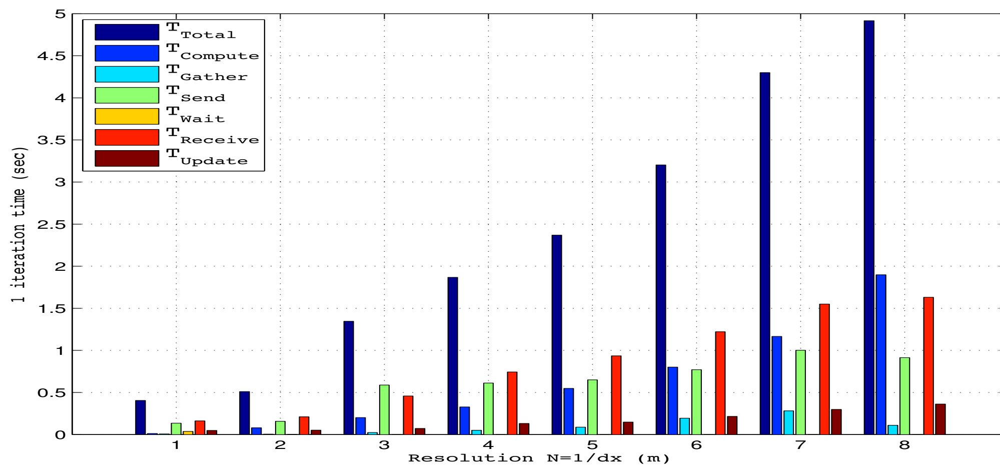
In this measurement, we changed the *Gordias* cluster by a cloud cluster mounted on Amazon-EC2 (EC2-Cluster-1 and EC2-Cluster-2). By this, we would like to demonstrate the feasibility of using the MMSF tools to connect local resources with the cloud. This study will also give the order of magnitude of the difference of performance between a cloud and a HPC resource.

Each section of the cavity3D problem runs over 50 cores on each machine. Once the EC2-Cluster is mounted, we manually connect to the master node of EC2-Cluster in order to configure the MUSCLE2 connection layer and run the computation. Note that it is possible to install and configure MUSCLE2 as a plug-in package before starting up the EC2-Cluster. We can also build it using personalized virtual machine images.

As depicted in Figure 6.6, we define T_{Wait} as the time required for a section to wait until the border data is available for reading. Note that after the *compute()* operation, each section firstly sends data, and then, waits for incoming data. We define T_{Send} as the time required to only drop data in the network and $T_{Receive}$ as the time required for the data



(a) section 1 on *Scylla*

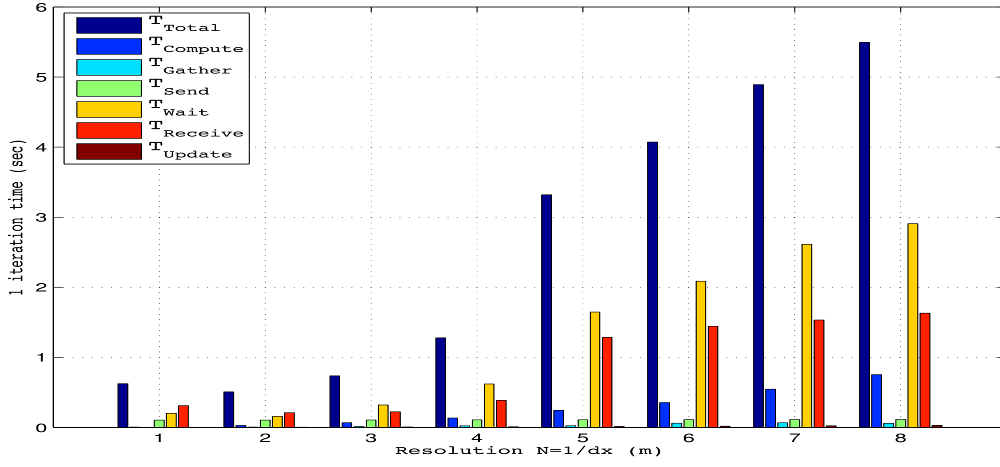


(b) section 2 on EC2-Cluster-1

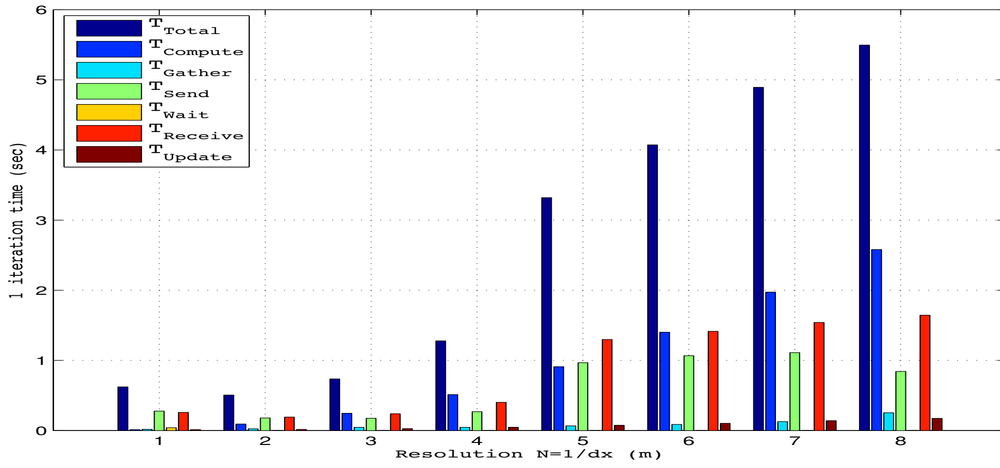
Figure 6.7 – The runtime of different operations in the distributed computations cases, during one iteration, as described in algorithm 5 and Fig. 6.6. Figure 6.7a and Figure 6.7b present the results of the same execution using *Scylla* and EC2-Cluster-1. T_{Total}, $T_{Compute}$, T_{Gather}, T_{Update} correspond to the times of the operations `compute()`, `getBoundaryData()`, `updateBoundaryData()`, respectively. T_{Send} and T_{Wait} are defined as the time required to only drop data in the network and $T_{Receive}$ as the time required for the data to move from the sender to the receiver.

to move from the sender to the receiver.

The measured computation time is depicted in Figures 6.7-6.8. On the one hand, Figure 6.7b shows a larger time $T_{Compute}$ than that shown in Figure 6.7a. This can be explained by the difference of the connection speed between nodes (see network description in Table 6.1). On the other hand, the times T_{Send} and $T_{Receive}$ depend on the inter-cluster network bandwidth: we have measured 22 MBytes/sec from *Scylla* to EC2-Cluster and 13.8 MBytes/sec from EC2-Cluster to *Scylla*.



(a) section 1 on *Scylla*



(b) section 2 on EC2-Cluster-2

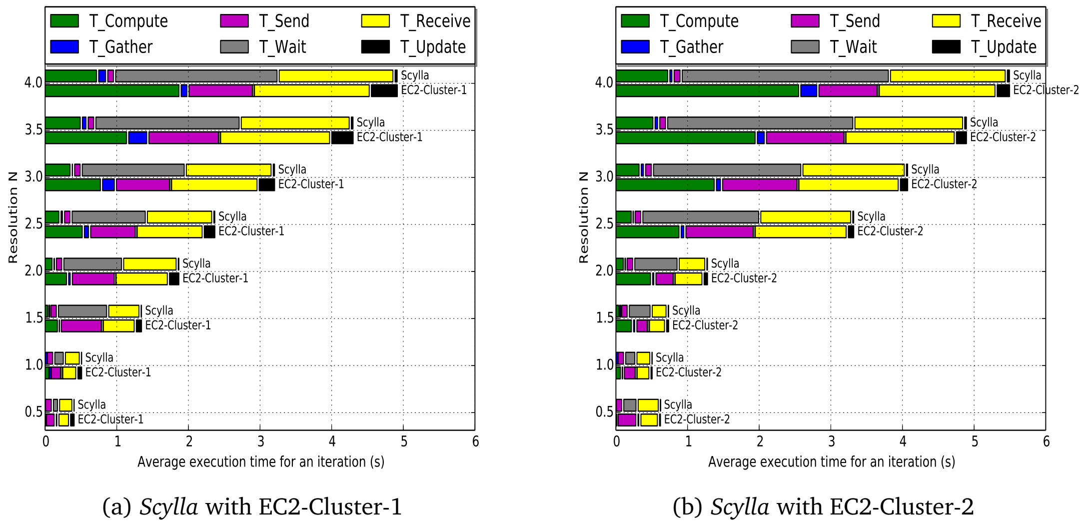
Figure 6.8 – The runtime of different operations in the distributed computations cases, during one iteration, as described in algorithm 5: Figure 6.8a and Figure 6.8b present the results of the same execution using *Scylla* and EC2-Cluster-2.
T_{Total}, $T_{Compute}$, T_{Gather}, T_{Update} correspond to the times of the operations `compute()`, `getBoundaryData()`, `updateBoundaryData()`, respectively. T_{Send} and T_{Wait} are defined as the time required to only drop data in the network and $T_{Receive}$ as the time required for the data to move from the sender to the receiver.

We recorded the time-stamps of the computing operations at each iteration in order to measure T_{Wait}. We combined the measurements of Figures 6.7-6.8 into a timeline graph (see Figure 6.9). In Figure 6.9a, and for high resolution (N=4.0), T_{Wait} represents 46% of the total execution time on *Scylla*. Similarly in Figure 6.9b, T_{Wait} represents 53% of the total execution time on the *Scylla* cluster.

The spread of performance shown in Figure 6.9 simply reflects the synchronization time required to tightly couple submodels over different machines. The measurements clearly

Table 6.5 – 1 iteration time (sec) of the scenario 3. T_{mpi} is the average time of a monolithic execution on 100 cores of EC2-Cluster-1. θ^{grid} is the overhead defined in eq. 6.12.

Resolution	T_{mpi}	T_{DMC}	θ^{grid}
0.5	0.0131	0.405	29.91
1	0.1106	0.51	3.61
1.5	0.2001	1.346	5.72
2	0.3328	1.867	4.60
2.5	0.5815	2.37	3.07
3	0.9446	3.204	2.39
3.5	1.2365	4.3	2.47
4	1.7221	4.917	1.85



(a) *Scylla* with EC2-Cluster-1

(b) *Scylla* with EC2-Cluster-2

Figure 6.9 – Timeline chart of the runtime execution over 50 cores. $T_{PreCompute}$ stands for the time required to only compute the border domain in order to get the border data ready to be sent. The "compute()" operation (represented by $T_{Compute}$) is performed in parallel with the "gatherBoundaryData()" and "sendReceiveBoundaryData()" operations (represented by T_{Gather}, T_{Send} and $T_{Receive}$).

show that the distributed execution is approximately 5 times slower than the monolithic execution on *Scylla*. We can conclude that the coupling strategy should be based on the different demands each submodel places upon CPU performance and network bandwidth.

The performance measurement is summarized in Table 6.5. For high resolution ($N\!=\!4$), the value of θ^{grid} is about 1.85. This low performance is due to the difference of computing power and bandwidth speed between *Scylla* and EC2-Cluster.

6.6 Conclusion

In this chapter we presented the evaluation of the MMSF through the experiments conducted over different computing resources: EGI, university and cloud clusters. Measurement showed that the overhead is acceptable with a good ratio between the computation and the communication times. In particular, we have successfully connected Amazon cloud resources to a MAPPER computing infrastructure. We firstly built an MPI cluster on Amazon cloud platform using the cc2.8xlarge and c1.xlarge instance types. Then, we connected it to a local cluster (Scylla) located at the University of Geneva (Switzerland) using the MUSCLE library, thus performing a distributed execution of two tightly coupled MPI submodels.

Coupling of a PRACE machine (*superMUC* [136] supercomputer) with an EGI cluster has been also be done showing that the MMSF can be also handle supercomputers machines.

In chapter 7, we present the optimization strategy applied to our application to optimize the data transfer and evaluate further the `DMC` aspect of the MMSF approach.

7 Impact of optimization techniques on DMC simulation*

Introduction

In this chapter we present the optimization techniques that we used to reduce the time of data transfer and the waiting time in order to improve the simulation efficiency. Then we evaluate the performance of deploying stencil-based applications in general, and Lattice Boltzmann based applications in particular, over a distributed infrastructure after applying these optimization techniques. In this chapter, we consider the same benchmark application as in chapter 6 and the same University and cloud resources.

7.1 Boundaries in stencil based applications

A several range of scientific applications are modeled using mathematical equations. Resolving these equations can be performed on machines through simulation using numerical methods (such as the Lattice Boltzmann method). This requires to map the mathematical equations into a discrete data representation in order to resolve them numerically. Generally, a regular grid table is used as data structure. Thereby, a numerical simulation consists in applying a solver iteratively on each cell of this grid structure. This usually requires interaction and exchange of information between the neighboring cells. Stencil based applications fit well in this category especially for the scientific domains such as the computational fluid dynamics.

*This content of this chapter is derived from:

- Mohamed Ben Belgacem and Bastien Chopard. A hybrid HPC/cloud distributed infrastructure: Coupling EC2 cloud resources with HPC clusters to run large tightly coupled multiscale applications. *Future Generation Computer Systems*, 42(0):11–21, 2014.
- J. Borgdoroff, M. Ben Belgacem C. Bona-Casas L. Fazendeiro D. Groen O. Hoenen A. Mizeranschi J. L. Suter D. Coster P. V. Coveney W. Dubitzky A. G. Hoekstra P. Strand and Chopard, B. Performance of Distributed Multiscale Simulations. *Journal of Philosophical Transactions A*, 372(2021), August 2014.

Parallel computing is well adapted to this kind of simulations. Inter-process communications reflect the exchanges of information between neighboring cells. Depending on the solver boundary condition, a given cell may require information not only from its direct neighbors but also from other adjacent cells.

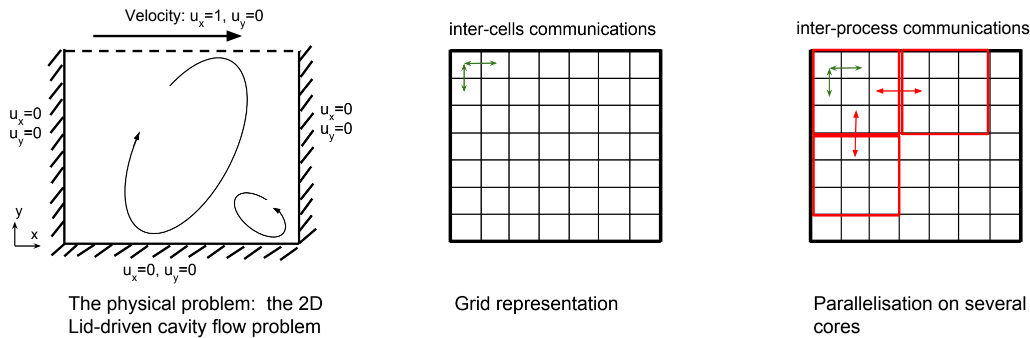


Figure 7.1 – Illustration of an example of a stencil based application: 2D Lid-driven cavity flow problem. Green arrows represent the communication between neighboring cells. A red block represent the domain to process by one machine core. Red arrows represent the inter-process communication.

In this chapter we describe the optimization techniques we used to reduce the overhead induced by the MMSF approach. They can be applied to stencil-based applications in general and to Lattice Boltzmann based applications in particular.

7.2 Optimization techniques

Chapter 6 showed that for the case of geographically distributed clusters the time to transfer data has an impact on the overall execution time, especially in the case of a cloud cluster connected to a University cluster. We improved the performance of the MMSF approach by using optimization techniques to reduce the impact of the time to send, wait and receive data in a distributed simulation. These techniques consist of an overlapping communication-and-computation and a load-balancing techniques.

7.2.1 The overlapping communication-and-computation technique

This technique will be illustrated with the Lattice Boltzmann method as follows. A given cell in the grid structure requires e adjacent cells in all directions to compute the boundary condition as specified in the steps of algorithm 5 (in chapter 6): *collide*, *stream* and *boundary update*. Thus, each sub-domain (submodel) (Figure 6.2 in chapter 6) requires an extra boundary block with e cells in the x-axis direction, including all the corresponding y-axis and z-axis cells as described in Figure 7.2.

For each submodel, the optimization technique consists in sending the extended boundary

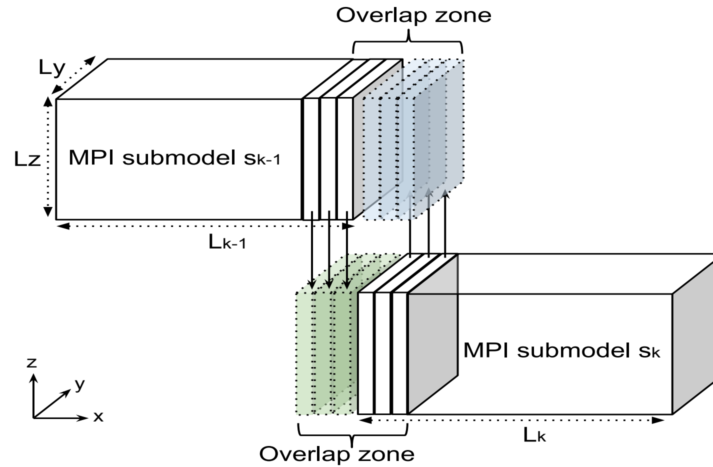


Figure 7.2 – Illustration of the overlap zone and the extended boundary when coupling two submodels in a stencil based application.

Algorithm 6: Applying the optimization technique on the computation algorithm

```
1 F_init();
2 while (iterations < max_Iteration_Number) do
3     PreCompute();
4     In parallel do = { gatherBoundaryData() then sendReceiveBoundaryData()
                       { compute()                                          ;
5     updateBoundaryData();
6 Observation();
```

block to the target submodel in parallel to the computation of the full sub-domain as described in algorithm 6. First, in the *PreCompute()* operation, the boundary domain is computed before performing the *compute()* operation of the sub-domain. This prepares the data to be transferred to the target submodel. Here, we can apply an other optimization technique by assigning the data of the boundary block to the main MPI process. This allows us to avoid the time overhead induced by the data "gathering" and "updating" of the boundary block over several cores. The next step consists in launching a thread on the same core as the main MPI process. This thread will only handle the *gatherBoundaryData()* and *sendReceiveBoundaryData()* operations in parallel to the *compute()* operation. The *updateBoundaryData()* operation is performed once the *compute()* operations had finished. This optimization practically enables us to run the main computation in parallel with the data transfer and thereby to reduce significantly the overall execution time.

In what follows, we evaluate the application of this overlapping technique using the same benchmark as in chapter 6.

Experiments on Universities HPC clusters

In this experiment, we consider the same scenario as in section 6.4 and we apply the overlap optimization of data transfer between *Gordias* and *Scylla*. We define the following parameters:

- p_{mono}: the number of cores used for the monolithic computation (T_{mpi}).
- $p_{\texttt{DMC}}^{local}$: the number of used cores for the distributed multiscale computation (DMC) on the same cluster ($T_{\texttt{DMC}}^{local}$).
- $p_{\texttt{DMC}}^{grid}$: the number of used cores for the distributed multiscale computation (DMC) on two different clusters ($T_{\texttt{DMC}}^{grid}$).

The first measurement considers $p_{\texttt{DMC}}^{local} = p_{\texttt{DMC}}^{grid} = 50 + 50$ cores. The second measurement consists in a strong scale using $p_{\texttt{DMC}}^{local} = p_{\texttt{DMC}}^{grid} = 100 + 100$ cores with the same problem size. Results are illustrated in Table 7.1.

Table 7.1 – Performance measurement after applying the overlap optimization technique. Low resolution corresponds to $N = 0.5$. High resolution corresponds to $N = 4$. Execution times are expressed in seconds. The overhead θ is computed based on eq. 6.12.

Simulation	p_{mono}	T_{mpi}	$p_{\texttt{DMC}}^{local}$	$T_{\texttt{DMC}}^{local}$(s)	$p_{\texttt{DMC}}^{grid}$(s)	$T_{\texttt{DMC}}^{grid}$(s)	θ^{local}	θ^{grid}
Low resolution	100	0.011	50+50	0.015	50+50	0.025	0.26	0.56
			100+100	0.019	100+100	0.023	0.42	0.52
High resolution	100	1.77	50+50	1.37	50+50	1.31	-0.29	-0.35
			100+100	0.78	100+100	0.71	-1.26	-1.49

The simulation is performed on *Gordias* and the *Scylla* clusters, with $T_{\texttt{DMC}}^{local}$ taken as the average of the $T_{\texttt{DMC}}^{local}$ on *Gordias* and on *Scylla*. It consists of a comparison between the execution of the two submodels on $50 + 50$ cores and the execution of a single monolithic model with the same total problem size on 100 cores.

After applying the overlap optimization technique, a speed-up is not realized for a low-resolution domain size, as the computation time is too short compared with the communication time. For a high resolution, combining the *Gordias* cluster with the *Scylla* cluster leads to reduce the time send/receive of data. $T_{\texttt{DMC}}^{grid}$ $(50 + 50)$ has a negative overhead $\theta^{\text{grid}} = -0.35$. This can be explained by the fact that bandwidth between *Scylla* and *Gordias* is good and the send/receive time is masked by the *compute()* operation. Comparing a distributed run on $100 + 100$ cores, the gain is even larger with a $\theta^{\text{grid}} = -1.49$. For time-dependent runs where high accuracy is required and local resources are limited, distributed computing turns out to be advantageous. The previous benchmarks presented in chapter6 showed small differences between local and distributed computations of the same problem size with the same number of cores.

Cloud measurements

The same configuration as in section 6.5 is used to evaluate the overlap optimization technique. The detailed timeline view shown in Figure 7.3 indicates that, for $N = 4$, the execution time is reduced by a factor of 1.4: from $4.91sec$ with an overhead $\theta^{\text{grid}} = 1.85$ to $3.52sec$ with a $\theta^{\text{grid}} = 0.98$. It also shows that $T_{PreCompute}$ is negligible compared to $T_{compute}$, and that T_{Gather} and T_{Update} are reduced since the boundary data run on one core.

7.2.2 Load-balancing technique

Another important issue when coupling two different architectures is to balance either the resources or the size of sub-domains each side so that the waiting time vanishes as much as possible.

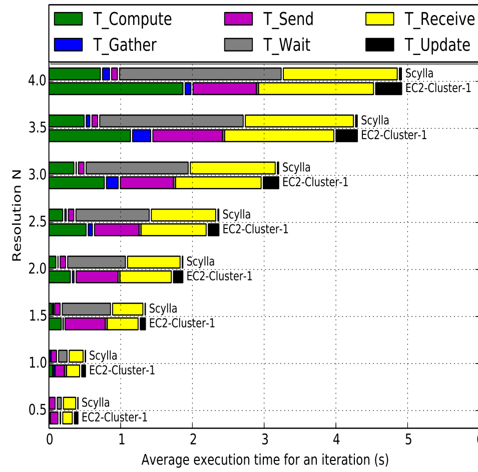
Accordingly, as a first experiment, we have used more EC2 nodes and less Scylla nodes. We ran a test using 70 cores on EC2-cluster-1 and 30 cores on Scylla. The overall execution time for the case $N = 4.0$ is reduced from $2.58sec$ to $1.82sec$. To further explore the impact of different load balancing strategies, we also varied in a second experiment the size of the sub-domain assigned to the 50 cores of each machine (see Table 7.2).

Table 7.2 – Measurements using 50 cores for each machine with a resolution $N = 4.0$ and applying the overlap and load-balancing optimization strategies. In the column *Size of sections*, the size of each section is expressed as a percentage value relatively to the overall size of the problem. The overhead θ^{grid} is computed based on eq. 6.12.

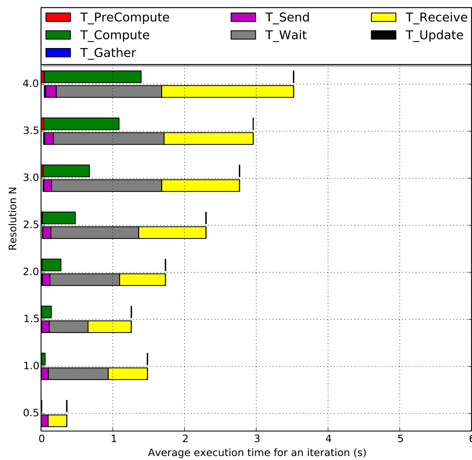
Size of sections in %		$\mathbf{T_{DMC}}$	θ^{grid}
Scylla	EC2-cluster-1		
60%	40%	3.79	1.20
65%	35%	2.46	0.43
70%	30%	2.52	0.46
75%	25%	2.60	0.51
80%	20%	3.32	0.93

This experiment shows that, by assigning 65% of the problem domain to Scylla and 35% to EC2-Cluster-1, the overhead θ^{grid} decreases from 1.85 to 0.43.

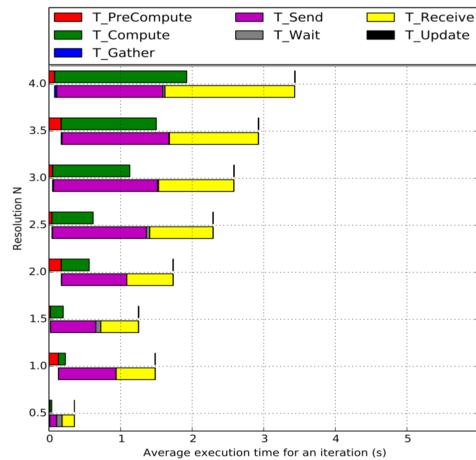
It should be noted that, in this overhead evaluation of the MMSF approach, we do not aim to exhaustively analyze the long-distance network throughput. Our objective is study the large scale simulation of tightly coupled scientific applications over a hybrid distributed HPC platform based on the MMSF approach. There exist other research works that studied more in details the performance and the variability of the network throughput (see for instance [88, 85, 93, 146, 75]) by considering a fine grained tuning of TCP parameters, network latency and memory management that may have an impact



(a) *Scylla* with EC2-Cluster-1 without optimization



(b) Scylla cluster with optimization

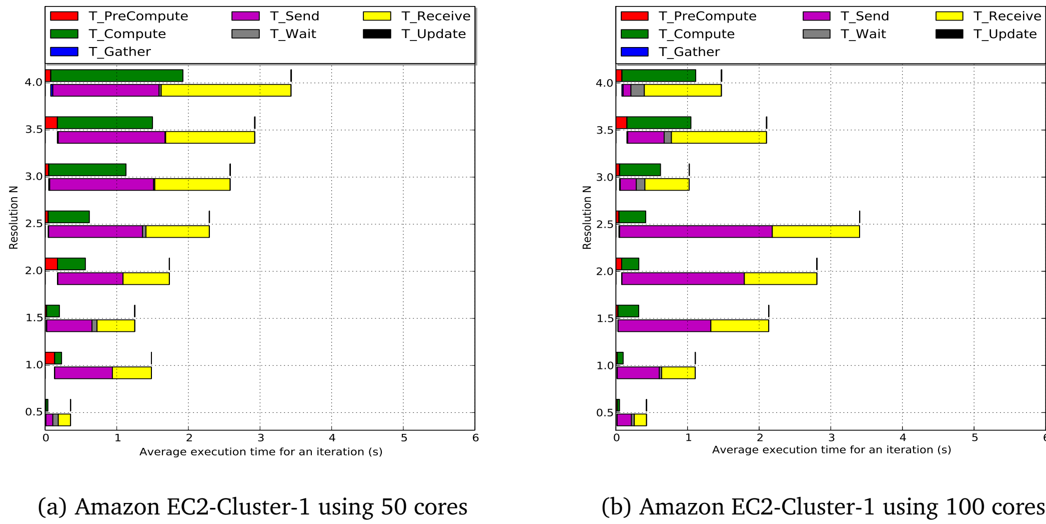


(c) Amazon EC2-Cluster-1 with optimization

Figure 7.3 – Timeline chart of the runtime execution before (Fig. 7.3a) and after (Fig. 7.3b-7.3c) applying the overlapping communication-and-computation optimization strategy over 50 cores between Scylla and EC2-Cluster-1. For the optimization case, the “compute()” operation (represented by $T_{Compute}$) is performed in parallel with the “gatherBoundaryData()” and “sendReceiveBoundaryData()” operations (represented by T_{Gather}, T_{Send} and $T_{Receive}$).

on the overall performance of these kind of large simulations.

The results presented in Table 7.2 are obtained using $50 + 50$ cores. We can investigate how they change when considering more than 50 cores on each machine. Figure 7.4 compares the runtime execution when using 50 and 100 cores for each cluster machine.



(a) Amazon EC2-Cluster-1 using 50 cores

(b) Amazon EC2-Cluster-1 using 100 cores

Figure 7.4 – Timeline chart comparing the runtime execution using the the communication-and-computation optimization strategy using either 50 or 100 cores on each machines. Only the behavior of the Amazon EC2-Cluster-1 is shown. The "compute()" operation (represented by $T_{Compute}$) is performed in parallel with the "gatherBoundaryData()" and "sendReceiveBoundaryData()" operations (represented by T_{Gather}, T_{Send} and $T_{Receive}$)

Only the data for the Amazon EC2-cluster-1 are shown, for simplicity. The decrease of $T_{Compute}$ observed when adding more cores is expected since $T_{Compute}$ mostly depends on the performance of the resources.

Note that when running experiments on the cloud, there is a variability in the time of sending and receiving data between the two clusters. All the measurements on the cloud were repeated three times and conducted during Saturday night and during the day, where the Amazon EC2 data center and the university network are likely to be less loaded. Several factors can influence the throughput of the long-distance TCP connection between Scylla and EC2-Cluster-1, such as unpredictable network congestion, re-transmission problem related to packets loss, TCP windows size, etc. This heterogeneity makes the measurement comparison difficult, since two runs are completely depending on the status of the traffic-load of the network in the Amazon data center and the Internet transatlantic backbone connecting the two continents.

With the proposed domain decomposition in linear sections, the communication overhead is not reduced by increasing the number of cores. Clearly, for big domains the coupling is more attractive since we can take advantage of using more computing cores.

Also, coupling university clusters with cloud resources can be a very good solution if,

for instance, the problem size does not fit in the available local resource. In our case, a monolithic simulation over 50 cores on *Scylla* with a resolution $N > 4.0$ is not feasible due to the cluster memory limitation. Moving a part of the domain on a cloud makes such computation possible with an acceptable overhead.

As shown in section 2.6 (in chapter 2), the hydrology application wants to study the water behavior and the sedimentation process over a long distance ($\sim 30\ Km$) of a river section. To better explore the benefit of using a remote machine to augment the local resource, we considered a cavity3D problem of length $30\ km$ (before splitting). We ran a simulation using 50 cores on each cluster (N is fixed to 4.0). Results show that with a good load-balancing, $T_{compute}$ is now greater than the time required to transfer boundary data (see Figure 7.5). In this instance, with a partition size of 55% of the full section on Scylla and 45% on EC2-Cluster-1, $T_{compute}$ on EC2-Cluster-1 (resp. on Scylla) is equal to $3.74sec$ (resp. $3.87sec$) and the time required to exchange boundary data is equal to $3.35sec$ (resp. $3.34sec$). In this case, the coupling overhead is totally masked since the inter-cluster communication is performed in parallel with the computation, as depicted in Figure 7.5. Moreover, when comparing the time of a monolithic execution of the same problem on Scylla using 50 cores, we see that the distributed execution over $50 + 50$ cores enables us to reduce the execution time by a factor of 1.8. Although this is lower than the ideal speedup of 2, we get a parallel efficiency of 90%. This clearly shows the benefit of using the MMSF approach in this case.

An other situation that makes the coupling with cloud resources a viable solution for multiscale simulations is to consider national or private cloud data centers, likely to have a fast and stable network bandwidth, allowing a significant reduction of the time required to exchange the boundary data.

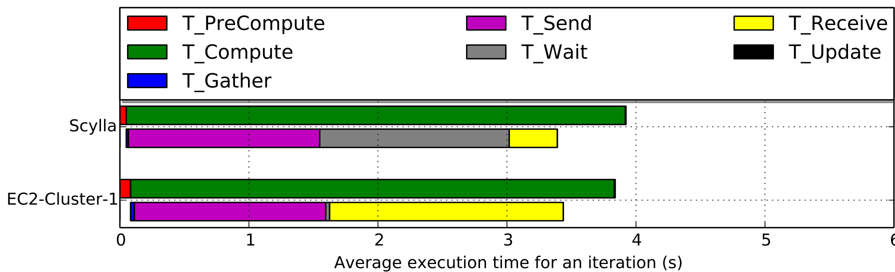


Figure 7.5 – Simulation of a problem with length size of $30Km$ using 50 cores for each section. The simulation is run with a load-balancing of 55% of the full problem size on Scylla and 45% on EC2-Cluster-1.

7.3 Conclusion

In this chapter, we explained the overlap and load-balancing optimization techniques used to minimize the data transfer and the execution time for our benchmarks.

For the case of University clusters, we showed that for strong scaling (50 cores for monolithic execution compared to $100 + 100$ cores for distributed one with the same problem size), we obtained a good speed-up with a high resolution. For cloud resources, we showed that load-balancing and the overlapping techniques have been instrumental in order to minimize the synchronization time between the two connected machines. Hence, while current cloud resources are not as powerful as classic HPC machines, they may still be a good solution to scale up computing power beyond the limitation of a single machine. Cloud providers keep improving their infrastructures and resources, but improvements of the network configuration are therefore desirable to make the cloud more useful for daily e-science computations.

An other optimization technique "the garbage computing" that may reduce the time of data transfer has been also implemented but not benchmarked in this work. The "garbage computing" consists in extending the overlap boundary between two submodels so that they exchange their boundary data after each x iterations of computation. Obviously, this will increase the time of computation since the submodel domain is extended. So, the idea is to find the optimal value of x that will reduce the time of data transfer and the overall execution time within the MMSF approach.
Our performance measurements has been developed for a specific coupling scenario, but it can be indicative to the stencil based applications in general.

8 Conclusion and perspectives

8.1 Conclusion

The work presented in this thesis work the modeling, programing and deploying of e-science applications on large distributed HPC platforms. We mainly aimed to find a tandem between the jungle of the existing computing resources and the variety of the scientific applications. Computing resources, as shown in the Introduction chapter of this thesis, range from normal Desktops to distributed infrastructures such HPC grid of clusters, supercomputers, Desktop grid and cloud platforms.

We have considered two categories of applications as defined in chapter 2 (Fig. 2.13): i) *embarrassingly parallel*: concerns the applications that have a workflow with a convergence condition where the number of jobs is not known in advance and ii) *communication intensive*: concerns the applications that have a tightly coupled workflow with a highly parallel computation where jobs communicate. In chapter 2, we have enumerated the set of applications that we have worked on: *NeuroWeb*, *MetaPIGA*, *GIFT*, *Selector*, *CleanCity* and *Hydrology*. They are of different complexities, belong to different scientific domains, are CPU demanding and some of them are multiscale. The accumulated experience during this thesis has led us to move from specific solutions of porting scientific applications to more general ones.

To deploy the first category of these applications on distributed infrastructures, we have firstly developed a dedicated programming API which is built on top of the Desktop platform XtremWeb-CH [16, 106]. This API provides all requested functionalities to monitor jobs, express parallelism, manage precedence rules and data dependency. However, the use of this API by non-expert computer developers is not trivial: source codes are often too verbose and difficult to understand and maintain. In this way, we have presented two high level programming paradigms in chapter 3. The two programming paradigms targeted loosely coupled applications having a convergence condition. They prevent the users to be exposed to the technical complexity of programming and deploying their

applications. Porting an application is thereby reduced to a simple program written in a *Object Oriented* style. Besides, these two programming models support several types of computing resources such as cloud platform which has been illustrated through the MetaPIGA application.

For the second category of applications, chapter 4 presented the Multiscale Modeling and Simulation Framework (MMSF), developed within the European project MAPPER [102]. It presented a new rationale to design, implement and run heavily parallel multiscale applications on distributed infrastructures. The conceptual phase of the MMSF approach clarifies the understanding of a given multiscale phenomena: it defines the temporal and spatial scales, separates the physical phenomena into single-scale ones and models the interaction between them. To this end, the Multiscale Modeling Language (*MML*), part of the MMSF, is used to describe graphically the workflow of a given multiscale application. In *MML*, each submodel should obey to a defined execution loop (*SEL*) and interaction between submodels are done based on a set of defined coupling templates. This phase is crucial since it offers well defined steps to model and design multiscale applications. The implementation phase of the MMSF approach is done compliantly to the conception phase. It offers a modular approach to program and couple together submodels, *mapper*s and *filter*s. For instance, one can couple a `c++` MPI based submodel with `Java` or `Phyton` ones.

Applying the MMSF approach was illustrated in chapter 5 and chapter 6 through the hydrology application case. Building an irrigation canal on the fly was detailed in chapter 5, showing the potential of the MMSF approach. The use case we studied is an irrigation canal composed of two canal sections, each modeled numerically with a 1D shallow water equations, coupled through a gate and a spillway junctions. Firstly, we presented the *MML* description of this scenario. Then, we presented an explicit implementation of the gate and spillway junctions using the Lattice Boltzmann method (LBM). The applications is thus tightly coupled since that, at each iteration, the water junctions receive information from both canal sections and return the proper boundary conditions that ensure the right discharge between the two sections. It is also multiscale since each canal section is considered as an independent submodel and so it can run with different spatial and temporal scales. We validated this coupling by applying the full steps of the MMSF approach and we showed the convergence of our implementation.

Chapters 6 and 7 addressed this question: *is-it beneficial to use distributed multiscale computing for tightly coupled parallel applications*?. This depends on several aspects like the type of the distributed infrastructures, the implementation of submodels, their scales and the characteristics of their multiscale coupling. Based on the simulations performance, the answer that chapters 6 and 7 gave is: yes, dependently. Indeed, the multiscale distributed computing performance is lower than the monolithic one without a proper adjustment of the CPUs power, bandwidth of computing machines and the problem size. Once the resources are adequately chosen, results showed that adding

more cores to an intensive computation can bring a very good speedup, even though the coupled machines are linked with a modest and fluctuating bandwidth network. This also depends on the type of the application and the degree of the communication between the submodels composing it.

This brings us to the question: *How did MMSF changed doing computational science?*. Based on the experience of developing several applications within the MAPPER project, the MMSF approach enabled to access to unprecedented computing resources by coupling several supercomputers. In addition, a multiscale application can be composed of several codes implemented in different programming languages. Within the same computation, one can choose the adequate resource such GPUs, multicore machines and clusters that optimizes the execution for each code. The MMSF approach also gave a flexibility to build complex multiscale simulations with an excellent reuse of components, and without loss of performance. Furthermore, MMSF brought research at a new level: For example, afford a “fully resolve" simulation of a long section of a river.

8.2 Limitations

The two programming models proposed in chapter 3 are only restricted to dynamic workflow (DW) e-science applications. Besides, the end-user should have a very basic knowledge about the *Object Oriented Programming* concept in order to understand and use the two models. When using public cloud resources (such as Amazon-EC2), short jobs are penalized by the time overhead needed to start and setup the required virtual machines.

The MMSF approach is attractive for distributed multiscale computations, but it has some limitations. In the conceptual phase of the MMSF approach, submodels are coupled either directly or through *mappers/filter*s. A mapper has a fixed number of ports which limits the number of submodels to be connected to it. The reason behind this is that each *mapper* port should be predefined before starting the simulation. This limitation can be resolved by considering a parameterized number of ports when instantiating a *mapper* at the design phase of the workflow (MAD tool). In this work we did not explore a dynamic topology where the programmer can create in the code a dynamic number of ports.

For many heavy simulations like the Hydrology case, decomposing the computation domain into several submodels and scheduling them over the available computing resources should be done in an efficient way. The performance model presented in chapter 6 treated the case of a regular decomposition. However, a more advanced scheduling algorithm should be elaborated and take into account the HPC resources configuration which are already multiscale in nature.

The MMSF presents also some limitations on the practical level. The MAPPER frame-

work was built to operate on the European e-infrastructure. However, the European e-infrastructure federates heterogeneous and complex clusters and supercomputers, where softwares and services are governed by European organizations (such as PRACE, EGI) having a different access and security policies. The sustainability of using this framework is therefore limited in lifetime, and requires considerable logistic effort to maintain it operational. It is worth saying here that MUSCLE2, MaMe-MAD, QCG middleware, part of the MAPPER project tools, can operate in standalone mode. For instance, one can use MUSCLE2 to perform a distribution multiscale computation.

8.3 Perspectives

In our opinion, we believe that the scientific community needs a paradigm shift in the term of doing computational sciences in the future. This domain should take benefit from the unprecedented power of existing computing resources. This addresses the programming and code development challenges of large parallel applications.

We aim in a future work to make the MMSF framework more closer to the Swiss scientific community by deploying the MAPPER softwares on several machines at the Swiss National Supercomputing Center (CSCS) [45]. In addition, we want to augment the framework with a performance modeling tool, that will allow the users to map optimally a multiscale application to a given HPC computing infrastructure. The objective is either to minimize execution time or energy consumption, or the maximize resource utilization.

Finally, we will develop and deploy a multiscale, multiphysics university application using the MMSF formalism. This application will be used to predict the transport of volcanic ashes in the atmosphere. It will advance the sate of the art in hazard assessment in geoscience and will be an other demonstrator of the MMSF approach.

Publications

A technical part of the work in this thesis does not appear in the present document. For instance, important development was invested to extend the Desktop grid platform XtremWeb-CH [148] programmed with the advanced J2EE environment.

International journals:

1. Mohamed Ben Belgacem and Bastien Chopard. A hybrid HPC/cloud distributed infrastructure: Coupling EC2 cloud resources with HPC clusters to run large tightly coupled multiscale applications. *Future Generation Computer Systems*, 42(0):11–21, 2014
2. Mohamed Ben Belgacem, Bastien Chopard, Latt Jonas, and Andrea Parmigiani. A Framework for Building a Network of Irrigation Canals on a Distributed Computing Environment: a case study. *Journal of Cellular Automata. Special Issue: Cellular Automata Applications for Research and Industry*, 9(2-3):225–240, 2014
3. M. Ben Belgacem, N. Abdennadher, and M. Niinimaki. Virtual EZ Grid: A Volunteer Computing Infrastructure for Scientific Medical Applications. *International Journal of Handheld Computing Research*, 3(1):74–85, 2012
4. J. Borgdoroff, M. Ben Belgacem C. Bona-Casas L. Fazendeiro D. Groen O. Hoenen A. Mizeranschi J. L. Suter D. Coster P. V. Coveney W. Dubitzky A. G. Hoekstra P. Strand and Chopard, B. Performance of Distributed Multiscale Simulations. *Journal of Philosophical Transactions A*, 372(2021), August 2014
5. J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. Ben Belgacem, B. Chopard, D. Groen, P.V. Coveney, and A.G. Hoekstra. Distributed multiscale computing with MUSCLE2, the Multiscale Coupling Library and Environment. *Journal of Computational Science*, 5(5):719–731, 2014
6. Joris Borgdorff, Mariusz Mamonski, Bartosz Bosak, Krzysztof Kurowski, Mohamed Ben Belgacem, Bastien Chopard, Derek Groen, Peter V. Coveney, and Alfons G. Hoekstra. Distributed Multiscale Computing with MUSCLE 2, the Multiscale Coupling Library and Environment. *CoRR*, abs/1311.5740, 2013

7. Parmigiani,Andrea and Latt, Jonas and Ben Belgacem, Mohamed and Chopard, Bastien. A lattice Boltzmann Simulation of the Rhone river. *International Journal of Modern Physics C*, 11(24):1340008, 2013

International Conferences:

1. M. Ben Belgacem and N. Abdennadher. Skeleton paradigm for developing e-science applications on distributed platforms. In *The 6th IEEE International Workshop on Multicore and Multithreaded Architectures and Algorithms (M2A2 2014)*. IEEE Computer Society, 2014
2. Mohamed Ben Belgacem, Nabil Abdennadher, Marko Niinimaki. Programming distributed medical applications with XWCH2. In *Proceedings of HealthGrid 2010*, volume 159, France, June 2010
3. Mohamed Ben Belgacem, Bastien Chopard, Joris Borgdorff, Mariusz Mamonski, Katarzyna Rycerz, and Daniel Harezlak. Distributed Multiscale Computations Using the MAPPER Framework. In Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 1106–1115. Elsevier, 2013
4. Mohamed Ben Belgacem, Haithem Hafsi, and Nabil Abdennadher. A Hybrid Grid/-Cloud Distributed Platform: A Case Study. In JamesJ.(JongHyuk) Park, HamidR. Arabnia, Cheonshik Kim, Weisong Shi, and Joon-Min Gil, editors, *Grid and Pervasive Computing*, volume 7861 of *Lecture Notes in Computer Science*, pages 162–169. Springer Berlin Heidelberg, 2013
5. Mohamed Ben Belgacem, Bastien Chopard, and Andrea Parmigiani. Coupling Method for Building a Network of Irrigation Canals on a Distributed Computing Environment. In GeorgiosCh. Sirakoulis and Stefania Bandini, editors, *Cellular Automata*, volume 7495 of *Lecture Notes in Computer Science*, pages 309–318. Springer Berlin Heidelberg, 2012
6. MohamedBen Belgacem, Nabil Abdennadher, and Marko Niinimaki. Virtual EZ Grid: A Volunteer Computing Infrastructure for Scientific Medical Applications. In Paolo Bellavista, Ruay-Shiung Chang, Han-Chieh Chao, Shin-Feng Lin, and PeterM.A. Sloot, editors, *Advances in Grid and Pervasive Computing*, volume 6104 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin Heidelberg, 2010
7. Borgdorff, Joris and Mamonski, Mariusz and Bosak, Bartosz and Groen, Derek and Ben Belgacem, Mohamed and Kurowski, Krzysztof and Hoekstra, Alfons G. Multiscale Computing with the Multiscale Modeling Library and Runtime Environment. *Procedia Computer Science*, 18(0):1097–1105, 2013. 2013 International Conference on Computational Science

8. Sébastien Miquée, Raphaël Couturier, David Laiymani, Nabil Abdennhader, Mohamed Ben Belgacem, Marko Niinimaki, and Marc Sauget. Gridification of a radiotherapy dose computation application with the xtremweb-CH environment. In *Proceedings of the 6th international conference on Advances in grid and pervasive computing*, GPC'11, pages 188–197, Berlin, Heidelberg, 2011. Springer-Verlag
9. M. Ben Belgacem and N. Abdennadher. Towards a high level programming paradigm to deploy e-science applications with dynamic workflows on large scale distributed systems. In *The 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (IEEE/ACM CCGrid 2015)*. IEEE Computer Society, 2015. Accepted (10 January 2015)

Book chapter:

1. Mohamed Ben Belgacem, Nabil Abdennadher, Marko Niinimaki. *Desktop Grid Computing*, chapter The XtremWebCH Volunteer Computing Platform (3). Numerical Analysis and Scientific Computing Series. Chapman and Hall/CRC, June 25, 2012

Bibliography

[1] A. Hoekstra and E. Lorenz and J.-L. Falcone and B. Chopard. Towards a Complex automata formalism for multuscale modeling. *Int. J. Multiscale Multiscale Computational Engineering*, 5(6):491–502, 2008.

[2] N. Abdennadher and R. Boesch. Deploying PHYLIP phylogenetic package on a Large Scale Distributed System. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 673–678, May 2007.

[3] Emmen Ad and Versweyveld Leslie. IDGF/IDGF-SP International Desktop Grid federation. Technical report, IDGF consortium, December 2013. version 4.2.

[4] Emmanuel Agullo, Camille Coti, Thomas Hérault, Julien Langou, Sylvain Peyronnet, Ala Rezmerita, Franck Cappello, and Jack Dongarra. QCG-OMPI: MPI applications on grids. *Future Generation Comp. Syst.*, 27(4):357–369, 2011.

[5] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus Striped GridFTP Framework and Server. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, page 54, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Amazon EC2 cloud computing. http://aws.amazon.com/ec2/.

[7] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In Rajkumar Buyya, editor, *GRID*, pages 4–10. IEEE Computer Society, 2004.

[8] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@Home: An Experiment in Public-resource Computing. *Commun. ACM*, 45(11):56–61, November 2002.

[9] Advanced Resource Connector (ARC). http://www.nordugrid.org/arc/.

[10] Henri Bal and Kees Verstoep. Large-Scale Parallel Computing on Grids. *Electronic Notes in Theoretical Computer Science*, 220(2):3–17, 2008. Proceedings of the 7th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2008).

[11] Bartosz Bosak and Jacek Komasa and Piotr Kopta and Krzysztof Kurowski and Mariusz Mamoński and Tomasz Piontek. New Capabilities in QosCosGrid Middleware for Advanced Job Management, Advance Reservation and Co-allocation of Computing Resources – Quantum Chemistry Application Use Case. In Marian Bubak, Tomasz Szepieniec, and Kazimierz Wiatr, editors, *Building a National Distributed e-Infrastructure–PL-Grid*, volume 7136 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin / Heidelberg, 2012.

[12] David Baum. Reading a Phylogenetic Tree: The Meaning of Monophyletic Groups. *Nature Education*, 1(1), 2008.

[13] G Behrmann, P Fuhrmann, M Grønager, and J Kleist. A distributed storage system with dCache. *Journal of Physics: Conference Series*, 119(6):062014, 2008.

[14] M. Ben Belgacem and N. Abdennadher. Skeleton paradigm for developing e-science applications on distributed platforms. In *The 6th IEEE International Workshop on Multicore and Multithreaded Architectures and Algorithms (M2A2 2014)*. IEEE Computer Society, 2014.

[15] M. Ben Belgacem and N. Abdennadher. Towards a high level programming paradigm to deploy e-science applications with dynamic workflows on large scale distributed systems. In *The 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (IEEE/ACM CCGrid 2015)*. IEEE Computer Society, 2015. Accepted (10 January 2015).

[16] M. Ben Belgacem, N. Abdennadher, and M. Niinimaki. Virtual EZ Grid: A Volunteer Computing Infrastructure for Scientific Medical Applications. *International Journal of Handheld Computing Research*, 3(1):74–85, 2012.

[17] Mohamed BEN BELGACEM. Etude, conception et implémentation d'un générateur dynamique de tâches pour l'environnement de calcul volontaire XtremWeb-CH. Master's thesis, ENSI, Tunisia, 2009.

[18] Mohamed Ben Belgacem and Bastien Chopard. A hybrid HPC/cloud distributed infrastructure: Coupling EC2 cloud resources with HPC clusters to run large tightly coupled multiscale applications. *Future Generation Computer Systems*, 42(0):11–21, 2014.

[19] Mohamed Ben Belgacem, Bastien Chopard, Joris Borgdorff, Mariusz Mamonski, Katarzyna Rycerz, and Daniel Harezlak. Distributed Multiscale Computations Using the MAPPER Framework. In Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 1106–1115. Elsevier, 2013.

[20] MohamedBen Belgacem, Nabil Abdennadher, and Marko Niinimaki. Virtual EZ Grid: A Volunteer Computing Infrastructure for Scientific Medical Applications.

In Paolo Bellavista, Ruay-Shiung Chang, Han-Chieh Chao, Shin-Feng Lin, and PeterM.A. Sloot, editors, *Advances in Grid and Pervasive Computing*, volume 6104 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin Heidelberg, 2010.

[21] Mohamed Ben Belgacem, Bastien Chopard, Latt Jonas, and Andrea Parmigiani. A Framework for Building a Network of Irrigation Canals on a Distributed Computing Environment: a case study. *Journal of Cellular Automata. Special Issue: Cellular Automata Applications for Research and Industry*, 9(2-3):225–240, 2014.

[22] Mohamed Ben Belgacem, Bastien Chopard, and Andrea Parmigiani. Coupling Method for Building a Network of Irrigation Canals on a Distributed Computing Environment. In GeorgiosCh. Sirakoulis and Stefania Bandini, editors, *Cellular Automata*, volume 7495 of *Lecture Notes in Computer Science*, pages 309–318. Springer Berlin Heidelberg, 2012.

[23] Mohamed Ben Belgacem, Haithem Hafsi, and Nabil Abdennadher. A Hybrid Grid/Cloud Distributed Platform: A Case Study. In JamesJ.(JongHyuk) Park, HamidR. Arabnia, Cheonshik Kim, Weisong Shi, and Joon-Min Gil, editors, *Grid and Pervasive Computing*, volume 7861 of *Lecture Notes in Computer Science*, pages 162–169. Springer Berlin Heidelberg, 2013.

[24] BlueGene/Q. http://www-03.ibm.com/systems/technicalcomputing/solutions/bluegene/.

[25] BOINC. http://boinc.berkeley.edu/.

[26] J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. Ben Belgacem, B. Chopard, D. Groen, P.V. Coveney, and A.G. Hoekstra. Distributed multiscale computing with MUSCLE2, the Multiscale Coupling Library and Environment. *Journal of Computational Science*, 5(5):719–731, 2014.

[27] Joris Borgdorff, Jean-Luc Falcone, Eric Lorenz, Carles Bona-Casas, Bastien Chopard, and Alfons G. Hoekstra. Foundations of distributed multiscale computing: Formalization, specification, and analysis. *Journal of Parallel and Distributed Computing*, 73(4):465–483, 2013.

[28] Joris Borgdorff, Mariusz Mamonski, Bartosz Bosak, Krzysztof Kurowski, Mohamed Ben Belgacem, Bastien Chopard, Derek Groen, Peter V. Coveney, and Alfons G. Hoekstra. Distributed Multiscale Computing with MUSCLE 2, the Multiscale Coupling Library and Environment. *CoRR*, abs/1311.5740, 2013.

[29] Borgdorff, Joris and Mamonski, Mariusz and Bosak, Bartosz and Groen, Derek and Ben Belgacem, Mohamed and Kurowski, Krzysztof and Hoekstra, Alfons G. Multiscale Computing with the Multiscale Modeling Library and Runtime Environment. *Procedia Computer Science*, 18(0):1097–1105, 2013. 2013 International Conference on Computational Science.

[30] Inc Bright Computing. *Cloudbursting Manual*. ClusterVision, revision: 5640 edition, 10 2014.

[31] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid. In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, volume 1, pages 283–289 vol.1, May 2000.

[32] A. Caiazzo, D. Evans, J-L. Falcone, J. Hegewald, E. Lorenz, B. Stahl, D. Wang, J. Bernsdorf, B. Chopard, J. Gunn, R. Hose, M. Krafczyk, P. Lawford, Rod Smallwood, D. Walker, and Alfons Hoekstra. A Complex Automata approach for In-stent Restenosis: two-dimensional multiscale modeling and simulations. *J. of. Computational Sciences*, 2010. DOI: 10.1016/j.jocs.2010.09.002.

[33] CERN. http://home.web.cern.ch/.

[34] Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro, editors. *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*, volume 7312 of *Lecture Notes in Computer Science*. Springer, 2012.

[35] B. Chopard and M. Droz. *Cellular Automata Modeling Of Physical Systems*. Aléa-Saclay. Cambridge University Press, 1998.

[36] Bastien Chopard, Mohamed Ben Begacem, Andrea Parmigiani, and Jonas Latt. A lattice Boltzmann Simulation of the Rhone river. *International Journal of Modern Physics C*, page 1340008, 2013.

[37] Bastien Chopard, Jean-Luc Falcone, Alfons G Hoekstra, and Joris Borgdorff. A Framework for Multiscale and Multiscience Modeling and Numerical Simulations. In Cristian Calude, Jarkko Kari, Ion Petre, and Grzegorz Rozenberg, editors, *LNCS 6714*, pages 2–8. Springer, Berlin, Heidelberg, 2011.

[38] E. Ciepiela, D. Harezlak, J. Kocot, T. Bartyński, M. Kasztelnik, P. Nowakowski, T. Gubala, M. Malawski, and M. Bubak. Exploratory programming in the virtual laboratory. In *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*, pages 621–628, October 2010.

[39] Ciepiela, E. and Nowakowski, P. and Kocot, J. and Haręźlak, D. and Gubała, T. and Meizner, J. and Kasztelnik, M. and Bartyński, T. and Malawski, M. and Bubak, M. Managing Entire Lifecycles of e-Science Applications in the GridSpace2 Virtual Laboratory–From Motivation through Idea to Operable Web-Accessible Environment Built on Top of PL-Grid e-Infrastructure. *Building a National Distributed e-Infrastructure–PL-Grid*, pages 228–239, 2012.

[40] Cilk Plus. https://software.intel.com/en-us/intel-cilk-plus.

[41] ClusterVision Company. https://azure.microsoft.com/.

[42] Corona. https://github.com/facebook/hadoop-20/tree/master/src/contrib/corona.

[43] Alessandro Costantini, Riccardo Murri, Sergio Maffioletti, and Antonio Laganà. A Grid Execution Model for Computational Chemistry Applications Using the GC3Pie Framework and the Apppot VM Environment. In *Proceedings of the 12th International Conference on Computational Science and Its Applications - Volume Part I*, ICCSA'12, pages 401–416, Berlin, Heidelberg, 2012. Springer-Verlag.

[44] Cray-XC supercomputer. http://www.cray.com/Products/Computing/XC/.

[45] Swiss National Supercomputing center (CSCS). http://cscs.ch/.

[46] Joseph O Dada and Pedro Mendes. Multi-scale modelling and simulation in systems biology. *Integrative Biology*, 2(3):86–96, 2011.

[47] Darest. http://www.darest.com/.

[48] Da Di. *Histoire du peuplement de l'Asie orientale révèlée par le système HLA*. PhD thesis, University of Geneva, Switzerland, 03/25 2013.

[49] E. Laure et al. Programming the Grid with gLite. Technical Report EGEE-TR-2006-001, CERN, march 2006.

[50] Enabling Desktop Grids for e-Science (EDGeS). http://www.edges-grid.eu/.

[51] European desktop Grid Initiative (EDGI). http://edgi-project.eu/.

[52] P. Eerola, B. Kónya, O. Smirnova, T. Ekelöf, M. Ellert, J. R. Hansen, J. L. Nielsen, A. Wäänänen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter. The NorduGrid Production Grid Infrastructure, Status and Plans. In *Proceedings of the 4th International Workshop on Grid Computing*, GRID '03, pages 158–, Washington, DC, USA, 2003. IEEE Computer Society.

[53] EGI. http://www.egi.eu/.

[54] Integrated Sustainable Pan-European Infrastructure for Researchers in Europe (EGI-InSPIRE project). https://www.egi.eu/about/egi-inspire/.

[55] Elasticluster web project. http://gc3-uzh-ch.github.io/elasticluster/.

[56] EMOTIVE Cloud. http://autonomic.ac.upc.edu/emotive/.

[57] enviroGRIDS FP-7 project. http://www.envirogrids.net/.

[58] Eucalyptus project. www.eucalyptus.com.

[59] Facebook. https://www.facebook.com/.

[60] Jean-Luc Falcone, Bastien Chopard, and Alfons Hoekstra. MML: towards a Multi-scale Modeling Language. *Procedia Computer Science*, 1(1):819–826, 2010.

[61] Tiziana Ferrari and Luciano Gaido. Resources and Services of the EGEE Production Infrastructure. *Journal of Grid Computing*, 9(2):119–133, 2011.

[62] Ian T. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *J. Comput. Sci. Technol.*, 21(4):513–520, 2006.

[63] James Frey, Todd Tannenbaum, Miron Livny, Ian T. Foster, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[64] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogeneous computing environment. In Vassil Alexandrov and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 180–187. Springer Berlin Heidelberg, 1998.

[65] GEANT project. http://www.geant.net/.

[66] Wolfgang Gentzsch, Denis Girou, Alison Kennedy, Hermann Lederer, Johannes Reetz, Morris Riedel, Andreas Schott, Andrea Vanni, Mariano Vazquez, and Jules Wolfrat. DEISA-Distributed European Infrastructure for Supercomputing Applications. *Journal of Grid Computing*, 9(2):259–277, 2011.

[67] Horacio González-Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[68] Google App Engine. https://cloud.google.com/appengine/.

[69] W.H. Graf and M.S. Altinakar. *Hydraulique fluviale: écoulement et phénomènes de transport dans les canaux à géométrie simple*. Traité de génie civil de l'Ecole polytechnique fédérale de Lausanne. Presses Polytechniques Universitaires Romandes, 2008.

[70] D. Groen, S.J. Zasada, and P.V. Coveney. Taxonomy of Multiscale Computing Communities. In *e-Science Workshops (eScienceW), 2011 IEEE Seventh International Conference on*, pages 120–127, dec. 2011.

[71] Derek Groen, Steven Rieder, Paola Grosso, Cees de Laat, and Simon Portegies Zwart. A Light-Weight Communication Library for Distributed Computing. *CoRR*, abs/1008.2767, 2010.

[72] Derek Groen, Simon Portegies Zwart, Tomoaki Ishiyama, and Jun Makino. High-performance gravitational N-body simulations on a planet-wide-distributed supercomputer. *Computational Science & Discovery*, 4(1):015001, 2011.

[73] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[74] Tomasz Gubala, Marian Bubak, and Peter M.A. Sloot. *Semantic Integration of Collaborative Research Environments*, chapter XXVI, pages 514–530. Information Science Reference IGI Global, 2009.

[75] Thilina Gunarathne, Tak-Lon Wu, Jong Youl Choi, Seung-Hee Bae, and Judy Qiu. Cloud computing paradigms for pleasingly parallel biomedical applications. *Concurrency and Computation: Practice and Experience*, 23(17):2338–2354, 2011.

[76] Hadoop system. http://hadoop.apache.org/.

[77] Clark AG. Hartl DL. *Principles of Population Genetics*. Sinauer Associates, Inc., Harvard and Cornell Universities, fourth edition, 2007.

[78] Haiwu He, Gilles Fedak, Péter Kacsuk, Zoltan Farkas, Zoltán Balaton, Oleg Lodygensky, Etienne Urbah, Gabriel Caillat, Filipe Araujo, and Ad Emmen. Extending the EGEE Grid with XtremWeb-HEP Desktop Grids. In *CCGRID*, pages 685–690. IEEE, 2010.

[79] Raphael Helaers and Michel Milinkovitch. MetaPIGA v2.0: maximum likelihood large phylogeny estimation using the metapopulation genetic algorithm and other stochastic heuristics. *BMC Bioinformatics*, 11(1):379, 2010.

[80] HES-SO. High-Level API for XtremWeb-CH: Concurrent Computing with a Uniform Interface. Technical report, University of Applied Sciences, Western Switzerland, 2011.

[81] A. Hoekstra, Falcone J.-L, A. Caiazzo, and B. Chopard. Multi-scale modeling with cellular automata: The complex automata approach. In H. Umeo et al., editor, *ACRI 2008*, volume LNCS 5191, pages 192–199. Springer-Verlag Berlin Heidelberg 2008, 2008.

[82] AlfonsG. Hoekstra, Alfonso Caiazzo, Eric Lorenz, Jean-Luc Falcone, and Bastien Chopard. Complex Automata: Multi-scale Modeling with Coupled Cellular Automata. In Jiri Kroc, Peter M.A. Sloot, and Alfons G. Hoekstra, editors, *Simulating Complex Systems by Cellular Automata*, volume 0 of *Understanding Complex Systems*, pages 29–57. Springer Berlin Heidelberg, 2010.

[83] G.D. Ingram, I.T. Cameron, and K.M. Hangos. Classification and analysis of integrating frameworks in multiscale modelling. *Chemical Engineering Science*, 59:2171–2187, 2004.

[84] Intel Xeon Phi. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[85] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(6):931–945, jun 2011.

[86] J. Borgdorff and E. Lorenz and A.G. Hoekstra and J. Falcone and B. Chopard. A Principled Approach to Distributed Multiscale Computing, from Formalization to Execution. In *e-Science Workshops (eScienceW), 2011 IEEE Seventh International Conference on*, pages 97–104, dec. 2011.

[87] J. Borgdoroff, M. Ben Belgacem C. Bona-Casas L. Fazendeiro D. Groen O. Hoenen A. Mizeranschi J. L. Suter D. Coster P. V. Coveney W. Dubitzky A. G. Hoekstra P. Strand and Chopard, B. Performance of Distributed Multiscale Simulations. *Journal of Philosophical Transactions A*, 372(2021), August 2014.

[88] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, Harvey J. Wasserman, and N.J. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168, 2010.

[89] K-computer. http://www.aics.riken.jp/en/k-computer/about/.

[90] Nicholas T. Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-enabled Implementation of the Message Passing Interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, May 2003.

[91] Susan Coghlan Brent Draney Katherine Yelick and Richard Shane Canon. The Magellan Report on Cloud Computing for Science. Technical report, U.S. Department of Energy Office of Advanced Scientific Computing Research (ASCR), 2011.

[92] Daniel Katz, Scott Callaghan, Robert Harkness, S. Jha, Krzysztof Kurowski, Steven Manos, Sudhakar Pamidighantam, Marlon Pierce, Beth Plale, Carol Song, and John Towns. Science on the TeraGrid. *Computational Methods in Science and Technology*, 05/2010 2010. Special Issue 2010.

[93] Piotr Kopta, Michal Kulczewski, Krzysztof Kurowski, Tomasz Piontek, Pawel Gepner, Mariusz Puchalski, and Jacek Komasa. Parallel application benchmarks and performance evaluation of the Intel Xeon 7500 family processors. *Procedia Computer Science*, 4(0):372–381, 2011. Proceedings of the International Conference on Computational Science, {ICCS} 2011.

[94] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *SIGPLAN Not.*, 35(5):145–156, May 2000.

[95] E Laure, F Hemmer, F Prelz, S Beco, S Fisher, M Livny, L Guy, M Barroso, P Buncic, Peter Z Kunszt, A Di Meglio, A Aimar, A Edlund, D Groep, F Pacini, M Sgaravatto,

and O Mulmo. Middleware for the next generation Grid infrastructure. Technical Report EGEE-PUB-2004-002, CERN, 2004.

[96] M. Leyton and J.M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296, Feb 2010.

[97] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988.

[98] Charles Loomis, Mohammed Airaj, Marc-Elian Bégin, Christophe Blanchet, Vangelis Floros, and Clément Gauthey. Final Report on StratusLab Adoption. Technical report, StratusLab project, 2012.

[99] MAD software. http://www.mapper-project.eu/web/guest/mad-mame-ew.

[100] Steven Manos, Marco Mazzeo, Owain Kenway, Peter V. Coveney, Nicholas T. Karonis, and Brian Toonen. Distributed mpi cross-site run performance using mpig. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 229–230, New York, NY, USA, 2008. ACM.

[101] MAPPER project. http://www.mapper-project.eu/.

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

[103] MetaPIGA 2 - Large phylogeny estimation. http://www.metapiga.org/.

[104] Microsoft Cloud: Azure. https://azure.microsoft.com/.

[105] Sébastien Miquée, Raphaël Couturier, David Laiymani, Nabil Abdennhader, Mohamed Ben Belgacem, Marko Niinimaki, and Marc Sauget. Gridification of a radiotherapy dose computation application with the xtremweb-CH environment. In *Proceedings of the 6th international conference on Advances in grid and pervasive computing*, GPC'11, pages 188–197, Berlin, Heidelberg, 2011. Springer-Verlag.

[106] Mohamed Ben Belgacem, Nabil Abdennadher, Marko Niinimaki. Programming distributed medical applications with XWCH2. In *Proceedings of HealthGrid 2010*, volume 159, France, June 2010.

[107] Mohamed Ben Belgacem, Nabil Abdennadher, Marko Niinimaki. *Desktop Grid Computing*, chapter The XtremWebCH Volunteer Computing Platform (3). Numerical Analysis and Scientific Computing Series. Chapman and Hall/CRC, June 25, 2012.

[108] M.M. Morgan and AS. Grimshaw. Genesis II - Standards Based Grid Computing. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 611–618, May 2007.

[109] MPICH. http://www.mpich.org/.

[110] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[111] MUSCLE2. http://apps.man.poznan.pl/trac/muscle.

[112] NVIDIA. http://www.nvidia.com/.

[113] NVIDIA. NVIDIA CUDA Programming Guide 2.0. Technical report, NVIDIA Corporation, 2008.

[114] Palabos toolkit. http://www.palabos.org/.

[115] OPEN FOAM. http://www.openfoam.com/.

[116] OpenMPI. http://www.open-mpi.org/.

[117] OpenNebula. http://opennebula.org/.

[118] OpenStack poject. http://www.openstack.org/.

[119] Parmigiani,Andrea and Latt, Jonas and Ben Belgacem, Mohamed and Chopard, Bastien. A lattice Boltzmann Simulation of the Rhone river. *International Journal of Modern Physics C*, 11(24):1340008, 2013.

[120] Pham van Thang and Bastien Chopard and Laurent Lefèvre and Diemer Anda Ondo and Eduardo Mendes. Study of the 1D lattice Boltzmann shallow water equation and its coupling to build a canal network. *Journal of Computational Physics*, 229(19):7373–7400, 2010.

[121] PRACE. http://www.prace-ri.eu/.

[122] Weizhong Qiang and Aleksandr Konstantinov. Towards cross-middleware authentication and single sign-on for ARC Grid middleware. *Computer Science - Research and Development*, 23(3-4):267–274, 2009.

[123] Lavanya Ramakrishnan. Magellan: experiences from a Science Cloud. Technical report, Lawrence Berkeley National Laboratory, 2013.

[124] Lavanya Ramakrishnan and Beth Plale. A Multi-dimensional Classification Model for Scientific Workflow Characteristics. In *Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*, Wands '10, pages 4:1–4:12, New York, NY, USA, 2010. ACM.

[125] Frank B. Schmuck and Roger L. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In Darrell D. E. Long, editor, *FAST*, pages 231–244. USENIX, 2002.

[126] Riccardo Murri Sergio MAFFIOLETTI and Tyanko Aleksiev. Computational workflows with GC3Pe. In *EGI Community Forum 2012 / EMI Second Technical Conference*, March 2012.

[127] Shibboleth. http://shibboleth.net/.

[128] SixSq Sàrl. http://sixsq.com/.

[129] SLURM batch system. http://www.schedmd.com/.

[130] Swiss Multi-Science Computing Grid. http://www.switch.ch/aaa/projects/detail/UZH.2.

[131] David Snelling. *Unicore and the Open Grid Services Architecture*, pages 701–712. John Wiley & Sons, Ltd, 2003.

[132] David McG. Squire, Wolfgang Müller, Henning Müller, and Thierry Pun. Content-based query of image databases: inspirations from text retrieval. *Pattern Recognition Letters*, 21(13–14):1193 – 1198, 2000. Selected Papers from The 11th Scandinavian Conference on Image.

[133] StartCluster Package. http://star.mit.edu/cluster/.

[134] stratusLab project. http://stratuslab.eu/.

[135] Michael C. Sukop and Daniel T. Thorne. *Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers*. Springer, 1 edition, January 2007.

[136] SuperMUC machine. http://www.lrz.de/services/compute/supermuc/.

[137] Soil and Water Assessment Tool. http://swat.tamu.edu/.

[138] SZTAKI Desktop Grid. http://szdg.lpds.sztaki.hu/szdg/.

[139] S.J.R. Taylor, M. Ghorbani, N. Mustafee, S.J. Turner, T. Kiss, D. Farkas, S. Kite, and S. Strassburger. Distributed computing and modeling amp; simulation: Speeding up simulations and creating large models. In *Simulation Conference (WSC), Proceedings of the 2011 Winter*, pages 161–175, December 2011.

[140] The Globus Security Team. Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective. Technical report, University of Chicago, September 2005.

[141] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, February 2005.

[142] Top500 ranking. http://www.top500.org/lists/2014/11/.

[143] TORQUE. http://www.pbsworks.com/?AspxAutoDetectCookieSupport=1.

[144] VENUS-C European project. http://www.venus-c.eu/.

[145] E W., X. Li, W. Ren, and E. Vanden-Eijnden. Heterogeneous Multiscale Methods: A Review. *Commun. Comput. Phys.*, 2:367–450, 2007.

[146] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 1163–1171, Piscataway, NJ, USA, 2010. IEEE Press.

[147] Extreme Science and Engineering Discovery Environment (XSEDE). https://www.xsede.org/.

[148] XtremwebCH Desktop platform (XWCH). http://www.lsds-rg.org/xtremwebch/.

[149] Akinori Yonezawa, Tadashi Watanabe, Mitsuo Yokokawa, Mitsuhisa Sato, and Kimihiko Hirao. Advanced Institute for Computational Science (AICS): Japanese National High-Performance Computing Research Institute and Its 10-petaflops Supercomputer K. In *State of the Practice Reports*, SC '11, pages 13:1–13:8, New York, NY, USA, 2011. ACM.

[150] Jia Yu and Rajkumar Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34(3):44–49, September 2005.

[151] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.